RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN
INSTITUT FÜR INFORMATIK II
ARBEITSGRUPPE TECHNISCHE INFORMATIK

Diplomarbeit

Strukturelle und funktionale Beschreibung dynamisch rekonfigurierbarer Hardware in SystemC

Armin Felke

27. Dezember 2007

Betreut durch:
Prof. Dr. Joachim K. Anlauf
Dipl.-Inform. Andreas Raabe

Eidesstattliche Erklärung

Hiermit erkläre ich, Armin Felke, dass ich die vorliegende Arbeit selbstständig verfasst und die darin beschriebenen Implementierungen selbstständig entwickelt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und habe diese kenntlich gemacht.

Inhaltsverzeichnis

Aufgabenstellung

Aufgabe dieser Arbeit ist es, die Simulation von dynamischer Rekonfiguration (DR) auf höheren Abstraktionsebenen in SystemC zu ermöglichen. Dazu soll eine existierende Klassenbibliothek namens ReChannel weiterentwickelt werden, die speziell als SystemC-Erweiterung zur Simulation von DR konzipiert worden ist.

Die aktuelle Version dieser Klassenbibliothek soll um Sprachkonstrukte zur Modellierung grundlegender Synchronisationsfunktionalität erweitert werden, die zur Simulation von dynamischer Rekonfiguration auf funktionaler sowie transaktionaler Abstraktionsebene benötigt werden.

Um den Funktionsumfang von SystemC möglichst vollständig abzudecken, soll ferner die Klassenbibliothek ReChannel um die Möglichkeit zur Rekonfiguration von Exports erweitert werden.

Es soll in der ReChannel-Bibliothek außerdem ein Refactoring an einigen bereits bestehenden Klassen durchgeführt werden. Dieses betrifft unter anderem z. B. die Methoden der Rekonfigurationskontrolle sowie die Bedienerfreundlichkeit des Verfahrens zur Anpassung von ReChannel an benutzerdefinierte Kommunikationskanäle.

Motivation

Seit einigen Jahren ist sowohl in der Forschung als auch in der Industrie ein zunehmendes Interesse an dynamisch rekonfigurierbarer Hardware (DRHW) zu beobachten, da diese die Effizienz von Hardware mit der Flexibilität von Software in sich vereint. Die technischen Fortschritte in der Hardwareentwicklung haben u. a. dazu geführt, dass FPGA-Chips mittlerweile serienmäßig die Fähigkeit zur dynamischen Rekonfiguration besitzen. Auf dem Gebiet der Hardwarebeschreibungs- und Entwicklungssoftware hingegen konnte bisher nicht mit der rapiden technischen Entwicklung Schritt gehalten werden, so dass für die Beschreibung und Simulation von komplexen, rekonfigurierbaren Systemen momentan noch kein lückenloser Design-Flow existiert.

Die für abstrakte Systembeschreibungen vorzugsweise verwendete und mittlerweile standardisierte Sprache, SystemC, bietet nativ keine Möglichkeit, DR zu modellieren bzw. zu simulieren. Es sind bereits einige Lösungen bekannt, wie SystemC um die Aspekte von dynamischer Rekonfiguration erweitert werden kann. Doch sind diese entweder lediglich auf eine einzige Abstraktionsebene beschränkt oder sie erzwingen eine Methodologie, die stark von den üblichen SystemC-Beschreibungsstilen abweicht.

Wünschenswert wäre eine Lösung, die es erlaubt, bei der Entwicklung von dynamisch rekonfigurierbaren Systemen auch weiterhin den bewährten und kostensparenden SystemC-Design-Flow – inklusive Refinements und Wiederverwendbarkeit bestehender Module (IP-Reuse) – zu verwenden.

1 Grundlagen

Dieses Kapitel gibt einen Überblick über die grundlegenden Konzepte und Techniken, auf die im weiteren Verlauf aufgebaut wird bzw. die in den folgenden Kapiteln als bekannt vorausgesetzt werden.

1.1 C++

C++ [5] ist eine höhere Programmiersprache, mithilfe derer die Entwicklung von Software unter verschiedensten Programmierparadigmen[1] sowie Abstraktionsstufen möglich ist. Das Anwendungsspektrum reicht von maschinennaher Programmierung (z. B. für Betriebssysteme und Treiber) bis hin zu abstrakterer Programmierung, wie sie z. B. für komplexe Klassenbibliotheken und Anwendungsprogramme benötigt wird. C++ wurde von der Internationalen Organisation für Normung (ISO) standardisiert. Die Sprache verfügt darüber hinaus über eine standardisierte Erweiterungsbibliothek und es existieren zahlreiche Compiler, die hochoptimierten Maschinencode erzeugen können.

Aufgrund ihrer Vielseitigkeit und Effizienz sowie ihrer besonderen Eignung für die Erstellung von Klassenbibliotheken, ist die Sprache C++ auch in hohem Maße für die Beschreibung und Simulation von komponentenbasierten Hardwaremodellen geeignet. Die objektorientierten Sprachfähigkeiten von C++ erfüllen bereits viele der Anforderungen, die an komponentenbasierte Programmierung gestellt werden. Es mangelt aber z. B. an einer nativen Unterstützung für Nebenläufigkeit sowie an einem Zeitmodell. Um diese bestehenden Lücken zu schließen, wurden Hardwarebeschreibungssprachen konzipiert, die sowohl sprachlich als auch funktional auf C++ basieren. Einer der prominentesten Vertreter solcher Hardwarebeschreibungssprachen ist SystemC (siehe Abschnitt 1.3).

Aufbauend auf SystemC und deren Simulationsumgebung, sind daher sämtliche Implementierungen dieser Arbeit ebenfalls in C++ verfasst. Für grundlegende Problemstellungen der Programmierung werden die Datenstrukturen und Algorithmen aus der C++ *Standard Template Library* (STL) verwendet. Die STL ist Bestandteil von C++ und steht standardmäßig in sämtlichen C++-Compilern zur Verfügung. Hier sind u. a. alle benötigten Standard-Container, wie Vektoren, Sets und Maps bereits enthalten.

[4] und [11] wurden als Nachschlagewerk für C++-spezifische Themen verwendet. Als Referenz für potentiell anzuwendende Entwurfsmuster (Design Patterns) diente [6].

[1] Hierzu zählen u. a. die prozedurale, die objektorientierte und die generische Programmierung.

1.2 Boost Quellcode-Bibliothek

Boost [3] ist eine frei verfügbare Klassenbibliothek für C++. Die Boost-Bibliothek gehört zu den bekanntesten und verbreitetsten Klassenbibliotheken für C++. Sie enthält zahlreiche Pakete mit Klassentemplates zur Lösung häufig vorkommender Programmierproblemstellungen. Einige dieser Pakete werden sogar in den nächsten C++-Standard übernommen. Alle Boost-Klassen befinden sich im Namensraum `boost`.

1.2.1 Funktionsobjekte

Ein Funktionsobjekt ist definiert als ein Objekt, das wie eine Funktion verwendet werden kann. Demzufolge muss mit einem Funktionsobjekt eine Notation erlaubt sein, die syntaktisch identisch zu der eines normalen Funktionsaufrufs ist.

Um dies zu erreichen, kann in C++ der Klammeroperator für die Benutzung mit einem Objekt überladen werden. Die Klasse eines Funktionsobjektes enthält zu diesem Zweck ein oder mehrere Methoden, die den Namen `operator()` tragen. Diese sogenannten Klammeroperator-Methoden werden auf analoge Weise wie alle anderen Objektmethoden deklariert und definiert. Sie besitzen also ebenfalls einen Rückgabetyp, können beliebige Funktionsparameter erhalten und dürfen mit denselben Modifizierern (z. B. `inline` oder `const`) ausgestattet werden – wie der folgende Ausschnitt aus einer Klasse namens `FuncObj` illustriert:

```
int FuncObj::operator()(int a) const {
    return (a * a);
}
int FuncObj::operator()(int a, int b) const {
    return (a * b);
}
```

Der einzige Unterschied der Methode `operator()` zu einer normalen Methode besteht darin, dass sie wie ein Klammeroperator auf dem Objekt aufgerufen werden kann. Die Syntax entspricht somit exakt der eines Funktionsaufrufs:

```
FuncObj f;
int x = f(2);
int y = f(3, 4);
```

Wie die obigen Codezeilen zeigen, ist der Aufruf syntaktisch nicht von dem einer Funktion namens `f` zu unterscheiden. In Wahrheit ist `f` hier aber der Variablenname eines zuvor erzeugten Objekts der Klasse `FuncObj`.

Compiler sind in der Lage, einen solchen Funktionsaufruf dahin gehend zu optimieren, dass hierbei kein zusätzlicher Overhead durch die Verwendung eines Objektes entsteht und außerdem von eventuellen `inline`-Deklarationen profitiert werden kann.

boost::function

Für viele Anwendungsfälle wäre es äußerst praktisch, wenn man gleichartige Funktionsobjekte in einer vereinheitlichten Art und Weise verwenden könnte. Jedoch ist die Aufruf-Schnittstelle, die durch Typ und Reihenfolge der Übergabeparameter sowie eines Rückgabetyps spezifiziert wird, bei Funktionsobjekten nicht eindeutig einem einzigen Datentyp zuordenbar, wie es im Gegensatz dazu bei Funktionspointern der Fall ist. Funktionsobjekte, die eine identische Aufruf-Schnittstelle aufweisen und möglicherweise sogar demselben Zweck dienen, müssen somit nicht zwangsläufig vom selben Typ sein. Eine allgemeine Verwaltung von Funktionsobjekten unterschiedlicher Klassen ist in C++ folglich nicht ohne weiteres möglich.

Die BOOST-Bibliothek bietet als Lösung dieses Problems ein Klassentemplate namens `boost::function` an. Sie ist als Wrapper-Klasse für beliebige Funktionsobjekte mit einer bestimmten Aufruf-Schnittstelle konzipiert und repräsentiert einen speziellen Typ von Funktionsobjekt, dessen Implementation sich während der Laufzeit eines Programms beliebig ändern kann. Mithilfe von `boost::function` ist es daher möglich, alle Funktionsobjekte (sowie Funktionspointer) mit derselben Aufruf-Schnittstelle auf eine einheitliche Art und Weise zu verwalten. Es können `function`-Objekte überall dort eingesetzt werden, wo bisher einfache Funktionspointer zur Anwendung kamen.

Über einen Typparameter des `function`-Klassentemplates wird deklariert, welche Aufruf-Schnittstelle derartige Objekte haben werden. Es wird dazu einfach der gewünschte C++-Funktionstyp übergeben. Die maximal mögliche Parameteranzahl, die dieser Funktionstyp aufweisen darf, ist in der aktuellen BOOST-Implementation auf zehn beschränkt. Es wird davon ausgegangen, dass ein solches Limit für die meisten Anwendungen mehr als ausreichend ist.

Die folgende Codezeile demonstriert die Deklaration eines `function`-Objektes für die Verwendung mit Funktionen, die zwei Parameter sowie einen Rückgabewert vom Typ **int** besitzen:

```
boost::function<int (int, int)> f1;
```

`f1` ist so definiert, dass ihr z. B. eine Funktion `h`, die den Typ **int** h(**int** x, **int** y) hat, zugewiesen und ausgeführt werden kann:

```
f1 = h;              // setzt f1 auf h()
int x = f1(3, 4)     // berechnet x = h(3, 4)
```

Einem `function`-Objekt können ebenfalls beliebige kompatible Funktionsobjekte zugewiesen und anschließend ausgeführt werden, wie folgendes Beispiel unter Verwendung der (auf der vorigen Seite eingeführten) Klasse `FuncObj` zeigt:

```
f1 = FuncObj();
int y = f1(3, 4); // berechnet x = 3*4
```

Bei einer Zuweisung wird standardmäßig eine Kopie des übergebenen Funktionsobjektes abgespeichert, da dies von den BOOST-Autoren als am vorteilhaftesten angesehen

worden ist.

Abgesehen davon kann ein `function`-Objekt grundsätzlich analog zu einem herkömmlichen Funktionspointer verwendet werden. Um eine bereits zugewiesene Funktion wieder zu entfernen, kann explizit der Wert `NULL` gesetzt oder alternativ die Methode `clear()` aufgerufen werden. Es ist auch nicht unbedingt erforderlich, dass ein `function`-Objekt bereits bei der Erzeugung initialisiert wird. Wenn dieses zu Beginn keine Funktion enthält, entspricht dies - in Analogie zu Funktionspointern - der Initialisierung mit einem `NULL`-Pointer. Ob ein `function`-Objekt eine gültige Funktion darstellt, kann jederzeit mit der Methode `empty()` erfragt werden.

1.2.2 Bindung von Funktionsparametern

Das Paket *bind* der BOOST-Bibliothek enthält Konstrukte, mit deren Hilfe Parameterwerte an beliebige Funktionen gebunden werden können. Hierdurch ist es möglich, aus einer bestehenden Funktion eine neue Funktion zu bilden, die eine veränderte Anzahl von Parametern besitzt oder deren Parameterreihenfolge vertauscht wurde. Bei der Ausgangsfunktion kann es sich sowohl um Funktionspointer als auch um Funktionsobjekte oder Pointer auf Objektmethoden handeln. Die neu gebildete Funktion ist ein Funktionsobjekt, das die ursprüngliche Funktion mit gebundenen Parameterwerten oder veränderter Parameterreihenfolge repräsentiert. Dieses neue Funktionsobjekt wird mittels eines Funktionstemplate namens `boost::bind` unter Angabe einer Funktion und der daran zu bindenden Parameter erzeugt.

boost::bind

Das Konzept der Funktion `boost::bind` basiert auf einer Technik der generischen Programmierung, die als *Objekt Generator* (dt.: Objekt-Erzeuger) bezeichnet wird. Ein Benutzer kann `bind` zur Bindung von Parametern an Funktionen verwenden, ohne dass dieser Kenntnis über die in der Implementation tatsächlich verwendeten, generischen Typnamen besitzt. Die jeweiligen Templatefunktionen von `bind` liefern als Rückgabe ein Funktionsobjekt von einem boostinternen Typ zurück.

Sei eine Funktion `h` des Typs **int** h(**int** x, **int** y) gegeben. Mittels des Ausdrucks `boost::bind(h, 3, 4)` wird ein Funktionsobjekt erzeugt, dass an `h` zwei Parameterwerte bindet. Die resultierende Funktion hat keine Übergabeparameter und berechnet `h(3, 4)`, wenn sie aufgerufen wird, wie folgende Codezeile demonstriert:

```
int x = boost::bind(h, 3, 4)(); // berechnet x = h(3, 4)
```

Durch die Verwendung von Platzhalterausdrücken (`_1`, `_2`, `_3`, ...), können auch Funktionsobjekte mit Übergabeparametern erzeugt werden. `_1` steht hierbei für den ersten Übergabeparameter, `_2` für den zweiten, usw. Mittels Platzhalterausdrücken kann u. a. auch die Reihenfolge der Parameter vertauscht werden, wie folgendes Beispiel zeigt:

```
int z = boost::bind(h, _2, _1)(3, 4); // liefert z = h(4, 3)
```

Weiterhin lassen sich mit `bind` beliebige Methoden in Funktionsobjekte verwandeln. Wenn an eine Objektmethode das entsprechende Objekt als erster Parameter gebunden wird, kann das daraus resultierende Funktionsobjekt wie eine normale Funktion verwendet werden.

In folgendem Codeausschnitt wird eine Objektmethode `h` der Klasse `A` an ein Objekt `a` der Klasse `A` gebunden und das resultierende Funktionsobjekt in einem `boost::function`-Objekt (siehe Abschnitt 1.2.1) namens `f` abgespeichert. Für den anschließenden Aufruf von `f` ist weder die Kenntnis des Objektes `a` noch der Klasse `A` erforderlich.

```
class A { public: bool h(int x); } a;

boost:function<bool (int)> f;
f = boost::bind(&A::h, &a, _1);

bool b = f(100); // berechnet b = a.h(100)
```

In Verbindung mit der Klasse `boost::function` ist somit die Schaffung von flexiblen Callback-Mechanismen möglich, bei denen beliebige Funktionen bzw. Methoden abgespeichert werden müssen, um diese zu einem späteren Zeitpunkt ausführen zu können.

1.2.3 Typinformationen

Bei der Programmierung von Templates besteht häufig das Problem, dass man Informationen über die Eigenschaften eines gegebenen Typs bereits zur Kompilationszeit benötigt. Wenn die Funktionalität eines Klassentemplates von einem Typparameter abhängen soll, ist es z. B. wichtig zu wissen, ob es sich bei dem Typ um einen Pointer oder eine Referenz oder um einen skalaren oder zusammengesetzten Datentyp handelt. C++ verfügt über keine Schlüsselwörter, mit denen diese Information ermittelt werden kann.

Die Boost-Bibliothek stellt mit dem Paket *type_traits* Klassentemplates zur Verfügung, mit denen grundlegende Typinformationen zur Kompilationszeit ermittelt werden können. Für jede Eigenschaft, die Datentypen in C++ besitzen können, existiert in Boost eine entsprechende Klasse. Zum Beispiel kann mit `boost::is_pointer` überprüft werden, ob es sich bei einem Typ um einen Pointer handelt. Mit `boost::is_scalar`, `boost::is_function` und `boost::is_object` kann festgestellt werden, ob es sich um einen skalaren, einen Funktions- oder einen Objektdatentyp handelt. Es gibt auch Klassentemplates für komplexere Überprüfungen, wie z. B. `boost::is_base_of` oder `boost::is_polymorphic`.

Die Klassentemplates des *type_traits*-Pakets sind so genannte Traits-Klassen, da sie keine Funktionalität, sondern ausschließlich **typedef**-Deklarationen enthalten.

Ein *type_traits*-Klassentemplate wird mit dem zu überprüfenden Typ parametrisiert und vom Compiler ausgewertet. Mittels der zugrunde liegenden Templatetechniken sind *type_traits*-Klassen entweder von der Basisklasse `boost::true_type` abgeleitet, wenn die zu überprüfende Eigenschaft vorliegt oder von der Basisklasse `boost::false_type`

abgeleitet, wenn der Typ nicht über die gesuchte Eigenschaft verfügt. Ein **enum** namens `value` enthält die boolesche Entsprechung der jeweiligen Basisklasse, d.h. **true** oder **false**.

Der folgende Ausdruck repräsentiert den Wert **true**, da **int**& eine Referenz ist:

```
(boost::is_reference<int&>::value == true)
```

Wohingegen der nachfolgende Ausdruck folglich dem Wert **false** entspricht:

```
(boost::is_pointer<int&>::value == true)
```

Da die Auswertung der Ausdrücke zur Kompilationszeit erfolgt, gibt es für *type_traits* vielfältige Anwendungsmöglichkeiten in der generischen Programmierung. Diese Klassen ermöglichen z. B. die Überprüfung von Bedingungen zur Kompilationszeit sowie eine von Typeigenschaften abhängige Spezialisierung von Klassentemplates.

Darüber hinaus enthält die Boost-Bibliothek auch Klassen, die den Modifizierer eines Typs verändern können. Ein Beispiel hierfür ist `boost::remove_const`, die einen Datentyp von einem eventuell vorhandenen **const**-Modifizierer befreit.

1.2.4 Multi-Index-Container

Das Boost-Paket *multi_index* stellt ein Klassentemplate zur Verfügung, das zur Konstruktion von Containern mit mehreren Indexen verwendet werden kann. Im Gegensatz zu den in der STL bereitgestellten Container-Klassen kann auf die in einem solchen Multi-Index-Container enthaltenen Elemente mittels benutzerdefinierter Indexe zugegriffen werden. Durch die Verwendung mehrerer Indexe können verschiedene Sortierungen für ein und denselben Container definiert werden.

Die Klasse des Multi-Index-Containers, `multi_index_container`, ist generisch. Bei deren Deklaration werden Anzahl und Art der verwendeten Indexe mittels Templateparametern spezifiziert. Hierzu stehen in Boost verschiedene, vordefinierte Indexe, wie z. B. `random_access`, `ordered_unique` und `ordered_non_unique`, zur Verfügung.

Ein einfaches Beispiel eines Set-Containers mit nur einem Index über einer Menge von Integer-Werten sieht z. B. folgendermaßen aus:[2]

```
multi_index_container<
  int,                                  // Elementdatentyp
  indexed_by<                           // Deklaration eines
    ordered_unique<identity<int> > //   einfachen Set-Indexes
  >
> mySetContainer;
```

Ein Index repräsentiert die Schnittstelle für den Zugriff auf die Elemente des Containers. Da sich die Schnittstellen der Indexe an der Standardschnittstelle für STL-Container orientieren, ist deren Benutzung zu großen Teilen mit diesen identisch.

[2] Alle im Codeausschnitt verwendeten Klassen befinden sich im Namensraum „`boost::multi_index`“, welcher aus Übersichtlichkeitsgründen weggelassen worden ist.

Durch die Deklaration von nur einem einzigen Index können mittels Multi-Index-Containern, laut [3], effizientere Varianten der Set-, Map- und Multimap-Container der STL erzeugt werden.

Eine erschöpfende Dokumentation der gesamten Multi-Index-Container-Funktionalität findet sich in [3].

1.3 SystemC

SystemC [7] ist eine standardisierte, auf C++ basierende Hardwarebeschreibungssprache mit einem besonderen Fokus auf Systembeschreibungen. Darüber hinaus ist SystemC gleichzeitig auch eine vollwertige Simulationsumgebung für strukturelle Hardware- und funktionale Systembeschreibungen. Da der Simulations-Kernel und alle Sprachkomponenten als C++-Klassenbibliothek zur Verfügung stehen, kann eine SystemC-Designbeschreibung unmittelbar und ohne zusätzliche Simulationssoftware simuliert und verifiziert werden. Eine standardkonforme Referenzimplementation von SystemC wurde von der *Open SystemC Initiative* (OSCI) entwickelt und steht zur freien Verfügung [13].

Mit SystemC ist es möglich, Funktionen zu beschreiben, die innerhalb eines Simulationszeitmodells wie nebenläufige Prozesse ausgeführt werden. Diese Prozesse können untereinander synchronisiert werden und kommunizieren mittels der so genannten Channel-Komponenten miteinander. Ein Channel besitzt zu diesem Zweck ein Interface, auf das ein Prozess durch einen Methodenaufruf zugreifen kann. Dieses Aufrufprinzip ist auch unter dem Namen „Interface Method Call“ (IMC) bekannt. In SystemC stehen bereits einige vordefinierte Channels zur Verfügung, die bei der Modellierung von Hardware- und Software-Systemen benötigt werden, wie z. B. Signale, FIFOs und Semaphoren. Beliebige zusätzliche Channels können vom Designer definiert werden.

Alle in SystemC verfügbaren Simulationsobjekte können innerhalb von Modulen gekapselt werden. Die Schachtelung von Modulen bildet eine Modulhierarchie, welche die Struktur eines Systems definiert. Module können beliebig viele Ports besitzen, die mit kompatiblen Channeln verbunden werden. Über diese Channels wird die Kommunikation zwischen den Modulen (bzw. den darin enthaltenen Prozessen) modelliert. Mittels der sogenannten Exports kann ein Modul einen in ihm gekapselten Channel (bzw. dessen Interface) für den Zugriff durch externe Prozesse verfügbar machen.

In SystemC gibt es drei verschiedene Arten von Prozessen, den Method-, den Thread- sowie den CThread-Prozess. Die Synchronisation von Prozessen und das Fortschreiten von Simulationszeit werden dadurch modelliert, dass die Ausführung eines Prozesses an festgelegten Punkten ausgesetzt wird und erst beim Auftreten von bestimmten Ereignissen (Events) wieder fortgesetzt wird. Dieser Mechanismus erfolgt je nach Prozesstyp nach unterschiedlichen Regeln.

Ein Method-Prozess kann als einfacher Funktionsaufruf betrachtet werden, der ein Mal komplett ausgeführt wird und dessen Ausführung erst bei Verlassen der Funktion beendet ist. Die Funktion wird jedes Mal erneut aufgerufen, sobald das Aktivierungsereignis aufgetreten ist. Ein Thread-Prozess läuft in einer Endlosschleife ab. Seine Ausführung wird

ausgesetzt, sobald er eine SystemC-Funktion namens `wait()` aufruft. Die Ereignisse, die diesen Prozess wieder aufwecken, werden beim Aufruf von `wait()` als Parameter angegeben. Ein CThread-Prozess verhält sich ähnlich wie ein Thread-Prozess, doch ist dessen Aktivierung ausschließlich an die Taktflanke eines bestimmten Clock-Signals gebunden.

Die Menge der Aktivierungsereignisse (Events), die einem Prozess bereits bei seiner Erzeugung zugewiesen worden sind, werden in der SystemC-Terminologie als *static sensitivity list* (dt.: „statische Sensitivitätsliste“) bezeichnet. Diese Menge kann während der Simulation vorübergehend durch eine andere ersetzt werden, z. B. dadurch, dass `wait()` aufgerufen wird. Die hierbei beteiligten Events bilden dann die so genannte *dynamic sensitivity list*.

Eine ausführliche Beschreibung der Sprache SystemC sowie deren Verwendung kann in [2] nachgelesen werden.

1.3.1 Berechnungsmodelle und Abstraktionsebenen

Hardware-Software-Systeme werden ständig komplexer und erfordern einen immer höheren Entwicklungsaufwand. Da heutzutage bereits ganze Systeme auf einem einzigen Chip (SoC, „System on Chip“) Platz finden, können normale Hardwarebeschreibungssprachen aufgrund ihres geringen Abstraktionsgrades bei Entwurf und Verifikation nicht mehr effektiv verwendet werden. Um die Komplexität eines gesamten Systems bewältigen zu können, muss der Entwicklungsprozess auf einer höheren Abstraktionsebene beginnen.

Der Begriff **Refinement** (dt.: Verfeinerung, Weiterentwicklung) bezeichnet die Fortentwicklung eines Designs von einer hohen, abstrakten Ebene bis hin zu einer hardwarenahen, synthetisierbaren Ebene. Anders als bei der traditionellen Entwicklung, bei der ein System direkt auf einer hardwarenahen Ebene realisiert werden muss, setzt die Strategie der Systementwicklung in SystemC auf einen schrittweisen Verfeinerungsprozess. Beim Refinement wird ausgenutzt, dass in SystemC ein Design in den vielfältigsten Berechnungsmodellen beschrieben werden kann [12].

Ein **Berechnungsmodell** ist definiert durch

(a) ein Zeitmodell (kontinuierlich, diskret oder zeitlos),

(b) die Kommunikation zwischen den einzelnen nebenläufigen Prozessen und

(c) seine Regeln bei der Prozess-Aktivierung.

In traditionellen Hardwarebeschreibungs-Sprachen (wie z.B. VHDL, Verilog) steht meist nur ein einziges statisches Berechnungsmodell zur Verfügung, welches keine Benutzeranpassungen oder Erweiterungen zulässt. Anders dagegen der Simulations-Kernel von SystemC. Diesem liegt ein Berechnungsmodell zugrunde, bei dem es sich um ein sehr allgemein gehaltenes Ereignismodell handelt. Auf dieses Ereignismodell können benutzerangepasste Berechnungsmodelle intuitiv und effizient aufgesetzt werden. Erreicht wird dies mittels der Verfügbarkeit von Events sowie der Möglichkeit, spezialisierte Channels und Interfaces zu erstellen. Im Prinzip kann jedes Berechnungsmodell in SystemC beschrieben werden, das auf einem diskreten Zeitmodell basiert.

Zu den gebräuchlichsten Berechnungsmodellen, die in SystemC auf natürliche Weise modelliert werden können, zählen u. a. Datenfluss-Modelle [9] (wie z. B. Kahn Process Network (KPN), Static-Multirate-Dataflow und Dynamic-Dataflow) sowie ereignisbasierte Modelle, die sich z. B. in Transaction Level Modelling (TLM) und Register Transfer Level Modelling (RTL) klassifizieren lassen.

Datenflussmodelle

Datenflussmodelle, wie KPN, sind effiziente Beschreibungen für digitale, signalverarbeitende Systeme, wie sie z.B. in Multimedia- oder Kommunikationsanwendungen vorkommen. Mittels Datenflussmodellen können beliebige, komplexe Systeme auf einem hohen Abstraktionsniveau beschrieben werden.

Der Datenfluss zwischen den einzelnen Prozessen erfolgt *point-to-point*, d.h. ein Datenweg verbindet immer nur jeweils zwei Prozesse unidirektional miteinander. Jeder dieser Datenwege verhält sich wie eine FIFO, die auf der einen Seite mit Daten gefüllt und auf der anderen Seite wieder ausgelesen wird. Die Synchronisation verläuft implizit über blockierende Lesezugriffe. Ein Prozess, der eine leere FIFO auslesen möchte, wird solange blockiert, bis wieder neue Daten zur Verfügung stehen. Da in der Praxis meist FIFOs mit endlicher Länge verwendet werden, kann auch ein schreibender Prozess blockiert werden. Bei der Modellierung muss also immer darauf geachtet werden, dass es nicht zu einem Deadlock kommt. Static-Multirate- und Dynamic-Dataflow stellen Spezialfälle des allgemeinen KPN dar. Bei diesen Berechnungsmodellen unterliegen die Reihenfolge und die Anzahl der übermittelten Datenpakete bestimmten Regeln.

Transaction Level Modelling

Das Transaction Level Modelling wird vorzugsweise zur intuitiven Beschreibung von Bus-Systemen herangezogen. Die einzelnen Systemkomponenten werden hierbei über einen Bus miteinander verbunden, der die Kommunikation über Transaktionen zwischen ihnen ermöglicht. Eine Transaktion besteht aus einem Datentransfer zwischen zwei Modulen, welche den Bus für diese Zeit für sich allein beanspruchen. Für das tatsächlich verwendete Protokoll interessiert man sich hierbei i. d. R. noch nicht. Dennoch muss die Synchronisation z. B. von einem Bus-Arbiter sichergestellt werden. Dieser vergibt dann die Bus-Zeit an den *Master* und identifiziert den dazugehörigen *Slave* anhand seiner Adresse (Ein Beispiel hierfür ist eine CPU, die auf ein externes RAM-Modul zugreift). Das gesamte System verhält sich i. d. R. *bus-cycle-accurate* (dt.: "Bustakt-getreu"). Eine Simulation in TLM gibt Aufschluss über die spätere Bus-Auslastung eines Systems.

[15] beschreibt die Vorteile von TLM bei der Entwicklung von komplexen Systemen. [14] ist eine von der OSCI entwickelte Klassenbibliothek, die dazu konzipiert ist, eine einheitliche Methodologie für transaktionale Beschreibungen zu definieren.

Register Transfer Level Modelling

Die Modellierung auf Register Transfer Level (RTL, dt.: „Register-Transfer-Ebene", „RT-Ebene") gehört zu den hardwarenächsten Beschreibungen, die in der Praxis Verwendung

finden. RTL-Berechnungsmodelle beschreiben digitale Hardware, die *pin-accurate* (dt.: „anschlussgetreu“) ist und einem Taktgeber gehorcht. Die Taktzyklen entsprechen hierbei schon der endgültigen Taktung der Hardware (*cycle-accurate*, dt.: „taktzyklengetreu“). Jeglicher Datenaustausch findet hier zwischen getakteten Registern statt. Der Datenweg wird mittels Signalleitungen zwischen Modulen und Prozessen gebildet. Die Funktionalität eines Prozesses wird häufig als endlicher Automat beschrieben. In den meisten Hardwarebeschreibungssprachen kann auf der RT-Ebene modelliert werden. RTL-Modelle werden standardmäßig als Ausgangspunkt zur vollautomatisierten Hardware-Synthese verwendet.

Abstraktionsebenen in SystemC

In jedem der oben vorgestellten Berechnungsmodelle wird ein System auf einer unterschiedlichen Abstraktionsebene modelliert. Diese Abstraktionsebenen unterscheiden sich durch ihre Nähe zum endgültigen Produkt. Auf der abstraktesten und damit höchsten Ebene rangieren die Datenflussmodelle. Sie enthalten noch keine Details über die reale Hardware und ähneln in ihrem Stil einem parallelen Softwareprogramm. In der SystemC-Refinement-Terminologie fallen sie in die Kategorie der rein funktionalen Beschreibungen, welche in den Varianten mit (TF, *timed functional*) sowie ohne Zeit (UTF, *untimed-functional*) vorkommen. Eine Stufe darunter ist das Transaction Level angesiedelt. Das abstrakte Bus-System wird nur unter dem Aspekt von Transaktionen beschrieben. Genaue Details über die Hardware-Implementierung werden hierbei noch vermieden. Die RT-Ebene ist normalerweise die niedrigste Abstraktionsstufe auf der in SystemC Hardware beschrieben wird.

Abbildung 1.1 zeigt die Hierarchie der Abstraktionsebenen in SystemC. Die obersten drei Ebenen werden auch als Systemebene (engl.: „System Level“) bezeichnet. Nach oben hin nimmt der Abstraktionsgrad der Beschreibung zu. Dieser „Mangel“ an Detailgrad in den oberen Ebenen erlaubt im Gegensatz zu hardwarenahen Beschreibungen deutlich höhere Geschwindigkeiten bei der Simulation. Zudem fällt der Entwicklungsaufwand, der benötigt wird, um ein System direkt in einer jeweiligen Ebene zu implementieren, mit zunehmendem Abstraktionsgrad geringer aus.

1.3.2 Simulationssemantik

Der SystemC-Simulations-Kernel unterscheidet zwischen der Konstruktions- und der Simulationsphase. In der Konstruktionsphase wird die Modulhierarchie angelegt und die innere Struktur dieser Module (d.h. Prozesse, Channels, usw.) erzeugt. In der Simulationsphase beginnt die Simulation, in der Prozesse ausgeführt werden und die Simulationszeit voranschreitet.

[7] spezifiziert genauestens, wann welche SystemC-Funktionen aufgerufen werden dürfen und wann es erlaubt ist, eine bestimmte Simulationskomponente zu erzeugen. Module (abgeleitet von `sc_module`) und Channels (abgeleitet von `sc_prim_channel`) sowie Ports und Exports können ausschließlich während der Konstruktionsphase erzeugt werden. Port- und Exportbindungen dürfen ebenfalls ausschließlich in dieser Phase einge-

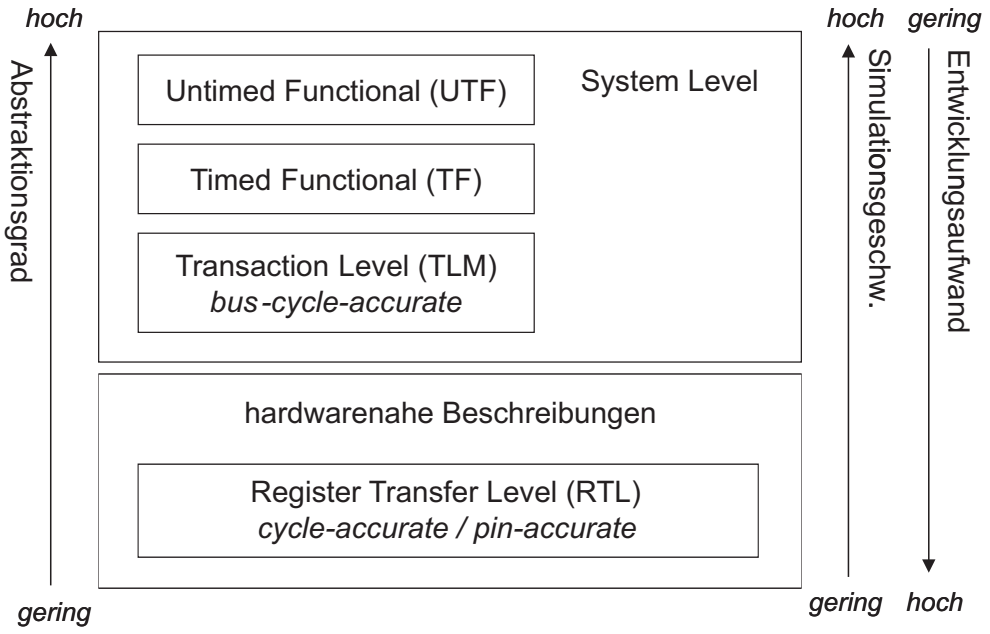

Abbildung 1.1: Hierarchie der Abstraktionsebenen in SystemC

gangen werden. Wenn die Simulation startet, müssen Modulhierarchie und Verbindungsstruktur eines Designs vollständig angelegt worden sein. Prozesse können sowohl in der Konstruktionsphase als auch später noch während der Simulationsphase erzeugt werden. Die während der Simulationsphase erzeugten Prozesse nennen sich dynamische Prozess und werden mittels der SystemC-Funktion `sc_spawn()` erzeugt.

Module, Ports und Channels besitzen zusätzlich einige Callback-Methoden, die vom Simulations-Kernel zu bestimmten Zeitpunkten aufgerufen werden, damit die jeweilige Komponente bestimmte Aktionen, wie z. B. Initialisierungen oder Aufräumarbeiten, durchführen kann. Die Methode `before_end_of_elaboration()` wird vor Ende und `end_of_elaboration()` am Ende der Konstruktionsphase aufgerufen. Die Methoden `start_of_simulation()` und `end_of_simulation()` werden zu Beginn bzw. nach Beendigung der Simulationsphase aufgerufen.

Die Simulation beginnt mit einer Initialisierungsphase in der alle Prozesse[3] zu Initialisierungszwecken ein Mal ausgeführt werden. SystemC verwendet, wie in Hardwarebeschreibungssprachen üblich, Delta-Zyklen zur Emulation von nebenläufigen Prozessen durch einen sequentiell arbeitenden Simulator (wie ihn SystemC darstellt). Des Weiteren arbeiten primitive Channels (wie z. B. Signale und FIFOs) nach dem so genannten *Request-Update-Verfahren*, bei dem ein geschriebener Wert erst am Ende eines Delta-Zyklus in der *Updatephase* übernommen wird.

Eine detaillierte Spezifikation des Simulationsablaufs findet sich in [7].

[3] Mit Ausnahme der Prozesse, die mit `dont_initialize()` deklariert worden sind.

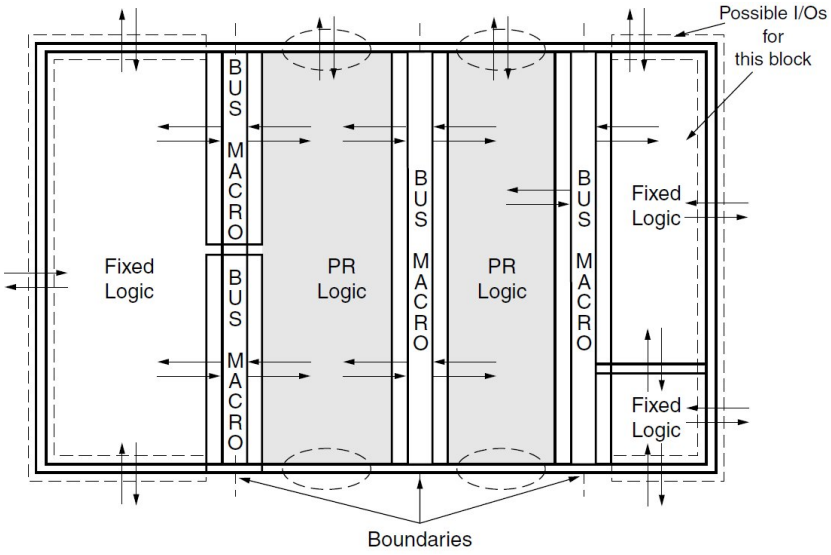

Abbildung 1.2: Schematische Darstellung eines FPGAs mit einem Beispieldesign mit zwei dynamisch rekonfigurierbaren Modulen, deren Kommunikationsschnittstellen durch Bus-Makros fixiert sind (aus [23])

1.4 Dynamisch rekonfigurierbare Hardware

Dynamisch rekonfigurierbare Hardware (DRHW) ist ein allgemeiner Begriff, der sämtliche Hardware umfasst, die sich im laufenden Betrieb neu programmieren lässt. Ein gängiges Beispiel hierfür sind FPGAs (Field Programmable Gate Arrays), die über die Fähigkeit zur dynamischen Rekonfiguration (DR) verfügen. Da es sich bei der FPGA-Technik um Chips handelt, die elektronisch neu programmierbar sind, war der nächste Schritt in deren Entwicklung, die Rekonfiguration auch während des laufenden Betriebs zu ermöglichen.

Ein FPGA besitzt eine Vielzahl an programmierbaren Logikblocken (CLBs), die sich aus kombinatorischer Logik und Flip-Flops zusammensetzen und zu einfachen Schaltfunktionen und/oder Speicherelementen konfiguriert werden können. Die CLBs werden mithilfe eines (ebenfalls rekonfigurierbaren) matrixförmig angeordneten Leitungsnetzwerks verbunden. Die momentane Konfiguration eines FPGAs wird mittels flüchtiger SRAM-Zellen gespeichert. Der Datenstrom, mit dem der SRAM des FPGAs programmiert wird, wird gemeinhin einfach als *Bitstream* bezeichnet. Ein Bitstream enthält u. a. die Informationen darüber, welche Schaltfunktionen in den CLBs berechnet werden und wie die CLBs untereinander verbunden sind. Ein FPGA besitzt meist noch weitere, komplexere Bauteile (z. B. BlockRAMs, Multiplizierer, etc.), die ebenfalls mittels der im Bitstream enthaltenen Informationen programmiert bzw. verschaltet werden können. Auf diese Weise können mit FPGAs hochkomplexe Schaltungen für beliebige Anwendungen gebildet werden. Eine detaillierte Beschreibung der FPGA-Technologie findet sich in [22].

Dynamische Rekonfiguration

Bei der dynamischen Rekonfiguration werden bestimmte Bereiche des Chips neu programmiert, während die restlichen Bereiche hiervon unberührt bleiben. Hierfür ist daher ein partieller Bitstream vonnöten.

Die Firma Xilinx bietet in den FPGAs ihrer Virtex-II-Serie z. B. die Möglichkeit, eine modulbasierte, dynamische Rekonfiguration durchzuführen (siehe [23]). Bei dieser wird ein FPGA während des Betriebs nur in bestimmten, rechteckigen Chip-Bereichen rekonfiguriert. Die Verbindungsleitungen zwischen den statischen und dynamischen Bereichen werden mittels so genannter *Bus-Makros* (engl.: „bus macros“, siehe Abbildung 1.2) fixiert. Die Bus-Makros bilden dabei eine vereinbarte Schnittstelle, so dass die grenzüberschreitende Kommunikation immer über die richtigen Verbindungswege erfolgt. Das Layout der Aufteilung in statische und dynamische Chip-Bereiche sowie die Position der Bus-Makros können vom Designer während der Entwicklung eines Designs beliebig gewählt werden.

1.5 ReChannel

ReChannel [19, 17] ist eine C++-Klassenbibliothek, die als Erweiterung von SystemC um die Aspekte dynamisch rekonfigurierbarer Hardware (DRHW) konzipiert wurde. Die Bibliothek wurde 2005/2006 in der Abteilung für Technische Informatik der Universität Bonn im Rahmen einer studentischen Projektgruppe implementiert (eine Liste der an diesem Projekt beteiligten Personen findet sich in Anhang A).

Motiviert wurde die Entwicklung der ReChannel-Bibliothek durch die Tatsache, dass die in weiten Teilen statischen Spracheigenschaften von SystemC die Möglichkeiten zur Beschreibung und Simulation von dynamischer Rekonfiguration in erheblichem Maße einschränken. Für die korrekte Modellierung des Verhaltens von DRHW müssen einige Voraussetzungen erfüllt sein, die in SystemC nativ nicht zur Verfügung stehen. SystemC verbietet z. B. die Konstruktion bzw. Destruktion von Modulen und Channeln während der Simulationszeit, ebenso wie es auch untersagt ist, Port-Channel-Bindungen nach Abschluss der Konstruktionsphase noch zu schließen oder zu verändern (vgl. dazu 1.3.2).

Die ReChannel-Bibliothek enthält spezielle Konstrukte, mit denen die Einschränkungen hinsichtlich der dynamischen Rekonfigurierbarkeit von Modulen in SystemC überwunden werden können. Mit der in ReChannel verwendeten Methodologie ist es möglich, die Simulation von DR nahtlos in den normalen Design-Flow zu integrieren. Dadurch kann der Kosten und Ressourcen sparende Refinement-Prozess (siehe Abschnitt 1.3.1), wie gewohnt, auch bei der Entwicklung von Systemen mit dynamisch rekonfigurierbarer Hardware durchgeführt werden. Im Gegensatz zu anderen bekannten Lösungsansätzen auf diesem Gebiet (wie z. B. [20] oder [16]), können hierbei sogar bestehende, statische SystemC-Module in rekonfigurierbare Module verwandelt werden. Dies erlaubt dann u. a. von Drittanbietern entwickelte Module wiederzuverwenden sowie die vorhandenen Standard-Tools für die Synthese zu nutzen. Außerdem sieht das Konzept von ReChannel vor, dass SystemC um die erforderliche Funktionalität erweitert wird, ohne dabei einen Eingriff in dessen Simulations-Kernel vorzunehmen. Hierdurch wird die Portabilität von ReChannel zwischen verschiedenen SystemC-Implementationen gewährleistet.

1.5.1 Prinzipielle Funktionsweise

Bevor auf die einzelnen in ReChannel zum Einsatz kommenden Komponenten eingegangen wird, soll im Folgenden kurz die in ReChannel zugrundeliegende Simulationssemantik vorgestellt werden.

Wie in [10] und [19] beschrieben, ist die einfachste und intuitivste Herangehensweise an die Simulation von DR in einer statischen Hardwarebeschreibungssprache, dass man die Kommunikation zwischen dem statischen und den rekonfigurierbaren Design-Bereichen kontrolliert. Nur einige, als „geladen“ ausgewiesene, rekonfigurierbare Module (RM) bekommen die Möglichkeit, auf die Channels der statischen Seite zuzugreifen, während alle anderen als „ungeladen“ gelten und vom Zugriff auf diese Channels ausgeschlossen werden. Entsprechend dazu in umgekehrter Richtung, werden auch die Events der Channels

nur an die geladenen Module weitergeleitet, mit der Folge, dass ungeladene Module von sämtlichen Ereignissen der Außenwelt abgeschnitten sind. Dadurch, dass die ungeladenen Module vom Rest des Designs isoliert vorliegen, wird deren Nicht-Existenz simuliert.

Der Zustand, in dem sich das Gesamtdesign befindet, kann als dessen momentane Konfiguration angesehen werden. Wird nun am Ladezustand mindestens eines rekonfigurierbaren Moduls etwas geändert, so hat man bereits einen Rekonfigurationsvorgang simuliert. Geschieht dies bei laufender Simulation, so handelt es sich definitionsgemäß um eine dynamische Rekonfiguration.

Geladene Module sind durch ihre Bindung mit externen Kommunikationskanälen an einen bestimmten Ort gebunden. Ein Rekonfigurationsvorgang kann daher scheitern, wenn ein Modul geladen werden soll, für das kein freier Platz mehr verfügbar ist. Ein möglicher Grund hierfür ist, wenn es denselben Platz wie ein anderes Modul beansprucht. In einem solchen Konfliktfall wird die Simulation mit einer Fehlermeldung abgebrochen.

Wenn die Zahl der Rekonfigurationszustände noch etwas vergrößert wird, dann können auch Initialisierungs- sowie Sicherungsvorgänge (bei Aktivierung bzw. Deaktivierung) simuliert werden. ReChannel unterscheidet daher im Prinzip zwischen ungeladenen, inaktiven und aktiven Modulen.

Ungeladene Module sind für die Simulation inexistent. Ein inaktives Modul wird als geladen, aber nicht bereit für den normalen Betrieb angesehen, beispielsweise weil der interne Zustand noch nicht fertig initialisiert worden ist. Eine Aktivierung (d. h. der Wechsel von „inaktiv“ nach „aktiv“) kann erst im Anschluss an eine erfolgreiche Ladeoperation erfolgen. Der umgekehrte Vorgang ist die Deaktivierung eines momentan aktiven Moduls, mit der Folge, dass dieses sich daraufhin wieder im Zustand „inaktiv“ befindet. Ein inaktives Modul kann entladen werden, um es aus der momentanen Konfiguration zu entfernen. Der von ihm belegte Platz wird dadurch freigegeben und steht hiernach für Rekonfigurationsvorgänge anderer Module zur Verfügung.

Die Rekonfigurationskontrolle ist diejenige Komponente im Design, die Rekonfigurationsvoränge veranlassen kann. Es darf mehrere dieser Kontrollkomponenten in einem Design geben, doch muss jedes rekonfigurierbare Modul genau einer einzigen Rekonfigurationskontrolle zugeordnet werden. Diese zugewiesene Kontrollkomponente ist dann exklusiv für die Rekonfiguration des entsprechenden Moduls zuständig.

Der eigentliche Rekonfigurationsvorgang wird durch das Verstreichen einer bestimmten Zeitspanne simuliert, gefolgt von dem jeweiligen Zustandsübergang des Moduls. Es besteht die Möglichkeit, für jedes Modul gesondert festzulegen, wieviel Zeit ein jeweiliger Rekonfigurationsvorgang dauern soll. Auf diese Weise können z. B. von der Bitstreamgröße eines Moduls abhängige Konfigurationszeiten modelliert werden.

Rekonfigurierbare Module entscheiden selbstständig, wann sie bereit sind, deaktiviert zu werden. Bevor ein aktives Modul deaktiviert werden kann, wird zuerst abgewartet bis alle zu diesem Zeitpunkt noch laufenden, externen Zugriffe beendet sind, um einen korrekten Simulationsablauf garantieren zu können.

Die zentralen Komponenten, die in ReChannel zur Umsetzung der oben beschriebenen Simulationssemantik zum Einsatz kommen, sind neben den rekonfigurierbaren Modulen und der Rekonfigurationskontrolle die sogannten *Portals* und *Accessoren*.

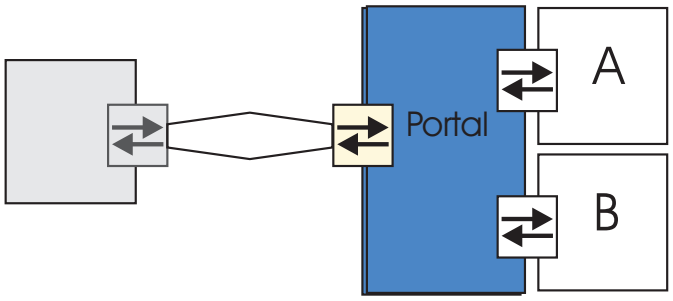

Abbildung 1.3: Portals werden verwendet, um die Kommunikation zwischen einem Channel des statischen Design-Bereichs (linke Seite) und den rekonfigurierbaren Modulen (rechte Seite) kontrollieren zu können. (aus [19])

1.5.2 Das Portal

Die zentrale Voraussetzung für das in ReChannel verwendete Verfahren zur Simulation von DR ist, dass die Kommunikation zwischen dem statischen Design-Bereich und den rekonfigurierbaren Modulen kontrolliert werden kann. Zu diesem Zweck führt ReChannel eine spezielle Hilfskomponente ein, die durch eine generische Klasse namens `rc_portal` repräsentiert wird – das sogenannte *Portal* (siehe Abbildung 1.3).

In einem Design müssen zu Simulationszwecken sämtliche Ports der rekonfigurierbaren Module mit einem kompatiblen Portal verbunden werden. An ein Portal können beliebig viele Ports jeweils unterschiedlicher rekonfigurierbarer Module gebunden werden. Es ist allerdings nicht erlaubt, dass ein Modul mehr als einen seiner Ports an ein und dasselbe Portal bindet.

Jedes Portal wird zudem mit einem Channel verbunden, der dem statischen Design-Bereich angehört. Dieser Channel – im Folgenden auch als **statischer Channel** bezeichnet – stellt das Ziel der Zugriffe, die von den Modulen auf der rekonfigurierbaren Seite ausgehen, dar. Portals repräsentieren in der Simulation somit (in Analogie zu den in 1.4 beschriebenen Bus-Macros) die Verbindungsstellen zwischen rekonfigurierbaren und nichtrekonfigurierbaren Bereichen.

Im Normalfall darf während der Simulation maximal eines der an einem Portal gebundenen rekonfigurierbaren Module zur gleichen Zeit geladen sein. Für eventuelle Optimierungszwecke bzw. wenn die Ortsbindung von Modulen nicht modelliert werden muss, kann auch eine höhere Maximalzahl gleichzeitig ladbarer Module für das jeweilige Portal festgelegt werden.

1.5.3 Der Accessor

Bei der Bindung eines Portals mit dem Port eines rekonfigurierbaren Moduls wird jener in Wirklichkeit nicht mit dem Portal selbst, sondern mit einem neu erzeugten Exemplar eines weiteren Hilfsobjektes, dem sogenannten *Accessor*, verbunden. Der Accessor implementiert das jeweilige Interface des statischen Channels und kann somit anstelle dessen an den Port gebunden werden.

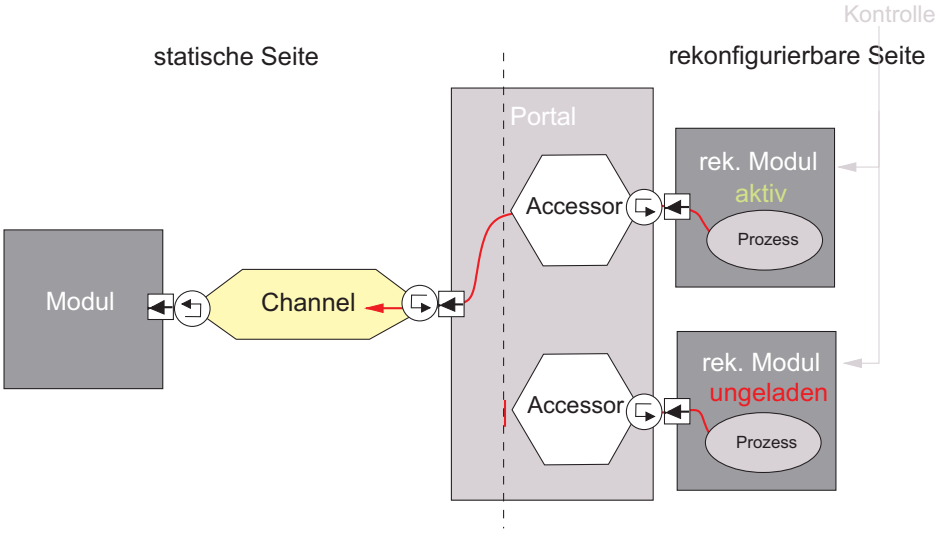

Abbildung 1.4: Weiterleitung von Zugriffen: Der Accessor des aktiven Moduls leitet Interface Method Calls (IMCs) an den statischen Channel weiter. Der Accessor des ungeladenen Moduls hingegen blockt die IMCs auf den Channel ab, wodurch dieses Modul vom Rest des Designs isoliert wird.

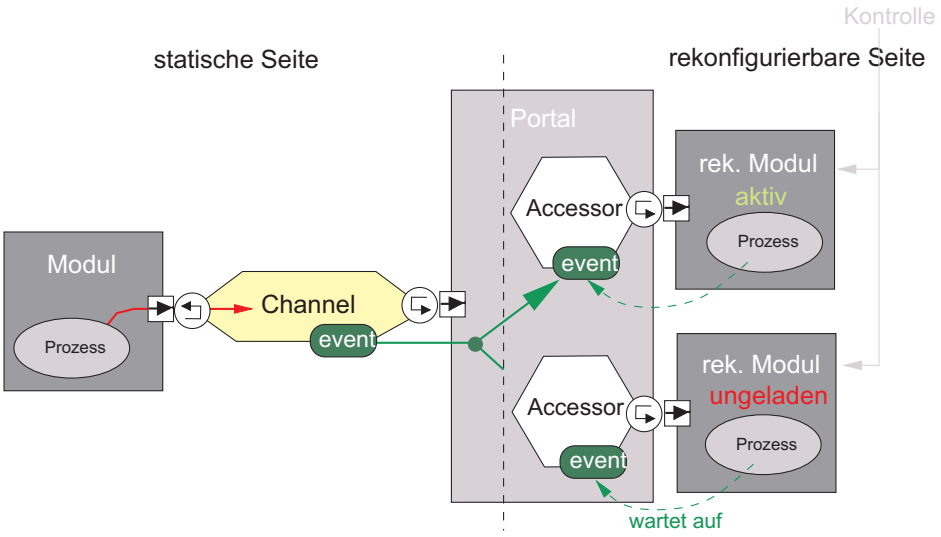

Abbildung 1.5: Weiterleitung von Events: Für jedes Event im statischen Channel besitzt der Accessor ein korrespondierendes Event. Die Prozesse innerhalb der rekonfigurierbaren Module übernehmen diese Events in ihre Sensitivity-List und sind aufgrund dessen nicht direkt mit den Events des statischen Channels verbunden. Alle Events, die im Channel auftreten, werden vom Portal nur an den aktiven Accessor weitergemeldet. Der Prozess im ungeladenen Modul wird somit nicht aktiviert und ist daher von allen externen Ereignissen abgeschnitten.

Der Accessor hat mehrere Aufgaben. Er sorgt dafür, dass – so wie es SystemC vorschreibt – der Port mit einem Interface verbunden ist, und verhindert, dass ein rekonfigurierbares Modul über diesen Port direkten Zugriff auf einen statischen Channel bekommt. Jegliche Kommunikation zwischen dem Modul und dem statischen Channel kann daher ausschließlich über den Accessor erfolgen, was ihn in die Lage versetzt, die Kommunikation zu kontrollieren (siehe Abbildung 1.4 und 1.5).

Der Accessor wird als fester Bestandteil des verbundenen, rekonfigurierbaren Moduls angesehen, da sein Verhalten direkt vom Rekonfigurationszustand dieses Moduls abhängt:

- Wenn das Modul aktiv ist, hat der Accessor die Aufgabe, die externen Zugriffe des zugehörigen, rekonfigurierbaren Moduls an den statischen Channel weiterzuleiten. In der umgekehrten Richtung dient der Accessor als Medium für die Weiterleitung der Events des Channels an die Prozesse des rekonfigurierbaren Moduls.

- In allen anderen Rekonfigurationszuständen dient der Accessor dazu, die vom rekonfigurierbaren Modul initiierte Kommunikation nach außen hin abzublocken.

  Alle potentiell blockierenden Zugriffe werden dazu solange angehalten, bis das Modul das nächste Mal wieder aktiviert wird.

  Zugriffe, die nicht blockieren dürfen („non-blocking“-Zugriffe), können vom Accessor nicht aufgehalten oder abgeblockt werden und müssen daher zum statischen Channel durchgelassen werden. Es wird in diesem Zusammenhang aber davon ausgegangen, dass ein deaktiviertes Modul keine auslösenden Events mehr von außen erhält und somit automatisch in einen künstlichen Schlafzustand verfällt. Ein Modul in diesem Zustand könnte von sich aus dann keine externen Zugriffe mehr initiieren.

  Da sämtliche, zu einem rekonfigurierbaren Modul gehörigen Prozesse ihre Events nicht vom statischen Channel, sondern von den mit den Ports verbundenen Accessoren bekommen, können alle äußeren Events dadurch effektiv unterdrückt werden.

Die Weiterleitung der Events vom statischen Channel an die aktiven Accessoren wird vom Portal durchgeführt. Die nichtaktiven Accessoren erhalten keine Benachrichtigungen über aufgetretene Events.

Das Portal ist ein Container für Accessor-Objekte, da es eine Liste aller – bei ihm erzeugter – Accessoren verwaltet. Andererseits weiß auch jeder Accessor, mit welchem Portal er in Verbindung steht. Dies ist insofern wichtig, da der Accessor das Portal über die Zustandsänderungen des Moduls informieren muss.

1.5.4 Rekonfiguration

Ein SystemC-Modul erhält die Eigenschaft „rekonfigurierbar“ dadurch, dass es zusätzlich noch von der ReChannel-Klasse `rc_module` abgeleitet wird. Ein durch eine Ableitung von einem SystemC-Modul und der Klasse `rc_module` neu entstandenes Modul besitzt alle Eigenschaften, die für die Simulation von DR benötigt werden. Jedes bestehende Modul kann somit auf einfache Weise in ein **rekonfigurierbares Modul** verwandelt

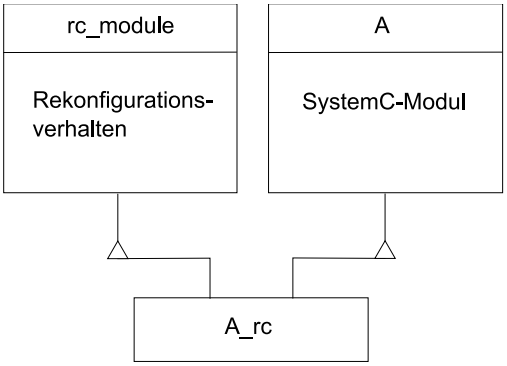

Abbildung 1.6: Ein rekonfigurierbares Modul `A_rc` wird durch Ableitung aus einem bestehenden Modul `A` und der Klasse `rc_module` gebildet.

werden (siehe Abb. 1.6). Die Klasse `rc_module` repräsentiert das Rekonfigurationsverhalten, das benötigt wird, um den Zustand des Moduls und dessen Zusammenwirken mit den anderen ReChannel-Komponenten (wie z. B. den Accessoren) zu modellieren. Rekonfigurierbare Module befinden sich initial immer im ungeladenen Zustand. Bevor die Simulation beginnt, müssen alle Ports des rekonfigurierbaren Moduls an kompatible Portals gebunden und das Modul bei einer Rekonfigurationskontrolle registriert worden sein.

Die **Rekonfigurationskontrolle** wird in ReChannel durch eine Komponente namens `rc_control` repräsentiert. Diese stellt die Benutzerschnittstelle des Rekonfigurationsmechanismus von ReChannel dar. Auf Benutzerseite stellt der Zugriff auf eine Rekonfigurationskontrolle die einzige Möglichkeit dar, Rekonfigurationsvorgänge von Modulen zu veranlassen.

`rc_control` hat die Aufgabe, sämtliche Rekonfigurationsvorgänge der von ihr kontrollierten Module auszuführen. Grundsätzlich stehen in `rc_control` zu diesem Zweck vier verschiedene Methoden zur Verfügung, die jeweils einem Rekonfigurationsvorgang entsprechen. Diese sind `load()`, `activate()`, `deactivate()` sowie `unload()`. Den Methoden wird ein Modul als Parameter übergeben, für das die entsprechende Operation ausgeführt werden soll. Es gibt sowohl blockierende als auch nichtblockierende Varianten dieser Methoden, wobei die nichtblockierenden Methoden mit dem Prefix „nb_“ versehen sind. Die blockierenden Methoden warten, bis der durch sie veranlasste Rekonfigurationsvorgang erfolgreich ausgeführt worden ist. Die nichtblockierenden Methoden hingegen geben lediglich den Befehl zur Rekonfiguration an das Modul weiter, aber warten nicht bis zum Abschluss der Operation.

Durch den Aufruf der Methode `load()` kann der Befehl zum Laden von bisher ungeladenen Modulen gegeben werden. Mit `activate()` wird veranlasst, dass ein bereits geladenes, aber inaktives Modul sich aktiviert und die Kommunikation mit der Außenwelt wieder aufnimmt. Liegt ein Modul im ungeladenen Zustand vor, wenn es aktiviert werden soll, dann wird zuvor automatisch auch ein Ladevorgang durchgeführt. Mittels `deactivate()` wird ein aktives Modul wieder in den inaktiven Zustand versetzt. `unload()`

sorgt dafür, dass ein Modul entladen wird – falls nötig, inklusive eines vorherigen Deaktivierungsvorganges.

Die **Kontrollhierarchie** ist dergestalt, dass die Kontrolle einen Rekonfigurationsbefehl zur Ausführung an das zu rekonfigurierende Modul weiterreicht. Falls das Modul diese Operation ausführen kann, wird es die Durchführung der Operation simulieren, indem es für eine bestimmte Zeit wartet und im Anschluss daran seinen Rekonfigurationszustand selbstständig ändert. Der neue Zustand des Moduls wird an die (mit dessen Ports verbundenen) Accessoren übertragen. Da die Portals überwachen sollen, ob ein Rekonfigurationsvorgang in Konflikt mit einem anderen Modul gerät bzw. ob noch genügend freier Platz zur Verfügung steht, müssen die Accessoren auch das jeweils verbundene Portal über alle Rekonfigurationsvorgänge des Moduls informieren. Dies ist nötig, da die Portals für die Weiterleitung von Events an die Accessoren zuständig sind. Die Portals müssen den Zustand der Accessoren kennen, da ausschließlich aktive Accessoren über das Auftreten von Events im statischen Channel benachrichtigt werden dürfen.

Während der Simulation gibt der Accessor die Anzahl der gerade durch ihn ausgeführten und noch andauernden, externen Zugriffe an das rekonfigurierbare Modul weiter. Zu diesem Zweck wird je begonnenem Zugriff eine Zählervariable namens `pending_accesses` um eins erhöht sowie pro beendetem Zugriff der Zähler wieder um eins verringert.

Wenn ein aktives Modul die Aufforderung zum Deaktivieren erhält, kann es anhand `pending_accesses` überprüfen, ob dies sofort möglich ist (wenn gleich null), oder ob zuerst noch auf die Beendigung eines Zugriffs gewartet werden muss (wenn größer null). Die Accessoren sorgen von da an dafür, dass keine neuen, potentiell blockierenden Zugriffe mehr gestartet werden können.

1.5.5 Funktionsumfang der Spracherweiterungen

Ein rekonfigurierbares Modul wird anhand eines bereits bestehenden, statischen SystemC-Moduls erzeugt. ReChannel stellt ein Makro zur Verfügung, das die hierfür erforderliche Ableitung von der Basisklasse `rc_module` syntaktisch vereinfacht.

Vorausgesetzt ein Modul namens `A` existiert, dann kann daraus ein rekonfigurierbares Modul namens `A_rc` mithilfe des Makros `RC_MODULE` gebildet werden. Das Suffix „_rc“ wird hierbei immer standardmäßig an den Namen des neu gebildeten rekonfigurierbaren Moduls angehängt.

```
RC_MODULE(A) {
    set_loading_time(sc_time(20, SC_MS));
    set_activation_time(sc_time(1.5, SC_MS));
    set_removal_time(sc_time(2, SC_MS));
};
```

Innerhalb der geschweiften Klammern können die Eigenschaften des rekonfigurierbaren Moduls festgelegt werden. Die Dauer eines Lade-, Aktivierungs- sowie Deaktivierungsvorgangs des Moduls kann mit den drei zur Verfügung stehenden Methoden (`set_..._time()`) festgelegt werden.

```
class A_rc : public rc_module, public A {
public:
    A_rc( sc_module_name name_ ) : A(name_) {
        rc_init(); // Initialisierung
        rc_setup(); // Aufruf von rc_setup()
    }
protected:
    inline void rc_setup();
};
virtual inline void A_rc::rc_setup() {
    set_loading_time(sc_time(20, SC_MS));
    set_activation_time(sc_time(1.5, SC_MS));
    set_removal_time(sc_time(2, SC_MS));
}
```

Listing 1.1: Definition eines rekonfigurierbaren Moduls namens `A_rc` durch Ableitung von `rc_module` und einem bestehenden Modul `A`

Das Makro `RC_DERIVE_TEMPLATE_MODULE()` muss anstelle von `RC_MODULE()` verwendet werden, wenn `A` über einen einzelnen Templateparameter verfügt. Ist die Verwendung eines Makros – z. B. aufgrund der Existenz eines von `SC_CTOR()` abweichenden Konstruktors oder des Vorliegens von mehr als einem Templateparameter – nicht möglich, so muss das neue Modul explizit von `rc_module` abgeleitet werden. Die in Listing 1.1 gezeigte, ausgeschriebene Notation ist identisch mit der Definition von `A_rc` mittels des Makros `RC_MODULE()`. Es ist erforderlich, dass `rc_init()` immer im Konstruktor einmal aufgerufen wird, um den innerhalb der Klasse `rc_module` zur Modellierung der Rekonfigurationszustände verwendeten, endlichen Automaten (engl.: "final state machine") zu erzeugen. Die Methode `rc_setup()` wird verwendet, um die Zuweisung aller Eigenschaften, die das rekonfigurierbare Modul betreffen, zu kapseln. Sie wird bei der Konstruktion einmal vom Konstruktor aufgerufen.

Zur Verwendung in `rc_setup()` stehen auch Schlüsselwörter zur Verfügung, die das **Rücksetzverhalten von Datenelementen** des abgeleiteten Basismoduls bei der Deaktivierung festlegen. Es kann z. B. das Methodentemplate `rc_reset()` verwendet werden, um bei Deaktivierung eine Variable oder ein Signal des zugrunde liegenden Basismoduls auf einen festgelegten Wert zurückzusetzen. Das Methodentemplate `rc_preserve()` kann verwendet werden, um explizit kenntlich zu machen, dass eine Variable oder ein Signal ihren Wert bei Deaktivierung behalten sollen.

In Listing 1.2 werden, unter Verwendung von `rc_reset()`, eine Variable `i` und ein Signal names `j` mit dem Wert 0 initialisiert sowie dazu vorgemerkt, bei jeder zukünftigen Deaktivierung des Moduls auf ihren initialen Wert zurückgesetzt zu werden. Für die boolesche Variable `k` wird durch die Verwendung von `rc_preserve()` explizit darauf hingewiesen, dass diese ihren Wert bei einer Deaktivierung beibehalten wird.

```
virtual inline void A_rc::rc_setup() {
        [...]
    rc_reset<int>(i, 0);
    rc_reset<sc_signal<int> >(j, 0);
    rc_preserve<bool>(k);
}
```

Listing 1.2: Festlegung des Rücksetzverhaltens von Variablen und Signalen innerhalb der Methode `rc_setup()`.

```
template<class T>
struct my_accessor : public rc_accessor<my_channel_if<T> > {
    my_accessor(rc_module& m)
      : rc_accessor<my_channel_if<T> >(m) {}
    void write(const T& data)
      { RC_NONBLOCKING_ACCESS(this->channel->write(data)); }
    RC_EVENT_FINDER(my_event);
};
```

Listing 1.3: Die Definition eines Accessors für ein benutzerdefiniertes Interface vom Typ `my_channel_if<T>`. Hierbei wird die Weiterleitung einer nichtblockierenden Schreib-Methode implementiert sowie eine Event-Methode des Interfaces deklariert.

Der **Typ eines Portals** wird bestimmt durch den Typ des Ports, an den es auf seiner rekonfigurierbaren Seite gebunden werden kann. Zu diesem Zweck hat die Klasse `rc_portal` einen Templateparameter, der den Typ des Ports festlegt.

Die Portal-Klasse ist eine vollständig generische Klasse. Für die Verwendung mit beliebigen Channeln müssen keine Anpassungen an ihr vorgenommen werden. Da ReChannel für sämtliche SystemC-Channels eine Implementation eines kompatiblen Accessors enthält, sind die entsprechenden Portals somit bereits fertig zum Gebrauch.

Sollen allerdings **benutzerdefinierte Channels** mit Portals verbunden werden, so schreibt ReChannel vor, dass eine benutzerangepasste Version eines Accessors definiert werden muss, wie es z. B.in Listing 1.3 illustriert wird. Ein Accessor muss von der Klasse `rc_accessor` abgeleitet werden und sämtliche Methoden des Channelinterfaces implementieren, für das er definiert wird. Die Implementation der Methoden besteht lediglich aus der Weiterleitung des Befehls an den statischen Channel. Für die Deklaration der Weiterleitungen stehen spezielle Makros sowohl für potentiell blockierende als auch für nichtblockierende Zugriffe zur Verfügung (siehe [17]). Diese Makros enthalten das gesamte Zugriffsverhalten des Accessors (vgl. Abschnitt 1.5.3). Enthält das Channelinterface Methoden, die Events zurückliefern, so muss anstelle der Definition einer Weiterleitungsmethode das Makro `RC_EVENT_FINDER()` aufgerufen werden (mit dem Na-

```
template<class T>
class rc_port_traits<my_port<T> > {
public:
    typedef my_channel_if<T> if_type;
    typedef my_port<T>       port_type;
    typedef my_accessor<T>   accessor_type;
    static rc_event_descriptor::p_vector events() {
        rc_event_descriptor::p_vector result;
        result.push_back( RC_EVENT_DESCRIPTOR( my_event ));
        return result;
    }
};
```

Listing 1.4: Eine Spezialisierung der Klasse `rc_port_traits`, die alle Typinformationen deklariert, die für die Generierung eines Portals des Typs `rc_portal<my_port<T> >` benötigt werden. Es muss außerdem eine Liste der Events des Channelinterfaces deklariert werden, damit das Portal diese Events weiterleiten kann. Hier enthält die Liste nur das Event `my_event`.

men der Methode als Parameter). Dieses wird benötigt, um ein Event im Accessor zu erzeugen, auf das die Prozesse des rekonfigurierbaren Moduls anstelle des eigentlichen statischen Channels zugreifen sollen.

Zusätzlich zu der Definition eines benutzerdefinierten Accessors muss auch eine Spezialisierung der Klasse `rc_port_traits` definiert werden, wie es beispielsweise in Listing 1.4 illustriert wird. Die Klasse `rc_port_traits` dient dem Zweck, die Typen der beteiligten Kommunikationskomponenten zu kapseln und enthält somit die Typen für Interface, Port sowie Accessor. Des Weiteren muss eine statische Liste aller im Interface des Ports verfügbaren Events angelegt werden. Diese wird benötigt, damit das Portal Kenntnis darüber erlangt, über welche Events das Inteface verfügt, um diese vom statischen Channel an den Accessor weiterleiten zu können. `rc_port_traits` wird über einen einzelnen Typparameter spezialisiert, der einen Port-Typ repräsentieren soll. Durch die Existenz einer solchen Spezialisierung kann die Portal-Klasse – welche ebenfalls über einen Port-Typ parametrisiert wird – auf generische Weise an die Defintionen aller benötigten Typen gelangen.

Die **Deklaration eines Portals** für einen Signal-Port sieht z. B. folgendermaßen aus:

```
rc_portal<sc_out<bool> > out_portal;
```

Für die **Bindung des Portals mit einem statischen Channel** wird der im Portal enthaltene statische Port verwendet. Z. B. ein Signal namens `out_signal` an ein Portal namens `out_portal` zu binden, sieht wie folgt aus:

```
out_portal.static_port(out_signal);
```

Die **Bindung der Ports der rekonfigurierbaren Module mit den Portals** wird über die Methode `bind()` der Portals abgewickelt. Der folgende Code-Ausschnitt zeigt ein rekonfigurierbares Modul `m1`, das zwei FIFO-Ports namens `in` und `out` besitzt, die jeweils an ein kompatibles Portal gebunden werden:

```
my_module_rc m1;
rc_portal<sc_fifo_in<int> >  fifo_in_portal;
rc_portal<sc_fifo_out<int> > fifo_out_portal;
[...]
fifo_in_portal.bind(m1.in);
fifo_out_portal.bind(m1.out);
```

Die **Registrierung eines rekonfigurierbaren Moduls** bei einer Rekonfigurationskontrolle wird mittels der Methode `add()` der Klasse `rc_control` durchgeführt.

Im folgenden Code-Ausschnitt wird eine Rekonfigurationskontrolle namens `ctrl` deklariert, drei Module (`m1`, `m2` und `m3`) mittels `add()` zu dieser Kontrolle hinzugefügt und anschließend das Modul `m1` aktiviert.

```
rc_control ctrl;
[...]
ctrl.add(m1 + m2 + m3);
ctrl.activate(m1);
```

Der „+“-Operator vereinigt rekonfigurierbare Module zu einer speziellen Menge des Typs `rc_module_set`, mithilfe derer beliebig viele Module gleichzeitig an eine Methode der Kontrollkomponente übergeben werden können.

Während der Simulation kann dann mittels der von `rc_control` zur Verfügung gestellten Kontrollmethoden die Rekonfiguration des Designs gesteuert werden:

```
ctrl.unload(m1);    // deaktiviert und entlädt m1
ctrl.load(m2);      // lädt Modul m2
ctrl.activate(m2); // aktiviert Modul m2
```

Des Weiteren gibt es in ReChannel eine Funktionalität, mit der die Deaktivierung eines rekonfigurierbaren Moduls mit speziell präparierten Thread-Prozessen synchronisiert werden kann. Dazu werden diese Prozesse durch ein spezielles Makro bei `rc_module` registriert. An einer bestimmten Stelle in deren Prozessmethode kann nun ein so genannter **Synchronisations-Punkt** gesetzt werden. Dieser besteht aus einem Aufruf der Methode `rc_syncsuspend()`, der den jeweils aufrufenden Prozess exakt so lange warten lässt, bis sämtliche anderen zu synchronisierenden Prozesse ebenfalls einen Synchronisations-Punkt erreicht haben. Dieser Mechanismus hat den beabsichtigten Zusatzeffekt, dass eine potentielle Deaktivierung des Moduls zwingend mit dem Erreichen aller Synchronisations-Punkte verbunden ist. Dadurch wird erreicht, dass sich die Prozesse in einem ganz bestimmten Zustand befinden, wenn abgeschaltet wird.

2 Analyse und Vorarbeiten

In diesem Kapitel soll systematisch untersucht werden, welche Probleme bei der Simulation von DR auf funktionaler bzw. transaktionaler Abstraktionsebene mit der ReChannel-Bibliothek bestehen.

Der letzte Abschnitt enthält eine kurze Auflistung der Änderungen, die an ReChannel vorgenommen wurden. Da im Laufe dieser Arbeit der Entschluss zur Durchführung eines Re-Designs der ReChannel-Bibliothek gefasst worden ist (siehe Kapitel 3 und 4), werden die Änderungen an der ursprünglichen Version zu den Vorarbeiten gezählt.

2.1 Fehlende Funktionalität und ungelöste Probleme

Bevor mit Änderungen an ReChannel bzw. der Implementierung begonnen werden kann, muss zuvor analysiert werden, auf welche Art sich die bekannten Problemstellungen in der vorliegenden Implementation der ReChannel-Bibliothek darstellen.

Da die Lösung von Synchronisationsproblemen eine der primären Zielsetzungen dieser Arbeit ist, wird beschrieben, wie sich die ReChannel-Bibliothek derzeit diesbezüglich verhält (2.1.1). Des Weiteren wird erörtert, worin die Ursache einer prinzipiellen Inkompatibilität mit dem SystemC-Sprachstandard begründet ist (2.1.2) und welche SystemC-Funktionalität zurzeit noch nicht bei der Simulation von DR mit ReChannel verwendet werden kann (2.1.3).

Einige Abschnitte befassen sich mit Thematiken und Problemstellungen, die darüber hinaus während der Analyse der Funktionsweise von ReChannel aufgefallen sind. Es werden die Annahmen, die ReChannel über das Verhalten von deaktivierten Modulen (2.1.4) und über die Verwendbarkeit von beliebigen SystemC-Modulen (2.1.5) macht, diskutiert. Auch wird das Fehlen eines Schalterverhaltens in Portal-Komponenten und dessen Folgen thematisiert (2.1.6). Weiterhin werden einige Beschränkungen erörtert bezüglich der Verwendung von bestimmten Channeltypen mit ReChannel (2.1.7). Ferner wird die interne Verwaltung der ReChannel-Klassen hinsichtlich möglicher Optimierungen untersucht (2.1.8).

2.1.1 Synchronisation des Rekonfigurationsvorgangs

Bei der in ReChannel verwendeten Simulationssemantik (siehe Abschnitt 1.5.1) handelt es sich im Grunde um einen sehr hardwarenahen Ansatz. Dieser setzt voraus, dass die Kommunikation zwischen den Modulen mittels Multiplexern beliebig kontrolliert werden kann. Was bei RTL-Beschreibungen bzw. der Verwendung von einfachen Hardwaresignalen grundsätzlich ohne weiteres möglich ist, führt bei Übertragung auf abstraktere Beschreibungen zu einer Reihe von Problemen. Diese werden im Folgenden beschrieben

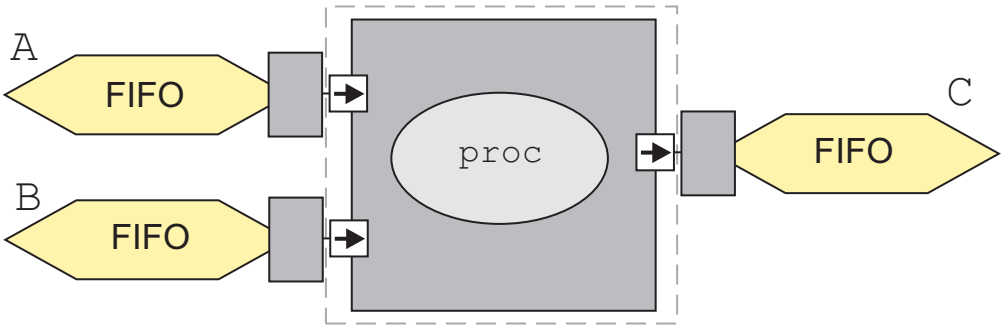

Abbildung 2.1: TF-Beispiel: Ein rekonfigurierbares Modul, das über Portals mit drei FIFO-Channeln auf der statischen Seite verbunden ist. Ein Thread-Prozess namens `proc` liest in einer Endlosschleife jeweils ein Datenpaket von A und B und gibt einen berechneten Wert an C aus.

und analysiert. Im Anschluss daran wird der Kontrollmechanismus von ReChannel auf die Existenz von implementationsbedingten Synchronisationsproblemen überprüft.

Rekonfiguration abstrakter SystemC-Beschreibungen

Abstrakte SystemC-Beschreibungen sind dadurch charakterisiert, dass sie von der durch sie modellierten Hardware abstrahieren. Meistens hat dies zur Folge, dass ein abstraktes SystemC-Modul weder *cycle-accurate* noch *pin-accurate* ist. Zur Kommunikation werden anstelle von Signalen abstrakte Channels mit beliebig komplexen Interfaces verwendet. Häufig werden Thread-Prozesse eingesetzt, die das Verhalten der Module auf eine funktionale Art beschreiben, die mehr Gemeinsamkeiten mit einem Softwareprogramm als mit einer Hardwarebeschreibung hat. Die Verwendung von abstrakten Channeln und Thread-Prozessen stellt daher eine besondere Herausforderung bei der Simulation von DR dar.

Zu welchen Synchronisationsproblemen es bei der Rekonfiguration von abstrakten Beschreibungen kommen kann, soll anhand eines einfachen Datenfluss-Beispiels veranschaulicht werden. Die dort bei der Verwendung von SystemC-FIFO-Channeln festgestellten Dateninkonsistenz-, Timing- und Deadlock-Problematiken können auch auf komplexere Channelinterfaces verallgemeinert werden. Abbildung 2.1 zeigt ein rekonfigurierbares Modul, das über Portals mit drei FIFO-Channeln auf der statischen Seite verbunden ist. In diesem Modul befindet sich ein Thread-Prozess namens `proc` mit folgendem Verhalten:

```
1 while(true) {
2     a = portA.read();
3     b = portB.read();
4     c = berechnen(a, b);
5     portC.write(c);
6 }
```

Da `proc` sich in einer Endlosschleife befindet, wird dessen Ausführung lediglich durch blockierende Lese- und Schreibzugriffe beschränkt. Hierbei ergeben sich folgende Probleme:

1. **Dateninkonsistenzen.** Wird das rekonfigurierbare Modul während eines beliebigen Zeitpunktes deaktiviert, so befindet sich `proc` dementsprechend auch an einer beliebigen Stelle (d.h. in einer beliebigen `wait()`-Anweisung) auf dessen Ausführungspfad.

   Wenn `proc` beim Lesen in Codezeile 3 aufgehalten wird, dann wurde `A` bereits gelesen, aber das dazugehörige Datenpaket in `B` noch nicht. In Codezeile 4 ist der Prozess noch mit der Berechnung der eingelesenen Datenpakete beschäftigt. Und in Zeile 5 ist lediglich die Ausgabe des zuletzt berechneten Ergebnisses noch nicht erfolgt.

   Da es für die Korrektheit von Datenflussbeschreibungen vor allem auf die Reihenfolge und Zuordenbarkeit der Daten-Pakete ankommt, führen alle diese Fälle zu Dateninkonsistenzen und somit potentiell zu undefiniertem Verhalten.

2. **Deadlocks.** Sind alle Eingangs-FIFOs leer, blockiert `proc` innerhalb eines dieser FIFO-Channels solange, bis wieder Daten zur Verfügung stehen. Aufgrund dieses externen Zugriffs ist der Zugriffszähler `pending_accesses` größer Null und daher eine Rekonfiguration unmöglich (vgl. Abschnitt 1.5.4). Soll das Modul nun – da es alle seine Daten berechnet hat – durch ein anderes ausgetauscht werden, resultiert dies in einem Deadlock in der Rekonfigurationskontrolle.

3. **Timing.** Um einen Deadlock sowie Dateninkonsistenzen zu verhindern, muss das rekonfigurierbare Modul exakt zu einem bestimmten Zeitpunkt (bzw. Delta-Zyklus) deaktiviert werden können.

   Dieses Vorhaben scheitert daran, dass von außen nicht feststellbar ist, wann das Modul gerade seine letzte Ausgabe geschrieben hat. Wenn die Ausgabe in `C` gelesen werden kann, dann ist es bereits zu spät: Da sich `proc` in einer Endlosschleife befindet, beginnt es im Anschluss an den Schreibvorgang direkt wieder mit dem Lesen von neuen Daten.

Weitere Probleme ergeben sich in Fällen, in denen ein Method-Prozess oder mehrere Prozesse gleichzeitig in einem rekonfigurierbaren Modul vorhanden sind:

4. *Method-Prozesse.* Die Verwendung von abstrakten Channeln mit Method-Prozessen ist grundsätzlich problematisch, da Accessoren nichtblockierende Zugriffe nicht aufhalten können (vgl. Abschnitt 1.5.3).

   Falls es sich im gegebenen Beispiel bei `proc` nicht um einen Thread-Prozess, sondern um einen Method-Prozess handelt, kann dieser nicht daran gehindert werden, in einer Schleife alle Eingangs-FIFOs komplett auszulesen. Eine Deaktivierung des Moduls zwischen den einzelnen Zugriffen ist in einem solchen Fall nicht möglich.

5. *Nebenläufige Prozesse.* Im Falle von nebenläufigen Prozessen besteht die Möglichkeit, dass zu jedem Zeitpunkt immer mindestens einer dieser Prozesse einen externen Zugriff durchführt. Ein solches Modul kann niemals deaktiviert werden, da der Zugriffszähler `pending_accesses` immer größer Null ist.

6. *Verschiedene Prozesstypen.* Da Accessoren nichtblockierende Prozesse nicht vom Zugriff auf den statischen Channel abhalten können, gibt es Probleme, wenn sowohl Thread-Prozesse als auch Method-Prozesse in einem Modul vorhanden sind.

   Auch wenn bereits alle Thread-Prozesse blockiert sind, bedeutet dies nicht, dass die Method-Prozesse ihre Arbeit daraufhin einstellen. Hier besteht eine mögliche Gefahr für Dateninkonsistenzen und Deadlocks im Inneren eines Moduls, da nur eine Teilabschaltung des Moduls erreicht werden kann.

7. *Interne Prozesse.* Völlig unbeeinflusst vom Zustand der Accessoren und des rekonfigurierbaren Moduls zeigen sich alle internen Prozesse, die keine Ein- oder Ausgaben an externen Ports vornehmen. Aufgrund dessen besteht hier ebenfalls die Möglichkeit von Dateninkonsistenzen und Deadlocks im Inneren eines Moduls.

Die primäre Zielsetzung von ReChannel ist, dass jedes bestehende Modul ohne Änderungen in ein rekonfigurierbares Modul verwandelt werden kann [19]. Aus diesem Grund ist es erforderlich, dass die in ReChannel bestehenden Synchronisationsprobleme durch äußere Mechanismen (d.h. ohne Modifikation der inneren Struktur der jeweiligen Module) gelöst werden.

Um die Rekonfiguration von abstrakten Modulen von außen synchronisieren zu können, muss es ermöglicht werden,

a) **den inneren Zustand eines Moduls anhand externer Zugriffe zu bestimmen.** Hierfür müssen nicht nur die Anzahl der Zugriffe, sondern auch die versendeten Daten selbst, unmittelbar überwacht werden können.

b) **nichtblockierende Zugriffe auf einen Channel abzublocken[1].** Hierfür ist es notwendig, dass die von einem Zugriff zurückgelieferten Daten beliebig manipuliert werden können.

c) **Transaktionen für Eingabe- und Ausgabezugriffe zu definieren**, die eine Deaktivierung des Moduls währenddessen verhindern.

Wie diese drei Anforderungen erfüllt werden können, wird in Kapitel 4.13 beschrieben. Problemstellung 7 ist durch die Anwendung von äußeren Synchronisationsmechanismen prinzipiell nicht lösbar. Diesbezüglich bleibt nur die Möglichkeit, dass ein Modul explizit durch rekonfigurierbares Verhalten beschrieben wird (siehe Kapitel 4.12).

Kontrollmechanismus

Der Kontrollmechanismus für die Durchführung der Rekonfigurationsvorgänge ist der zentrale Algorithmus der ReChannel-Bibliothek. Dieser definiert, wie die einzelnen Simulations-Komponenten miteinander interagieren, und repräsentiert somit die Umsetzung der Simulationssemantik (siehe Abschnitt 1.5.1).

Die bei der Durchführung von Rekonfigurationsvorgängen verwendete Kontrollhierarchie korrespondiert mit der Verbindungsstruktur der Module, Accessoren und Portals. Sie

[1] Bei FIFOs würde dies z. B. bedeuten, dass man dem Modul vortäuscht, dass die FIFO leer ist.

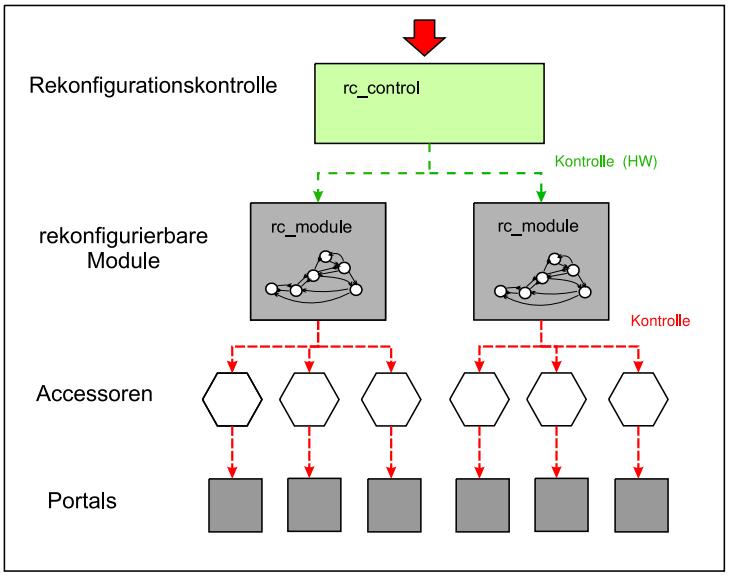

Abbildung 2.2: Die Hierarchie des Kontrollmechanismus von ReChannel zur Durchführung der Rekonfigurationsvorgänge

hat daher die Topologie eines Baumes (siehe Abbildung 2.2). Ganz oben in der Hierarchie (d.h. an der Wurzel des Baumes) befindet sich die Rekonfigurationskontrolle. Da diese die Schnittstelle zum Design repräsentiert, nimmt sie die Befehle der Kontrollprozesse entgegen und veranlasst die entsprechenden Rekonfigurationsvorgänge in den betreffenden Modulen. Die Module kontrollieren die Zustände der mit ihnen verbundenen Accessoren. Die Accessoren kontrollieren ihrerseits das ihnen zugeordnete Portal[2].

Die Kommunikation zwischen den einzelnen Komponenten sowie deren Zustandsübergänge erfolgen dabei auf sehr unterschiedliche Weise:

Abstrakter Channel. Ein Kontrollprozess greift auf die Rekonfigurationskontrolle wie auf einen abstrakten Channel zu. Der Zugriffstyp kann blockierend oder nichtblockierend sein.

Hardwarekomponente / endlicher Automat. Die Rekonfigurationskontrolle kommuniziert mit einem rekonfigurierbaren Modul auf dieselbe Weise wie mit einer nebenläufigen Hardwarekomponente. Es wird ein Rekonfigurationsvorgang angestoßen und danach auf eine Rückmeldung in Form eines Erfolgsereignisses gewartet. Die Kontrolle hat ansonsten keine weiteren Einflussmöglichkeiten auf die Ausführung des Rekonfigurationsvorganges.

Die Zustandsübergänge innerhalb des rekonfigurierbaren Moduls werden mittels eines endlichen Automaten modelliert, der durch einen SystemC-Prozess repräsentiert wird. Alle Zustandsübergänge werden vollständig parallel zum restlichen Design

[2] Accessoren unterschiedlicher Module dürfen sich hierbei ein Portal teilen. Dies ist aus Gründen der Übersichtlichkeit in Abbildung 2.2 nicht explizit dargestellt.

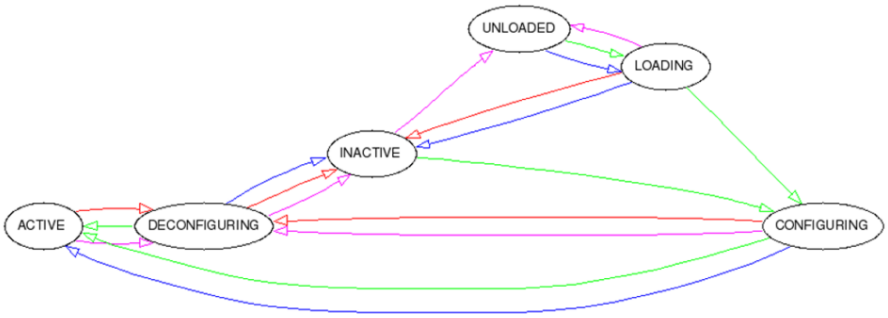

Abbildung 2.3: Der endliche Automat, der in `rc_module` die Rekonfigurationszustandsübergänge modelliert (aus [17]) [Anm.: Der undefinierte Zustand *UNDEF* ist in dieser Darstellung nicht enthalten.]

durchgeführt und benötigen jeweils mindestens einen Delta-Zyklus. Das Erreichen eines neuen Zustandes wird mittels Events nach außen gemeldet.

Obwohl in ReChannel im Prinzip lediglich drei Rekonfigurationszustände modelliert werden, gibt es sieben verschiedene Zustände, in denen sich der Automat befinden kann (siehe Abbildung 2.3). Der Grund hierfür ist, dass nicht nur die stationären Zustände eines Moduls, sondern auch temporäre Übergangszustände explizit dargestellt werden.

Signal. Wenn ein Modul seinen Rekonfigurationszustand wechselt, dann sorgt es dafür, dass die mit seinen Ports verbundenen Accessoren ebenfalls ihren Zustand entsprechend ändern. Der Accessor verhält sich hierbei analog zu einem Signal, d.h. sein neuer Zustand wird erst mit dem nächsten Delta-Zyklus übernommen.

Abbildung 2.4 zeigt den Ausschnitt einer Simulation zum Zeitpunkt t, in dem die Deaktivierung eines rekonfigurierbaren Moduls durchgeführt wird. Innerhalb eines Delta-Zyklus ist die tatsächliche Ausführungsreihenfolge der Prozesse unbestimmt, da in einem Delta-Zyklus die gleichzeitige Ausführung von Prozessen simuliert wird (siehe [7]). Somit wechselt der Rekonfigurationszustand des Moduls an beliebiger Stelle während des ersten Delta-Zyklus Δ_0 des Zeitpunktes t. Vor der Deaktivierung wurden bereits Prozesse ausgeführt und nach der Deaktivierung müssen noch weitere Prozesse ausgeführt werden. Da alle im Delta-Zyklus Δ_0 auszuführenden Prozesse als gleichzeitige Vorgänge angesehen werden, muss auch eine Ungleichbehandlung der Prozesse des rekonfigurierbaren Moduls vermieden werden. Es muss folglich erst abgewartet werden, bis alle gleichzeitig auszuführenden, externen Zugriffe der Prozesse abgearbeitet sind, bevor der Accessor beginnt, Zugriffe zu blockieren.

Der Zustand des Accessors ist daher als Signal modelliert, so dass dieser sich erst mit dem Übergang zum darauf folgenden Delta-Zyklus ändert. Dies bewirkt, dass der Accessor für den Rest des Delta-Zyklus noch alle externen Zugriffe durchlässt und das blockierende Verhalten erst mit dem Übergang zum nächsten Delta-Zyklus beginnt, da das Signal dann seinen neuen Wert (Zustand) übernommen hat.

Ein Accessor kann sich in sechs verschiedenen Zuständen befinden, die sein Verhalten bestimmen (siehe dazu auch Abschnitt 2.1.8).

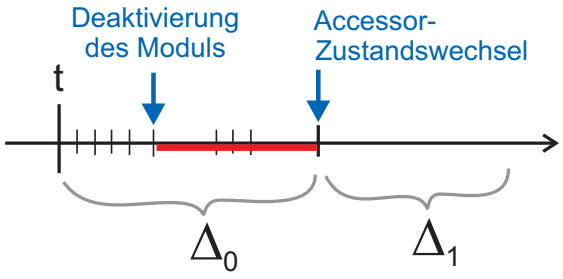

Abbildung 2.4: Deaktivierung eines rekonfigurierbaren Moduls zum Zeitpunkt t. Der Zustand der Accessoren wechselt erst mit dem Übergang zum nächsten Delta-Zyklus. Zwischen der Deaktivierung des Moduls und dem Zustandswechsel des Accessors sind sowohl nichtblockierende als auch blockierende Zugriffe des Moduls auf die Channels des statischen Design-Teils möglich.

Methodenaufruf. Wenn ein Accessor durch das Modul einen neuen Zustand zugewiesen bekommt, dann teilt es diese Änderung seinem Portal mit. Dies geschieht ohne Verzögerung mittels eines Methodenaufrufs, so dass es sich hierbei um einen rein funktionalen Zugriff handelt.

Probleme mit dem Kontrollmechanismus

Der Kontrollmechanismus von ReChannel hat die Eigenschaft, dass er instabil werden kann, sobald mehrere Kontrollprozesse konkurrierend auf eine Rekonfigurationskontrolle zugreifen. An den meisten Stellen innerhalb der oben beschriebenen Kontrollhierarchie ist es erlaubt, dass ein Rekonfigurationsvorgang potentiell scheitern oder vorzeitig abgebrochen werden kann. Dies ist insofern problematisch, da der Mechanismus teilweise parallel arbeitet und dabei unterschiedlichste Zeitverzögerungen und Zugriffsarten verwendet werden.

Es gibt z. B. bestimmte Zugriffsfolgen, die dazu führen, dass der Accessor-Zustand mit dem Modul-Zustand aus dem Takt gerät.

In einem anderen Szenario können widersprüchliche Rekonfigurationsbefehle dazu führen, dass ein Kontrollprozess auf unbestimmte Zeit auf die Benachrichtigung eines Events wartet. Dies folgt daraus, dass innerhalb des endlichen Automaten jeder Rekonfigurationsvorgang aus zwei bis vier Zustandsübergängen modelliert wird (siehe Abbildung 2.3) und zu jedem Zeitpunkt der Zielzustand des Automaten geändert werden darf. So ist es möglich, dass ein veranlasster Zustandsübergang nicht durchgeführt wird, wenn währenddessen ein anderer Vorgang gestartet wird. Im schlimmsten Fall führt dies zu Deadlocks im gesamten Design.

Ein weiteres unerwünschtes Verhalten resultiert daraus, dass im Accessor intern ein Signal zur Repräsentation des Zustandswertes verwendet wird. Wenn ein Accessor auf einen neuen Zustand gesetzt wird, behält das Signal seinen alten Zustandswert noch für den Rest des aktuellen Delta-Zyklus (siehe Abbildung 2.4). Im Falle einer Deaktivierung beginnt daher das blockierende Verhalten des Accessors erst mit Erreichen des nächsten

Delta-Zyklus. Aus diesem Grund ist es möglich, dass im selben Delta-Zyklus, in dem ein Modul gerade erfolgreich deaktiviert wurde, noch blockierende Zugriffe an die statischen Channels weitergeleitet werden. Hierbei handelt es sich um ein systematisches Problem, das für alle potentiell blockierenden Channelinterfaces besteht.

Die Rekonfigurationskontrolle `rc_control` besitzt neben den Methoden `deactivate()` und `nb_deactivate()` auch noch eine weitere Kontrollmethode: `force_deactivate()`. Diese kann benutzt werden, um ein rekonfigurierbares Modul sofort zu deaktivieren, d.h. unter Nichtbeachtung des momentanen Zählerwertes für die noch laufenden externen Zugriffe (vgl. Abschnitt 1.5.4, `pending_accesses`). Der Zweck dieser Methode ist zweifelhaft, da jener offensichtlich mit den Hauptprinzipien der Simulationssemantik von ReChannel bricht (vgl. Abschnitt 1.5.1). Da die Verwendung von `force_deactivate()` im Zusammenhang mit abstrakten Channeln (wie z. B. FIFO, Bus, etc.) mit hoher Wahrscheinlichkeit zu undefiniertem Verhalten führt, hilft diese Methode bei der Lösung der Synchronisationsproblematik nicht weiter.

Das Fazit der Analyse des Kontrollmechanismus ist, dass dieser einer tief gehenden Überarbeitung bedarf. Die Funktionsweise des Accessors und des endlichen Automaten des rekonfigurierbaren Moduls ist für einige Synchronisationsprobleme direkt verantwortlich. Des Weiteren ist der Mechanismus momentan noch nicht robust genug, um von mehreren Kontrollprozessen gleichzeitig verwendet zu werden. Die Verfügbarkeit von nichtblockierenden Kontrollmethoden sowie `force_deactivate()` birgt auch bereits bei der Verwendung von nur einem Kontrollprozess gewisse Risiken in sich.

2.1.2 Treiberkonflikte

Seit ihrer ersten Version (1.0) fordert SystemC, dass es als Fehler zu betrachten ist, wenn zwei Prozesse sich gemeinsam einen schreibenden Zugriff auf ein Signal teilen, da aus Hardwaresicht mit einem solchen Vorgang ein Treiberkonflikt bzw. ein Kurzschluss beschrieben wird. Jedoch wird ein Treiberkonflikt in SystemC bis zur Version 2.1 in der Standardeinstellung nicht als Laufzeitfehler gemeldet, um die hierfür benötigte Prozessüberprüfung aus Effizienzgründen in der normalen Simulation auslassen zu können. Um eine Fehlermeldung zu erhalten, muss bei der Kompilation des Designs manuell das Präprozessor-Makro `DEBUG_SYSTEMC` definiert werden. Die Definition von `DEBUG_SYSTEMC` ist immer dann vonnöten, wenn ein Designfehler im System aufgespürt werden soll, noch bevor die entsprechende Hardware synthetisiert wird.

In SystemC besitzen Prozesse eine Repräsentation als normales Simulationsobjekt (`sc_object`). Mittels einer Methode `sc_get_curr_process()` kann der aktuell ausgeführte Prozess seine Objektrepräsentation erfragen. Anhand eines Pointers auf ein solches Objekt können Prozesse identifiziert und voneinander unterschieden werden.

Wenn `DEBUG_SYSTEMC` für ein Design definiert worden ist und auf ein Signal (z. B. `sc_signal`) das erste Mal schreibend zugegriffen wird, dann speichert dieses daraufhin den Objektpointer des zugreifenden Prozesses. Bei jedem weiteren Schreibzugriff auf dieses Signal wird das abgespeicherte Objekt mit dem des aktuell zugreifenden Prozesses verglichen. Entsprechen sich diese nicht, so wird ein Laufzeitfehler gemeldet, der auf

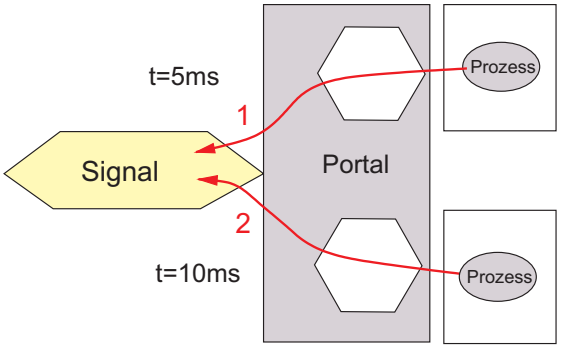

Abbildung 2.5: Die Rekonfiguration von Modulen, die mit Signalen verbunden sind, führt bei standardkonformen SystemC-Implementationen zu Treiberkonflikten. Die Accessoren (weißes Sechseck) leiten Zugriffe direkt an den statischen Channel (Signal) durch, so dass sich die Identität der schreibenden Prozesse bei jeder Rekonfiguration der angeschlossenen Module ändert.

diesen Treiberkonflikt hinweist und zum Abbruch der Simulation führt.

Aufgrund der Wichtigkeit einer Treiberüberprüfung für die korrekte Funktionsweise der synthetisierten Hardware, ist mit dem Erscheinen des SystemC-Sprachstandards spezifiziert worden, dass eine SystemC-Implementation Treiberkonflikte grundsätzlich als Laufzeitfehler melden muss. Ein Makro `DEBUG_SYSTEMC` oder eine Methode, die Treiberüberprüfung auszuschalten, ist in diesem Standard nicht enthalten. Somit ist sichergestellt, dass Kurzschlüsse in einem Design nicht mehr unentdeckt bleiben können.

ReChannel ist von dieser Änderung betroffen, da es – im Gegensatz zur Simulation von DR mit Hardware-Multiplexern – die vorteilhafte Eigenschaft hat, dass es die Zugriffe der Prozesse auf der rekonfigurierbaren Seite direkt und ohne Delta-Verzögerung auf den statischen Channel durchleitet [19]. Wenn es sich beim statischen Channel um ein Signal handelt, stellt dies ein prinzipielles Problem dar: Da die rekonfigurierbaren Module während der Simulation ausgetauscht werden, ändert sich auch jedes Mal die Identität der zugreifenden Treiberprozesse (siehe Abbildung 2.5). Dies führt dazu, dass die Verwendung von ReChannel zusammen mit SystemC-Signalen nicht möglich ist, sobald eine standardkonforme SystemC-Implementation verwendet wird (z. B. OSCI-SystemC seit der Version 2.2b).

2.1.3 Abdeckung des SystemC-Sprachumfangs

Dieser Abschnitt befasst sich mit SystemC-Funktionalität, die derzeit noch nicht zusammen mit ReChannel bei der Simulation von DR verwendet werden kann.

Exports

Bei hardwarenahen Beschreibungen, wie z. B. der RT-Ebene, kann ein Modul ausschließlich Ports besitzen, da dies die traditionelle Verbindungsart von Modulen ist. Der

SystemC-Sprachstandard schreibt genau vor, welche Komponenten wie zu verbinden sind. Zum Beispiel müssen Ports mit Signalen verbunden werden, die sich außerhalb des jeweiligen Moduls befinden. Und da immer ein Channel als Kommunikationsmedium vorhanden sein muss, ist es z. B. verboten zwei Module direkt über ihre Ports miteinander zu verbinden.

Während auf hardwarenahen Abstraktionsebenen hauptsächlich primitive Channels mit write- und read-Zugriffen verwendet werden, sind auf höheren Abstraktionsebenen die weiterführenden Eigenschaften des Interface-Konzeptes von Bedeutung. Hier ergibt sich häufig die Problemstellung, dass sich ein Channel oder Interface innerhalb eines Moduls befindet, auf den von außerhalb des Moduls zugegriffen werden soll.

Mit SystemC in der Version 2.1 sind zu diesem Zweck die sogenannten *Exports* zur Sprache hinzugefügt worden. Diese Konstrukte ermöglichen die Bindung von Channeln in der umgekehrten Richtung wie bei einem Port. Der Channel kann sich hierbei an beliebiger Stelle innerhalb der Struktur eines Moduls befinden. Durch die Bindung mit einem Export steht das Interface des Channels dann auch außerhalb des Moduls zur Verfügung. Mit Exports kann außerdem beschrieben werden, dass ein Modul bzw. hierarchischer Channel mehrere Interfaces gleichen Typs besitzt. Dies war bei der herkömmlichen Vorgehensweise mittels Ableitung des Interfaces bisher nicht zu beschreiben.

Obwohl SystemC, Version 2.1, während der Entwicklung und Implementierung der ReChannel-Bibliothek bereits verfügbar war, ist die Rekonfiguration von Modulen mit Exports hierin nicht vorgesehen worden.

Da mittels Interfaces die Schnittstellen eines Bus-Systems beschrieben werden, spielen diese besonders bei TLM-Beschreibungen eine wichtige Rolle. Die in [14] beschriebene Methodologie für die Beschreibung von transaktionalen Modellen setzt gleichermaßen auf Ports wie auf Exports. Für die Abdeckung des SystemC-Sprachumfangs und für die Simulation von DR auf transaktionaler und funktionaler Ebene in ReChannel wird eine Fähigkeit zur Rekonfiguration von Exports daher dringend benötigt.

Analog zu Portals, die für Ports zuständig sind, sollte es daher ein Konstrukt namens **Exportal** geben, das die gleiche Funktionalität auch für Exports bietet. Mit der Existenz von Exportals wäre es darüber hinaus auch möglich, einen Channel mit ReChannel zu rekonfigurieren. Dies ist bisher nicht umsetzbar, da Portals nur mit Channeln auf der statischen Seite verbunden werden können und die Kommunikation somit in der entgegengesetzten Richtung verläuft.

Ein Portal kann nicht für die Aufgaben eines Exportals verwendet werden, da die Anforderung sehr verschieden sind. Auf den ersten Blick sieht es so aus, als könnte einfach der Code der Portal-Klasse wiederverwendet werden. Dies ist allerdings nicht möglich, da u. a. die Weiterleitung von Events in der umgekehrten Richtung technisch nicht auf dieselbe Art und Weise implementiert werden kann. Im Gegensatz zu Portals ist es bei Exportals nicht das Ziel der Event-Weiterleitung, das sich bei einer Rekonfiguration ändert, sondern die Quelle für Events. Bei Exportals müsste der Prozess, der für die Weiterleitung eines Events zuständig ist, im Falle einer Rekonfiguration instantan seine dynamische Sensitivity-List ändern. Dies kann aber ausschließlich durch den Prozess selbst erfolgen. Dies bedeutet, dass der Prozess zuerst ausgeführt werden muss. Dies wiederum ist nur möglich, wenn ein Event seiner alten Sensitivity-List benachrichtigt wird. Das Eintre-

ten eines solchen Events kann nicht garantiert werden und sollte abgesehen davon auch nicht mehr beachtet werden, da es zu diesem Zeitpunkt bereits zu einem deaktivierten Modul gehört. Hier müssen somit zusätzliche Problematiken beachtet werden. Die Event-Weiterleitung eines Portals ist demgegenüber deutlich einfacher zu implementieren, da sich hierfür die Sensitivity-List des Prozesses nicht dynamisch ändern muss.

Eine weitere Schwierigkeit besteht in der aktuellen Implementation des Kontrollmechanismus von ReChannel (siehe Abschnitt 2.1.1), da die Kontrollhierarchie des Mechanismus auf einer direkten Verbindung der Module mit den Accessoren basiert. Wie in Abbildung 2.6 illustriert, verfügt ein Modul mit einem exportierten Channel über keinen Accessor. Ein Export kann mit einem Accessor prinzipiell keine Bindung eingehen, da jener nur mit seinem exportierten Channel, aber nicht mit einem weiteren Interface (Accessor) gebunden werden kann. Abgesehen davon ist es aktuell nicht möglich, einen Accessor wiederzuverwenden oder diesen vor einen exportierten Channel zu schalten, da dieser in umgekehrter Richtung betrieben werden müsste. Ein Accessor kann auch nicht auf der statischen Seite wiederverwendet werden, da er auf der einen Seite mit einem rekonfigurierbaren Modul und auf der anderen Seite mit einem Portal verbunden werden muss. Die Beziehung zwischen diesen Komponenten ist fest durch deren Implementation vorgegeben. Da der Accessor ein fester Bestandteil des Kontrollmechanismus von ReChannel ist, ist dessen reine Verwendung als Kommunikations-Weiterleitungskomponente nicht möglich.

Multiports

In SystemC können Ports die spezielle Eigenschaft besitzen, sich mit mehreren Interfaces zu verbinden. Dies kann auf abstrakteren Ebenen sinnvoll sein, um die Bindung zu vereinfachen und des Weiteren eine variable Portanzahl modellieren zu können.

Portals können nicht für die Rekonfiguration von Multiports verwendet werden, da pro Port nur ein Accessor für ein Interface angelegt wird. Es ist fraglich, ob es Sinn macht, einen rekonfigurierbaren Bereich mit einer variablen Anzahl von Ports zu versehen. Sicher ist allerdings, dass die Anpassung von Portals und Accessoren an Multiports einen erheblichen Aufwand bedeutet. Die Einschränkung durch das Fehlen von Multiports in ReChannel wird als gering eingeschätzt.

Variable Anzahl von Events

In SystemC kann beschrieben werden, dass ein Interface über eine variable Anzahl von Events verfügt. Dies wird z. B. dadurch erreicht, dass eine Eventmethode einen Index als Parameter erwartet und dem Index entsprechend ein anderes Event-Exemplar zurückliefert. Damit auch über einen Port auf diese Events zugegriffen werden kann, muss dieser Port über Eventfinder (siehe [7]) verfügen, die ebenfalls die entsprechenden Indizes erwarten.

In ReChannel können mittels `rc_port_traits` lediglich eine konstante Anzahl an Events für ein Interface deklariert werden (siehe Abschnitt 1.5.5). Die hierbei bestehende Einschränkung könnte möglicherweise dadurch behoben werden, dass die Registrierung

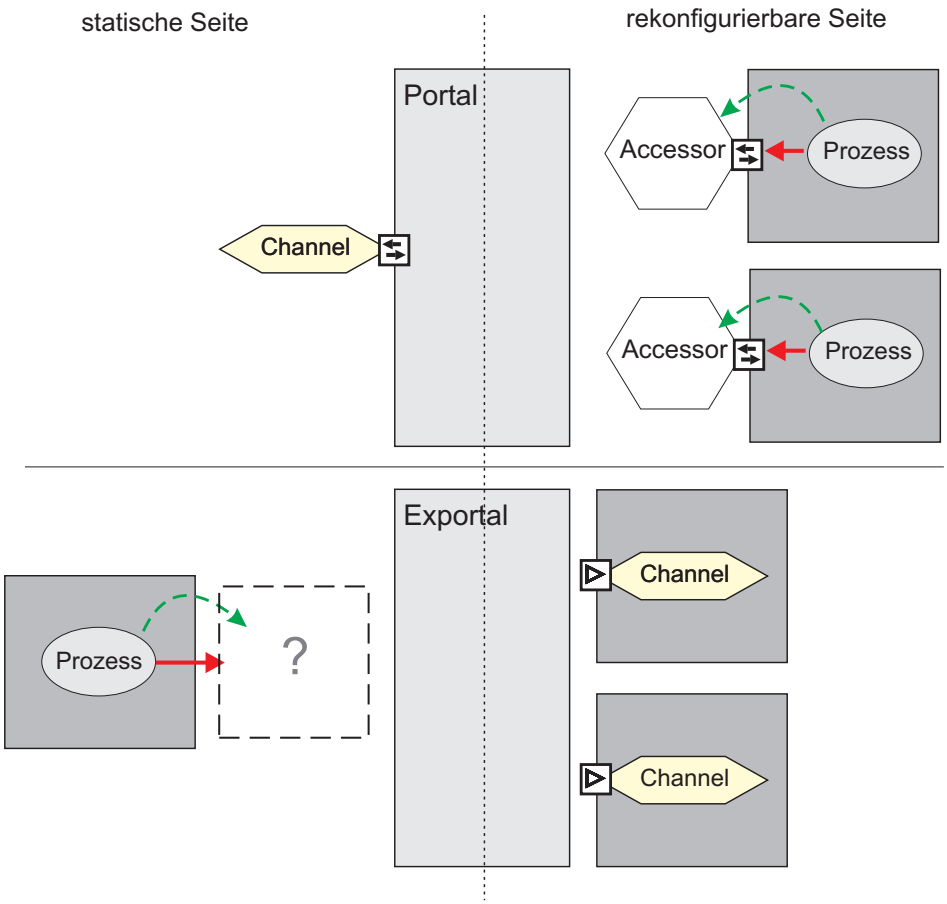

Abbildung 2.6: Die Nebeneinanderstellung von Portal und Exportal zeigt, dass der bestehende Kontrollmechanismus von ReChannel (vgl. Abschnitt 2.1.1) nicht mehr mit der Einführung eines Exportals verwendet werden kann. Des Weiteren sieht man, dass der Prozess auf der statischen Seite des Exportals kein eindeutiges Interface besitzt, mit dem er kommunizieren kann (roter Pfeil) bzw. von dem er die Events für seine Sensitivity-List erhält (grüner Pfeil). Wünschenswert wäre hier die Wiederverwendung des Accessors auf der statischen Seite. Dies ist allerdings nicht möglich, da der Accessor nicht in umgekehrter Richtung betrieben werden kann und durch seine Implementation fest auf die Bindung mit einem Portal und einem rekonfigurierbaren Modul ausgerichtet ist. Des Weiteren ist der Accessor fester Bestandteil der Kontrollhierarchie von ReChannel, was dessen reine Verwendung als Kommunikations-Weiterleitungskomponente nicht zulässt.

von Events nicht statisch durch eine Traits-Klasse vorgegeben ist, sondern variabel zur Laufzeit vorgenommen wird.

2.1.4 Verhalten deaktivierter Module

Bei der Simulation von DR mit ReChannel wird davon ausgegangen, dass rekonfigurierbare Module nach ihrer Deaktivierung in einen künstlichen Schlafzustand verfallen, da sie keine Events mehr erhalten. Diese Isolation eines Moduls von seiner Umgebung wird dadurch erreicht, dass Portals die Events des statischen Channels ausschließlich an diejenigen Accessoren weiterleiten, deren Modul momentan einen aktiven Rekonfigurationszustand hat (vgl. Abschnitt 1.5.3).

Ein RTL-Modul, das von äußeren Events und somit auch von seiner Clock getrennt worden ist, wird mit Sicherheit früher oder später seine Arbeit einstellen, da jeglicher externe Stimulus ausbleibt. Nicht verhindert werden kann allerdings ein Nachhall aufgrund der inneren Struktur des Moduls. Da Module der RT-Ebene meist eine innere Struktur besitzen, die sich aus Signalen und weiteren Untermodulen zusammensetzt, kann dies dazu führen, dass diese Module noch etliche Delta-Zyklen lang ihre Ausgabe-Signale beschreiben.

Die Ursache hierfür sind die Delta-Zyklus-Verzögerungen der einzelnen Signale. Ein Signal übernimmt einen neu geschriebenen Wert erst mit dem nächsten Delta-Zyklus. Wenn sich der Wert ändert, werden die entsprechenden Events (z. B. `value_changed`) benachrichtigt, was daraufhin zur erneuten Aktivierung von Prozessen führen kann.

Es hängt vom jeweiligen Modul ab, wie lange es dauert, bis alle Signalwerte einen konstanten Wert haben. Hierbei handelt es sich um ein mögliches Problem, da Accessoren nichtblockierende Zugriffe nicht abblocken können. Die Accessoren können daher auch nach erfolgter Deaktivierung nicht verhindern, dass das Modul die Signale auf der statischen Seite beschreibt. Dies kann zu unerwarteten Effekten führen. Verstärkt wird dieser Effekt, wenn innerhalb des Moduls Bauteile oder Verhalten mit Zeitverzögerung verwendet werden. Hierbei kann es nach einiger Zeit plötzlich noch zu Ausgaben kommen, die im Rahmen der Simulationssemantik von ReChannel vollkommen unerwartet auftreten und potentiell zu undefiniertem Verhalten im Design führen können.

Auch wenn in hardwarenahen Beschreibungen in einigen Fällen kurzzeitig instabile Signalwerte in der Simulation toleriert werden können, so kann dies in gemischten Beschreibungen – wie sie während des Refinements vorkommen – ernsthafte Folgen haben, da die Auswirkung eines nichtblockierenden Zugriffs auf einen Channel höheren Abstraktionsgrades meist von größerem Ausmaß ist.

Ein weiterer Aspekt ist die Verträglichkeit der internen Verhaltensbeschreibung eines Moduls mit der Simulationssemantik von ReChannel. Eine Fülle von Beschreibungsarten und Konstrukten führt innerhalb von rekonfigurierbaren Modulen zu einem Verhalten, das sich mit der Deaktivierung durch eine äußere Kontrolle nicht in Einklang bringen lässt.

Zum Beispiel stellt das Benachrichtigen von Events ein Problem dar, da die Auslösung des Events zu einem beliebigen, späteren Zeitpunkt stattfinden kann. Die Prozesse eines bereits deaktivierten Moduls können somit zu einem unerwarteten Zeitpunkt wieder

angestoßen werden.

2.1.5 Reset des inneren Modulzustands

Wie in Abschnitt 2.1.4 bereits angedeutet wurde, kann das innere Verhalten oder die Struktur eines Moduls zu Problemen mit der Simulationssemantik von ReChannel führen. Für viele funktionale Modulbeschreibungen bedeutet dies, dass diese nicht als rekonfigurierbare Module zu verwenden sind. Um ein Modul als rekonfigurierbares Modul verwenden zu können, darf dieses lediglich eine Teilmenge der Beschreibungsvielfalt von SystemC verwenden.

Der Kern des Problems ist darin begründet, dass Module höherer Abstraktionsstufen sich häufig nicht entsprechend einem Bitstream verhalten, der immer wieder von neuem unverändert auf einen FPGA geschrieben werden kann. Die dynamische Rekonfiguration von FPGAs kann nicht realitätsgetreu nachgebildet werden, wenn sich der Bitstream nach jedem Neubeschreiben auf unvorhersehbare Weise ändert. Da ein solcher Konfigurationsstream für einen FPGA auch Initialwerte für innere Register und RAMs enthalten kann, muss in einer realitätsgetreuen Simulation der innere Zustand sofort nach dem Laden eines Moduls hergestellt sein. Wenn zur Initialisierung des Moduls zuerst einige Prozesse anlaufen müssen, um dies zu erreichen, dann wird bereits Verhalten modelliert. Und da die Reihenfolge von Prozessen innerhalb eines Delta-Zyklus unbestimmt ist, kann es vorkommen, dass andere Prozesse vor der Initialisierung ausgeführt werden und somit noch auf die alten Werte zugreifen.

In der Simulation kann versucht werden, ein Modul mittels eines Reset-Vorganges wieder auf seinen originalen Zustand zu setzen. Hierbei gibt es zwei Möglichkeiten, die zur Auswahl stehen:

1. Wenn das Modul über einen eigenen Reset-Mechanismus verfügt, kann hierüber der innere Zustand des Moduls zurückgesetzt werden. Dies entspricht der Eigenschaft von DRHW allerdings nur in Fällen, in denen ein Reset ohne Zeitverzögerung durchgeführt werden kann, da ansonsten wiederum kein Initialzustand, sondern Verhalten modelliert wird.

   Inwieweit die zum Rücksetzen eines Moduls benötigte Zeit ein Problem darstellt, hängt von den an die Simulation gestellten Anforderungen und von der Dauer der Rekonfigurationsvorgänge ab. Beanspruchen die Rekonfigurationsvorgänge in einem Design verhältnismäßig viel Simulationszeit, so kann diese Zeit für die Durchführung eines aufwändigeren Resetvorganges verwendet werden. Sollen Rekonfigurationsvorgänge andererseits instantan oder innerhalb eines Taktzyklus erfolgen, kann es zu Problemen mit der Simulation von Initialzuständen der Module kommen, wie z. B. aus den folgenden Gründen:

   - Module auf hardwarenahen Abstraktionsebenen verfügen i. d. R. über Reset-Mechanismen, die mit einer externen Clock synchronisiert werden. Das implizite Rücksetzen eines solchen Moduls wird dadurch erschwert, dass hierfür die entsprechende Taktflanke dieses Clock-Signals benötigt wird.

- Viele auf funktionaler und transaktionaler Ebene verwendeten, primitiven Channels besitzen keine Reset-Möglichkeit (z. B. `sc_fifo`).
- Außerdem gibt es ein Problem mit Channeln, die nach dem Request-Update-Schema arbeiten (z. B. Signale und FIFOs), da deren neue Werte erst im nächsten Delta-Zyklus zur Verfügung stehen (bzw. sichtbar sind).

Ein weiteres Problem ist, dass das Auslösen des Resetmechanismus eines Moduls meist zur Folge hat, dass sich die Werte an den Output-Ports ändern. Da die Accessoren nichtblockierende Schreibzugriffe nicht abhalten können (vgl. Abschnitt 1.5.3), ändern sich daraufhin auch die Werte der verbundenen Signale. Demnach ist davon auszugehen, dass zum Zeitpunkt des Resets die in Abschnitt 2.1.4 beschriebene Problematik auftritt.

2. Die zweite Möglichkeit besteht darin, den inneren Zustand des Moduls von außen zu setzen. Dies soll mit der Methode `rc_reset()` (siehe Abschnitt 1.5.5) erreicht werden. Mit `rc_reset()` werden Variablen und Signale registriert, die bei Deaktivierung des Moduls auf ihre Initialwerte zurückgesetzt werden sollen.

   Aber wenn versucht wird, ein Signal durch einen Prozess außerhalb des Moduls zurückzusetzen, dann tritt das in Abschnitt 2.1.2 beschriebene Treiberproblem auf, das zum Abbruch der Simulation führt. Ein weiteres Problem besteht darin, dass die Variablen eines Moduls möglicherweise als `private` deklariert sind, so dass C++ den Zugriff darauf untersagt. Es kann auch sein, dass die innere Struktur eines Moduls unbekannt oder von außen unereichbar ist.

In SystemC besteht die Möglichkeit, dynamische Prozesse während der Laufzeit zu erzeugen. Wenn innerhalb eines rekonfigurierbaren Moduls ein dynamischer Prozess erzeugt worden ist, dann wird dieser mit der Deaktivierung des Moduls nicht zwangsläufig wieder beendet. Bei der nächsten Aktivierung desselben Moduls könnte dieser Prozess immer noch existieren. Das Modul befände sich somit nicht im selben Zustand wie bei dessen erster Aktivierung. Eine Beendigung eines Prozesses von außen ist in SystemC aufgrund des Fehlens entsprechender Prozess-API-Funktionen nicht möglich.

2.1.6 Schalterverhalten der Portals

Bei der Analyse der Funktionalität von ReChannel ist aufgefallen, dass Signale, die mit Portals verbunden sind, in einer Mischform von Register und einfacher Datenleitung betrieben werden. Wenn ein rekonfigurierbares Modul aktiv ist, dann greift es über seine externen Ports und Accessoren direkt auf die Signale zu. Jeder geschriebene Wert wird im nächsten Delta-Zyklus übernommen, was dazu führt, dass das entsprechende Signal eine einfache Hardwareleitung modelliert. Im Falle der Deaktivierung eines rekonfigurierbaren Moduls verhält sich das Signal dann aber wie ein Register, da es seinen vorherigen Wert behält. Im Gegensatz dazu hätte das Fehlen der Treiberleitung für eine Hardwareleitung eine Wertänderung zur Folge. Auch bei der Aktivierung eines rekonfigurierbaren Moduls wird nicht sofort mit dem Öffnen des Portals ein neuer Wert geschrieben, wie es von einem Schalter erwartet wird.

Es fehlt dem Portal somit ein korrektes Schalterverhalten für einfache Signalleitungen. Eine Öffnung oder Schließung eines Schalters, wie es das Portal darstellt, hat immer auch Auswirkungen auf das angeschlossene Signal.

Die Modellierung eines Schalterverhaltens kann auch bei abstrakten Channeln von Vorteil sein. Für einige abstrakte Channels macht es Sinn, eine Art „Anmelde-Abmelde"-Funktionionalität für rekonfigurierbare Module zur Verfügung zu haben. Wenn ein Modul rekonfiguriert wird, dann kann sich dieses bei dessen Aktivierung beim Channel anmelden und bei Deaktivierung wieder abmelden. Ein Beispiel für eine mögliche Anwendung ist ein Mutex-Channel, bei dem ein rekonfigurierbares Modul automatisch seinen gehaltenen Lock zurückgeben kann, wenn es deaktiviert wird. Auch bei transaktionalen Bus-Systemen könnte es von Vorteil sein, wenn eine solche Funktionalität vorhanden ist.

Die Verfügbarkeit eines Schalterverhaltens von Portals ist besonders wichtig bei Verwendung der *Resolved Signals* von SystemC (`sc_signal_resolved` und `sc_signal_rv`, siehe [7]). Diese Signale verfügen über eine vierwertige Logik, die die Werte '0' (low), '1' (high), 'Z' (hochohmig) und 'X' (undefiniert) umfasst. Es ist erlaubt, dass mehrere Prozesse auf ein solches Signal zugreifen. Jeder dieser Prozesse wird dann als eigenständiger Treiber dieses Signals betrachtet. Der resultierende Wert, den das Signal im nächsten Delta-Zyklus übernimmt, wird mittels der folgenden Tabelle aufgelöst:

	'0'	**'1'**	**'Z'**	**'X'**
'0'	'0'	'X'	'0'	'X'
'1'	'X'	'1'	'1'	'X'
'Z'	'0'	'1'	'Z'	'X'
'X'	'X'	'X'	'X'	'X'

Schreiben zwei Prozesse einen widersprüchlichen Wert, z. B. '0' und '1', so erhält das Signal den undefinierten Wert 'X'. Ein als Treiber dieses Signals in Erscheinung getretener Prozess kann seinen Einfluss auf die Wertbildung dieses Signals zurücknehmen, indem dieser den Wert 'Z' schreibt. Der Wert 'Z' entspricht in Hardwareterminologie einem hochohmigen Wert, der technisch z. B. durch den Einsatz von Tri-State-Treibern umgesetzt werden kann.

Ein Portal müsste in der Lage sein, bei der Öffnung den Wert 'X' zu schreiben, um anzudeuten, dass der Signalwert noch undefiniert ist. Wenn ein Prozess des rekonfigurierbaren Moduls dann zum ersten Mal schreibend auf dieses Signal zugreift, dann müsste jener Wert durch den entsprechenden geschriebenen Wert ersetzt werden.

Wenn das Portal für eine Deaktivierung eines Moduls geschlossen wird, dann müsste der Einfluss der Treiber des rekonfigurierbaren Moduls durch Schreiben des Wertes 'Z' zurückgenommen werden können.

Einem Portal ein Schalterverhalten für Signale hinzuzufügen, scheitert momentan an einem prinzipiellen technischen Problem, das eng mit dem in Abschnitt 2.1.2 beschriebenen Treiberproblem verwandt ist: Es ist in SystemC nicht möglich, stellvertretend für andere Prozesse einen Wert in einem Resolved-Signal zu ändern, da der ausführende Prozess beim Schreiben auf dieses Signal als eigenständiger Treiber in Erscheinung tritt.

Abgesehen vom Treiberproblem stellt sich die Frage, in welcher Klasse und auf welche Weise ein mögliches Schalterverhalten implementiert werden könnte. Ein Designer muss die Möglichkeit haben, für einen benutzerdefinierten Channel ein individuelles Schalterverhalten beschreiben zu können. Eine notwendige Anforderung an die Funktionalität des Schalterverhaltens ist daher, dass diese je nach Channeltyp beliebig anpassbar sein muss. Die Klasse `rc_port_traits` (siehe Abschnitt 1.5.5) scheint ungeeignet für den Ort einer allgemeinen Implementation des Schalterverhaltens zu sein, da die hierfür erforderlichen Komponenten (Ports, Channels) aus dem Kontext dieser statischen Klasse nicht erreichbar sind. Demgegenüber hätte es einen höheren Aufwand für einen Designer zur Folge, wenn die Funktionalität in der Portal-Komponente implementiert wird, da für benutzerdefinierte Channels in diesem Fall drei unterschiedliche Klassen zu definieren sind: Accessor, Portal und eine Traits-Klasse.

Bei einer Implementierung muss auch beachtet werden, dass das Schalterverhalten auch für Exportals definierbar sein muss. Dies sollte konzeptionell in der gleichen Weise wie für Portals erfolgen. Des Weiteren ist noch nicht absehbar, ob für die Erzeugung eines Exportals eine Traits-Klasse (wie z. B. `rc_export_traits`) benötigt wird.

2.1.7 Inkompatible Channeltypen

Die Simulationssemantik von ReChannel basiert darauf, dass inaktive und ungeladene Module vom statischen Designbereich isoliert werden können (vgl. Abschnitt 1.5.1). Um dies zu erreichen, muss die gesamte externe Kommunikation eines rekonfigurierbaren Moduls kontrolliert werden können. ReChannel verwendet zu diesem Zweck zwei Hilfskomponenten, das Portal und den Accessor (siehe Abschnitt 1.5.2 und 1.5.3).

Das Portal hat die Aufgabe, die Events des statischen Channels an den jeweils aktiven Accessor weiterzuleiten. Der Accessor hat die Aufgabe, *Interface Method Calls* (IMCs) an den statischen Channel weiterzuleiten oder diese abzublocken. Zu diesem Zweck muss der Accessor das Interface des Channels implementieren. Um die Kommunikation zwischen dem statischen Designbereich und dem rekonfigurierbaren Modul kontrollieren zu können, muss der Accessor in der Lage sein, sämtliche blockierenden und nichtblockierenden IMCs auf den statischen Channel unterbinden zu können. Des Weiteren muss der Accessor mit dem Port des rekonfigurierbaren Moduls verbunden werden können.

Diese an den Accessor gestellten Anforderungen sowie die in den vorangegangenen Abschnitten diskutierten Probleme führen dazu, dass einige Channeltypen momentan nicht auf die gewünschte Weise bei der Simulation von DR in ReChannel verwendet werden können. Die hiervon betroffenen Channels lassen sich anhand der Ursache für deren Inkompatibilität mit ReChannel in folgende Klassen einteilen:

2.1.7.a Abstrakte Channels mit nichtblockierenden Schreibzugriffen

Wie in Abschnitt 2.1.1 und 2.1.4 bereits thematisiert wurde, besitzt der Accessor keine entsprechende Funktionalität, um nichtblockierende Zugriffe auf den statischen Channel abblocken zu können. Daher kann die Kommunikation, die über einen Channel mit nichtblockierenden Schreibzugriffen erfolgt, nicht in ausreichendem Maße kontrolliert werden.

Im Gegensatz zu einfachen Signalen besteht bei Channeln höherer Abstraktionsebenen die Gefahr, dass ein ungewollter Zugriff weitreichende Konsequenzen für die Korrektheit eines Systems hat.

2.1.7.b Channels mit blockierenden Zugriffen (ohne Statusabfragen und Events)

Greift ein Prozess eines rekonfigurierbaren Moduls mittels eines blockierenden IMCs auf einen statischen Channel zu, kann es zu der in Abschnitt 2.1.1 beschriebenen Deadlock-Situation kommen: Wenn der zugreifende Prozess durch eine `wait()`-Anweisung innerhalb des Channels auf unbestimmte Zeit blockiert wird, führt dies zu einem andauernden, externen Zugriff, der die Durchführung eines Rekonfigurationsvorganges verhindert (vgl. Abschnitt 1.5.4, `pending_accesses`).

Channels mit Interfaces, die ausschließlich blockierende Zugriffe besitzen und zudem über keine Events verfügen, sind Extrembeispiele solcher Channeltypen, wie es beim folgenden FIFO-Interface aus [14] der Fall ist:

```
template<typename T>
class tlm_blocking_get_if : public virtual sc_interface
{
public:
  virtual T get() = 0;
  virtual void get(T& t) = 0;
};
```

Das Interface `tlm_blocking_get_if` besitzt zwei blockierende `get`-Methoden, mit denen Datenpakete aus der FIFO ausgelesen werden können. Falls die FIFO leer ist, blockieren diese Methoden einen aufrufenden Prozess solange, bis neue Daten verfügbar sind.

Besitzt ein rekonfigurierbares Modul einen Port des Interfaces `tlm_blocking_get_if`, so muss auch der für die Kontrolle dieser Kommunikationsverbindung verwendete Accessor dieses Interface implementieren.

Bei der Verwendung des Interfaces `tlm_blocking_get_if` besteht ein prinzipielles Problem, das einer direkten Lösung der Deadlock-Problematik innerhalb des Accessors im Wege steht. Um Deadlocks zu vermeiden, die aufgrund des Blockierens innerhalb der FIFO entstehen, muss bereits vor dem Aufruf einer `get`-Methode absehbar sein, ob Daten zur Verfügung stehen oder nicht. Sollte die FIFO leer sein, so muss der Aufruf im Accessor solange aufgehalten werden können, bis wieder Daten verfügbar sind. Dies ist aber aufgrund des Fehlens von weiteren Interface-Methoden nicht möglich. Das Interface verfügt einerseits über keine Methode zur Statusabfrage, ob bzw. wie viele Datenpakete in der FIFO momentan vorhanden sind. Und es existiert andererseits auch keine Methode, die ein Event zurückliefert, das die Verfügbarkeit neuer Daten ankündigen könnte.

Channels mit Interfaces des oben beschriebenen Typs sind aufgrund der bestehenden Deadlock-Problematik derzeit bei der Simulation von DR mit ReChannel nicht verwendbar.

2.1.7.c Channels mit Prozessidentifikation

Falls eine standardkonforme SystemC-Implementation verwendet wird, gehören auch normale SystemC-Signale zu den inkompatiblen Channeltypen. Die Ursache hierfür ist das in Abschnitt 2.1.2 dargestellte Treiberproblem. Innerhalb eines SystemC-Signals wird die Identität jedes zugreifenden Prozesses überprüft, um Treiberkonflikte zu erkennen. Wenn während der Simulation von DR mindestens zwei Module nacheinander auf ein Signal zugreifen, führt dies unweigerlich zu einem Treiberkonflikt, der zum Abbruch der Simulation führt. Da Accessoren einen IMC direkt an den statischen Channel weiterleiten, tritt jeder zugreifende Prozess als eigenständiger Treiber auf.

Da durch einen Rekonfigurationsvorgang in ReChannel die Prozesse eines Moduls nicht tatsächlich ersetzt werden, sondern durch jedes weitere rekonfigurierbare Modul zusätzliche Prozesse in Erscheinung treten, ist eine für die Verwendung eines statischen Channels sinnvolle Prozessidentifikation nicht möglich.

Auch die Resolved-Signals von SystemC, `sc_signal_resolved` und `sc_signal_rv`, stellen Channels dar, deren Schreibmethoden eine Identifikation der zugreifenden Prozesse vornehmen. Der Wert eines solchen Signals wird aus den geschriebenen Werten aller bisher zugreifenden Prozesse gebildet (siehe [7]). Wenn Prozesse aus verschiedenen rekonfigurierbaren Modulen einen Wert ungleich 'Z' geschrieben haben, resultiert dies im Signal mit hoher Wahrscheinlichkeit zu einem undefinierten Wert 'X', der für den Rest der Simulation bestehen bleibt. Die ReChannel-Bibliothek besitzt vermutlich aus diesem Grund bisher keine vordefinierten Accessoren und Port-Traits für Resolved-Signals.

Aber nicht nur Signale sind betroffen. Auch beliebige abstrakte Channels können auf eine Identifikation von Prozessen angewiesen sein. Unter den vordefinierten SystemC-Channeln ist der Mutex-Channel (`sc_mutex`) als Beispiel hierfür zu nennen. Ein Mutex legt einen Lock exklusiv für einen bestimmten Prozess an, der anhand seines Prozessobjekts identifiziert wird. Ausschließlich dieses Prozessexemplar kann diesen Lock wieder aufheben. Die Rekonfiguration eines rekonfigurierbaren Moduls, das mit einem Mutex verbunden ist, kann somit dazu führen, dass ein alter Lock auf diesem Mutex nicht mehr gelöst werden kann und der Mutex somit für andere rekonfigurierbare Module dauerhaft blockiert wird.

2.1.7.d Channels mit Transaktionseigenschaften

Die in Abschnitt 2.1.1 beschriebene Synchronisationsproblematik kann auch innerhalb eines einzelnen Channels vorliegen, falls dieser Funktionalität besitzt, die einer Transaktion ähnlich ist. Dies ist z. B. der Fall, wenn ein Interface über Methoden verfügt, die den Channel in einen Zustand versetzen, der durch den Aufruf einer anderen Methode wieder aufgehoben werden muss. Dadurch, dass ein oder mehrere Zugriffe in zeitlicher Reihenfolge aufeinander folgen müssen, besitzt dieser Vorgang die Eigenschaften einer Transaktion.

Es ist davon auszugehen, dass solche Vorgänge in den meisten Fällen nicht von einem Rekonfigurationsvorgang unterbrochen werden dürfen. Da hierbei die Synchronisationsproblematik durch den Channel selbst hervorgerufen wird, sind derartige Channel-

typen derzeit nur sehr bedingt als Verbindungschannel von rekonfigurierbaren Modulen verwendbar. Zu den Vertretern dieses Channeltyps zählen unter den vordefinierten SystemC-Channeln in erster Linie die Resolved-Signals (vgl. Abschnitt 2.1.6) sowie der Mutex- und der Semaphor-Channel.

2.1.7.e Channels, die mit Spezialports gebunden werden

Ob ein Port mit einem Channel verbunden werden kann, muss in SystemC nicht allein vom Typ des Interfaces abhängen. In manchen Fällen überprüft ein Port bei der Bindung auch direkt den Typ des Channels. Die Ports, die in SystemC für die Bindung mit Resolved-Signals zur Verfügung stehen, sind ein Beispiel hierfür. Aufgrund einer vom SystemC-Standard vorgeschriebenen Typüberprüfung in der Callback-Methode `end_of_elaboration()` lässt sich ein `sc_in_resolved`-Port ausschließlich mit einem Channel vom Typ `sc_signal_resolved` binden.

Da es sich bei einem Accessor nicht um den Channel selbst, sondern um ein Objekt handelt, welches das entsprechende Channelinterface implementiert, ist die Bindung eines Accessors mit derartigen Spezialports nicht möglich. Module, die mit `sc_in_resolved`- oder `sc_in_rv`-Ports ausgestattet sind, scheiden somit für die Verwendung als rekonfigurierbare Module aus.

2.1.8 Interne Verwaltung

In diesem Abschnitt wird die interne Verwaltung der ReChannel-Bibliothek hinsichtlich möglicher Optimierungen untersucht. Wenn die Simulationssemantik von ReChannel (siehe Abschnitt 1.5.1) mit ihrer Repräsentation durch den Kontrollmechanismus (siehe Abschnitt 2.1.1) verglichen wird, ist festzustellen, dass der Mechanismus zur Durchführung der Rekonfigurationsvorgänge verhältnismäßig aufwändig implementiert wurde.

Zustandsmodellierung der Rekonfigurationsvorgänge

Die Anzahl der Zustände des endlichen Automaten im Inneren eines rekonfigurierbaren Moduls beträgt insgesamt sieben (siehe Abbildung 2.3). Die Zustände werden durch Objekte repräsentiert. Deren jeweilige Klasse leitet sich von einer Klasse namens `rc_state` ab. Somit gibt es sieben verschiedene Zustandsimplementationen.

Jede Zustandsklasse besitzt u. a. die beiden Methoden `do_action()` und `next()`. Die Methode `do_action()` enthält den Code, der bei Erreichen dieses Rekonfigurationszustands ausgeführt werden soll. Da sich dieser auszuführende Code auf das rekonfigurierbare Modul bezieht, existiert innerhalb der Klasse `rc_module` jeweils eine `friend`-Deklaration für jede Zustandsklasse. Die Methode `next()` bestimmt den Folgezustand in Abhängigkeit des gerade durchzuführenden Rekonfigurationsvorgangs. Bei jedem Zustandswechsel wird ein neues Zustandsobjekt erzeugt und das alte vernichtet.

Der endliche Automat ist somit zwar formal nach dem Zustands-Entwurfsmuster (State-Pattern) implementiert worden, doch wird kein Vorteil dieses Entwurfsmusters tatsächlich ausgenutzt. Erstens ist die Übersichtlichkeit und Wartbarkeit des Codes stark eingeschränkt, da sich der zugrundeliegende Algorithmus über zahlreiche Klassen

(`rc_module`, `rc_state_machine`, 7 `rc_states`) erstreckt und viele große `switch`-Blöcke verwendet werden. Zweitens können auch keine Effizienzvorteile erreicht werden, da die Zustände lediglich einmalige Aktionen durchführen, die nicht laufzeitkritisch sind. Keiner dieser Zustände ist direkt für das Laufzeitverhalten des Moduls zuständig. Diese Aufgabe wird von den Accessoren übernommen. Die Accessoren kontrollieren die Kommunikation in Abhängigkeit von ihrem Zustand. Jedoch werden im Accessor große `switch`-Blöcke anstelle des State-Patterns verwendet (siehe [17]).

Wie aus Kapitel 1.5.1 hervorgeht, befindet sich ein rekonfigurierbares Modul aus konzeptioneller Sicht lediglich in einem von drei Zuständen. Für die Modellierung des Verhaltens eines rekonfigurierbaren Moduls reichen folglich bereits die Zustände „ungeladen“, „inaktiv“ und „aktiv“ aus. Demgegenüber werden in der derzeitigen Implementation zusätzlich alle Zustandsübergänge explizit durch einen eigenständigen Zustand dargestellt. Die Zwischenzustände haben die Aufgabe, die Durchführungszeit eines Rekonfigurationsvorgangs zu simulieren, aber sie bewirken keine Verhaltensänderung der Module.

Die Idee liegt daher nahe, eine **hierarchische Zustandsmodellierung** zu verwenden, die die Hauptzustände „ungeladen“, „inaktiv“ und „aktiv“ besitzt. Zustandsabfragen bzw. -überprüfungen werden hierbei, insbesondere durch die Verwendung von lediglich drei Hauptzuständen, deutlich vereinfacht. In der Beschreibungsmächtigkeit steht die hierarchische Zustandsmodellierung der linearen in keiner Hinsicht nach.

Aufgrund der deutlichen Verringerung der Zustandsanzahl sowie den vorangegangenen Betrachtungen scheint es nicht sinnvoll zu sein, Zustandsobjekte zu verwenden. Die Hauptzustände des rekonfigurierbaren Moduls können in einer einfachen **enum**-Variablen mit drei möglichen Werten gespeichert werden. Zur Repräsentation der Unterzustände können die bereits vorhandenen Informationen verwendet werden. Aus den existierenden Datenelementen (wie z. B. der Variablen für den Folgezustand) geht jederzeit hervor, in welchem Unterzustand sich das Modul befindet.

Die Überprüfung von (Unter-)Zuständen gestaltet sich einfach, da alle benötigten Information hierfür mittels weniger, zielgerichteter Vergleiche zur Verfügung stehen. Um z. B. festzustellen, ob ein Modul geladen aber noch nicht aktiv ist, muss lediglich der aktuelle Hauptzustand auf Gleichheit mit der Zustandskonstante für „inaktiv“ getestet werden. Ein Vergleich mit mehreren Zustandskonstanten ist hierbei, im Gegensatz zur derzeitigen Implementation, nicht notwendig.

Accessor-Zustände

Der Accessor hat Kenntnis über sämtliche Implementationsdetails des Kontrollmechanismus, da er zusätzliche, über die Kontrolle von IMCs hinausgehende Aufgaben übernimmt. Dies führt u. a. dazu, dass der Accessor nicht nur die gleichen Zustände wie das Modul, sondern auch alle dazu korrespondierenden Kontrollmethoden besitzt. Des Weiteren beinhalten die Makros zur Definition der Interface-Methoden des Accessors, `RC_NONBLOCKING_ACCESS()` und `RC_BLOCKING_ACCESS()`, eine `switch`-Anweisung, die alle sechs Zustände überprüft und den jeweils entsprechenden Code ausführt (siehe [17]). Es besteht daher Grund zu der Annahme, dass die Effizienz des Accessors durch eine Verringerung der Zustandsanzahl gesteigert werden kann.

rekonf. Modul	Accessor
UNDEF	UNDEF
ACTIVE	ACTIVE
DECONFIGURING	ACCESSES_PENDING
INACTIVE	INACTIVE
UNLOADED	”
LOADING	CONFIGURING
CONFIGURING	”
–	RESERVED

Tabelle 2.1: Eine Auflistung der möglichen Zustände des rekonfigurierbaren Moduls und des Accessors. Zustände in derselben Zeile entsprechen sich in ihrer Bedeutung für die Simulation und treten in Kombination auf. Die Gegenüberstellung zeigt, dass hier eine Redundanz in der Zustandsmodellierung vorliegt.

Aus konzeptioneller Sicht kann das Verhalten des Accessors durch lediglich zwei Zustände beschrieben werden: „Kommunikation möglich“ und „Kommunikation gesperrt“ (vgl. Abschnitt 1.5.3). In der derzeitigen Implementation besitzt der Accessor einen eigenständigen Satz von sechs Zuständen. Da das Verhalten des Accessors direkt an den Rekonfigurationszustand eines Moduls gekoppelt ist, stellt die explizite Modellierung von sechs separaten Accessorzuständen eine unnötige Redundanz dar (siehe Tabelle 2.1). Der damit verbundene Verwaltungsaufwand kann eingespart werden. Hierdurch wäre zu erwarten, dass sich eventuelle Erweiterungen um zusätzliche Funktionalität dann wesentlich leichter und fehlerfreier durchführen lassen.

Verteilung des Kontrollmechanismus

Ein weiterer Aspekt ist die Verteilung des Kontrollmechanismus über verschiedenste Klassen, wobei aber nicht die Auslagerung von Funktionalität in unterschiedliche Klassen das Problem darstellt, sondern dass viele Klassen spezifische Detailkenntnisse über den Algorithmus des Kontrollmechanismus besitzen. Dies erschwert die Wiederverwendbarkeit und Erweiterbarkeit der Komponenten.

Verwendete Ressourcen

Da ReChannel als zusätzliche Bibliothek in ein SystemC-Design eingebunden wird, sollten Speicherverbrauch und Anzahl der verwendeten SystemC-Komponenten möglichst gering gehalten werden.

Der endliche Automat innerhalb des rekonfigurierbaren Moduls ist in einem Modul namens `rc_state_machine` gekapselt. Dieses ist somit als Untermodul in jedem rekonfigurierbaren Modul enthalten. Innerhalb von `rc_state_machine` befindet sich ein Thread-Prozess, der den endlichen Automaten repräsentiert und die Rekonfigurationsvorgänge für das zugehörige Modul durchführt. Dies bedeutet, dass pro rekonfigurierbarem Modul ein zusätzliches SystemC-Modul und ein Thread-Prozess erzeugt werden.

Der Speicherbedarf eines Thread-Prozesses setzt sich zusammen aus dem Speicherplatz für die Objektrepräsentation in SystemC, dem Overhead für die Kontextwechsel sowie mehreren Kilobytes für den bei der Ausführung erforderlichen Stapelspeicher[3]. Da Thread-Prozesse somit zu den „teuren" Ressourcen gehören und auch Module einen nicht zu vernachlässigenden Speicherbedarf haben, sollten diese Komponenten in ReChannel möglichst sparsam verwendet werden.

Wenn der Kontrollmechanismus auf andere Weise implementiert wird, kann der gesamte endliche Automat, d.h. sowohl das Modul als auch der Prozess, eingespart werden. Wenn die Kontrollprozesse dazu verwendet werden, den gesamten Rekonfigurationsvorgang selbstständig durchzuführen, dann ist es nicht mehr notwendig, jedes rekonfigurierbare Modul, das in einem Design vorhanden ist, mit einem eigenen Thread-Prozess auszustatten. Dies hätte zur Folge, dass zur Kontrolle und Durchführung sämtlicher Rekonfigurationsvorgänge in einem Design maximal ein Thread-Prozess pro rekonfigurierbarem Bereich benötigt wird. Bei der Modellierung von Systemen auf Basis heutiger, auf dem Markt befindlicher FPGA-Chips, ist die Verwendung von einigen wenigen Kontrollprozessen bereits ausreichend.

`rc_state_machine` verwaltet ferner eine Menge von 17 SystemC-Event-Objekten, von denen circa 14 nicht benötigt werden und somit ebenfalls eingespart werden können.

Der Accessor verwendet ein Signal zur Repräsentation des Accessor-Zustands. Wie bereits erörtert, handelt es sich bei der Modellierung eines eigenständigen Accessor-Zustands um eine verzichtbare Redundanz. Gleiches gilt für das Signal sowie für die Aufgabe, die dieses im Kontrollmechanismus zu erfüllen hat (siehe Abschnitt 2.1.1). Aufgrund der voraussichtlich hohen Anzahl an Accessoren in einem Design und der Tatsache, dass sich alle Accessoren eines rekonfigurierbaren Moduls zu jedem Zeitpunkt im selben Zustand befinden, ist es nicht sinnvoll, jeden Accessor mit einem eigenen Signal auszustatten. Dieser Teil der Accessor-Funktionalität sollte in das rekonfigurierbare Modul verlagert werden.

Der Mechanismus, der in Portals eingesetzt wird, um die Events des statischen Channels an die Accessoren weiterzuleiten, setzt sich modular aus den so genannten `rc_pager`-Komponenten zusammen. Die `rc_pager`-Komponente enthält einen Method-Prozess, der für die Weiterleitung eines Events zuständig ist. In jedem Portal muss daher pro Event des statischen Channels eine `rc_pager`-Komponente angelegt werden. Da es sich bei dieser Komponente um ein Modul handelt, führt dies zu einem zusätzlichen Speicherverbrauch, der sich zu etlichen hundert Bytes pro Portal aufsummieren kann. Auf ein Modul kann verzichtet werden, wenn der Method-Prozess mittels `sc_spawn()` erzeugt wird. Im Gegensatz zum Makro `SC_METHOD()` kann die SystemC-Funktion `sc_spawn()` unabhängig von einem Modul-Kontext aufgerufen werden. Die Anzahl der Method-Prozesse kann jedoch nicht reduziert werden, da ein Prozess nicht für die Weiterleitung von mehr als einem Event verwendet werden kann. Die Ursache hierfür ist, dass in SystemC nicht abgefragt werden kann, welche Events zur Aktivierung eines Prozesses geführt haben. Nur im trivialen Fall, d.h. wenn die Sensitivity-List des Prozesses genau ein Event enthält, kann eine eindeutige Aussage bezüglich des Aktivierungsereignisses getroffen werden.

[3] Die Standardeinstellung der Stapelspeichergröße in OSCI-SystemC (v2.2) beträgt 64 Kilobyte.

2.2 Durchgeführte Änderungen an ReChannel-v1

Modul-Registrierung: Eine Klasse namens `rc_module_registry` wurde implementiert, bei der sich alle rekonfigurierbaren Module automatisch bei deren Erzeugung registrieren. Die Modul-Registrierung dient dem Zweck, die in einem Design vorhandenen rekonfigurierbaren Module global anhand ihres SystemC-Modulnamens ermitteln zu könnnen. Zusätzlich wird eine Syntax zur Bildung von relativen Namen zur Verfügung gestellt, mit der ein Modul von jedem Punkt in der Modulhierarchie aus adressiert werden kann.

Änderungen an rc_control: Die Klasse der Rekonfigurationskontrolle (`rc_control`) wurde um Kontrollmethoden ergänzt, denen anstelle eines Pointers der SystemC-Name des zu rekonfigurierenden Moduls übergeben wird. Die hierfür erforderliche Namensauflösung erfolgt mittels der Funktionalität, die durch die Modul-Registrierung bereitgestellt wird. Mit den neuen Kontrollmethoden ist es nun möglich, in einem Design auf die Verwendung bzw. Weitergabe von Pointern auf rekonfigurierbare Module zu verzichten.

Channelinterface für rc_control: Ein Interface namens `rc_control_if` wurde definiert, das alle Kontrollmethoden der Kontrollkomponente enthält. Da `rc_control` dieses Interface implementiert, ist die Kontrollkomponente ein Channel und kann mit einem Port gebunden werden. Dadurch ist nun eine wie in SystemC übliche, modulare Verwendung möglich.

Refactoring von rc_module_set: Die Klasse `rc_module_set` (siehe Kapitel 1.5.5) sollte einem Refactoring unterzogen werden, da hierin ein nicht näher spezifizierter Fehler vermutet wurde. Die Schnittstelle der Klasse und deren Implementation wurde vereinfacht und auf eine korrekte Funktionsweise überprüft.

Änderungen am Accessor: Die Makros zur Definition von Interface-Methoden im Accessor (siehe [17]) wurden durch Objektmethoden ersetzt. Im Accessor stehen dazu nun die beiden Methoden `rc_forward()` und `rc_nb_forward()` zur Verfügung. Diese liefern bei Aufruf ein temporäres Objekt zurück, mittels dessen auf den statischen Channel zugegriffen werden kann. Die Funktionalität der Makros befindet sich nun in leicht optimierter Form in diesen Zugriffsobjekten. Die Zählung der externen Zugriffe mittels `pending_accesses` wird durch die Konstruktoren und Destruktoren der Objekte durchgeführt. Die in Listing 1.3 definierte Interface-Methode `write` kann nun in der folgenden, intuitiveren Schreibweise implementiert werden:

```
void write(const T& data)
    { this->rc_nb_forward()->write(data); }
```

Die Notation mittels `rc_forward()` bzw. `rc_nb_forward()` bietet einen besonderen Vorteil bei der Implementierung von Interface-Methoden mit Rückgabewert, da hierbei bisher eine mehrzeilige Notation (bestehend aus einer lokalen Variablendeklaration, dem Makroaufruf sowie einer `return`-Anweisung) notwendig war.

3 Vorüberlegungen zum Re-Design von ReChannel

Die Analyse in Kapitel 2 hat aufgezeigt, welche Probleme bei der Simulation von DR auf funktionaler bzw. transaktionaler Abstraktionsebene bestehen. Allerdings hat sich hierbei auch herausgestellt, dass in der bestehenden Implementation der ReChannel-Bibliothek noch technische und konzeptionelle Probleme vorliegen, die zuvor gelöst werden müssen. Hierfür sind vielfältige, funktionale Änderungen erforderlich, die über ein Refactoring hinausgehen und fast alle Klassen betreffen.

Bevor in ReChannel Sprachkonstrukte zur Lösung der Synchronisationsproblematik integriert werden können, müssen alle implementationsbedingten Ursachen (siehe Abschnitt 2.1.1, „Probleme mit dem Kontrollmechanismus“) beseitigt werden.

Der Accessor muss dazu befähigt werden, nichtblockierende Zugriffe vom verbundenen statischen Channel abhalten zu können.

Zudem muss das Problem gelöst werden, dass blockierende Zugriffe noch nach der Deaktivierung eines Moduls zugelassen werden. Dies ist auf die derzeitige Implementation des Accessors zurückzuführen, da hier ein Signal zur Zustandsrepräsentation verwendet wird. Da das Signal nach dem Request-Update-Verfahren arbeitet, ändert sich der Accessor-Zustand trotz sofortiger Deaktivierung des Moduls immer erst beim Übergang zum nächsten Delta-Zyklus. Dies bewirkt, dass alle externen Zugriffe innerhalb eines Delta-Zyklus gleich behandelt werden (siehe Abschnitt 2.1.1). Dies gewährleistet einen deterministischen Simulationsablauf, führt aber im Gegenzug auch dazu, dass zum Zeitpunkt der Deaktivierung noch dauerhaft blockierende, externe Zugriffe gestartet werden können. Da diese blockierenden Zugriffe bei der Feststellung eines möglichen Deaktivierungszeitpunktes nicht beachtet werden, besteht hier eine Verletzung der Simulationssemantik von ReChannel.

Derzeit ist sämtliche Funktionalität starr auf eine Verwendung von Portals ausgerichtet. Bevor die ReChannel-Bibliothek um eine Exportal-Komponente erweitert werden kann, ist daher eine grundlegende Neukonzeption des Kontrollmechanismus erforderlich (vgl. Abschnitt 2.1.3, „Exportal“).

Weitere Gründe, die für eine Neukonzeption des Kontrollmechanismus sprechen:

- Die interne Verwaltung der ReChannel-Klassen (wie z. B. die Zustandsmodellierung) sollte hinsichtlich der Beobachtungen aus Abschnitt 2.1.8 umgestaltet werden, da andernfalls alle vorzunehmenden Erweiterungen auf einer unnötig aufwändigen Infrastruktur aufbauen müssten.

- Die direkten Abhängigkeiten vieler ReChannel-Klassen untereinander behindern die Implementierung von neuer Funktionalität und müssen daher zuvor aufgelöst werden. Der häufige Einsatz von `friend`-Deklarationen ist z. B. ein Grund für die engen Beziehungen der Klassen untereinander, da mittels `friend`-Deklarationen klassenübergreifende Zugriffe auf private Methoden und Datenelemente implementiert wurden.

- Die Nebenläufigkeit des endlichen Automaten innerhalb der rekonfigurierbaren Module führt in einigen Fällen zu undefiniertem Verhalten. Daher sind Fehlerkorrekturen und weitere Überprüfungen notwendig.

- Der Rekonfigurationsalgorithmus liegt über viele Klassen verstreut vor. Eventuell vorhandene Fehler sind schwer ausfindig zu machen.

- In seiner derzeitigen Form ist der Kontrollmechanismus nicht für die nebenläufige Benutzung durch mehrere Kontrollprozesse geeignet. Einem Kontrollprozess fehlt z. B. die Möglichkeit, die von ihm momentan verwendeten, rekonfigurierbaren Module reservieren zu können, um sie vor dem Zugriff durch andere Kontrollprozesse zu schützen.

- Die Benutzerschnittstelle der Rekonfigurationskontrolle muss überdacht werden, da einige der bereitgestellten Kontrollmethodentypen potentiell riskante Eigenschaften besitzen.

Für das in Abschnitt 2.1.2 beschriebene Treiberproblem muss eine allgemeine Lösung gefunden werden. Nicht nur einfache Signale sind hiervon betroffen, sondern es hat sich herausgestellt, dass sämtliche Channels, die eine Prozessidentifikation vornehmen, davon betroffen sind. Daher muss eine Lösung gefunden werden, die die Vorteile von ReChannel gegenüber der Simulation von DR mittels Multiplexern (siehe [19]) nicht aufhebt und gleichzeitig für beliebige Channels anwendbar ist. Dabei muss beachtet werden, dass eine solche Lösung gleichermaßen bei Portals und Exportals zum Einsatz kommen kann. Bei der Verwendung von Exportals können Treiberkonflikte ebenso wie beim Portal auftreten, falls in ReChannel später eine Möglichkeit zur Mobilität von rekonfigurierbaren Modulen integriert wird. Daher muss hier bereits die grundsätzliche Designentscheidung getroffen werden, in welcher Komponente bzw. in welcher Klasse die entsprechende Funktionalität implementiert werden muss, um allen Anforderungen gerecht werden zu können.

Da zwischen den ReChannel-Klassen viele feste Abhängigkeiten bestehen, sind die meisten Änderungen nicht lokal durchzuführen, sondern können nur klassenübergreifend gelöst werden. Aufgrund des Umfangs dieser Änderungen wird der Aufwand einer schrittweisen Veränderung des ReChannel-Codes deutlich aufwändiger und fehleranfälliger eingeschätzt als von Grund auf ein sauberes Re-Design durchzuführen.

3.1 Experimentelle Implementationen

Solange einige wichtige technische und konzeptionelle Fragestellungen noch nicht beantwortet sind, können keine begründeten Designentscheidungen getroffen werden. Bevor mit einem Re-Design begonnen werden kann, müssen daher zuerst Lösungskonzepte für die folgenden, grundlegenden Probleme gefunden werden:

- Treiberproblem
- Integration einer Exportal-Komponente in die Klassenhierarchie
- Umsetzbarkeit der in Abschnitt 2.1.8 beschriebenen Optimierungen
- sinnvollste Aufteilung der Funktionalität auf die Klassen

Zu diesem Zweck wurden zwei Testimplementierungen durchgeführt:

1. Da ein Re-Design auch fundamentale Änderungen nicht ausschließt, hatte die erste dieser experimentellen Implementierungen das Ziel, mögliche alternative Verfahren zur Umsetzung der Simulationssemantik von ReChannel zu finden. Hierbei wurde u. a. ausprobiert, wie eine Implementation ohne Accessoren aussehen könnte.

   Das Portal ist wie bisher auf der statischen Seite mit einem Channel verbunden, erzeugt aber für die Bindungen auf der rekonfigurierbaren Seite dynamisch jeweils einen Port pro rekonfigurierbarem Modul. Für einen benutzerdefinierten Channeltyp muss keine Accessor- oder Traits-Klasse definiert werden, sondern es wird anstelle dessen lediglich die Portal-Klasse spezialisiert. Es gibt keinen endlichen Automaten, der durch einen nebenläufigen Prozess repräsentiert wird. Die Zustände des Portals und des rekonfigurierbaren Moduls sind reduziert. Die Rekonfigurationskontrolle ist in einem Sockel namens `rc_area` integriert, der einen rekonfigurierbaren Bereich repräsentiert und die rekonfigurierbaren Module enthält. Mittels `rc_area` sollte eine Schnittstelle für die Ortsbindung der Module modelliert werden, die möglicherweise für das Verschieben von Modulen in einem Design (Mobilität) verwendet werden könnte.

   Obwohl es viele implementationsbedingte Unterschiede gibt, beschreibt dieser experimentelle Implementationsansatz dieselbe Simulationssemantik wie die derzeitige ReChannel-Implementation.

   Da der Accessor fehlt und somit die rekonfigurierbaren Module fest mit den Portals verbunden sind, ist diese Variante allerdings später nicht um die Mobilität von rekonfigurierbaren Modulen erweiterbar. Auch ist die Kontrolle der Kommunikation hierbei zwangsläufig etwas aufwändiger, da das Portal die Herkunft eines Prozesses bei jedem Zugriff überprüfen muss. Keinen Accessor zu verwenden, wird daher als Designfehler betrachtet. Abgesehen davon konnte aber gezeigt werden, dass die Modellierung der Rekonfigurationszustände deutlich vereinfacht werden kann und ein nebenläufiger, endlicher Automat nicht benötigt wird.

Das Treiberproblem wird dadurch gelöst, dass eine **spezielle Zugriffsart für Channels mit Prozessidentifikation** zur Verfügung gestellt wird. Hierbei wird ein IMC nicht direkt von dem Prozess des rekonfigurierbaren Moduls ausgeführt, sondern in einem Funktionsobjekt gekapselt und zur Ausführung an einen anderen, speziell für diesen Zweck angelegten Prozess weitergereicht. Dieser zusätzliche Prozess tritt im Channel als einziger Treiber in Erscheinung. In dieser experimentellen Umsetzung können nur Treiberzugriffe von Method-Prozessen weitergeleitet werden. Für eine robuste und allgemein verwendbare Lösung sind noch diverse Verbesserungen erforderlich.

Bei Überlegungen während der Vorbereitung zu dieser Arbeit ist ein Konzept entstanden, wie sich **rücksetzbare Prozesse** und **rücksetzbare Komponenten** als Spracherweiterung in SystemC integrieren lassen, ohne dass Änderungen am Simulationskernel von SystemC vorgenommen werden müssen. Einige Kernaspekte dieses Konzepts wurden einmal probeweise implementiert, um die technische Umsetzbarkeit zu belegen. Mittels rücksetzbarer Prozesse kann rekonfigurierbares Verhalten in SystemC simuliert werden. In der Kombination mit implizit rücksetzbaren Komponenten stehen dann alle Mittel zur Verfügung, um ein beliebiges, rekonfigurierbares Modul direkt beschreiben zu können. Hierbei ist ein Beschreibungsstil möglich, der formal mit dem von SystemC gewohnten bis auf wenige Ausnahmen identisch ist. Alle Prozesse, Channels und Variablen eines derart beschriebenen Moduls werden implizit bei der Rekonfiguration auf ihren initialen, internen Zustand zurückgesetzt. Daher unterliegen solche Module nicht den in Abschnitt 2.1.4 und 2.1.5 beschriebenen Problematiken. Sie verhalten sich bei einer mit ReChannel simulierten Rekonfiguration exakt wie dynamisch rekonfigurierbare Hardware.

2. Da Portals und Exportals nicht nur Unterschiede, sondern auch einige Gemeinsamkeiten besitzen, könnte es von Vorteil sein, diese Klassen auf eine gemeinsame Basisklasse zurückzuführen. Die Klasse des Portals (`rc_portal`) ist in ReChannel derzeit vom Klassentemplate `rc_portal_b` abgeleitet. Diese wiederum ist von der Klasse `rc_portal_base` abgeleitet. Beide Basisklassen sind auf die Funktionalität des Portals und auf die Verwaltung von Accessoren spezialisiert. Daher hat eine Zusammenführung von Portal und Exportal – vergleichbar mit einer Zusammenführung des Multiplexer- und Demultiplexer-Konzepts – eine grundlegende Neukonzeption dieser Klassen zur Folge.

   Die zweite Testimplementierung hatte das Ziel, einige Möglichkeiten für die Integration des Exportals in ReChannel auszuprobieren, ohne hierfür bereits konkrete und umfangreiche Änderungen an der bestehenden Implementation vornehmen zu müssen. Die bei dieser Testimplementierung gewonnenen Erkenntnisse haben ergeben, dass sowohl die Erweiterung um eine zusätzliche Basisklasse `rc_exportal_base` als auch die Integration der Exportal-Funktionalität in `rc_portal_base` als Designfehler zu betrachten sind. Diese Ansätze verlagern dasselbe Problem, das derzeit bei der Erweiterung um das Exportal besteht, lediglich

in die Zukunft. Denn dieselbe Situation ergäbe sich von neuem für eventuelle, zum jetzigen Zeitpunkt noch unbekannte SystemC-Komponenten oder zusätzlich benötigte Implementationsvarianten von Portal und Exportal.

Durch die durchgeführten, experimentellen Implementationen konnten wichtige Erkenntnisse hinsichtlich der Vor- und Nachteile verschiedener Ansätze gewonnen werden. Außerdem ist nun sicher, dass ein Re-Design von ReChannel erfolgreich durchgeführt werden kann.

3.2 Designentscheidung zum Re-Design

In den vorangegangenen Abschnitten wurde erörtert, aus welchen Gründen ein Re-Design der ReChannel-Bibliothek wünschenswert ist. Dass ein Re-Design mit der angestrebten Zielsetzung auch technisch durchführbar ist, konnte mittels experimenteller Implementierungen bewiesen werden. Es wird aufgrund dessen die Entscheidung getroffen, ein komplettes Re-Design von ReChannel durchzuführen.

4 Entwurf und Implementation

Dieses Kapitel beschreibt die Durchführung des Re-Designs der ReChannel-Bibliothek. Da Entwurf und Implementierung hierbei nicht strikt voneinander zu trennen sind, werden diese gemeinsam in einem Kapitel dargestellt.

Die ursprüngliche Implementation der ReChannel-Bibliothek [19] wird im Folgenden als **ReChannel-v1** und die neue, aus dieser Arbeit hervorgegangene ReChannel-Version, mit **ReChannel-v2** bezeichnet. Die Namen mit Versionszusatz werden verwendet, wenn die beiden Implementationen miteinander verglichen werden. Ist die Versionsnummer nicht explizit angegeben, wird entweder das allgemeine Konzept von ReChannel oder die neue Implementation bezeichnet. Die jeweilige Bedeutung ist aus dem Kontext heraus ersichtlich.

4.1 Zielsetzung

Für das Re-Design gelten auch weiterhin die ursprünglichen, primären Zielsetzungen von ReChannel, die wie folgt zusammengefasst werden können:

1. ReChannel soll sich nahtlos in den gewohnten Design-Flow einfügen.
2. Bestehende Module können in rekonfigurierbare Module verwandelt werden, ohne Änderungen an deren Code vornehmen zu müssen.
3. Die Bibliothek ist für die Verwendung benutzerdefinierter Channels anpassbar.
4. Das Refinement von rekonfigurierbaren Modulen soll möglich sein.
5. Für die Synthese von rekonfigurierbaren Modulen sollen die üblichen Standardtools verwendet werden können.
6. ReChannel soll SystemC erweitern ohne Änderungen am SystemC-Simulationskernel vorzunehmen.

Die Simulationssemantik für die Simulation von DR soll wie in Kapitel 1.5.1 beschrieben umgesetzt werden. Mit dem einzigen Unterschied, dass nicht nur Module sondern auch beliebige Objekte rekonfiguriert werden können.

Für die Korrektheit einer Simulation ist entscheidend, dass die Simulationssemantik so exakt wie möglich umgesetzt wird. Daher soll z. B. nicht mehr toleriert werden, dass als „ungeladen“ ausgewiesene Module in bestimmten Fällen dennoch externe Zugriffe durchführen können.

Die Bibliothek soll speziell auf die Wiederverwendbarkeit von Code und die einfache Erweiterbarkeit um neue Komponenten und Implementationsvarianten ausgerichtet sein. Eine Klassenhierarchie soll entworfen werden, die die festen Abhängigkeiten zwischen den Klassen minimiert.

Nach wie vor sollen Hilfskomponenten in Form von Portal und Accessor zur Kontrolle der Kommunikation verwendet werden. Um mit ReChannel auch die Rekonfiguration von Channeln sowie Modulen mit Exports simulieren zu können, soll eine weitere Komponente namens Exportal hinzugefügt werden. Portal und Exportal sollen auf das Konzept eines allgemeinen, für Rekonfigurationszwecke verwendbaren Schalters (Switch) vereinigt werden, der durch ein *Abstract Base Interface* repräsentiert wird.

Mit dieser neuen Sichtweise wird erreicht, dass Portal, Exportal und Accessor lediglich Implementationsdetails darstellen und somit dem zentralen Rekonfigurationsalgorithmus nicht bekannt sind. Die Konzepte der bisher in ReChannel verwendeten Algorithmen und Datenstrukturen sollen dahingehend abstrahiert werden, dass die Erweiterung um beliebige, zusätzliche Switch-Implementationen (inklusive auch solchen, die z. B. keinen Accessor verwenden) keine Änderungen am Rekonfigurationsalgorithmus erforderlich machen.

Des Weiteren sollen alle in Kapitel 2 beschriebenen Problemstellungen allgemein gelöst werden (Treiberkonflikte, Synchronisation der DR, etc.). Die Bibliothek soll – soweit dies technisch möglich ist – die Verwendung des vollen Sprachumfangs des SystemC-Standards [7] bei der Simulation von DR ermöglichen.

Ferner soll die Bibliothek um die in Abschnitt 3.1 vorgestellten rücksetzbaren Prozesse und Komponenten erweitert werden. Mithilfe dieser Funktionalität soll ReChannel-v2 auch zur expliziten Modellierung von DRHW verwendet werden können. Diese Erweiterung soll sich nahtlos in das bestehende ReChannel-Konzept einfügen. Um eine größtmögliche Freiheit und Flexibilität zu gewährleisten, sollen die mit ReChannel-v2 modellierten Beschreibungen darüber hinaus auch außerhalb der Simulation von DR (d.h. in statischen Designs) verwendet werden können.

Da es sich bei ReChannel um eine Spracherweiterung von SystemC handelt, die von Systemdesignern bei der Modellierung von DRHW eingesetzt werden soll, sind nur solche Spracherweiterungen akzeptabel, die zu verständlichen Konzepten und Sprachkonstrukten mit intuitiver Verwendung führen. Ob und auf welche Weise die hierzu nötige Funktionalität implementiert werden kann, hängt größtenteils von den Eigenschaften und Besonderheiten von C++, SystemC und der verwendeten Compiler ab. Daher muss beim Entwurf und der Implementierung häufig zwischen der konzeptionellen Anforderung, der technischen Umsetzbarkeit und der Bedienerfreundlichkeit der Spracherweiterungen abgewogen werden.

Zusätzlich soll ReChannel Funktionalität hinzugefügt werden, die es ermöglicht, rekonfigurierbare Objekte während der Simulation beliebig in einem Design zu verschieben. Auch ein Wechsel der Zugehörigkeit einer Rekonfigurationskontrolle soll möglich sein. Die Verschiebbarkeit von rekonfigurierbaren Objekten (RO) wird im Folgenden als *Mobilität* bezeichnet. Eine Möglichkeit zur Mobilität von RO ist wünschenswert, da hierdurch weitere Eigenschaften eines Rekonfigurationsstreams in der Simulation abgebildet werden können. Das Konzept der Mobilität muss im Entwurf frühzeitig mit einbezogen werden,

da sich gezeigt hat, dass dieses Auswirkungen auf andere Konzepte haben kann (z. B. Treiberproblem bei Exportals, siehe Kapitel 3).

Die Portabilität zwischen verschiedenen SystemC-Implementationen und C++-Compilern soll durch die Standardkonformität der ReChannel-Bibliothek zum SystemC-Standard (*IEEE-1666-2005*) und zum aktuellen C++-Standard (*ISO/IEC 14882:2003*) gewährleistet werden. Spezifisches Wissen über Implementationsdetails einer bestimmten SystemC-Implementation oder eines C++-Compilers sollen nur in solchen Fällen verwendet werden, in denen eine Kompatibilität zu der jeweiligen Software nicht anders hergestellt werden kann (ein möglicher Grund hierfür wäre z.B. ein Bug oder eine spezielle Einschränkung einer Software).

Es soll grundsätzlich darauf geachtet werden, dass häufig ausgeführter und daher laufzeitkritischer Code möglichst effizient programmiert wird. Doch ist das Ziel bei der Implementation von ReChannel-v2 vorrangig auf Funktionsumfang und Bedienungskomfort und nicht auf Performance-Aspekte konzentriert.

Zusammengefasst ergeben sich hieraus die folgenden Zielsetzungen:

1. exakte Umsetzung der Simulationssemantik
2. Abstraktion der Algorithmen und Datenstrukturen
3. abstrakte Schnittstellen in Form von Abstract Base Interfaces (ABIs)
4. Modularer Aufbau (z. B. Paketstruktur, Wiederverwendbarkeit von Code)
5. Lösung der bestehenden Probleme / fehlende Funktionalität ergänzen
6. robuste und leicht verständliche Benutzerschnittstelle
7. Mobilität der RO zwischen rekonfigurierbaren Bereichen
8. Konstrukte zur expliziten Beschreibung von DRHW (Verhalten und Struktur)
9. bedienerfreundliche Spracherweiterungen / Syntax an SystemC orientiert
10. Kompatibilität zum SystemC- und C++-Sprachstandard

Die oben aufgeführten Zielsetzungen dienen zusammengenommen dem Zweck, in ReChannel die Simulation von DR auf allen Abstraktionsebenen zu ermöglichen.

Eine andere Diplomarbeit [1] beschäftigt sich zeitgleich zu dieser Arbeit mit der praktischen Verwendung der ReChannel*-Bibliothek in einem Anwendungsprojekt und dem hiermit verbundenen Workflow. Da ReChannel-v1 die in Kapitel 2 beschriebenen Probleme aufweist, ist es wünschenswert, wenn schon möglichst frühzeitig eine lauffähige ReChannel-v2-Bibliothek mit der benötigten Basisfunktionalität zur Verfügung steht.*

4.2 Konzeption

Die gewählte Vorgehensweise bei der Durchführung des Re-Designs ist wie folgt:

1. In einem ersten Schritt soll eine robuste Basisimplementation geschaffen werden, welche die in Abschnitt 1.5.1 beschriebene Simulationssemantik korrekt beschreibt und der ursprünglichen Implementation hinsichtlich des Funktionsumfangs bereits entspricht. Diese Basisimplementation soll abstrakte Schnittstellen verwenden und so modular wie möglich aufgebaut sein. Das Konzept hierfür soll bereits zu Beginn der Implementierung weitgehend ausgearbeitet sein. Spezielle Detailfragen können während der Implementierung geklärt werden, wenn die jeweils benötigten technischen Grundvoraussetzungen erreicht sind und die zur Lösung erforderlichen Erkenntnisse vorliegen.

2. Im nächsten Schritt soll die Bibliothek um weiterführende Funktionalität, wie z. B. Exportals, Synchronisationsfunktionalität und neue Sprachkonstrukte, erweitert werden. Aufgrund der konsequenten Trennung von Schnittstelle und Implementation in der Basisimplementation wird davon ausgegangen, dass die Integration neuer Funktionalität ohne großen Aufwand durchführbar ist.

Im Folgenden sollen anhand eines Klassendiagramms die Abhängigkeiten veranschaulicht werden, die Probleme bei der Erweiterung von ReChannel bereiten und es sollen entsprechende Lösungskonzepte für den Entwurf der Basisimplementation entwickelt werden. Das Klassendiagramm in Abbildung 4.1 skizziert die in ReChannel-v1 vorliegende Klassenstruktur. Es sind darin die zentralen ReChannel-v1-Klassen und deren Abhängigkeiten untereinander dargestellt.

PORT → Port-Traits → Accessor

In ReChannel-v1 bestimmt das Klassentemplate des Portals die von ihm zu verwendende Accessor-Klasse und den dazugehörigen Interfacetyp (IF) mithilfe einer Port-Traits-Klasse. Diese Port-Traits-Klasse wird mit dem Porttyp (PORT) des Portals parametrisiert. Da die Klasse des Accessors somit anhand eines Porttyps bestimmt wird, kann dieses Verfahren nicht bei Exportals angewandt werden. Da Traits-Klassen grundsätzlich auf Benutzerseite definiert werden müssen, ist es nicht praktikabel, eine weitere Traits-Klasse für das Exportal einzuführen.

Idealerweise sollten Portal und Exportal ohne Zutun des Anwenders an die Klasse des Accessors gelangen können. Um dies zu erreichen, soll in ReChannel-v2 die Konvention eingeführt werden, dass Accessor-Implementationen die Spezialisierung eines Klassentemplates namens `rc_accessor<IF>` sein müssen. Aufgrund der Parametrisierung mit dem Interfacetyp IF kann somit zu jedem Interface der dazu kompatible Standard-Accessor generisch ermittelt werden.

Doch besteht im Falle des Portals weiterhin das Problem, dass aus einem Porttyp der dazu kompatible Interfacetyp nicht auf generische Weise ermittelt werden kann. Obwohl in der SystemC-Referenzimplementation sämtliche Port-Klassen ein Typedef namens `if_type` mit dem dazugehörigen Interfacetyp besitzen, darf dieses nicht

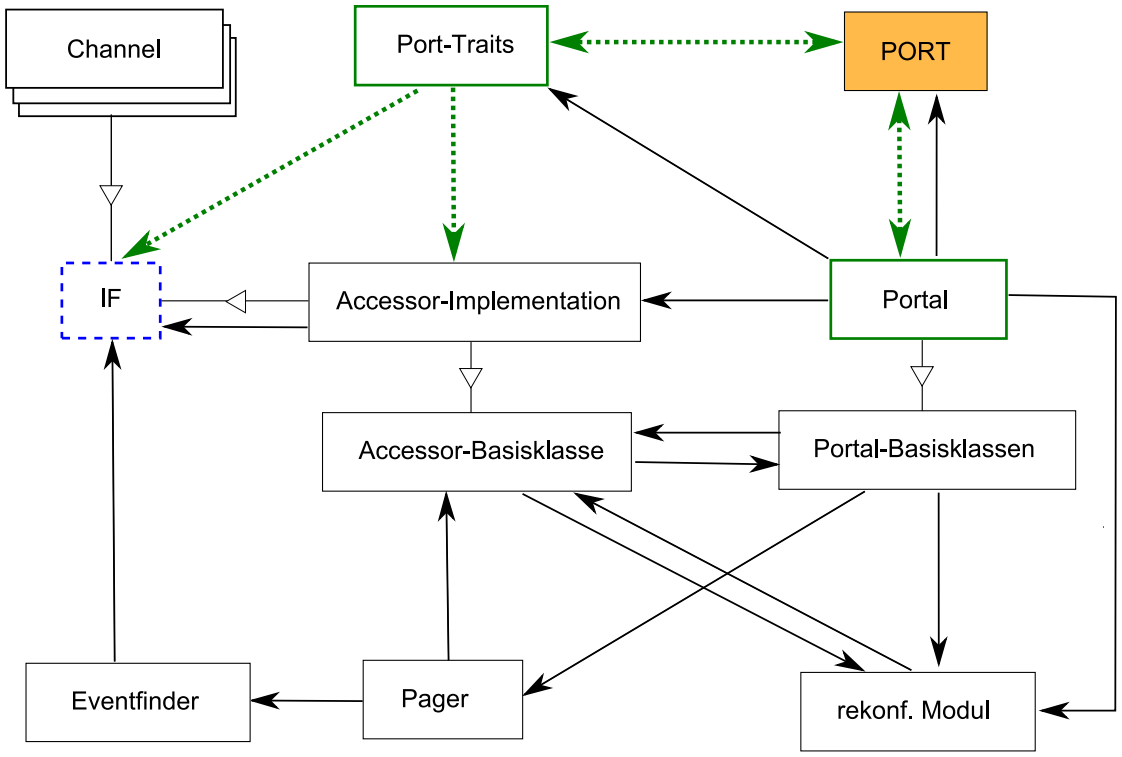

Abbildung 4.1: Übersicht über die Klassenabhängigkeiten von ReChannel-v1. Anhand eines gegebenen Porttyps (PORT) werden in ReChannel-v1 alle zur Rekonfiguration dieses Ports benötigten Klassentypen festgelegt. Hierzu zählen die Klasse des Portals, des Accessors sowie des Interfaces (IF). Die mit diesem Port verbindbaren Channels mit passendem Interfacetyp sind ebenfalls eingezeichnet. Aus dem Diagramm geht hervor, dass in ReChannel-v1 viele Klassen miteinander direkte Abhängigkeiten besitzen und die gesamte Klassenhierarchie somit fest auf die Rekonfiguration von Modulen mit Ports ausgerichtet ist. [Legende: siehe Anhang C]

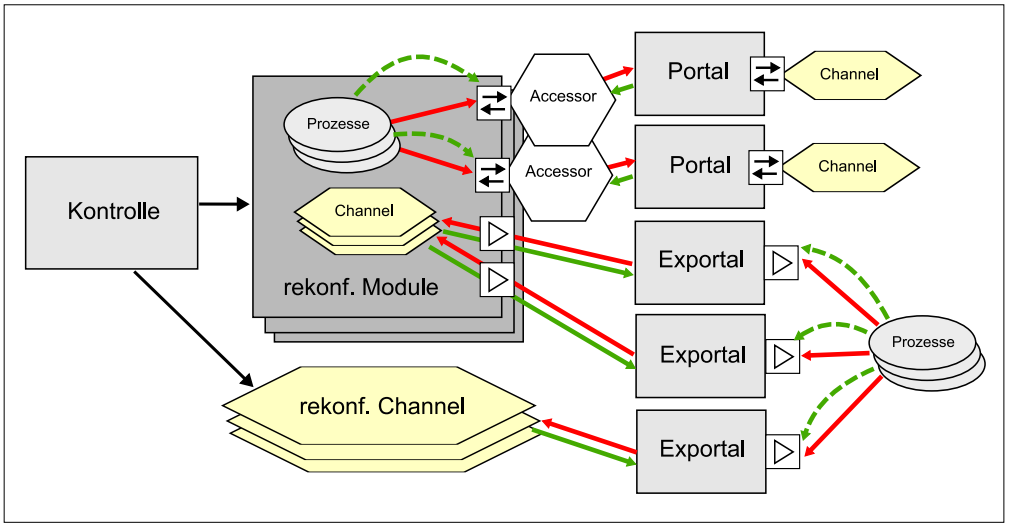

Abbildung 4.2: Grafische Darstellung der möglichen Kommunikationswege und Verbindungen zwischen ReChannel- und SystemC-Komponenten nach der Einführung des Exportals

verwendet werden, da ReChannel standardkonform bleiben soll. In der Spezifikation der Port-Klasse im SystemC-Standard (siehe [7]) fehlt eine derartige Typedef-Deklaration. Darüber hinaus existiert in C++ kein `typeof`-Operator, der bei dieser Problematik hilfreich sein könnte; und auch die Boost-Bibliothek kann keine allgemeine Auflösung unbekannter Typen bieten. Da es sich bei der Problematik nicht um ein rechannel-spezifisches Problem handelt und zudem die Möglichkeit besteht, dass diese in zukünftigen Versionen von SystemC bzw. C++ nicht mehr besteht, sollte es durch Hilfsfunktionalität gelöst werden, die von anderen ReChannel-Klassen (wie z. B. dem Accessor) unabhängig ist. In diesem Zusammenhang sind dann benutzerdefinierte Port-Traits-Klassen erforderlich, die ein Typedef namens `if_type` deklarieren, falls der entsprechende Port kein solches besitzt.

rekonf.Modul → Accessor → Portal → rekonf.Modul

Die Dreiecksbeziehung zwischen dem rekonfigurierbaren Modul, dem Accessor und dem Portal stellt eine der entscheidenen Schlüsselabhängigkeiten dar, die einer Erweiterung der ReChannel-Bibliothek um weitere Switches im Wege stehen. Wie in Abschnitt 2.1.1 beschrieben, verläuft in ReChannel-v1 die Kontrollhierarchie der Komponenten über den Accessor. Das Portal, das an das Modul aus Benutzersicht direkt angeschlossen wird, ist vom Modul aus nur über den Umweg durch den Accessor steuerbar. Dies entspricht der in SystemC vorliegenden Verbindungsstruktur, bei der das Modul über einen seiner Ports mit dem Accessor verbunden ist.

Die Situation, die sich bezüglich Kommunikationswegen und Verbindungsstruktur ergibt, wenn zusätzlich zum Portal ein Exportal zu ReChannel hinzugefügt wird, ist in Abbildung 4.2 skizziert. Hier wird ersichtlich, dass sich Portal und Exportal be-

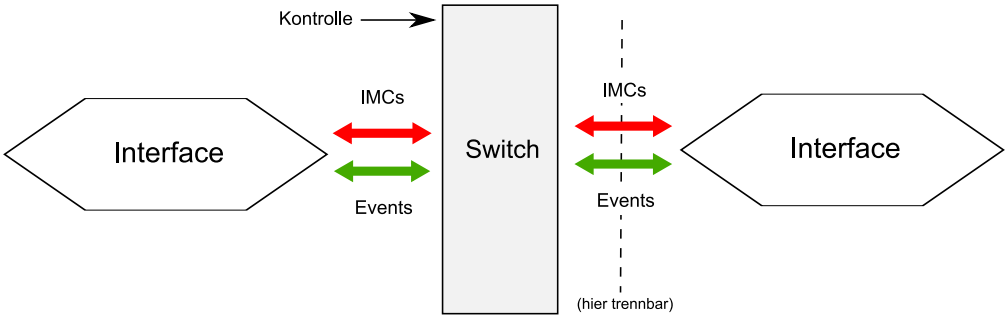

Abbildung 4.3: Verallgemeinerte Sichtweise einer Switch-Komponente.

züglich Kommunikation und Bindung sehr unterschiedlich verhalten. Beim Entwurf der Klassenhierarchie von ReChannel-v2 muss daher versucht werden, den Algorithmus und die Datenstrukturen zur Simulation von DR dahin gehend zu abstrahieren, dass diese in allgemein verwendbare, voneinander unabhängige Konzepte aufgespalten werden können. Die hierbei relevanten Aspekte sind:

a) die Bindung der Komponenten,

b) die Kontrolle der Rekonfiguration und

c) die Kontrolle der Kommunikation.

Aufgrund der Erkenntnisse aus Abschnitt 3.1 wird eine gemeinsame Schnittstelle für Portal und Exportal in Form eines Abstract Base Interfaces (ABI) als die sinnvollste Lösung erachtet. Um größtmögliche Flexibilität und Erweiterbarkeit zu gewährleisten, muss dieses ABI von der jeweiligen Implementation unabhängig sein und darf daher nicht über portal- oder exportalspezifische Methoden verfügen. Das Problem der Integration des Exportals in ReChannel konzentriert sich somit auf den Entwurf eines **Switch-ABIs**, das die *allgemeine Abstraktion eines für Rekonfigurationszwecke verwendeten Schalters* repräsentiert.

Welche Komponenten in Verbindung mit einem Switch verwendet werden (z. B. Accessor oder Channel) und wie diese gebunden werden (Port, Export, etc.), ist abhängig vom jeweiligen Switchtyp und kann nicht für alle Switches auf einheitliche Weise über ein ABI erfolgen. Die Erzeugung und Bindung des Accessors wird somit als internes Implementationsdetail eines Switches angesehen. Dies hat den Vorteil, dass die rechannelinternen Bindungen von Switches mit rekonfigurierbaren Modulen (bzw. Channeln) von außen betrachtet auf identische Weise erfolgen. Hierbei ist somit lediglich eine gegenseitige Registrierung erforderlich, die über das Switch-ABI vorgenommen werden kann.

Da Portal und Exportal bezüglich der Kommunikationsrichtung gegensätzlich arbeiten, muss die Weiterleitung von IMCs und Events in beiden Richtungen möglich sein. Hierbei ist festzustellen, dass die Gemeinsamkeit von Portal und Exportal darin besteht, dass unabhängig von der Richtung immer eine Weiterleitung zwischen einem Interface auf der statischen und vielen Interfaces auf der rekonfigurierbaren Seite er-

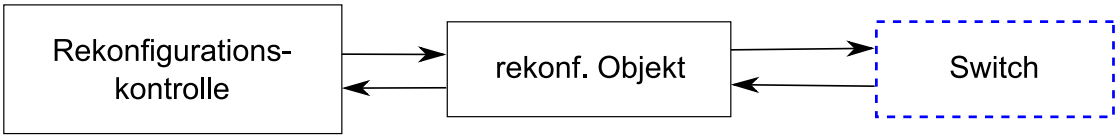

Abbildung 4.4: Abhängigkeiten der Klassen, die an der Kontrolle der Rekonfiguration aus Sicht des Rekonfigurationsalgorithmus beteiligt sind

folgt. Dies führt zu der in Abbildung 4.3 dargestellten, verallgemeinerten Sichtweise der Switch-Komponente, die lediglich SystemC-Interfaces kennt. Das Switch-ABI kann sich somit darauf beschränken, Kontroll-, Registrierungs- und Informationsmethoden für diese allgemeine Sichtweise bereitzustellen (siehe Abschnitt 4.4).

Aufgrund der Verwendung des Switch-ABIs kann der Rekonfigurationsalgorithmus nun unabhängig von den jeweils verwendeten Switch-Implementationen modelliert werden. Abbildung 4.4 zeigt die Abhängigkeiten der Klassen, wie sie sich für den Rekonfigurationsalgorithmus darstellen.

IF ← Accessor → Portal-Basisklasse / rekonf.Modul

In ReChannel-v1 sorgt der Accessor neben seiner Hauptaufgabe u. a. auch dafür, dass externe Zugriffe im rekonfigurierbaren Modul gezählt werden und dass das Portal über Zustandsänderungen informiert wird (siehe Abschnitt 1.5.1). Der Accessor hat daher direkten Zugriff auf das Portal, auf das rekonfigurierbare Modul sowie auf das Interface des statischen Channels.

Da in ReChannel-v1 aus diesem Grund eine Wiederverwendung des Accessors für den Einsatz im Exportal nicht möglich ist (vgl. Abschnitt 2.1.3), müsste für jeden Channel zusätzlich ein „Exportal-Accessor“ definiert werden. Für benutzerdefinierte Channels hätte der Anwender somit den doppelten Aufwand, obwohl in einem einzigen Accessor bereits alle zur Weiterleitung von IMCs benötigten Informationen enthalten sind. Um zu vermeiden, dass grundsätzlich sämtliche Funktionalität mehrmals vorhanden ist, muss eine einzelne Accessordefinition wiederverwendet werden können. Dies bedeutet, dass der Accessor unabhängig von dem ihn umgebenden Kontext (Portal, rekonf. Modul) verwendbar sein muss.

Um den Accessor als allgemeine Weiterleitungskomponente verwenden zu können, darf dieser nur über Wissen bzw. Funktionalitäten verfügen, die mit der Weiterleitung von IMCs zu tun haben. Um dies in ReChannel-v2 zu erreichen, soll das Konzept des Accessors von anderweitigen Aufgaben befreit werden. Zwischen die Kommunikation von Accessor und Channel wird hierfür eine Zwischenschicht eingefügt, wie in Abbildung 4.5 skizziert.

Diese Zwischenschicht muss durch ein ABI repräsentiert werden, das selbst nicht die Interfacemethoden implementiert. Dies ist unbedingt erforderlich, um eine generische Implementation dieser Zwischenschicht zu ermöglichen. Andernfalls wäre hierfür erneut eine Definition aller Interfacemethoden durch den Anwender notwendig.

Als Schnittstelle für die Zwischenschicht wird bewusst nicht das Switch-ABI gewählt.

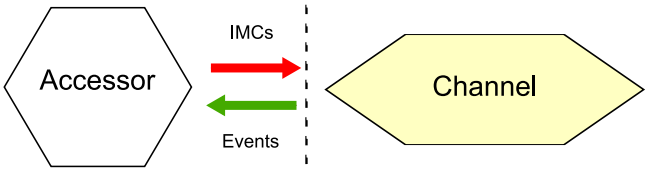

Abbildung 4.5: In ReChannel-v2 soll eine allgemeine Schnittstelle für die Kommunikation zwischen Accessor und Channel konzipiert werden, um die Accessor-Komponenten in beliebigem Kontext verwenden zu können.

Der Switch ist das Konzept eines allgemeinen Schalters, der vom Rekonfigurationsalgorithmus gesteuert wird. Daher hätte für den Accessor eine Abhängigkeit vom Switch wiederum die Folge, an einen bestimmten Verwendungszweck gebunden zu sein. Zudem würde dies eine erneute Abhängigkeit zwischen Rekonfigurationsalgorithmus und der verwendeten Kommunikationsform (Accessor) herstellen.

Aus den genannten Gründen wird das in Abschnitt 4.7 beschriebene ABI des **Interface-Wrappers** konzipiert, um das Kommunikationsziel eines Accessors zu repräsentieren.

Portal → Accessor-Implementation

In ReChannel-v1 ist der Typ des zu einem Portal gehörigen Accessors durch den Porttyp des Portals festgelegt. Das generische Portal ist somit direkt von einer ganz bestimmten Accessor-Implementation abhängig. Eine Variation der Accessor-Implementation ist nicht möglich. Daher ist in dieser Situation die Erzeugung der Accessor-Komponenten nach dem Factory-Entwurfsmuster vorzuziehen.

In ReChannel-v2 soll eine Switch-Implementation nur das Wissen über ein allgemeines **Accessor-ABI** besitzen und das konkrete Objekt der Accessor-Implementation von einer Factory-Klasse oder Factory-Methode erhalten. Einerseits entstehen dadurch mehr Freiheiten bei der Implementation beider Komponenten und andererseits kann durch eine solche Vorgehensweise redundanter Templatecode vermieden werden.

Pager → Eventfinder

Der Pager ist die Klasse in ReChannel-v1, die innerhalb des Portals als modulare Komponente für die Weiterleitung von Events eingesetzt wird. Eine Verwendung dieses Pagers ist im Exportal nicht ohne weiteres möglich, da die Funktionalität des Pagers auf Eventfinder ausgerichtet ist. Eventfinder werden in SystemC ausschließlich im Zusammenhang mit Ports verwendet. Wegen der speziellen Ausrichtung des Pagers auf die Anforderungen des Portals, wird der Pager in ReChannel-v2 durch eine allgemeinere Komponente (`rc_event_forwarder`) mit der Bezeichnung **Event-Forwarder** ersetzt. Der Event-Forwarder löst auch die Abhängigkeit mit dem Accessor, da anstelle dessen nur SystemC-Interfaces verwendet werden.

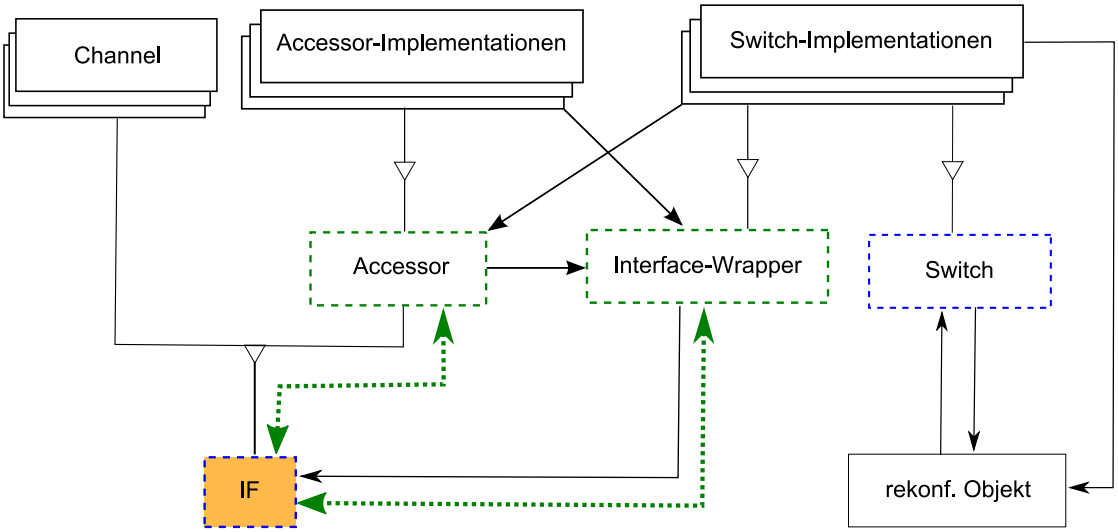

Abbildung 4.6: Die neue Sichtweise der Klassenhierarchie, wie sie sich in ReChannel-v2 durch die Verwendung der Konzepte von Switch, Interface-Wrapper und Accessor darstellt (schematisch)

Vereinigung der Konzepte

Die Sichtweise der Klassenhierarchie hat sich in ReChannel-v2 gegenüber ReChannel-v1 entscheidend gewandelt (siehe Abbildung 4.6 im Vergleich zu Abbildung 4.1). Die festen Abhängigkeiten zwischen Portal, Accessor und Modul wurden dadurch aufgelöst, dass der Kontrollmechanismus von ReChannel nun auf ABIs basiert. Durch die Verwendung des Konzepts des Switches (bzw. des Switch-ABIs) sind vielfältige Switch- und Accessor-Implementationen möglich, die mit dem Rekonfigurationsalgorithmus verwendet werden können und zudem voneinander unabhängig sind. Die einzigen vorgeschriebenen Abhängigkeiten bestehen lediglich zwischen dem Switch-ABI und der Klasse der rekonfigurierbaren Objekte sowie zwischen dem Interface-Wrapper-ABI und dem dazugehörigen Accessor-ABI.

Eine Switch-Implementation vereinigt das Switch- und das Interface-Wrapper-ABI in sich. Accessor- und Switch-Implementationen können somit im Rahmen der durch die ABIs spezifizierten Verwendungsregeln sowie unter der Voraussetzung eines kompatiblen Interfacetyps (IF) beliebig miteinander kombiniert werden.

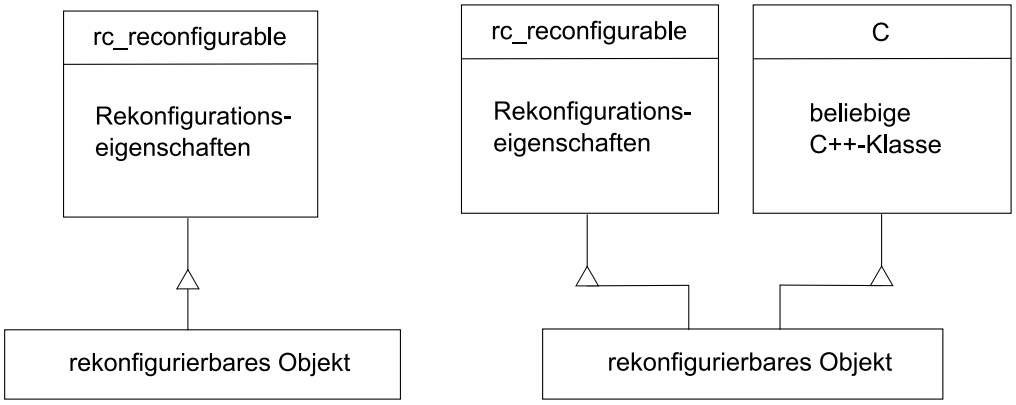

Abbildung 4.7: Jedes beliebige Objekt, das sich von `rc_reconfigurable` ableitet, besitzt aus Sicht von ReChannel die Eigenschaft „rekonfigurierbar“ und kann daher zusammen mit dem Rekonfigurationsalgorithmus von ReChannel verwendet werden.

4.3 Rekonfigurierbare Objekte

Rekonfigurierbares Objekte (RO) werden durch die Klasse `rc_reconfigurable` repräsentiert. Diese ist das Re-Design der Klasse `rc_module` aus ReChannel-v1 und nimmt daher in ReChannel-v2 deren Platz ein. Der Name „rc_module“ wird einerseits deshalb nicht mehr verwendet, da dieses Objekt nicht von `sc_module` abgeleitet ist und daher kein Modul ist, und andererseits ist der Name `rc_module` für einen anderen Verwendungszweck vorgesehen wird (siehe Abschnitt 4.12).

Jedes beliebige Objekt, das sich von `rc_reconfigurable` ableitet, besitzt aus Sicht von ReChannel die Eigenschaft „rekonfigurierbar“ (siehe Abbildung 4.7). Die Klasse `rc_reconfigurable` ist kein ABI, da sie sämtliche Rekonfigurationseigenschaften eines RO beinhalten muss.

Da `rc_reconfigurable`, im Gegensatz zu `rc_module` aus ReChannel-v1, nicht zwingend die Ableitung von einem Modul erfordert, können nun auch beliebige C++- und SystemC-Objekte (wie z. B. primitive Channels) mit dem Rekonfigurationsalgorithmus von ReChannel verwendet werden. Ein weiterer Vorteil in der Bedienung ergibt sich dadurch, dass die Notwendigkeit zum Aufruf der Methode `rc_init()` entfällt (vgl. Abschnitt 1.5.5).

Obwohl ein rekonfigurierbares Modul direkt mittels manueller Ableitung von einem SystemC-Modul und `rc_reconfigurable` erzeugt werden kann, wird für diesen Zweck die Verwendung der Klasse `rc_reconfigurable_module` (siehe Abschnitt 4.10) empfohlen. Diese nimmt die Ableitung von `rc_reconfigurable` automatisch vor. Die Verwendung von `rc_reconfigurable_module` ist nicht nur deutlich komfortabler, sondern auch die Voraussetzung für weiterführende Funktionalitäten (vgl. Abschnitt **??**).

Entsprechend der Simulationssemantik kann ein rekonfigurierbares Objekt entweder „ungeladen“, „inaktiv“ oder „aktiv“ sein. In der Implementation sind die entsprechenden

Zustände mit `UNLOADED`, `INACTIVE` und `ACTIVE` bezeichnet. Definiert sind sie in einem **enum**-Aufzählungstyp namens `rc_reconfigurable::state_type`. Die Konstanten werden im Folgenden auch als die **Zustandskonstanten** des RO bezeichnet.

Die Klasse `rc_reconfigurable` kapselt sämtliche Funktionalität, die benötigt wird, um die Lade-, Entlade-, Aktivierungs- und Deaktivierungsvorgänge eines RO zu simulieren. Diese vier Aktionstypen werden ebenfalls jeweils durch eine Konstante repräsentiert. Sie sind im Aufzählungstyp `rc_reconfigurable::action_type` definiert und heißen `LOAD`, `ACTIVATE`, `DEACTIVATE` und `UNLOAD`. Im Folgenden werden diese als **Aktionskonstanten** bezeichnet.

Der nebenläufige Automat zur Modellierung von Rekonfigurationszuständen und alle damit verbundenen Klassen (wie z. B. die sieben Zustands-Klassen) sind ersatzlos weggefallen. Anstelle dessen wird die in Abschnitt 2.1.8 beschriebene **hierarchische Zustandsmodellierung** verwendet. Die sich daraus ergebenden Vorteile sind:

- kein Initialisierungsaufruf durch den Anwender erforderlich
- keine interne Nebenläufigkeit bzw. Parallelität
- eine verringerte Zustandsanzahl und daher einfachere Zustandsüberprüfungen
- Lokalität von Code (Wartbarkeit, Übersichtlichkeit)
- kein zusätzlicher Thread-Prozess / keine interne Modulstruktur

In ReChannel-v1 wurde die Anzahl der momentan bestehenden externen Zugriffe in einer Variablen namens `pending_accesses` gezählt. Ein Wert dieser Variablen größer Null verhinderte, dass ein rekonfigurierbares Modul deaktiviert werden kann. In ReChannel-v2 gibt es für diesen Zähler eine Entsprechung in Form einer **Transaktionszählervariable**. Eine solche befindet sich in jedem rekonfigurierbaren Objekt.

Die Transaktionszählervariable in ReChannel-v2 ist eine Verallgemeinerung des Zählers aus ReChannel-v1 auf beliebige Transaktionen, die vor der Deaktivierung eines RO beendet sein müssen. Hierbei sind nun externe wie auch interne Transaktionen mit eingeschlossen. Diese neue Sichtweise dient dem Zweck, dass nicht nur der Rekonfigurationsalgorithmus sondern nun auch beliebige andere Mechanismen die Transaktionszählervariable verwenden können, um den Deaktivierungszeitpunkt eines RO zu beeinflussen. Auf die Variable wird von außen nicht direkt, sondern über Methoden zugriffen, um dem RO die Möglichkeit zu geben, auf Änderungen des Zählers reagieren zu können.

Die Zeit, die ein RO für einen Rekonfigurationsvorgang in Anspruch nehmen soll, wird mittels der Methode `rc_set_delay()` zugewiesen. Als Parameter wird dieser Methode die entsprechende Aktionskonstante und Simulationszeit übergeben. Weiherhin steht eine Methode `rc_set_default_delay()` zur Verfügung, da zwischen der momentan gesetzten Zeit und den voreingestellten Standardwerten für diese Zeiten unterschieden wird. Die Unterscheidung zwischen aktuellen und voreingestellten Zeiten wird eingeführt, da die Rekonfigurationszeiten eines RO auch von einer benutzerdefinierten Rekonfigurationskontrolle modifiziert werden können. Die Rekonstruktion der ursprünglichen Zeiten soll nachträglich noch möglich sein.

Die Klasse `rc_reconfigurable` besitzt allgemeine, öffentliche Informationsmethoden zur Abfrage des aktuellen Zustands und des SystemC-Namens sowie SystemC-Objekts (falls ein RO von `sc_object` abgeleitet ist). Auch gibt es Methoden, die Auskunft über den Status eines aktuell stattfindenden Rekonfigurationsvorgangs geben.

Die Kontrollklasse `rc_control` hat als einzige Klasse direkten Zugriff auf die Kontrollmethoden von `rc_reconfigurable`. Die Klasse des RO ist auf der anderen Seite die einzige Klasse, die von außen Zugriff auf die Kontrollmethoden der Switches hat. Die Funktionsweise des Rekonfigurationsalgorithmus wird in Abschnitt 4.5 beschrieben.

Verschiedene weitere Aspekte der in der Klasse `rc_reconfigurable` enthaltenen Funktionalität werden später themenbezogen in den folgenden Abschnitten des Kapitels geschildert.

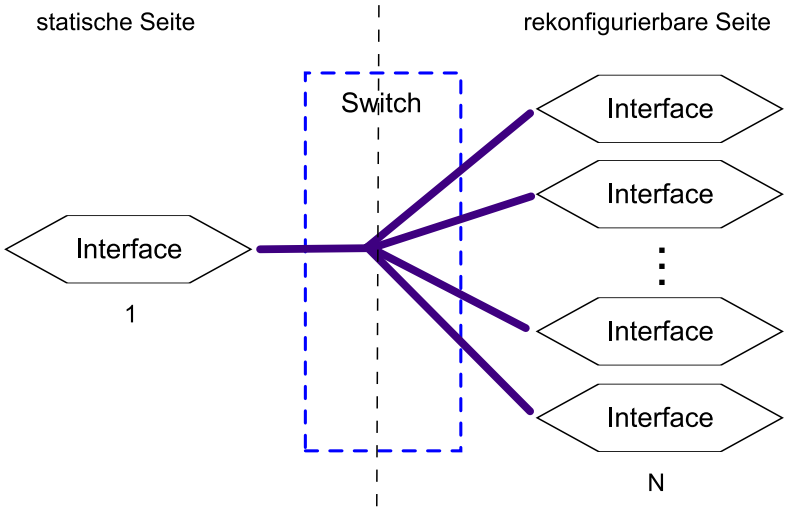

Abbildung 4.8: Schematische Darstellung des Swich-ABIs. Switch-Komponenten werden in der Simulation verwendet, um den statischen Designteil mit einem rekonfigurierbaren Bereich zu verbinden und – getreu der Simulationssemantik von ReChannel – sämtliche Kommunikation zwischen diesen kontrollieren zu können. Das Konzept des Switches ist eine Verallgemeinerung des Multiplexer-/Demultiplexer-Prinzips für die Kommunikation zwischen beliebigen SystemC-Interfaces.

4.4 Der Switch

Der Switch ist die Abstraktion eines Schalters, der zur Verbindung von rekonfigurierbaren Bereichen mit dem statischen Designteil verwendet wird. Er stellt die Verallgemeinerung des Multiplexer-/Demultiplexer-Prinzips für Kommunikationsverbindungen zwischen beliebigen SystemC-Interfacetypen dar. Das Konzept des Switches wird durch das ABI `rc_switch` repräsentiert. Portal und Exportal sind definitionsgemäß Switches. Beide implementieren daher das ABI `rc_switch`.

Das ABI enthält allgemeine Kontrollmethoden, so dass die Verwendung aller möglichen Switches aus Sicht des Rekonfigurationsalgorithmus auf exakt dieselbe Weise funktioniert. Kenntnisse über den Unterschied von Portals, Exportals oder anderen Switch-Implementationen sind nicht erforderlich. Dem Kontrollmechanismus von ReChannel bzw. den rekonfigurierbaren Objekten sind nur Komponenten vom Typ `rc_switch` bekannt. Dies führt zu einer Entkopplung der Beziehungen zwischen den Klassen und lässt viele Freiheiten bei der Implementation eines Switches.

Durch das ABI `rc_switch` werden zusätzlich zur Methodenschnittstelle auch einige Verwendungsregeln spezifiziert. Im Folgenden wird beschrieben, was die Eigenschaften eines Rekonfigurationsschalters sind und welches Schalterverhalten eine Switch-Implementation aufweisen muss, um sinnvoll bei der Simulation von DR verwendet werden zu können.

4.4.1 Spezifikation des Switches

Jeder Switch ist mit genau einem Interface auf der statischen Seite verbunden sowie mit einer beliebigen Anzahl $N >= 0$ von Interfaces auf der rekonfigurierbaren Seite (siehe Abbildung 4.8). Die N Interfaces auf der rekonfigurierbaren Seite sind jeweils genau einem RO zugeordnet, von dem das Interface stammt. Alle diese N RO-Interface-Paare werden mittels speziell dafür vorgesehener Registrierungsmethoden beim Switch angemeldet. Ein nicht näher spezifizierter, äußerer oder innerer Mechanismus hat dafür zu sorgen, dass sich Switch und RO gegenseitig beieinander registrieren.

Für jede Switch-Implementation muss eine Kommunikationsrichtung festgelegt werden. Diese Richtung bestimmt, ob IMCs und Events von der statischen auf die rekonfigurierbare oder umgekehrt weitergleitet werden. Diese Richtung ist aber nicht zu verwechseln mit der durch Channels simulierten Kommunikationsrichtung, wie es z. B. bei In-, Out- oder In/Out-Ports der Fall ist. Ein Switch ist unabhängig von der Semantik der durch ihn laufenden Kommunikation, da für ihn lediglich die technischen Aspekte der Weiterleitung von IMCs und Events von Bedeutung sind.

Die Kommunikationsrichtung eines Switches ist nicht durch das ABI vorgegeben, sondern wird erst durch dessen jeweilige Implementation bestimmt. Die Vermittlung der Kommunikation zwischen den Interfaces kann somit analog zu einem Demultiplexer $(1 : N)$ oder einem Multiplexer $(N : 1)$ erfolgen. Der Aufbau einer Kommunikationsverbindung unterliegt dabei den folgenden zwei Regeln: Erstens darf ein Switch zur gleichen Zeit maximal für eine $1 : 1$ Kommunikationsverbindung geöffnet sein. Zweitens muss das Interface auf der rekonfigurierbaren Seite einem rekonfigurierbaren Objekt zugeordnet sein, das sich aktuell im aktiven Zustand befindet.

Durch das ABI wird ebenfalls nicht spezifiziert, welche Objekte dem Interface zu Grunde liegen oder wie diese erzeugt werden sowie auch nicht, über welchen Mechanismus die Kommunikation kontrolliert bzw. weitergeleitet wird. Somit können je nach Aufgabe des Switches Accessoren, Channels oder sonstige Objekte zum Einsatz kommen. Die einzige Voraussetzung ist, dass diese Objekte über ein SystemC-Interface verfügen (d.h. von `sc_interface` abgeleitet sind). Eine Switch-Implementation ist somit allein verantwortlich für Typüberprüfungen, die Durchführung von Bindungen und die Kontrolle der Kommunikation. Einer Implementation ist es weiterhin erlaubt, beliebige Verwendungsregeln und Vorgaben zusätzlich zu den durch das ABI vorgeschriebenen aufzustellen, solange diese sich nicht mit jenen widersprechen.

Schalterzustände

Da es sich um einen Schalter handelt, kann sich der Switch im geöffneten Zustand (`OPEN`) oder im geschlossenen Zustand (`CLOSED`) befinden. Ein Switch im Zustand `OPEN` ist für genau ein aktives rekonfigurierbares Objekt und dessen zugehöriges Interface geöffnet, sodass eine direkte Kommunikationsverbindung zu dem Interface der statischen Seite besteht. Im Zustand `CLOSED` ist keine Kommunikation zwischen statischer und rekonfigurierbarer Seite erlaubt. Sämtliche Kommunikation muss daher vom Switch abgeblockt werden. Zur Umsetzung der Simulationssemantik wird ein Switch vom Kontrollmecha-

Switch-Methode	Beschreibung
register_reconfigurable()	Registriert ein RO-Interface-Paar beim Switch
unregister_reconfigurable()	Entfernt die Registrierung eines RO-Interface-Paares
get_registered_interface()	Liefert das zu einem registrierten RO gehörige Interface
is_registered()	Erfragt, ob ein RO bzw. Interface registriert ist
open(RO)	Öffnet den Switch für ein zuvor registriertes RO
open()	Öffnet den Switch (einfache Schalteroperation)
close()	Schließt den Switch (einfache Schalteroperation)
set_undefined()	Setzt den Switch direkt auf den Zustand UNDEF

Tabelle 4.1: Die Registrierungs- und Kontrollmethoden des Switch-ABIs.

nismus entsprechend der Rekonfigurationszustände der RO geöffnet oder geschlossen.

Zusätlich gibt es einen dritten Zustand `UNDEF`. Wie im Zustand `CLOSED` ist die Weiterleitung der Kommunikation in diesem Zustand untersagt. Aber im Unterschied zu `CLOSED` entspricht `UNDEF` einem undefinierten Verhalten der Kommunikationsverbindung. Worin konkret der Unterschied zwischen „undefiniert“ und „geschlossen“ besteht, wird von der jeweiligen Switch-Implementation bestimmt.

Wird von außen versucht, einen Switch zu öffnen, ohne dass ein Interface verfügbar ist oder dass ein verbundenes RO aktiv ist, so soll der Switch in den Zustand `UNDEF` zurückfallen. In der Konstruktionsphase ist es meist sinnvoll, wenn sich der Switch kurz nach seiner Erzeugung (und daher noch unverbunden) standardmäßig im Zustand `UNDEF` befindet. Während der Simulation ist mit diesem Zustand nur dann zu rechnen, wenn ein Switch direkt vom Anwender angesteuert wird. Einer Switch-Implementation liegt es frei zu entscheiden, ob eine derartige Möglichkeit zu einer direkten Kontrolle bestehen soll.

Mittels der Switch-Komponenten wird die Ortsbindung von RO modelliert und gewährleistet. Zur Vereinfachung der Simulationssemantik und des Kontrollmechanismus sind in ReChannel-v2 zwei aktive RO an einem Ort (Switch) grundsätzlich nicht zulässig. Eine solche Situation kann entstehen, wenn ein Switch sich für ein RO öffnen soll, obwohl bereits eine aktive Kommunikationsverbindung zu einem anderen RO besteht. Der Switch hat in einem solchen Konfliktfall unmittelbar einen SystemC-Error zu melden, der zum Abbruch der Simulation führt.

4.4.2 Methoden des Switch-ABIs

Das Switch-ABI (siehe Anhang B.1) besitzt Methoden mit zwei unterschiedlichen Zugriffsrechten.

Zum einen gibt es die als `public` deklarierten, öffentlichen Methoden. Hierbei handelt es sich um Informationsmethoden, die die allgemeinen Eigenschaften des Switches wiederspiegeln. Es können Name, Typ und Zustand des Switches sowie das verbundene statische oder das momentan aktive, dynamische Interface abgefragt werden.

Zum anderen gibt es die als `protected` deklarierten, vor äußerem Zugriff geschützten Methoden. Diese dienen der Kontrolle des Schalterverhaltens sowie der Registrierung von rekonfigurierbaren Objekten und Interfaces. Sie sind nichtöffentlich, da diese Methoden lediglich vom rechanneleigenen Kontrollmechanismus sowie vom jeweiligen Switch selbst

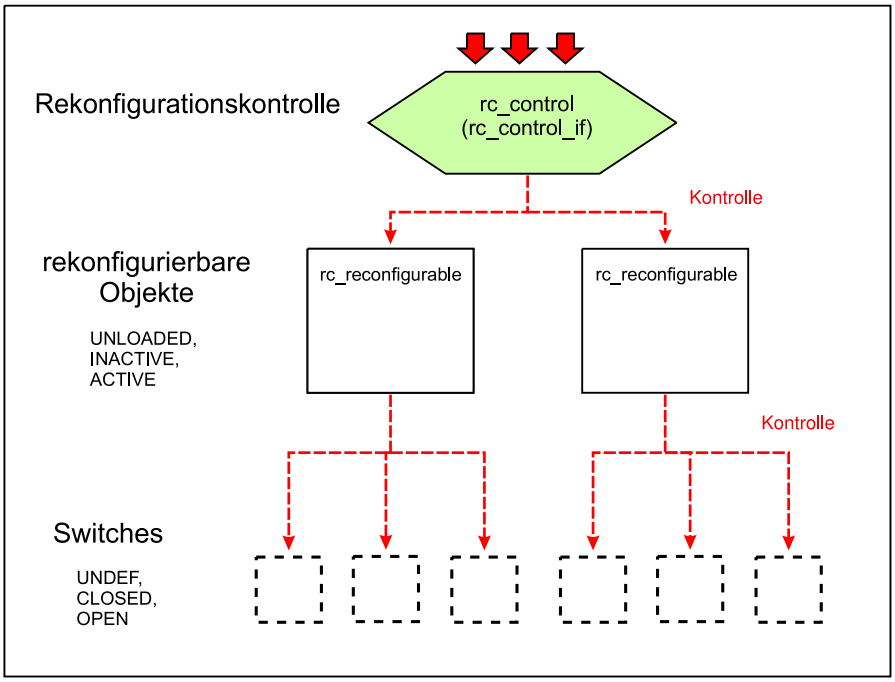

Abbildung 4.9: Die Hierarchie des Kontrollmechanismus von ReChannel-v2 zur Durchführung der Rekonfigurationsvorgänge

verwendet werden sollen. Die Methoden zur Steuerung des Schalters und der Registrierung der RO-Interface-Paare sind in Tabelle 4.1 aufgelistet.

Zusätzlich zu den durch das ABI vorgeschriebenen Methoden kann eine Switch-Implementation beliebige Methoden, Operatoren oder Datenelemente besitzen.

4.5 Der Kontrollmechanismus

Der Kontrollmechanismus in ReChannel-v2 ist in Abbildung 4.9 skizziert. In ReChannel-v2 werden anstelle von rekonfigurierbaren Modulen rekonfigurierbare Objekte verwendet. Diese Abstraktion ermöglicht eine allgemeinere Verwendung des Rekonfigurationsalgorithmus, wie z. B. für die Rekonfiguration von primitiven Channels. Ein weiterer Unterschied zu ReChannel-v1 ist, dass die rekonfigurierbaren Objekte (RO) nicht mit einem Accessor kommunizieren, sondern direkt mit dem Switch. Wenn sich der Rekonfigurationszustand eines RO ändert, dann sorgt es dafür, dass die bei ihm registrierten Switches ihren Schalterzustand entsprechend ändern. Wenn das RO aktiviert wird, werden alle Switches geöffnet, wenn es deaktiviert wird, werden diese geschlossen. Da ein RO mit dem Switch ausschließlich über dessen Switch-ABI (siehe Abschnitt 4.4) kommuniziert, benötigt es keine Kenntnis darüber, ob es ein Portal oder ein Exportal vor sich hat.

Einer der Hauptunterschiede zum Kontrollmechanismus von ReChannel-v1 (siehe Abbildung 2.2) ist, dass die rekonfigurierbaren Objekte keinen Hardware-Automaten mehr

beherbergen. Der gesamte Rekonfigurationsvorgang liegt somit im Ausführungspfad des Kontrollprozesses. Dies bringt Vorteile für den Rekonfigurationsalgorithmus mit sich, da dieser dadurch die volle Kontrolle über die Ausführung erhält. Da keine Automatenzustandsübergänge durch einen anderen Prozess ausgeführt werden müssen, kommt der Algorithmus ohne Nebenläufigkeit und den Verbrauch zusätzlicher Delta-Zyklen aus. Das Verhalten des Algorithmus kann dadurch exakter definiert werden und ist somit robuster.

4.5.1 Rekonfigurationskontrolle

Die Rekonfigurationskontrolle `rc_control` wird in ReChannel-v2 als Channel höherer Abstraktionsebene angesehen und besitzt daher ein SystemC-Interface namens `rc_control_if` mit sämtlichen Kontrollmethoden. Der einzige verfügbare Zugriffstyp für diese Kontrollmethoden ist „blockierend“, so dass alle Kontrollprozesse Thread-Prozesse sein müssen. Auf höheren Abstraktionsebenen stellt die Forderung nach Thread-Prozessen kein Problem dar, da hier hauptsächlich dieser Prozesstyp verwendet wird. Auf hardwarenahen Abstraktionsebenen wird hingegen meistens ausschließlich mit Method-Prozessen modelliert. Doch ist davon auszugehen, dass auf diesen Ebenen eine explizite Modellierung von benutzerdefinierten Hardware-Rekonfigurationskontrollen vorgenommen wird (siehe [19]). In solchen, vom Anwender definierten Komponenten können intern ein oder mehrere Thread-Prozesse arbeiten, die die Rekonfiguration durchführen. Die direkte Ansteuerung durch Method-Prozesse ist somit auch hier nicht erforderlich.

In `rc_control` gibt es weder nichtblockierende noch mit `force_deactivate()` vergleichbare Kontrollmethoden, da diese zu den in Kapitel 2.1.1 festgestellten Problemen führten. Nichtblockierende Kontrollmethoden haben sich als problematisch bei Zugriffen durch konkurrierende Kontrollprozesse erwiesen, wenn diese auf dieselben RO-Exemplare zugreifen. Zudem erfordern diese die Existenz eines endlichen Automaten. Durch den Verzicht auf nichtblockierende Kontrollmethoden kann somit der vormals für jedes Modul notwendige Thread-Prozess eingespart werden.

Die Anzahl an Thread-Prozessen, die für die Kontrolle der Rekonfiguration in einem Design maximal verwendet werden müssen, ist somit proportional zur Anzahl der rekonfigurierbaren Bereiche und nicht mehr wie in ReChannel-v1 proportional zur Anzahl der rekonfigurierbaren Module (vgl. Kapitel 2.1.8). Die Entscheidung über die Anzahl der in einem Design vorhandenen Kontrollprozesse liegt beim Anwender. Je nach Anwendung kann diese Zahl verschieden sein. Bei vielen heute auf dem Markt befindlichen dynamisch rekonfigurierbaren FPGAs reicht z. B. bereits eine Rekonfigurationskontrolle mit einem Kontrollprozess, um dessen Fähigkeiten korrekt modellieren zu können.

In ReChannel-v2 kann ein Kontrollprozess die von `rc_control` bereitgestellte **Lock-Funktionalität** nutzen, um einen exklusiven Zugriff auf beliebige RO zu reservieren. Während ein Kontrollprozess einen Rekonfigurationsvorgang für ein RO durchführt, so kann man sicher sein, dass kein fremder Prozess eine gegenteilige Rekonfigurationsanweisung geben kann. Auch der Prozess selbst kann keine widersprüchlichen Befehle geben, da dieser immer direkt mit der Durchführung des Vorgangs beschäftigt ist.

```
// Verwaltungsmethoden
virtual void add(const rc_reconfigurable_set& rs) = 0;
virtual void remove(const rc_reconfigurable_set& rs) = 0;
virtual bool has_control(const rc_reconfigurable_set& rs) const = 0;

// Kontrollmethoden für die Rekonfigurationsvorgänge
virtual void load(const rc_reconfigurable_set& rs) = 0;
virtual void activate(const rc_reconfigurable_set& rs) = 0;
virtual void deactivate(const rc_reconfigurable_set& rs) = 0;
virtual void unload(const rc_reconfigurable_set& rs) = 0;

// Methoden für den Lock von rekonfigurierbaren Objekten
virtual void lock(const rc_reconfigurable_set& rs) = 0;
virtual bool trylock(const rc_reconfigurable_set& rs) = 0;
virtual void unlock(const rc_reconfigurable_set& rs) = 0;
virtual bool is_locked(const rc_reconfigurable_set& rs) const = 0;
```

Listing 4.1: Die Methoden des Kontrollinterfaces `rc_control_if`

4.5.2 Benutzerschnittstelle

Da in ReChannel die Rekonfigurationskontrolle nach dem Fassaden-Entwurfsmuster (Facade Pattern) implementiert ist, ist die Komponente `rc_control` die Benutzerschnittstelle für den gesamten dahinter liegenden Kontrollmechanismus. Die Klasse `rc_control` kann in ReChannel vom Anwender überladen werden, um benutzerdefinierte Rekonfigurationskontrollen zu erzeugen.

In ReChannel-v1 besaß `rc_control` mehrere Varianten einer Kontrollmethode. Es gab für jeden Rekonfigurationsvorgang sowohl blockierende als auch nichtblockierende Zugriffsmethoden. Von diesen lagen die meisten in zwei bis drei Varianten vor, um Parameter vom Typ `rc_module_set`, `rc_module` sowie `const char*` (Namensauflösung) zu akzeptieren.

Um den Aufwand für die Implementation benutzerdefinierter Rekonfigurationskontrollen zu minimieren, wird in ReChannel-v2 die Anzahl und Komplexität der Interfacemethoden von `rc_control_if` so gering wie möglich gehalten. Alle Interfacemethoden besitzen nur noch einen Parameter vom Typ `rc_reconfigurable_set`, der eine Menge von RO darstellt. Auch einzelne RO können diesen Methoden übergeben werden, da die Klasse `rc_reconfigurable` einen Cast-Operator für eine solche Menge besitzt. Ein RO kann somit implizit wie eine einelementige Menge verwendet werden.

Auf Kontrollmethoden, denen ein Namensparameter übergeben werden kann (vgl. Abschnitt 2.2), wird verzichtet, da es in der neuen SystemC-Version bzw. im SystemC-Standard eine Funktion namens `sc_find_name()` gibt, die für das Auffinden von Objekten in einem Design verwendet werden kann. Eine rechanneleigene Modulregistrierung zur Namensauflösung wie in ReChannel-v1 wird daher nicht mehr verwendet.

Es gibt nach wie vor die Methoden `load()`, `activate()`, `deactivate()` und `unload()`, um Rekonfigurationsvorgänge zu initiieren. Mit `add()` werden RO der Kontrolle zugewie-

sen, mit `remove()` können diese wieder entfernt werden. `remove()` entfernt zudem alle bestehenden Locks auf die entfernten RO, damit das RO direkt für die Weiterverwendung in einer anderen Kontrollkomponente bereit ist und ein solcher Kontrollwechsel nicht zu Deadlocks führt.

Da vornehmlich auf höheren Abstraktionsebenen eine direkte Verwendung der Benutzerschnittstelle durch einen Anwenderprozess zu erwarten ist, ist das Verhalten der Methoden in Bezug auf Fehlermeldungen auf diesen Anwendungsfall hin optimiert. Im Gegensatz zu ReChannel-v1 wird in ReChannel-v2 davon ausgegangen, dass von dem Befehl „activate()“ erwartet wird, dass auch ungeladene RO in den Endzustand `ACTIVE` überführt werden können. Daher werden alle Zwischenzustandsübergänge, wie z. B. in diesem Fall „load()“, implizit ausgeführt. Der implizite Aufruf von `load()` dient somit dazu, dem Anwender die Verwendung auf abstrakten Beschreibungsebenen zu vereinfachen. Sollte eine Fehlermeldung erforderlich sein – z. B. im Fall eines ungeladenen Moduls –, so kann der Anwender dies vor dem Aufruf überprüfen. Wenn eine benutzerdefinierte Kontrollkomponente verwendet wird, könnten solche Überprüfungen dort gekapselt werden. Hier kann dann ein Befehl z. B. auch in mehrere aufgespalten werden, falls die impliziten Übergänge nicht erwünscht sind.

Gestaffelte Zugriffsberechtigungen

Da die Kontrollkomponente wie ein Channel verwendet werden kann, können Ports und Exports dazu genutzt werden, um Kontrollaufgaben auf modulare Weise in einem Design zur Verfügung zu stellen. Um einem Designer hierbei die Modellierung einer gestaffelten Zugriffsberechtigung auf die Kontrolle zu ermöglichen, ist `rc_control_if` in zwei Unterinterfaces aufgeteilt.

`rc_control_elab_if` ist das Interface, das die Methode `add()` enthält und in der Konstruktionsphase für die globale Registrierung von RO in einem Design verwendet werden kann. Das zweite Unterinterface ist `rc_control_sim_if`, welches nur die Methoden enthält, die während der Simulation benötigt werden (wie z. B. `activate()`, `lock()`, etc.). Die Methode `remove()` wird nur im Hauptinterface deklariert, so dass für das Entfernen eines RO aus der Kontrolle der Vollzugriff auf die Kontrollkomponente erforderlich ist.

4.5.3 Paralleler Zugriff durch mehrere Kontrollprozesse

Der Aufruf einer Kontrollmethode zur Durchführung eines Rekonfigurationsvorgangs ist immer atomar. Kein anderer Kontrollprozess kann einen gestarteten Rekonfigurationsvorgang unterbrechen, da implizit die Lock-Funktionalität verwendet wird. Ein Kontrollprozess kann bei `rc_control` den Zugriff auf RO auch explizit reservieren. Das explizite Reservieren von RO ist immer dann erforderlich, wenn ein Prozess mehrere zusammengehörige Rekonfigurationsvorgänge ausführen möchte. Hierzu stehen die Methoden `lock()`, `trylock()`, `unlock()` zur Verfügung. Sie funktionieren wie ein Mutex auf Prozessbasis, d.h. ein Lock auf ein RO ist immer exklusiv an ein bestimmtes Prozess-Exemplar gebunden. Daher können zwei Prozesse nicht gleichzeitig einen Lock auf ein RO besitzen. Wenn ein Prozess `lock()` für ein bereits von einem anderen Prozess reserviertes RO aufruft,

blockiert der Aufruf solange, bis der Lock erlangt werden kann. Ein Aufruf der Methode `trylock()` hingegen ist nichtblockierend und liefert nur einen `bool`-Wert zurück, ob der Lock erfolgreich erlangt werden konnte oder nicht.

Hinter der Lock-Funktionalität steht ein Mutex-Objekt der Klasse `rc_mutex_object`, von dem sich jeweils eins in jedem RO befindet. Dieses Mutex-Objekt hat die Besonderheit, dass es einem Prozess ermöglicht, auch mehr als einen Lock zu besitzen. Ein erneuter Aufruf von `lock()` wird somit als zusätzlicher Lock gezählt. Mit `unlock()` wird jeder angelegte Lock einzeln wieder freigegeben.

Dieses spezielle Verhalten des Mutex dient dazu, Paare von Lock-Unlock-Aufrufen beliebig schachteln zu können (z. B. beim Aufruf von Hilfsfunktionen). Dies ermöglicht eine flexiblere Verwendung der Lock-Funktionalität, als es mit einem gewöhnlichen Mutex der Fall wäre.

Die Lock-Methoden reservieren die ihnen übergebenen RO in der Reihenfolge, wie sie in der Menge `rc_reconfigurable_set` vorliegen. Da `rc_reconfigurable_set` eine nach Objektpointern geordnete Menge ist, können bei vorschriftsmäßiger Verwendung der Lock-Methoden keine Deadlocks auftreten. Rufen zwei Kontrollprozesse $P1$ und $P2$ parallel die Methode `lock()` auf, kann es aufgrund der sortierten Reihenfolge nicht vorkommen, dass $P1$ und $P2$ über Kreuz auf ein RO warten, welches der jeweils andere Prozess durch seinen Aufruf reserviert hat. Wenn alle Kontrollprozesse diejenigen RO, die sie gleichzeitig verwenden möchten, immer durch einen einzigen Aufruf einer Lock-Methode reservieren, kann es somit nicht zu einem gegenseitigen Deadlock kommen.

4.5.4 Der Rekonfigurationsvorgang

Ein Rekonfigurationsvorgang für ein oder mehrere RO beginnt mit dem Aufruf einer Kontrollmethode (`load()`, `activate()`, `deactivate()` oder `unload()`) von `rc_control` durch einen Kontrollprozess.

1. Zuerst wird der Lock für die RO erzeugt, indem `lock()` aufgerufen wird. Hierbei wird auch überprüft, ob die RO bei der Kontrolle registriert sind. Falls dies nicht der Fall sein sollte, wird ein Fehler gemeldet (nachdem alle Locks wieder freigegeben worden sind).

2. Auf welche Weise der Rekonfigurationsbefehl an die RO weitergegeben wird, hängt von der Anzahl der übergebenen RO ab:

   a) Falls ein Rekonfigurationsvorgang nur für ein RO ausgeführt werden soll, übernimmt der Kontrollprozess selbst die Ausführung, indem die private Methode `reconfigure()` des `rc_reconfigurable`-Objekts aufgerufen wird. Dieser Methode wird die dem Vorgang entsprechende Aktionskonstante übergeben.

   b) Falls der Vorgang für mehrere RO gleichzeitig ausgeführt werden soll, kann der auftraggebende Kontrollprozess dies nicht übernehmen. Hierzu wird für jedes RO (das sich noch nicht im gewünschten Zielzustand befindet) ein temporärer Thread-Prozess mittels `sc_spawn()` erzeugt. Durch die Verwendung von `sc_bind` wird deren Einsprungmethode auf die Methode `reconfigure()` des

entsprechenden `rc_reconfigurable`-Objekts gesetzt. Der Parameter mit der entsprechenden Aktionskonstante wird dabei an den Aufruf gebunden. Der Kontrollprozess wartet, bis alle erzeugten Prozesse terminiert sind, was der Beendigung aller Rekonfigurationsvorgänge entspricht.

Dass der Kontrollprozess dynamisch auf eine beliebige Menge von Terminierungsevents warten kann, wird dadurch erreicht, dass eine `sc_event_and_list` in einer `for`-Schleife durch wiederholte Anwendung des „`&`“-Operators erzeugt und anschließend in einer `wait()`-Funktion als Parameter verwendet wird. Die Erzeugung einer Event-Liste in einer Schleife ist laut SystemC-Standard erlaubt (vgl. [8]).

Den Kontrollmethoden sollten nur dann mehrere RO übergeben werden, wenn der Overhead durch die Erzeugung von temporären Prozessen im Verhältnis zur restlichen Simulationszeit vernachlässigt werden kann. Andernfalls sollten für die Simulation gleichzeitiger Vorgänge mehrere Kontrollprozesse auf Anwenderseite vorgesehen werden.

3. Nach erfolgter Durchführung werden die zu Anfang für die RO angelegten Locks wieder freigegeben.

Die Methode `reconfigure()` der `rc_reconfigurable`-Klasse führt den Rekonfigurationszustandswechsel des RO durch. Hierbei wird ebenfalls wieder überprüft, ob ein Prozess einen Lock besitzt. Andernfalls muss er solange warten, bis ein Lock frei wird. Die temporären Prozesse teilen sich den Lock mit dem Kontrollprozess (bedingt durch einen vorherigen Aufruf der Methode `share_lock()` durch den Besitzer des Locks), so dass hier kein Problem besteht. Anhand der übergebenen Aktionskonstante wird in `reconfigurable()` entschieden, welche Methoden weiterhin aufgerufen werden müssen. Wenn die Aktion z. B. ACTIVATE ist, aber das RO sich noch im Zustand UNLOADED befindet, so müssen die internen Methoden `_rc_load()` und `_rc_activate()` aufgerufen werden. Diese Entscheidung trifft ein C++-Switch-Case-Block, der den gesamten Zustandsübergangsautomaten repräsentiert. Dessen Code befindet sich in ReChannel-v2 somit in übersichtlicher Form in einer einzigen Methode.

Die Methoden `_rc_load()`, `_rc_activate()`, `_rc_deactivate()` und `_rc_unload()` simulieren das Verhalten des RO während der Durchführung des jeweiligen Rekonfigurationsvorgangs und steuern die Schalterzustände der verbundenen Switches. Dies erfolgt streng nach der in Abschnitt 1.5.1 beschriebenen Simulationssemantik.

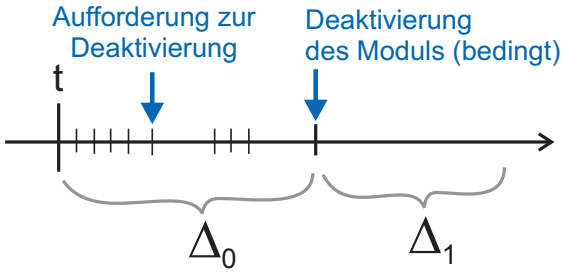

Abbildung 4.10: Die Aktivierung und Deaktivierung eines rekonfigurierbaren Objekts (bzw. Moduls) wird mit Delta-Zyklus-Grenzen synchronisiert.

4.5.5 Synchronisierung mit Delta-Zyklus-Grenzen

Um zu verhindern, dass das in Abschnitt 2.1.1 festgestellte Problem oder eine Ungleichbehandlung von externen Zugriffen innerhalb eines Delta-Zyklus auftritt, erfolgt in `rc_reconfigurable` der Rekonfigurationszustandswechsel im Fall der Aktivierung und Deaktivierung ausschließlich zwischen den Delta-Zyklen (siehe Abbildung 4.10). Hierzu wird eine Hilfsklasse namens `rc_delta_sync_object` (Delta-Sync-Objekt) implementiert, die die Synchronisierung mit Delta-Zyklus-Grenzen ermöglicht.

Die Klasse `rc_delta_sync_object` ist von der SystemC-Klasse für primitive Channels (`sc_prim_channel`) abgeleitet. Mittels des *Update-/Request-Update*-Mechanismus von SystemC ist es möglich, den Zustandswechsel in der Updatephase von SystemC und somit zwischen den Delta-Zyklen durchzuführen.

`rc_reconfigurable` verwendet intern ein Exemplar eines solchen Delta-Sync-Objekts. Diesem Objekt wird das Funktionsobjekt einer Methode übergeben, die für den Wechsel des Rekonfigurationszustands verantwortlich ist. Das Funktionsobjekt wird vom Delta-Sync-Objekt als Callback-Funktion verwendet, die es in der Updatephase von SystemC aufruft. Die Callback-Methode wird aufgerufen, wenn das Objekt eingeschaltet ist. Standardmäßig ist es ausgeschaltet. Eingeschaltet wird es immer dann, wenn ein Aktivierungs- oder Deaktivierungsvorgang des RO durchgeführt werden soll.

Im Fall der Aktivierung eines RO wird der Rekonfigurationszustandswechsel von der Callback-Methode immer umittelbar vorgenommen, da keine weitere Bedingung zu beachten ist. Im Anschluss daran wird das Delta-Sync-Objekt sofort wieder ausgeschaltet.

Wenn ein RO deaktiviert werden soll, kann es jedoch vorkommen, dass noch auf externe Zugriffe gewartet werden muss. `rc_reconfigurable` besitzt eine interne Transaktionszählervariable (siehe Abschnitt 4.3), die in ReChannel-v2 die Aufgabe der Variablen `pending_accesses` (siehe Abschnitt 1.5.4) aus ReChannel-v1 übernimmt. Solange diese Transaktionszählervariable größer 0 ist, kann die Deaktivierung nicht stattfinden. Beim nächstmöglichen Simulationszeitpunkt, an dem der Zähler am Ende eines Delta-Zyklus (d.h. in der Updatephase) den Wert 0 hat, wird der Zustandswechsel durchgeführt. Damit bei dieser Technik die Simulationzeit fortschreiten kann und nicht endlos viele Delta-Zyklen abgewartet werden, wird das Delta-Sync-Objekt zeitweise deaktiviert und erst wieder aktiviert, wenn der Zähler 0 wird.

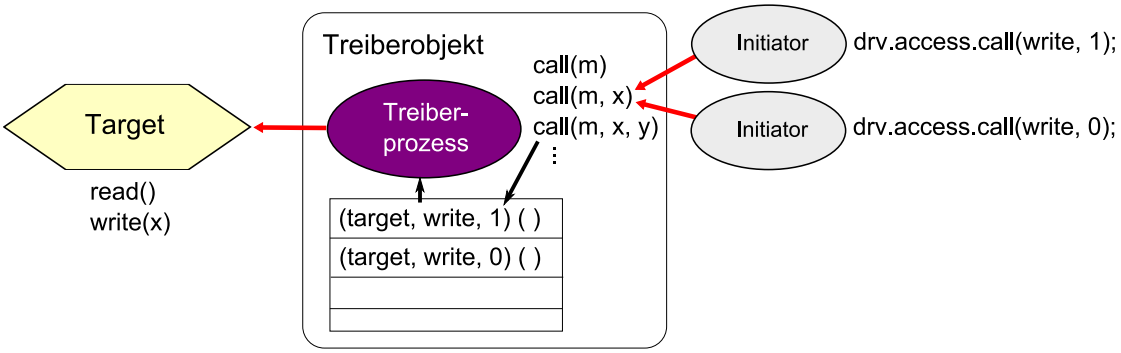

Abbildung 4.11: Illustration der prinzipiellen Funktionsweise des Treiberobjekts. Zwei Initiatorprozesse möchten schreibend auf den Channel zugreifen. Hierzu übergeben sie die Interfacemethode und die Parameter des IMCs an die Methode call() des Treiberobjekts (drv) weiter. Die IMCs werden als Funktionsobjekte gekapselt, in eine Liste eingefügt und noch im selben Delta-Zyklus in der richtigen Reihenfolge vom Treiberprozess ausgeführt. Da die Initiatorprozesse keinen direkten Zugriff auf den Channel haben, ist dem Channel nicht bewusst, dass tatsächlich mehrere Prozesse auf ihn zugreifen.

4.6 Lösung des Treiberproblems

Um das in Abschnitt 2.1.2 beschriebene Problem bei der Verwendung von Channels mit Prozessidentifikation allgemein zu lösen, wird ein so genanntes **Treiberobjekt** implementiert. Das Treiberobjekt stellt Funktionalität zur Verfügung, die es verschiedenen Prozessen erlaubt auf einen Channel zuzugreifen, ohne dabei in diesem Channel als unterschiedliches Prozess-Exemplar in Erscheinung zu treten. Ein zugreifender Prozess wird im Folgenden auch als **Initiatorprozess** oder kurz als **Initiator** bezeichnet. Der Channel, der das Ziel der Zugriffe durch die Initiatorprozesse darstellt, wird als das **Target des Treiberobjekts** bezeichnet.

Da in SystemC zwischen den beiden Zugriffstypen „blockierend„ und „nichtblockierend“ unterschieden wird, muss auch das Treiberobjekt entsprechend den Eigenschaften eines solchens Zugriffs ausgelegt sein. Außerdem ist ein IMC immer speziell auf einen bestimmten Interfacetyp ausgerichtet. Aus diesen beiden Gründen ist ein einzelnes Treiberobjekt immer auf einen bestimmten Zugriffs- und Interfacetyp spezialisiert.

Bei der Erzeugung des Treiberobjekts wird ein beliebiger, kompatibler Channel als dessen Target festgelegt. Dieses Target kann später beliebig ausgewechselt werden, jedoch muss zu jedem Zeitpunkt immer genau ein Target existieren. Ein Treiberobjekt nimmt den von einem Initiatorprozess übergebenen IMC entgegen und reicht diesen zur Ausführung an einen exklusiv für dieses Treiberobjekt-Exemplar erzeugten Treiberprozess weiter. Der hierbei verwendete Algorithmus ist wie folgt:

1. Der Initiator ruft eine `call()`-Methode auf dem Treiberobjekt auf. Hierbei werden alle Angaben bezüglich des IMC übergeben. Dies beinhaltet sowohl den Pointer der

aufzurufenden Interfacemethode sowie sämtliche Parameterwerte.

2. Die Methode `call()` erzeugt aus dem Methodenpointer, den übergebenen Parametern sowie dem Target-Pointer ein Funktionsobjekt, das weder Parameter noch einen Rückgabewert besitzt. Dieses den IMC repräsentierende Funktionsobjekt wird in eine Liste eingefügt. Falls nicht bereits durch einen anderen Aufruf geschehen, wird im Anschluss daran das Event benachrichtigt, das den Treiberprozess noch im selben Delta-Zyklus aktivieren soll.

3. Sobald der Treiberprozess aktiviert wird, führt dieser der Reihe nach jedes in der Liste enthaltene Funktionsobjekt genau ein Mal aus. Anschließend wird die Liste geleert und der Prozess gibt die Kontrolle wieder an den SystemC-Kernel ab. Der Treiberprozess sorgt selbstständig dafür, dass seine Sensitivity-List nicht durch die Zugriffe geändert wird.

Um eine allgemeine Verwendbarkeit zu gewährleisten, müssen Treiberobjekte generisch sein. `rc_driver_object` ist das Klassentemplate eines Treiberobjekts für blockierende Zugriffe, während `rc_nb_driver_object` das Treiberobjekt für nichtblockierende Zugriffe repräsentiert. Beide Klassentemplates sind mit dem Interfacetyp des Targets parametrisiert und leiten sich von einer gemeinsamen Basisklasse namens `rc_driver_object_base` ab. In `rc_driver_object` ist der interne Treiberprozess ein Thread-Prozess, in der anderen Klasse wird ein Method-Prozess verwendet. Die Basisklasse verwaltet die Liste mit den durchzuführenden Aufrufen und definiert die Einsprungmethoden der Treiberprozesse.

Treiberobjekte besitzen Templatemethoden namens `call()` für die Weiterleitung von IMCs mit bis zu zehn Übergabeparamtern (mit und ohne Funktionsrückgabewert). Um das zur Weiterleitung benötigte Funktionsobjekt zu erzeugen, verwendet das Treiberobjekt Boost-Funktionalität. Mittels `boost::bind()` (siehe Kapitel 1.2.2) werden alle Parameter des IMCs an die entsprechende Interfacemethode des Target-Exemplars gebunden. Da das hieraus resultierende Funktionsobjekt weder Parameter noch einen Rückgabewert hat, kann es in der Liste als Funktionsobjekt vom Typ `boost::function<void (void)>` (siehe Kapitel 1.2.1) abgespeichert werden. Da die Liste somit nur gleichartige Funktionsobjekte enthält, ist die vereinheitlichte Ausführung durch einen Treiberprozess möglich. Da alle Funktionsobjekte als einfache `void`-Funktionen ausgeführt werden, erfordert die Ausführung von IMCs mit Funktionsrückgabe eine zusätzliche Technik. Bei der Bildung des Funktionsobjekts kommen daher zwei Hilfsklassen (`retval`, `copy_retval`) zur Anwendung, die es ermöglichen, den Rückgabewert an den Initiatorprozess zurückzuleiten. Der Rückgabewert kann von beliebigem Datentyp sein (inkl. Referenztypen).

Da es aufgrund der Vorgaben seitens SystemC nicht möglich ist, einen Method-Prozess auf die Rückgabe des Treiberprozesses warten zu lassen, sind nichtblockierende IMCs mit Rückgabewert (dies schließt auch Rückgaben mittels Referenzparametern mit ein) nicht zusammen mit Treiberobjekten verwendbar. Für Signale bedeutet dies keine Einschränkung, da deren Schreibzugriffe keine Rückgaben liefern.

Führt der Initiator einen blockierenden Zugriff auf dem Target durch, so wird jener solange angehalten, bis der Aufruf erfolgt ist. Dazu wird ein temporäres Event erzeugt,

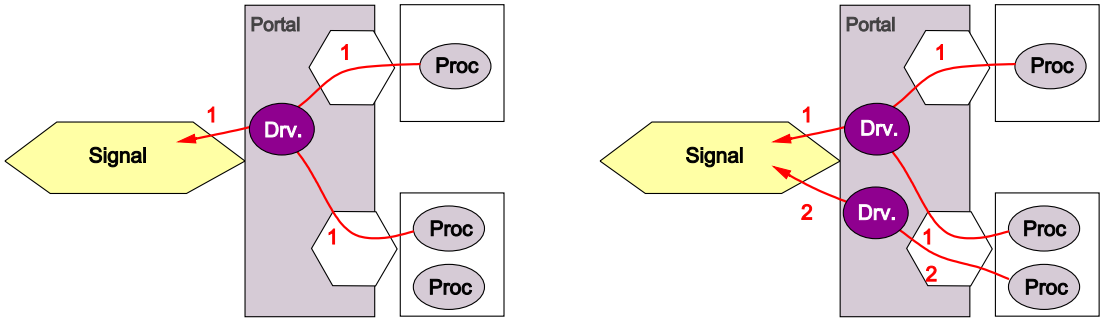

Abbildung 4.12: Anwendung des Treiberobjekts in einem Switch. Das Problem mit der Prozessidentifikation innerhalb von Channeln wird dadurch gelöst, dass im Accessor ein weiterer Zugrifftyp für derartige Channels zur Verfügung gestellt wird, bei der ein IMC nicht direkt vom Prozess des rekonfigurierbaren Moduls ausgeführt, sondern zur Ausführung automatisch an ein Treiberobjekt weitergereicht wird. (Siehe dazu auch Abschnitt 4.7.)

das der Treiberprozess nach erfolgtem Aufruf benachrichtigt. Die Rückgabe wird vom Treiberprozess in eine vorher vereinbarte Speicherstelle geschrieben. Sollte während des Aufrufs eine Exception vom rechannelinternen Typ `rc_throwable` auftreten, so wird diese an den Initiatorprozess weitergeleitet und korrekterweise im Kontext dieses Prozesses ausgelöst.

Das Target des Treiberobjekts kann zu jedem beliebigen Zeitpunkt geändert werden. Die in der Liste befindlichen Aufrufe werden immer auf dem korrekten Channel ausgeführt.

Um eine reine Aufrufschnittstelle zur Verfügung zu haben, die keine Änderung des Targets zulässt, sind die `call()`-Methoden in einem aggregierten Objekt namens `access` gekapselt. Folgender Codeausschnitt demonstriert die Verwendung des Treiberobjekts für ein SystemC-Signal:

```
// Dekl. eines Signals u. eines kompatiblen Treiberobjekts
sc_signal<int> mySignal;
rc_nb_driver_object<sc_signal_inout_if<int> > drv;
[...]
drv.set_channel(mySignal); // setze das Target auf mySignal
// Schreibe mySignal.write(32)
drv.access.call(sc_signal_inout_if<int>::write, 32);
```

Listing 4.2: Beispiel: Verwendung des Treiberobjekts für ein SystemC-Signal

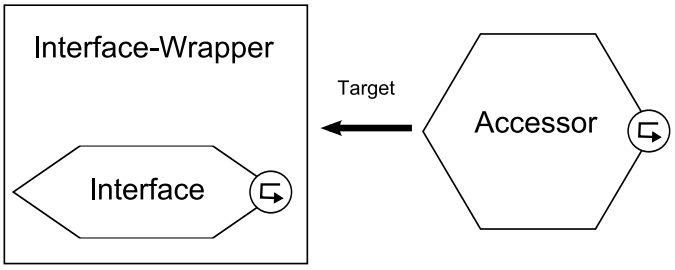

Abbildung 4.13: Der Interface-Wrapper „umhüllt“ das Interface eines Channels, so dass der Accessor keinen direkten Zugriff darauf hat.

4.7 Interface-Wrapper und Accessor

Das Konzept des Interface-Wrappers (IFW) wird eingeführt, um die völlige Unabhängigkeit des Accessors von allen anderen ReChannel-Komponenten (wie z. B. RO oder Switch) zu erreichen. Hierdurch ist es möglich, den Accessor als **allgemeine Weiterleitungskomponente** in verschiedensten Kontexten zu verwenden. Die Nützlichkeit der Wiederverwendung des Accessors wurde für die Exportalkomponente bereits in Kapitel 2.1.3 diskutiert. Darüber hinaus sind noch weitere Einsatzmöglichkeiten für den Accessor denkbar, wie z. B. die Verwendung als Filter (siehe Abschnitt 4.13.1).

Der Interface-Wrapper hat seinen Namen der Eigenschaft zu verdanken, dass er eine zusätzliche Implementationsschicht um ein Interface bildet und es somit umhüllt („wrapped“), wie in Abbildung 4.13 skizziert. Das Kommunikationsziel eines Accessors (im Folgenden auch als das **Target des Accessors** bezeichnet) ist nicht mehr ein Channelinterface, sondern ein IFW. Der IFW erfüllt den Zweck einer **Schnittstelle zwischen Accessor und Channelinterface**, die mit zusätzlicher Funktionalität ausgestattet werden kann.

Durch Callback-Aufrufe wird eine IFW-Implementation in die Lage versetzt, Treiberobjekte für die Zugriffe bereitzustellen sowie vor und nach blockierenden Zugriffen beliebigen Code auszuführen. Dies ist z. B. in Switches nützlich, um Treiberkonflikte in Signalen zu vermeiden oder die Anzahl der externen Zugriffe zählen zu können.

Die Callback-Aufrufe machen es somit möglich, sämtliche Verwaltungsaufgaben, die direkt nichts mit der Weiterleitung von IMCs zu tun haben, aus dem Accessor in den IFW zu verlagern. Dabei ist der IFW derartig konzipiert, dass ausschließlich der Accessor das Channelinterface implementieren muss und dass für die Implementation des IFWs keine Kenntnis der Interfacemethoden erforderlich ist.

4.7.1 Interface-Wrapper-ABI

Der Interface-Wrapper wird durch ein ABI namens `rc_interface_wrapper<IF>` repräsentiert, das mit einem Interfacetyp `IF` parametrisiert ist. Doch der IFW implementiert kein Channelinterface. Anstelle dessen bietet er spezielle Proxy-Funktionalität, um einen beliebigen IMC an ein Channelinterface weiterzuleiten. Ein Accessor erhält den Zugriff auf einen Channel ausschließlich über **temporäre Proxy-Objekte**, die dieser vom IFW

Zugriffsmethode	Rückgabewert
get_interface_access()	Proxy-Objekt für blockierende Zugriffe
get_nb_interface_access()	Proxy-Objekt für nichtblockierende Zugriffe)
get_driver_access(i)	Proxy-Objekt für blockierende Treiberzugriffe
get_nb_driver_access(i)	Proxy-Objekt für nichtblockierende Treiberzugriffe

Tabelle 4.2: Die im Interface-Wrapper zur Verfügung stehenden Zugriffsmethoden liefern jeweils ein Proxy-Objekt zurück. Ausschließlich über diese Proxy-Objekte kann ein Accessor auf das vom Wrapper „umhüllte“ Interface zugreifen.

erhält. Dazu stehen vier Methoden zur Verfügung, die jeweils einen der vier möglichen Zugriffstypen repräsentieren und das entsprechende Proxy-Objekt für den Zugriff liefern (siehe Tabelle 4.2).

Die Implementation der Proxy-Objekte ist fest vorgegeben. Jedes der vier weist bestimmte für den jeweiligen Zugriffstyp spezifische Unterschiede auf. Alle haben gemeinsam, dass sie bestimmte Callback-Methoden auf dem IFW aufrufen.

1. **interface_access.** Überlädt die Operatoren „−>“ und „−>∗“, durch die der blockierende Zugriff auf das Interface erfolgen soll. Das Interface wird mittels der Callback-Methode `interface_access_callback()` vom IFW erfragt. Der Zugriff wird durch die Aufrufe von zwei Callback-Methoden eingerahmt: `begin_access()` im Konstruktor und `end_access()` im Destruktor.

2. **nb_interface_access.** Überlädt die Operatoren „−>“ und „−>∗“, durch die der nichtblockierende Zugriff auf das Interface durchgeführt werden soll. Das Interface wird mittels der Callback-Methode `nb_interface_access_callback()` vom IFW erfragt. Da der Zugriff nicht blockieren kann, sind einrahmende Callbacks nicht unbedingt erforderlich. Aus Effizienzgründen wird daher auf den Aufruf von einrahmenden Callbacks verzichtet.

3. **nb_driver_access und driver_access.** Überladen beide den „−>“-Operator. Dieser liefert die Aufrufschnittstelle eines entsprechenden Treiberobjekts zurück, das vom IFW durch die Callback-Methode `nb_driver_access_callback()` bzw. `driver_access_callback()` zurückgeliefert wird. Die Treiber-Zugriffsmethoden des IFW erwarten einen Index `i` als Parameter, um verschiedene Treiberobjekt-Exemplare adressieren zu können. Jedes zurückgelieferte Proxy-Objekt steht somit in Verbindung mit einem Treiberobjekt-Exemplar des angebenen Indexes `i`. Wenn das Treiberobjekt eines angefragten Indexes noch nicht existiert, soll es vom IFW dynamisch erzeugt werden.

   `driver_access` ruft in dessen Konstruktor und Destruktor dieselben Callback-Methoden wie das Proxy-Objekt `interface_access` auf.

Der Vorteil bei der Verwendung von Proxy-Objekten besteht darin, dass für die Implementation eines Interface-Wrappers die Signaturen der Interface-Methoden nicht bekannt sein müssen. Dies macht den IFW potentiell zu einer vollständig generischen Komponente. Die Implementation von Interface-Methoden ist nur im Accessor erforderlich.

Interface-Wrapper-Methode	Beschreibung
get_interface_wrapper_name()	Liefert den Namen des Interface-Wrappers (IFW) zurück.
set_wrapped_interface()	Setzt das Interface, das der IFW „umhüllt“.
get_wrapped_interface()	Liefert das aktuelle Interface des IFWs zurück.
create_accessor()	Factory-Methode. Erzeugt kompatibles Accessor-Objekt.
get_nb_driver_count()	Liefert die Anzahl der Treiberobj. (rc_nb_driver_object).
get_driver_count()	Liefert die Anzahl der Treiberobj. (rc_driver_object).
register_port()	Accessor meldet hierüber seine Bindung mit einem Port.
begin_access_callback()	Wird vor einem blockierenden Zugriff aufgerufen.
end_access_callback()	Wird nach einem blockierenden Zugriff aufgerufen.
nb_interface_access_callback()	Callback-Methode für nichtblockierende Zugriffe.
interface_access_callback()	Callback-Methode für blockierende Zugriffe.
nb_driver_access_callback()	Callback-Methode für nichtblockierende Treiberzugriffe.
driver_access_callback()	Callback-Methode für blockierende Treiberzugriffe.

Tabelle 4.3: Übersicht über die grundlegenden Methoden des Interface-Wrapper-ABIs. [Anm.: Die IFW-Zugriffsmethoden sind in Tabelle 4.2 aufgeführt.]

Bei der Designentscheidung zum Interface-Wrapper-ABI und den Proxy-Zugriffen stand das Ziel einer hohen Erweiterbarkeit von ReChannel sowie ein möglichst geringer Aufwand für den Anwender im Mittelpunkt. Durch den Aufruf einer Callback-Methode wird ein gewisser Overhead von dem Ausmaß eines virtuellen C++-Funktionsaufrufs hervorgerufen. Da es sich um virtuelle Funktionsaufrufe auf einem ABI handelt, können Compiler diese nicht wegoptimieren. Der Overhead durch die Verwendung von temporären Proxy-Objekten und deren (als „inline“ deklarierten) Operatoren sollte vom Compiler wegoptimiert werden können. Auch können die Aufrufe der `call()`-Methode des Treiberobjekts von Compilern „ge-inlined“ werden.

4.7.2 Accessor-ABI

Das Konzept des Accessors wird in ReChannel durch ein ABI namens `accessor` repräsentiert. Der Accessor besitzt in ReChannel-v2 aufgrund der Überlegungen in Abschnitt 2.1.8 nicht mehr eine explizite Zustandsmodellierung, die auf dem Rekonfigurationsalgorithmus basiert. Anstelle dessen unterscheidet der Accessor nur noch zwischen den Zuständen „Kommunikation möglich“ und „Kommunikation gesperrt“. Diese Unterscheidung wird nicht durch einen Zustandswert bestimmt, sondern allein durch die Verfügbarkeit eines Targets. Mittels der Methode `rc_set_target()` wird dem Accessor ein Target zugewiesen, mittels der Methode `rc_clear_target()` wird dieses wieder entfernt (siehe Tabelle 4.4).

Da sich der Accessor ausschließlich auf seine Grundfunktionalität der Weiterleitung eines IMCs beschränkt, ist er vielseitig einsetzbar. Seine hauptsächliche Aufgabe, in einem Switch zur Kontrolle der Kommunikation eingesetzt zu werden, kann er weiterhin ohne Einschränkungen erfüllen.

Im Folgenden werden die grundlegenden Eigenschaften des Accessor-ABIs sowie die mit diesem verknüpften Verwendungsregeln dargestellt. Die Beschreibung der in ReChannel

Accessor-Methode	Beschreibung
[alle Interfacemethoden von IF] ...	Das Accessor-ABI ist vom Interface IF abgeleitet.
rc_set_target()	Setzt das Target des Accessors.
rc_get_target_wrapper()	Liefert das aktuelle IFW-Target zurück.
rc_clear_target()	Entfernt das Target des Accessors (⇒ „kein Target“).
rc_get_bound_port_count()	Liefert die Anzahl der gebundenen Ports zurück.
rc_get_bound_port()	Liefert den gebundenen Port des gegebenen Index zurück.

Tabelle 4.4: Übersicht über die grundlegenden Methoden des Accessor-ABIs für ein gegebenes Interface `IF`.

verwendeten Accessor-Implementation findet sich in Abschnitt 4.8.

Eigenschaften und Verwendung

Um die feste Abhängigkeit des IFWs und des Accessors kenntlich zu machen, ist das ABI des Accessors als eingebettete Klasse definiert und heißt daher mit vollem Namen: `rc_interface_wrapper_base<IF>::accessor`. Ein weiterer Grund hierfür ist, dass die Callback-Methoden und die ebenfalls eingebetteten Klassen der Proxy-Objekte als „`protected`“ deklariert werden können, ohne dass die gegenseitige Abhängigkeit der beiden ABIs die Einführung von „`template<..> friend`“-Deklarationen erforderlich macht.

Sowohl das Accessor-ABI als auch das Interface-Wrapper-ABI besitzen eine virtuelle Basisklasse mit allen Methoden, die auch ohne Kenntnis des Interfacetyps IF ausgeführt werden können. Die Namen dieser Basisklassen lauten `rc_interface_wrapper_base` und `rc_interface_wrapper_base::accessor_base`.

Für eine Accessor-Implementation ist die Kenntnis des Interfacetyps `IF` erforderlich, da das Accessor-ABI standardmäßig von diesem abgeleitet ist. Daher müssen zusätzlich zu den Methoden des ABIs auch alle Interfacemethoden implementiert werden (siehe Tabelle 4.4). Die Methoden, die durch das Accessor-ABI definiert werden, besitzen alle das Präfix „rc_“, um nicht mit den Interfacemethoden in Konflikt zu geraten.

Bei der Implementation der Interfacemethoden gelten die folgenden Regeln:

1. Eventmethoden leiten einen Aufruf nicht an den IFW weiter. Anstelle dessen muss ein accessorinternes Event-Exemplar zurückgeliefert werden. Dieses Event darf nicht dem Event des Targets entsprechen, da eine vollständige Separation des Accessors und des Channels erreicht werden soll (wie in Abbildung 1.5 auf S. 24 illustriert).

   Ein IFW ist dafür zuständig, dass die im Channel aufgetretenen Events an die Events des Accessors weitergeleitet werden.

2. Alle Methoden, die keine Eventmethoden sind, müssen einen Aufruf an den IFW weiterleiten. Dazu wird nach Zugriffstyp unterschieden und die entsprechende Zugriffsmethode des Targets zur Durchführung des Aufrufs verwendet. Sämtliche Rückgaben der Zugriffsmethoden des IFWs sind ausschließlich temporär zu verwenden. Eine Speicherung außerhalb des lokalen Gültigkeitsbereichs ist unbedingt zu vermeiden, da dies zu undefiniertem Verhalten im IFW führen kann.

3. Da es eine Methode `rc_clear_target()` gibt, die das Target entfernt, muss eine Accessor-Implementation wissen, wie sie mit nichtblockierenden Zugriffen verfährt, wenn aktuell kein Target verfügbar ist. Wird ein blockierender Zugriff aufgerufen und kein Target verfügbar ist, soll der Zugriff solange blockiert werden, bis wieder ein Target gesetzt worden ist. Sobald das neue Target verfügbar ist, soll auf diesem der Zugriff durchgeführt werden.

4. Wenn der Accessor Interfacemethoden mit Treiberzugriffen enthält, dann soll er für jeden über diese Methoden zugreifenden Prozess ein anderes Treiberobjekt verwenden. Dies ist bei Aufruf der Treiberzugriffsmethoden mittels der Übergabe eines unterschiedlichen Indexes möglich. Der Accessor soll dazu eine Map über die Indexzuordnung der Prozesse verwalten. Die Treiberobjekte sollen mit fortlaufender Nummerierung (bei 0 beginnend) indiziert werden. Dem Bedarf des Accessors entsprechend, soll die IFW-Komponente beliebig viele Treiberobjekte dynamisch neu erzeugen können.

   Mittels der Technik der Treiberobjekte können die durch die Simulation von DR verursachten Treiberkonflikte verhindert werden, wie in Abbildung 4.12 (S. 86) illustriert. Da in einem Accessor für jeden zugreifenden Prozess ein verschiedenes Treiberobjekt verwendet wird, können hierüber „reale“ Treiberkonflikte immer noch erkannt werden.

Das Accessor-ABI ist im Verlauf der Arbeit mehrmals erweitert worden. Auf einige weitere Aspekte und Methoden des Accessor-ABIs wird in späteren Abschnitten eingegangen.

4.8 Accessor-Implementation

Der Accessor wird im Sinne der in Abschnitt 4.2 skizzierten neuen Sichtweise sowie getreu der für das Accessor-ABI spezifizierten Verwendungsvorschriften (siehe Abschnitt 4.7.2) implementiert. Eine Accessor-Implementation leitet sich somit vom Accessor-ABI ab und kann mit einem beliebigen, kompatiblen Interface-Wrapper-Target kommunizieren. Die Accessor-Implementation, die standardmäßig für einen bestimmten Interfacetyp `IF` verwendet werden soll, ist eine Spezialisierung des Klassentemplates `rc_accessor<IF>`. Unter Angabe des Interfacetyps ist somit nun unmittelbar die Klasse eines kompatiblen Accessors bekannt.

In ReChannel sind für alle in SystemC definierten Interfaces bereits kompatible Standard-Accessoren verfügbar. Für Interfaces eines benutzerdefinierten Typs muss der Anwender eigens eine Accessor-Spezialisierung definieren.

Ein Accessor für einen bestimmten Interfacetyp wird definiert, indem er als Spezialisierung des Klassentemplates `rc_accessor<IF>` deklariert wird und sich von der Klasse `rc_abstract_accessor<IF>` ableitet. Das Klassentemplate `rc_abstract_accessor<IF>` ist die Basis sämtlicher Funktionalität der Accessor-Implementationen. Diese wiederum ist vom Accessor-ABI sowie von einer Basisklasse namens `rc_abstract_accessor_b` abgeleitet, in der alle Methoden definiert werden, die nicht vom Interfacetyp abhängen.

```
template<class T>
RC_ACCESSOR_TEMPLATE(my_channel_if<T>) {
    RC_ACCESSOR_TEMPLATE_CTOR(my_channel_if<T>) {}

    void write(const T& data) {
        rc_forward(if_type::write, data);
    }
    RC_EVENT(my_event);
};
```

Listing 4.3: Die Definition eines Accessors für ein benutzerdefiniertes Interface vom Typ `my_channel_if<T>`. Hierbei wird die Weiterleitung einer nichtblockierenden Schreib-Methode implementiert sowie eine Event-Methode des Interfaces deklariert. [Anm.: Da das Interface ein Template ist, muss `RC_ACCESSOR_TEMPLATE` anstelle von `RC_ACCESSOR` verwendet werden.]

Die Basisklasse dient dem Zweck, den für einen Accessor benötigten Templatecode zu verringern und somit die Gesamtcodegröße zu reduzieren.

Um einen lauffähigen Accessor zu erhalten, müssen bei der Spezialisierung der Klasse `rc_accessor<IF>` ferner alle Interfacemethoden von IF implementiert werden. Für die Deklaration und Ableitung werden Makros bereitgestellt, um diesen Vorgang für den Anwender zu erleichtern. Die Definition des Accessors für das in Listing 1.3 gezeigte Beispielinterface ist in Listing 4.3 (S. 92) illustriert.

4.8.1 Implementierung der Interfacemethoden

Die Implementierung der Interfacemethoden erfolgt aus Sicht des Anwenders ausschließlich über speziell hierfür bereitgestellte Weiterleitungsmethoden. Das Accessor-ABI ist dem Anwender nicht bekannt und stellt daher ein internes Implementationsdetail dar. Die Methoden kapseln die Funktionalität, die ein Accessor für die korrekte Weiterleitung eines IMCs an das Target benötigt. Hier ist auch der Wartevorgang im Falle des Fehlens eines Targets enthalten. Dabei wird auf ein bestimmtes accessorinternes Event gewartet, dass bei Setzen eines neuen Targets benachrichtigt wird. Im Falle von nichtblockierenden Zugriffen wird bei Fehlen eines Targets der IMC auf einem so genannten Fallback-Interface ausgeführt (siehe Abschnitt 4.8.2).

Für die Deklaration von Eventmethoden und dem dazugehörigen Event durch den Anwender stehen die Makros `RC_EVENT()` und `RC_EVENT_ALIAS()` zur Verfügung.

Für die Implementierung der Zugriffsmethoden des Interfaces kann der Anwender die folgenden vier Weiterleitungsmethoden verwenden:

- `rc_forward()` für blockierende Zugriffe,
- `rc_nb_forward()` für nichtblockierende Zugriffe,

- `rc_forward_driver()` für blockierende Treiberzugriffe und
- `rc_nb_forward_driver()` für nichtblockierende Treiberzugriffe.

Alle diese Weiterleitungsmethoden gehören jeweils zu einer Familie von Methodentemplates, mittels derer beliebige IMCs („`const`“ und „nicht `const`“, mit oder ohne Rückgabewert) mit bis zu zehn Parametern weitergeleitet werden können. `rc_nb_forward_driver()` unterliegt allerdings der in Abschnitt 4.6 beschriebenen Beschränkung bezüglich der Treiberweiterleitung von nichtblockierenden Zugriffen, so dass es von diesem Methodentemplate keine Varianten mit Funktionsrückgabe gibt.

Die Weiterleitungsmethoden sind alle mit dem Schlüsselwort „inline“ deklariert, so dass durch deren Verwendung kein zusätzlicher Methodenaufruf verursacht wird, wenn der Compiler die entsprechende Optimierung durchführt.

Gemischte Treiberzugriffstypen

Blockierende und nichtblockierende Treiberzugriffe verwenden zwei verschiedene Treiberobjekttypen (siehe Abschnitt 4.6 und 4.7.1). Die gemeinsame Verwendung beider Treiberzugriffstypen in einer Accessordefinition kann daher für einen zugreifenden Prozess dazu führen, dass dieser als doppelter Treiber im Channel in Erscheinung tritt.

Signal-Accessoren sind hiervon jedoch grundsätzlich nicht betroffen, da für sämtliche Signale ausschließlich nichtblockierende Zugriffe notwendig sind.

Spezielle Optimierungen

Im Fall von nichtblockierenden Treiberzugriffen kann eine Optimierung durch die Definition der Compilerkonstante `RC_USE_SHARED_METHOD_DRIVER` erreicht werden. Ist diese Konstante definiert, verwenden alle nichtblockierenden Treiberobjekte einen einzigen, gemeinsamen Method-Prozess. Der Accessor kann sich daher bei diesem Zugriffstyp den Aufwand für die Prozessidentifikation und die Indexzuordnung sparen. Bei dieser Optimierung muss zwar auf die Erkennung „realer“ Treiberkonflikte (siehe Abschnitt 4.7.2) verzichtet werden, doch ist im Gegenzug eine Performance-Steigerung z. B. bei der Verwendung von Signalen zu erwarten. Noch einen Schritt weiter geht die Compilerkonstante `RC_SIGNAL_WRITE_CHECK_DISABLED`, die als spezielle Optimierung für Signal-Accessoren gedacht ist. Deren Definiton ist sinnvoll in Verbindung mit einer SystemC-Implementation, bei der die Treiberkonflikterkennung in Signalen ausgeschaltet werden kann. Daraufhin verwenden die in ReChannel vordefinierten Signal-Accessoren überhaupt keine Treiberzugriffe mehr.

4.8.2 Fallback-Interface

Ein inaktiver Accessor in ReChannel-v1 entspricht in ReChannel-v2 einem Accessor ohne zugewiesenem Target. Wenn der Accessor kein Target besitzt, dann werden Prozesse innerhalb der blockierenden Zugriffsmethoden solange angehalten, bis wieder ein Target verfügbar ist. Für nichtblockierende Zugriffsmethoden stellt ein fehlendes Target hingegen

```
template<class T>
RC_FALLBACK_INTERFACE_TEMPLATE(my_channel_if<T>) {
    RC_FALLBACK_INTERFACE_TEMPLATE_CTOR(my_channel_if<T>) {}

    void write(const T& data) {} // Aufruf wird ignoriert
    const sc_event& my_event() const
        { [...] } // z.B. Warnung oder Error ausgeben
};
```

Listing 4.4: Die Definition eines Fallback-Interfaces für einen benutzerdefinierten Interfacetyp kann mithilfe von Makros erfolgen.

ein Problem dar. Einerseits darf nicht gewartet werden und andererseits darf der IMC nicht einfach ignoriert werden, da eine sinnvolle Rückgabe geliefert werden muss. Der IMC muss somit stattfinden.

In ReChannel-v1 hat ein inaktiver Accessor den IMC an den statischen Channel weitergeleitet, obwohl dies laut Simulationssemantik nicht gestattet ist (vgl. Kapitel 1.5.3). Die Problematik wird in ReChannel-v2 dadurch gelöst, dass der Accessor in einem solchen Fall den IMC an ein Hilfsinterface namens `rc_fallback_interface<IF>` weiterleitet, das als Ersatz für das fehlende Target dient.

`rc_fallback_interface<IF>` wird als Fallback-Interface bezeichnet, da es dazu konzipiert ist, alle Aufrufe entgegenzunehmen, die ausgeführt werden müssen, aber für die kein Target existiert. Je nachdem welche Funktionalität für einen Interfacetyp benötigt wird, kann es Standardrückgabewerte, Warnmeldungen oder Fehlermeldungen produzieren.

Beispielsweise ist aufgrund des Nachhalls in deaktivierten RTL-Modulen (siehe Abschnitt 2.1.4) bei Signal-Accessoren häufig mit Aufrufen auf dem Fallback-Interface zu rechnen. Das Fallback-Interface für Signal-Interfaces gibt daher keine Warnungen aus, sondern ignoriert `write()`-Zugriffe und liefert bei `read()`-Zugriffen einen Standardwert zurück. Bei Resolved-Signals ist der von der `read()`-Methode zurückgegebene Standardwert immer 'X' und bei den anderen Signaltypen entspricht dieser dem Standardwert des verwendeten Datentyps in C++. Der Standardwert eines Datentyps (im Zusammenhang mit den von ReChannel vordefinierten Fallback-Interfaces) kann vom Anwender mittels des Makros `RC_UNDEFINED_VALUE()` global festgelegt werden.

Für alle SystemC-Interfaces sind in ReChannel-v2 bereits entsprechende Fallback-Interfaces vordefiniert. Für benutzerdefinierte Interfacetypen ist das Fallback-Interface vom Anwender zu definieren.

Doch muss in ReChannel ein Fallback-Interface nicht zwingend für jeden Interfacetyp definiert werden. Falls in der Anwendung die Situation eines Aufrufs auf das entsprechende Fallback-Interface niemals vorkommt, kann dessen Definition gespart werden. Dass die Defintion des Interfaces optional ist, wird in ReChannel technisch mittels eines Factory-Klassentemplates ermöglicht. Diese Klasse erzeugt ein Fallback-Interface des entsprechenden Typs, falls dieser existieren sollte, andernfalls liefert es den NULL-Pointer

zurück. Eine Designbeschreibung kann daher kompiliert werden, ohne dass der Anwender ein Fallback-Interface definiert hat. Falls dieses dennoch von einem Accessor benötigt werden sollte, meldet dieser einen SystemC-Error, um den Anwender zur Definition des Fallback-Interfaces aufzufordern.

4.8.3 Verwendung mit rücksetzbaren Prozessen

Der Accessor ist definitionsgemäß eine Komponente, in der Prozesse angehalten werden können. Damit auch die rücksetzbaren Prozesse der ReChannel-Prozess-API (siehe Abschnitt 4.11.2) im Accessor korrekt zurückgesetzt werden können und dort nicht dauerhaft blockieren, muss der Accessor intern die präparierte Wartemethode `rc_wait()` der Prozess-API verwenden.

`rc_abstract_accessor<IF>` implementiert darüber hinaus auch eigene `wait()`- und `next_trigger()`-Methoden. Diese dienen als Umleitung zu den globalen Wartefunktionen, damit auch in abgeleiteten Oberklassen automatisch nur präparierte Wartemethoden aufgerufen werden. Dies erlaubt es dem Benutzer von ReChannel, im Accessor auch einfach „`wait()`“ bei der Implementierung der Interfacemethoden zu verwenden, ohne dass es mit rücksetzbaren Prozessen Schwierigkeiten gibt.

4.8.4 Zusätzliche Funktionalität

Die allgmeine Regel sollte sein, dass nur solche Zusatzfunktionalität in einen Standard-Accessor integriert wird, die grundsätzlich immer bei der Verwendung eines Interfacetyps Sinn macht. Funktionalität für Spezialanwendungen sollte ausschließlich in einer vom jeweiligen Standard-Accessor abgeleiteten Accessor-Implementation definiert werden.

Vermeidung von Deadlocks in FIFO-Channels

Der in ReChannel vordefinierte FIFO-In-Accessor enthält spezielle Zusatzfunktionalität, um die in Kapitel 2.1.1 beschriebenen Deadlocks zu verhindern. Bevor ein solcher Accessor einen `read()`-Zugriff startet, überprüft er erst mit `num_available()`, ob dies Erfolg haben wird. Wenn diese Statusmethode 0 zurückliefern sollte, wird schon im Accessor auf neue Daten gewartet und nicht erst im Channel. Hierdurch kann ein Modul auch dann problemlos abgeschaltet werden, wenn in den Eingangs-FIFOs keine Daten mehr verfügbar sein sollten.

Die gleiche Funktionalität bietet aus Symmetriegründen auch der FIFO-Out-Accessor. Dieser testet vor einem `write()`-Zugriff, ob `num_free()` größer 0 ist.

4.9 Switch-Implementationen

Die ReChannel-Bibliothek stellt zwei vordefinierte Switch-Varianten zur Verfügung, die das Portal- bzw. das Exportal-Konzept von ReChannel implementieren. Beide Switch-Implementationen verwenden Accessoren und Interface-Wrapper, um die durch sie fließende Kommunikation in Abhängigkeit ihres Schalterzustands kontrollieren zu können.

4.9.1 Interface-Wrapper

Da in Portal und Exportal Accessoren zur Kontrolle der Kommunikation verwendet werden, muss auch ein Target in Form eines Interface-Wrappers (IFW) bereitgestellt werden (vgl. Abschnitt 4.7). Beim Portal repräsentiert der Interface-Wrapper die Schnittstelle zum statischen Channel, da dieser das Ziel der Zugriffe durch Prozesse der rekonfigurierbaren Seite ist. Im Fall des Exportals stellt der Interface-Wrapper die Schnittstelle zu einem auf der rekonfigurierbaren Seite befindlichen Channel dar, da hier die Kommunikation in entgegengesetzter Richtung zum Portal abläuft.

Sowohl die Portal- als auch die Exportal-Klasse stellen (im Gegensatz zu der in Abbildung 4.6 auf S. 70 dargestellten Klassenhierarchie) nicht selbst das Target für den Accessor dar, sondern verwenden zu diesem Zweck eine separate, generische Interface-Wrapper-Klasse. Diese Aufteilung dient dazu, die Funktionalität des Interface-Wrappers auf modulare Weise wiederverwenden zu können und die Methoden-Schnittstellen der Switch-Klassen nicht zu groß werden zu lassen. Der Grund, weshalb Portal und Exportal dieselbe IFW-Komponente für ihre Zwecke nutzen können, wird in Abschnitt 4.9.5 beschrieben.

Aufgaben der Interface-Wrapper-Komponente im Switch

Die IFW-Komponente kapselt die Basisfunktionalität, die für die Verwaltung und Erzeugung von Treiberobjekten und die Weiterleitung von Events (Event-Forwarder-Mechanismus) benötigt wird.

Eine weitere wichtige Aufgabe, die diese Komponente im Auftrag des Switches übernimmt, ist die Zählung von Transaktionen im RO. Für jeden andauernden, externen Zugriff wird der Transaktionszähler im RO um Eins erhöht, um – wie von der Simulationssemantik (siehe Abschnitt 1.5.1) gefordert – währenddessen eine Deaktivierung des RO zu verhindern. Bei Beendigung des externen Zugriffs wird der Transaktionszähler um Eins erniedrigt. Dieser Mechanismus wird mittels der im Interface-Wrapper-ABI (siehe Abschnitt 4.7.1) spezifizierten Callback-Methoden implementiert, die jeden potentiell blockierenden Zugriff einrahmen.

Event-Forwarder-Mechanismus

Die IFW-Komponente ist intern darauf optimiert, die Anzahl der Method-Prozesse, die für die Event-Weiterleitung angelegt werden, minimal zu halten. Wenn ein und dasselbe Quellevent von mehreren Switches weitergeleitet werden soll (z. B. da diese alle mit

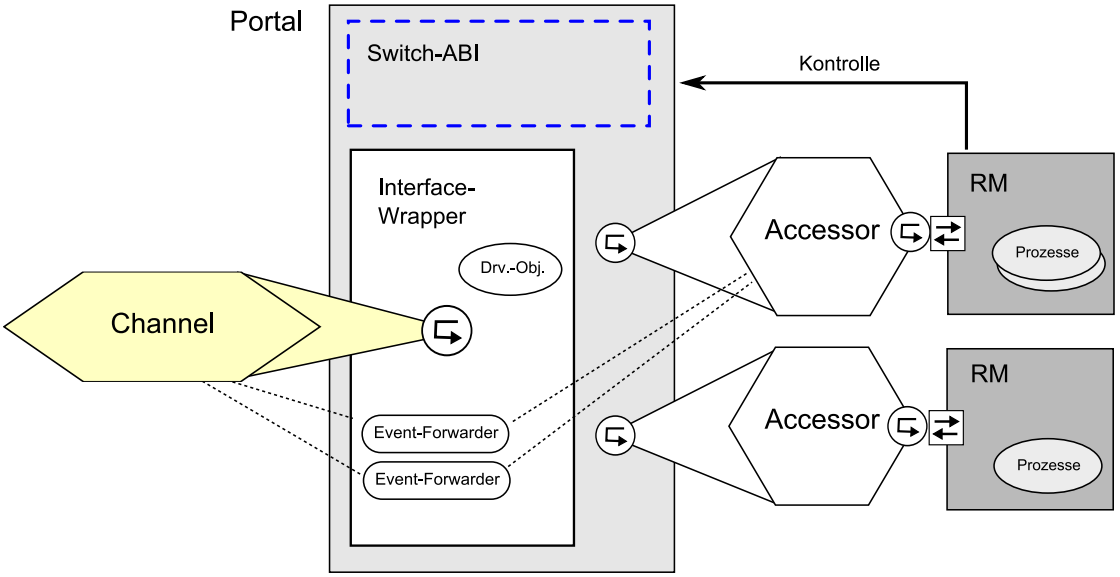

Abbildung 4.14: Schematische Darstellung des Portal-Switches

demselben statischen Channel verbunden sind), so wird hierfür nur ein Prozess verwendet. Anstatt mehrere Prozesse zu erzeugen, werden einem einzelnen Prozess mehrere Zielevents zugewiesen.

4.9.2 Portal

Das Portal wird durch das Klassentemplate `rc_portal<PORT>` repräsentiert, das mit einem beliebigen Porttyp (`PORT`) parametrisiert werden kann. Das Klassentemplate leitet sich von einer Basisklasse namens `rc_abstract_portal<PORT>` ab, die die generische Standardfunktionalität des Portals enthält. Die Basisklasse implementiert alle Methoden des Switch-ABIs und definiert darüber hinaus alle benötigten portalspezifischen Methoden (z. B. für die Bindung/Erzeugung der Accessoren) und Datenelemente (wie z. B. den statischen Port).

Das Klassentemplate `rc_portal<PORT>` enthält selbst keinen Code, damit beliebige Portal-Implementationen definiert werden können, die ebenfalls den Klassennamen `rc_portal` tragen. Dies wird durch die C++-Technik „Spezialisierung“ ermöglicht.

ReChannel stellt für jeden in SystemC vorhanden Porttyp ein spezialisiertes Portal zur Verfügung. Ein benutzerdefiniertes Portal wird definiert, indem das Klassentemplate `rc_portal<PORT>` für den entsprechenden Porttyp spezialisiert wird (siehe Listing 4.5). Dabei wird ebenfalls die Klasse `rc_abstract_portal<PORT>` als Basisklasse verwendet, um die Standardfunktionalität des Portals zu erben. Die Basisklasse besitzt keine als „rein virtuell“ deklarierten Methoden, so dass, abgesehen von der Bereitstellung eines Konstruktors, keine zusätzlichen Methoden definiert werden müssen.

Die Syntax zur Deklaration eines Portals sowie dessen Bindung an den statischen Channel sieht in ReChannel-v2 – hier am Beispiel des FIFO-Portals – wie in ReChannel-v1

```
template<class T>
class rc_portal<my_port<T> >
  : public rc_abstract_portal<my_port<T> >
{
    [...] // Implementation
};
```

Listing 4.5: Der obige Codeausschnitt zeigt, wie die Definition des Portals für einen benutzerdefinierten Porttyp namens `my_port<T>` aussehen würde. Das neu definierte Portal erbt dabei die gesamte Standardfunktionalität eines Portals von der generischen Basisklasse `rc_abstract_portal<my_port<T> >`.

aus:

```
rc_portal<sc_fifo_in<int> > fifo_in_portal;
[...]
fifo_in_portal.static_port(fifo_channel);
```

Die Bindung zu einem kompatiblen Port eines rekonfigurierbaren Moduls wird mittels der Methode `dynamic_port()` durchgeführt:

```
fifo_in_portal.dynamic_port(M_rc.fifo_port);
```

4.9.3 Benutzerdefinierte Portals

Durch die Möglichkeit der Spezialisierung des Klassentemplates `rc_portal<PORT>` kann das Verhalten des Portals an die Erfordernisse verschiedener Porttypen angepasst werden. Einem benutzerdefinierten Portal kann beliebige Funktionalität hinzugefügt werden, die für den jeweiligen Porttyp benötigt wird. ReChannel stellt Makros zur Verfügung, um den Vorgang der Spezialisierung bzw. Anpassung für den Benutzer zu automatisieren. Wie eine Portaldefinition mit Makros beispielsweise aussehen kann, ist in Listing 4.6 illustriert.

Event-Weiterleitung

Das Makro `RC_PORTAL_FORWARD_EVENT()` wird verwendet, um die Namen der Events beim Portal zu registrieren, die vom statischen Channel an den Accessor weitergeleitet werden sollen. Hinter diesem Makro verbirgt sich intern der Aufruf einer Methode namens `add_event_forwarder()`. Da es möglich ist, der Methode `add_event_forwarder()` auch direkt einen SystemC-Eventfinder zu übergeben, können in ReChannel-v2 auch Ports mit variabler Eventanzahl verwendet werden (vgl. Abschnitt 2.1.3, „Variable Anzahl von Events“).

```
RC_PORTAL(sc_inout_resolved)
{
    RC_PORTAL_CTOR(sc_inout_resolved) {
        // Deklaration aller weiterzuleitender Events
        RC_PORTAL_FORWARD_EVENT(value_changed_event);
        RC_PORTAL_FORWARD_EVENT(posedge_event);
        RC_PORTAL_FORWARD_EVENT(negedge_event);
    }
    // Aktion bei Öffnung des Portals
    RC_ON_OPEN() {
        this->refresh_notify(); // Prozesse im Modul aufwecken
    }
    // Aktion bei Schließung des Portals
    RC_ON_CLOSE() {
        // Schreiben von 'Z' für jeden registrierten Treiberprozess
        int count = this->get_interface_wrapper().get_nb_driver_count();
        for (int i=0; i < count; i++) {
            this->get_interface_wrapper().get_nb_driver_access(i)
                ->call( &if_type::write, SC_LOGIC_Z );
        }
    }
    // Aktion, bei undefiniertem Schalterzustand
    RC_ON_UNDEF() {
        [...] // wie RC_ON_CLOSE(), aber ein Treiber schreibt 'X'.
    }
    // benachrichtigt die Events des Accessors, um Prozesse aufzuwecken
    RC_ON_REFRESH_NOTIFY() {
        this->notify_event("value_changed_event");
        if (this->get_static_port().read() == SC_LOGIC_1) {
            this->notify_event("posedge_event");
        }
    }
};
```

Listing 4.6: Der Codeausschnitt zeigt, wie in ReChannel-v2 das Portal für den `sc_inout_resolved`-Port mittels Makros definiert wurde. Es werden Event-Forwarder für die zu diesem Channel gehörigen Events deklariert und das für diesen Porttyp notwendige Schalterverhalten implementiert.

```
template<class T>
struct rc_port_traits<sc_fifo_in<T> > {
    typedef sc_fifo_in<T>     type;
    typedef sc_fifo_in_if<T> if_type;
};
```

Listing 4.7: Port-Traits-Deklaration des FIFO-In-Ports `sc_fifo_in<T>`

Schalterverhalten

Ein weiterer Grund für eine Benutzeranpassung des Portals kann die Notwendigkeit eines Schalterverhaltens für einen bestimmten Channeltyp sein (vgl. Abschnitt 2.1.6). Das Schalterverhalten eines Portals wird in den Callback-Methoden `rc_on_open()`, `rc_on_close()` und `rc_on_unload()` definiert. Diese werden vom Portal automatisch aufgerufen, nachdem die entsprechende Schalteraktion durchgeführt wurde. Die Callback-Methoden dürfen weder Zeit noch Delta-Zyklen in Anspruch nehmen, um das Timing des Rekonfigurationsalgorithmus nicht negativ zu beeinflussen. Für jede der Callback-Methoden existiert auch ein entsprechendes Makro, um deren Definition für den Anwender zu erleichtern (siehe Listing 4.6).

ReChannel definiert für einige SystemC-Channeltypen ein angepasstes Schalterverhalten, damit diese bei der Simulation von DR verwendet werden können. Im Falle der Portals für Semaphor, Mutex, Signals sowie Resolved-Signals von SystemC ist ein Schalterverhalten implementiert, das die in Abschnitt 2.1.6 und 2.1.7.d beschriebenen Probleme löst.

Über die Interface-Wrapper-Komponente des Portals kann Einfluss auf den Channel genommen werden, da hierüber u. a. alle Treiberzugriffsmethoden aufgerufen werden können. Somit kann z. B. das Portal für Resolved-Signals bei dessen Schließung alle Treiberobjekte zurücksetzen, die zuvor auf den statischen Channel geschrieben haben. Dies wird dadurch erreicht, dass es jedes Treiberobjekt einmal den hochohmigen Wert 'Z' auf den Channel schreiben lässt.

Das Schalterverhalten ist ebenso in der anderen Richtung vonnöten, um die schlafenden, internen Prozesse eines RM aufwecken zu können. Beim Öffnen kann das Portal z. B. dafür sorgen, dass die richtigen Events auf dem Accessor in Abhängigkeit vom aktuellen Zustand des statischen Channels benachrichtigt werden.

Erzeugung des Accessors

Die Methode `create_accessor()` ist die Factory-Methode, die jene Accessor-Objekte erzeugt, welche das Portal an die Ports der rekonfigurierbaren Module bindet. Der Anwender kann diese Methode überladen, um die von einem Portal zu verwendende Accessor-Implementation zu variieren. Dies kann z. B. nützlich sein bei nachträglichen Erweiterungen bereits bestehender Portal-Spezialisierungen.

Zusätzliche Funktionalität

Ein Portal kann mit beliebiger Zusatzfunktionalität ausgestattet werden, die dessen Verwendung als Switch nicht widerspricht (vgl. Abschnitt 4.4). Da die Portal-Klasse auf einem SystemC-Modul basiert, können beliebige interne Strukturen oder Ports hinzugefügt werden. Weiterhin sind z. B. Methoden zur dynamischen Anpassung des Schalterverhaltens denkbar. Das Portal für Signal-Ports könnte z. B. um eine Möglichkeit zur direkten Ansteuerung des Schaltermechanismus erweitert werden.

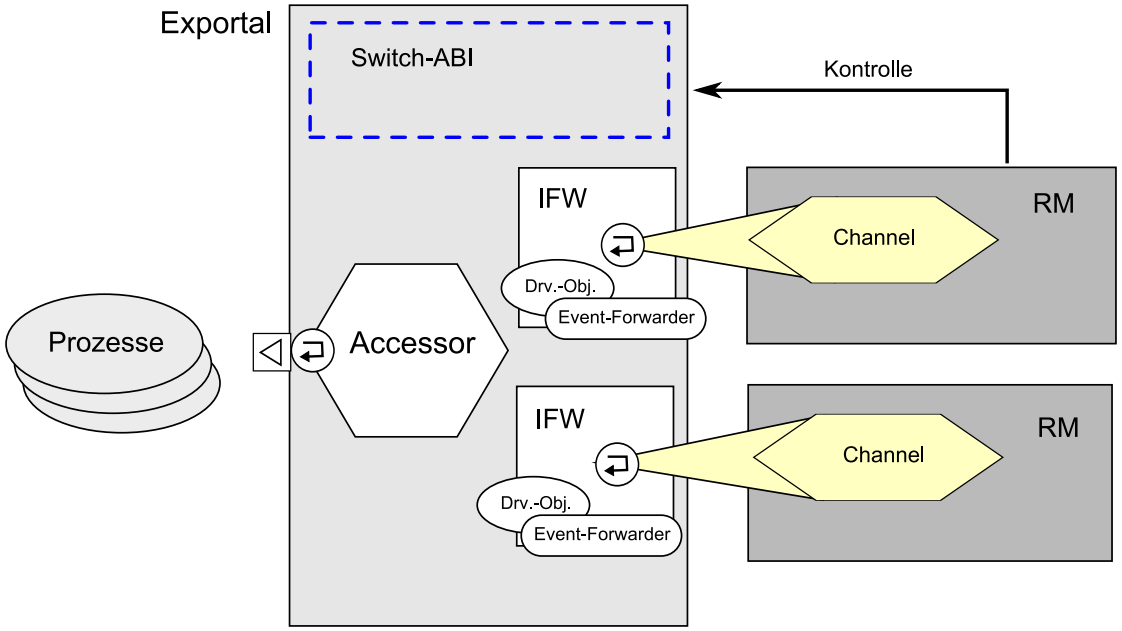

Abbildung 4.15: Schematische Darstellung des Exportal-Switches

4.9.4 Port-Traits

Der zum Port `PORT` gehörige Interfacetyp `IF` wird vom Portal mittels einer Traits-Klasse namens `rc_port_traits` ermittelt, die mit dem Porttyp parametrisiert wird. Im Unterschied zu der Traits-Klasse aus ReChannel-v1 hat diese keine Verbindung mehr mit dem Accessor, sondern ist ausschließlich für die Definition von Interface-Port-Paaren gedacht. Die Gründe, die diese Änderung veranlasst haben, wurden in Abschnitt 4.2 dargelegt. Die Port-Traits-Deklaration für ein FIFO-In-Port ist – exemplarisch für andere Port-Traits-Deklarationen – in Listing 4.7 zu sehen.

Der Anwender muss für einen benutzerdefinierten Porttyp lediglich dann eine Port-Traits-Klasse definieren, wenn der Port nicht ein öffentliches Typedef namens `if_type` mit dem Typ des Interfaces besitzt. Andernfalls kann ReChannel die Ermittlung des Interfacetyps automatisch vornehmen. Durch die Verwendung von C++-Templatetechniken funktioniert die automatische Typermittlung ebenfalls für das SystemC-Klassentemplate `sc_port`. Für alle weiteren Port-Klassen von SystemC stehen in ReChannel bereits vordefinierte Port-Traits-Definitionen zur Verfügung.

4.9.5 Exportal

Das Exportal wird durch das Klassentemplate `rc_exportal<IF>` repräsentiert. Dieses kann mit einem beliebigen Interfacetyp `IF` parametrisiert werden.

Zur statischen Seite hin besitzt das Exportal einen Export namens `static_export`, auf den die Prozesse des statischen Designbereichs zugreifen können. Intern wird an diesen Export ein Accessor gebunden, der die durch das Exportal fließende Kommunikation

kontrolliert. Dieser Accessor befindet sich im Unterschied zu den Accessoren des Portals nicht auf der rekonfigurierbaren, sondern auf der statischen Seite. Die Unabhängigkeit des Accessors vom Rekonfigurationsalgorithmus sowie einem bestimmten Verwendungszweck machen es möglich, dass der Accessor in der umgekehrten Richtung betrieben werden kann.

Die Deklaration eines Exportals für einen FIFO-Export sieht z. B. folgendermaßen aus:

```
rc_exportal<sc_fifo_out_if<int> > fifo_out_exportal;
```

Die Bindung zu einem kompatiblen FIFO-Export eines rekonfigurierbaren Moduls wird mittels einer Methode namens `dynamic_export()` durchgeführt:

```
fifo_out_exportal.dynamic_export(M_rc.exported_fifo);
```

ReChannel stellt für jeden in SystemC vorhandenen Interfacetyp ein spezialisiertes Exportal zur Verfügung. Um ein Exportal für einen benutzerdefinierten Interfacetyp zu implementieren, wird das Exportal-Klassentemplate `rc_exportal<IF>` auf zum Portal analoge Weise spezialisiert. Die Standardfunktionalität des Exportals wird dabei von der Basisklasse namens `rc_abstract_exportal<IF>` bereitgestellt. Um die Definition eines Exportals für den Anwender zu vereinfachen, stehen auch für das Exportal entsprechende Defintionsmakros zur Verfügung. Wie beim Portal können auch beim Exportal u. a. die Event-Weiterleitung und das Schlaterverhalten definiert werden.

Interface-Wrapper des Exportals

Im Gegensatz zum Portal, das eine einzige, feste Interface-Wrapper-Komponente verwendet, kann die vom Exportal verwendete Interface-Wrapper-Komponente während der Simulation ausgetauscht werden.

Dieser Umstand rührt daher, dass in ReChannel auch die Mobilität von RO ermöglicht werden soll (siehe Abschnitt 4.1). Bereits bei den Vorüberlegungen (siehe Kapitel 3) zeichneten sich die davon zu erwartenden Konsequenzen für den Entwurf des Exportals ab: Wenn ein mit einem Exportal verbundenes Interface zusammen mit seinem zugehörigen, rekonfigurierbaren Modul verschoben wird, wird es an anderer Stelle im Design mit einem anderen Exportal-Exemplar gebunden werden. Wenn der hinter dem Interface stehende Channel auf eine Prozessidentifikation angewiesen ist, trifft dieser daraufhin auf völlig neue Prozess-Identitäten. Im Fall von Signalen beispielsweise hätte diese Situation einen Treiberkonflikt inklusive Abbruch der Simulation zur Folge.

Um das Problem mit den Treiberkonflikten bei Exportals zu lösen, wird daher die Designentscheidung getroffen, jedem Interface, das mit einem Exportal verbunden ist, ein eindeutiges Interface-Wrapper-Exemplar zuzuordnen. Die Exportal-Klasse verwendet zu diesem Zweck einen globalen Map-Container, in dem die Zuordnung von Interfaces zu Interface-Wrapper-Komponenten gespeichert wird. Wird ein Interface das erste Mal mit einem Exportal verbunden, wird eine neue Interface-Wrapper-Komponente erzeugt und in die Map eingefügt. Außerhalb des Exportals ist dieser Mechanismus nicht sichtbar,

so dass auch in der Klasse `rc_reconfigurable` dafür keine Änderungen vorgenommen werden müssen.

Da das Exportal aus den oben genannten Gründen keinen eigenen, integrierten Interface-Wrapper besitzt, hat sich auch das Problem der unterschiedlichen Anforderungen für die Event-Weiterleitung von Portal und Exportal erübrigt (vgl. Abschnitt 2.1.3, „Export“). Denn beide können nun eine gemeinsame Interface-Wrapper-Basisfunktionalität mit exakt demselben Event-Weiterleitungsmechanismus verwenden. Der vom Interface-Wrapper umschlossene Channel und somit auch die weiterzuleitenden Events bleiben konstant. Daher wird bei beiden Switchtypen ein und dieselbe Interface-Wrapper-Implementation und Event-Forwarder-Technik verwendet.

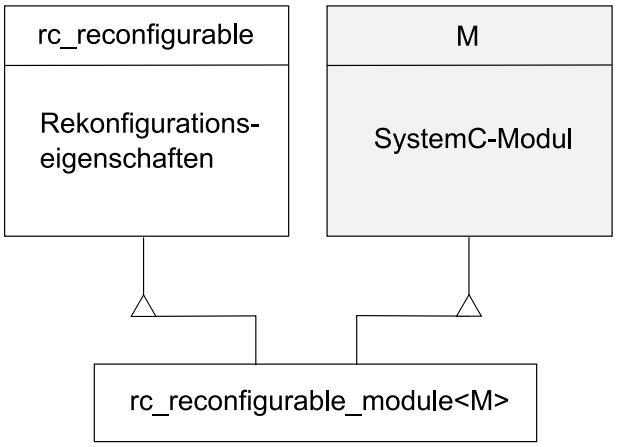

Abbildung 4.16: Um ein rekonfigurierbares Modul zu definieren, wird die Klasse `rc_reconfigurable_module` mit dem Typ eines abzuleitenden Moduls `M` parametrisiert. Die Ableitung von `rc_reconfigurable` und `M` wird hierdurch automatisiert und muss somit nicht manuell durchgeführt werden.

4.10 Rekonfigurierbare Module

Das Klassentemplate `rc_reconfigurable_module` repräsentiert in ReChannel ein rekonfigurierbares Modul (RM). Es dient dem Zweck, die bei der Erzeugung eines rekonfigurierbaren Moduls nötige Ableitung von `rc_reconfigurable` und einem SystemC-Modul zu automatisieren (siehe Abbildung 4.16). Für ein gegebenes statisches SystemC-Modul `M` repräsentiert der Typ `rc_reconfigurable_module<M>` dessen rekonfigurierbare Variante. Der erste Templateparameter ist das Modul, das in ein rekonfigurierbares verwandelt werden soll. Die Ableitung wird von der Klasse automatisch vorgenommen.

`rc_reconfigurable_module<M>` ist ohne Zutun des Benutzers ein verwendungsfertiges, rekonfigurierbares Modul. Dies wird mittels Templatekonstruktoren ermöglicht, mit denen beliebige Kontruktoren des ursprünglichen Moduls dupliziert werden. Alle Konstruktoren des Moduls `M` mit bis zu zehn Parametern können somit direkt verwendet werden. Zur Definition eines rekonfigurierbaren Moduls, an dem keine internen Anpassungen vorgenommen werden müssen, reicht bereits die folgende Codezeile aus:

```
typedef rc_reconfigurable_module<M> M_rc;
```

Einem Modul vom Typ `M_rc` können nach dessen Erzeugung beliebige Rekonfigurationszeiten mittels `rc_set_delay()` (siehe Abschnitt 4.3) zugewiesen werden.

Modifikationen oder Erweiterungen eines Moduls sind jedoch nur möglich, wenn das Modul durch Ableitung von der Klasse `rc_reconfigurable_module` definiert wird, wie in Listing 4.8 gezeigt. Dieses Beispiel dient dem direkten Vergleich mit der ursprünglichen ReChannel-v1-Notation, die in Listing 1.1 abgebildet ist. Eine Methode namens `rc_setup()` wird nicht mehr verwendet, da dem Anwender ansonsten eine unnötige, zusätzliche Konvention auferlegt würde.

```
class A_rc : public rc_reconfigurable_module<A> {
public:
    A_rc(sc_module_name name_)
        : rc_reconfigurable_module<A>(name_) {
        rc_set_delay(RC_LOAD, sc_time(20, SC_MS));
        rc_set_delay(RC_ACTIVATE, sc_time(1.5, SC_MS));
        rc_set_delay(RC_DEACTIVATE, sc_time(2, SC_MS));
    }
};
```

Listing 4.8: Definition eines rekonfigurierbaren Moduls namens `A_rc` aus einem bestehenden Modul `A` durch Ableitung von `rc_reconfigurable_module<A>`

```
RC_RECONFIGURABLE_MODULE_DERIVED(M_rc, M)
{
    RC_RECONFIGURABLE_CTOR_DERIVED(M_rc, M) {
        [...] // Initialisierungen
    }
    [...] // zusätzliche Komponenten, Variablen, Prozesse...
};
```

Listing 4.9: Definition eines rekonfigurierbaren Moduls namens `M_rc` aus einem bestehenden Modul `M` durch die Verwendung von ReChannel-Makros

```
template<class T, int W>
RC_RECONFIGURABLE_MODULE_DERIVED(M_rc, M<T>)
{
    RC_RECONFIGURABLE_CTOR_DERIVED(M_rc, M<T>) {
        [...] // Initialisierungen
    }
    [...] // zusätzliche Komponenten, Variablen, Prozesse...
};
```

Listing 4.10: Definition eines rekonfigurierbaren Modultemplates namens `M_rc` aus einem bestehenden Modultemplate `M<T>` durch die Verwendung von ReChannel-Makros

Für die Erzeugung eines RM werden von ReChannel auch Makros zur Verfügung gestellt, die der Definition eines Moduls in SystemC mittels `SC_MODULE` ähneln (siehe Listing 4.9). Bei diesen Makros kann im Gegensatz zu den Makros von ReChannel-v1 ein beliebiger Modulname angegeben werden. Auch können die Makros in ReChannel-v2 zur Defintion und Ableitung beliebiger Modultemplates verwendet werden, wie in Listing 4.10 illustriert.

Mehrere Templateparameter, z.B. `<T, W>`, müssen dem Makro allerdings mit folgender Syntax übergeben werden: `M<RC_PARAMS2(T, W)>`. Der Grund hierfür ist, dass ein Komma nicht ungeklammert einem Makro als Parameter übergeben werden kann. Die Makrofamilie `RC_PARAMSx()` steht zur Umgehung dieses Problems zur Verfügung. Diese Makros dienen der Klammerung von `x` Templateparametern. Es gibt sie für ein `x` von 2 bis 10. Um die Verwendung von `RC_PARAMSx()` zu umgehen, könnten hier auch die in C++ verfügbaren „Variadic Macros“ mit variabler Parameteranzahl verwendet werden. Doch wird dieser Weg nicht gegangen, da ReChannel die Kompatibilität mit dem Compiler *Microsoft Visual C++ 2003 .NET* gewährleisten soll. Mit diesem Compiler können „Variadic Macros“ nicht verwendet werden.

Wenn ein rekonfigurierbares Modul nicht von einem bestehenden Modul abgeleitet, sondern direkt definiert werden soll, dann kann `rc_reconfigurable_module<>` ohne die Angabe eines Templateparameters verwendet werden. Die folgende Codezeile definiert ein rekonfigurierbares Modul, das sich direkt auf `sc_module` zurückführt:

```
typedef rc_reconfigurable_module<> rekonfM;
```

Für die direkte Neudefinition sind ebenfalls Makros vorhanden, so dass dies für den Benutzer wie eine Moduldefinition in SystemC wirkt:

```
RC_RECONFIGURABLE_MODULE(M_rc) {
    RC_RECONFIGURABLE_CTOR(M_rc) { [...] }
    [...]
};
```

Der Anwender kann in einem rekonfigurierbaren Modul bestimmte Callback-Methoden überladen, die vom Rekonfigurationsalgorithmus jeweils aufgerufen werden, wenn ein bestimmter Rekonfigurationszustandswechsel erfolgt ist. Es stehen hierbei die Callback-Methoden `rc_on_load()`, `rc_on_activate()`, `rc_on_deactivate()` und `rc_on_unload()` zur Verfügung.

In einem RM gibt es ferner eine Methode namens `rc_find_object()`, die der SystemC-Funktion `sc_find_object()` zur Auffindung von Simulationsobjekten anhand ihres Namens entspricht. Der Unterschied besteht darin, dass diese Funktion auch relative Namen (wie in Abschnitt 2.2 beschrieben) auflösen kann.

4.11 Prozess-API

Eine der Zielsetzungen ist, dem Anwender die intuitive Beschreibung und Simulation von Prozessen zu ermöglichen, die bei der Deaktivierung eines rekonfigurierbaren Moduls automatisch in ihrer Ausführung abgebrochen werden. Bei der nächsten Aktivierung dieses Moduls sollen solche Prozesse ihre Ausführung wieder von vorne beginnen, so als ob sie gerade erst erzeugt worden wären.

ReChannel wird daher eine eigene Prozess-API hinzugefügt. Diese Prozess-API bietet Funktionen und Klassen, die die Simulation von rücksetzbaren Prozessen für die Simulation von DR ermöglichen. Der Kern dieser Funktionalität bildet eine Prozessregistrierung, bei der zusätzliche Informationen zu einem SystemC-Prozess abgespeichert werden können. Für jeden dort registrierten Prozess wird ein Datensatz angelegt, der u. a. einen Bool-Wert enthält, ob dieser Prozess rücksetzbar ist oder nicht, sowie einen Pointer auf das zu verwendende Rücksetz-Event.

Ein Prozess wird als „rücksetzbar“ bezeichnet, wenn dieser bei Eintreten einer bestimmten Rücksetzbedingung abgebrochen wird und daraufhin entweder direkt oder erst bei Auftreten eines bestimmten Aktivierungsereignisses neu gestartet wird.

Für einen Thread-Prozess bedeutet die Eigenschaft „rücksetzbar“, dass dieser bei Eintreten einer Rücksetzbedingung eine C++-Exception wirft, die ihn aus seiner aktuellen Ausführungsposition herauskatapultiert. Der C++-„Exception-Handling“-Mechanismus baut so lange den Funktionsaufrufstapel ab, bis eine Codeposition vorgefunden wird, die den entsprechenden Exception-Typ mittels eines Try-Catch-Blocks abfängt. Daher ist es notwendig, dass ein rücksetzbarer Prozess in seiner Einsprungfunktion auch ein solches Try-Catch-Konstrukt enthält, da ansonsten die gesamte Programmausführung abgebrochen wird. In der Einsprungfunktion des Prozesses gibt es daher eine Logik, die diesen Prozess z. B. auf die nächstmalige Aktivierung des zugehörigen RMs warten lässt und dann wieder von neuem mit der Ausführung fortfahren lässt. Im Falle des Thread-Prozesses läuft dieses Verfahren in einer unendlichen `while`-Schleife ab.

Ein Method-Prozess, der „rücksetzbar“ ist, hat hingegen ein anderes Verhalten. Dieser muss keine Exceptions verwenden, sondern es reicht, wenn dieser – nachdem er die Kontrolle an den SystemC-Kernel abgegeben hat – nicht mehr aktiviert wird, solange das RM deaktiviert ist. Hierzu ist ein Aufruf von `next_trigger()` am Ende der Einsprungfunktion erforderlich, mittels derer SystemC veranlasst wird, die Einsprungfunktion erst wieder bei der nächsten Aktivierung des RM aufzurufen.

Da idealerweise sämtliche für den Rücksetzvorgang nötige Logik für den Benutzer unsichtbar sein soll, kann die Notation eines Try-Catch-Konstruktes sowie die Angabe von zusätzlichen Events nicht vom Benutzer verlangt werden. Am vorteilhaftesten wäre es, wenn der Benutzer einen Prozess wie aus SystemC gewohnt deklarieren könnte, ohne sich um diesen Aspekt kümmern zu müssen. Um dies zu erreichen, werden von ReChannel Sprachkonstrukte zur Verfügung gestellt, die denen aus SystemC entsprechen (siehe Abschnitt 4.11.2). Die dahinter stehenden, internen Mechanismen bleiben dabei für den Anwender unsichtbar.

4.11.1 Eigenschaften des Resetmechanismus

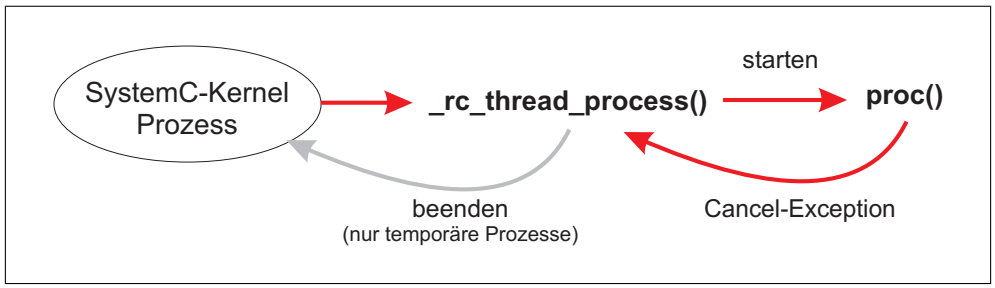

Abbildung 4.17: Prinzipielle Funktionsweise des Rücksetzmechanismus für Prozesse

Einsprungfunktion

Bei der Erzeugung eines Prozesses durch ein ReChannel-Sprachkonstrukt wird formal eine benutzerdefinierte Einsprungfunktion angegeben, doch wird anstelle dessen tatsächlich eine von ReChannel vorgegebene Einsprungfunktion mit der für den Rücksetzmechanismus benötigten Zusatzlogik verwendet (vgl. Abbildung 4.17). Diese Einsprungfunktion ermittelt anhand der Identität des Prozesses die zugehörige Benutzerfunktion aus einer internen HashMap (siehe Abschnitt 4.16) und kann diese somit aufrufen.

Präparierte Wartefunktionen

Da ein Hauptprinzip von ReChannel ist, keine Anpassungen am SystemC-Kernel vorzunehmen, muss die Rücksetzbedingung eines Prozesses auf Anwendungsebene (d.h. innerhalb der Simulation) überprüft werden. Daher werden alle 26 `next_trigger()`- und `wait()`-Methoden, die ein Prozess innerhalb eines RM aufrufen kann, speziell präpariert, indem sie überladen und auf die globalen Wartefunktionen der Prozess-API, `rc_wait()` und `rc_next_trigger()`, umgeleitet werden. In diesen Funktionen wird immer zusätzlich implizit auf das Event für die Rücksetzbedingung gewartet, da ansonsten das Eintreten der Rücksetzbedingung nicht bemerkt werden könnte. Im Falle einer erfüllten Rücksetzbedingung wird dort eine Exception ausgelöst (siehe Listing 4.11).

Initialisierung

Prozesse können in SystemC die Eigenschaft zugewiesen bekommen, dass sie nicht schon in der Initialisierungsphase ein Mal ausgeführt werden sollen. Diese Eigenschaft wird mit der Methode `dont_initialize()` deklariert. Sollte ein als „nicht zu initialisierend“ deklarierter Prozess rücksetzbar sein, so darf dieser bei einer Aktivierung des zugehörigen RM ebenfalls nicht sofort automatisch ausgeführt werden. Da dieses RM aus Sicht der Simulationssemantik von ReChannel vor dessen Aktivierung nicht existiert hat, muss eine solche Nicht-Initialisierung auch von ReChannel respektiert werden, um die korrekte Ausführung des Prozesses und damit die Korrektheit der Simulation zu gewährleisten.

```
void rc_wait(const sc_event& e)
{
  // Abfrage des Handles des aktuellen Prozesses
  rc_process_handle hproc = rc_get_current_process_handle();
  // Ist der Prozess rücksetzbar?
  if (hproc.is_cancelable()) {
    // Ist die Rücksetzbedingung bereits erfüllt?
    if (hproc.is_canceled()) {
      // Exception auslösen um Prozess abzubrechen
      throw new rc_process_cancel_exception();
    } else {
      // warte auf das Event e oder das Rücksetz-Event
      ::sc_core::wait(e | hproc.get_cancel_trigger_event());
      // Prozess abbrechen, falls Rücksetzbedingung erfüllt
      if (hproc.is_canceled() || hproc.is_cancel_event()) {
        // Exception auslösen um Prozess abzubrechen
        throw new rc_process_cancel_exception();
      }
    }
  } else { // der Prozess ist nicht rücksetzbar
    // warte auf das Event e
    ::sc_core::wait(e);
  }
}
```

Listing 4.11: Für jede SystemC-Wartefunktion steht auch eine spezielle ReChannel-Version mit „rc_Prefix zur Verfügung, um die Simulation von rücksetzbaren Prozessen zu ermöglichen. Der obige Codeausschnitt zeigt die Implementation der `rc_wait()`-Methode mit einem Event `e` als Übergabeparameter. Nichtrücksetzbare Prozesse warten ausschließlich auf das Event `e`, wohingegen rücksetzbare Prozesse zusätzlich auf ein spezielles Rücksetz-Event warten, das den Prozess aufweckt, sobald die Rücksetzbedingung erfüllt ist. Im Fall einer erfüllten Rücksetzbedingung wird eine Exception hervorgerufen, um die Ausführung des Prozesses an der aktuellen Position abzubrechen.

Dynamische Prozesse

In SystemC können nicht nur statisch definierte Prozesse, sondern auch die so genannten dynamischen Prozesse während der Simulation mittels der Funktion `sc_spawn()` erzeugt werden. Im Zusammenhang mit der Deaktivierung eines RM müssen solche während der Simulation erzeugten Prozesse als temporär angesehen werden. Daher werden mit ReChannel-Konstrukten erzeugte, temporäre Prozesse beendet, wenn ihr zugehöriges RM deaktiviert wird. Andernfalls bestünde die Möglichkeit, dass sich bei jedem Neustart des RM neue Prozess-Exemplare anhäufen.

Terminierungsevents

In SystemC besteht ferner die Möglichkeit, einen Prozess auf die Beendigung eines anderen Prozesses warten zu lassen. Wartet ein Prozess des statischen Designbereichs auf die Beendigung eines rücksetzbaren Prozesses, so muss ersterer benachrichtigt werden, wenn das RM deaktiviert wird, denn dann wird der Prozess zurückgesetzt und gilt laut Simulationssemantik als „nicht existent“. Ein Terminierungsevent muss demnach ausgelöst werden, obwohl der zugehörige Prozess tatsächlich noch existiert. Da das Terminierungsevent eines SystemC-Prozesses nicht von einer Anwendung ausgelöst werden darf, stattet ReChannel einen rücksetzbaren Prozess mit einem eigenen Terminierungsevent aus. Falls es sich um einen temporären Prozess handelt, ist dies nicht erforderlich, da der Prozess tatsächlich beendet wird und das originale Terminierungsevent von SystemC somit direkt verwendet werden kann.

CThread-Prozesse

In SystemC besitzen CThread-Prozesse bereits Rücksetzfunktionalität. Es ist möglich, einen solchen Prozess bei Auftreten eines bestimmten Signalwerts neu beginnen zu lassen. Dieser so genannte Reset, ist mit der Clock des Prozesses synchronisiert und tritt daher immer auf, wenn sich die entsprechende Taktflanke ereignet. Wenn ReChannel diese Rücksetz-Funktionalität für seine Zwecke nutzen würde, müsste es sich die Benutzung mit dem Anwender teilen, was zu Problemen führen kann. Zudem wäre hierbei ein Reset-Signal erforderlich, das ein eigenes Timing-Verhalten besitzt (Request/Update). Des Weiteren ist der CThread-Prozess in SystemC ein sehr eingeschränkter Thread-Prozess mit Sonderregeln. Es ist z. B. keine Änderung der dynamischen Sensitivity-List mittels `wait()` möglich, was das einfache Warten auf ein Aktivierungsereignis unmöglich macht. Die CThreads von SystemC können daher nicht zusammen mit der Prozess-API von ReChannel verwendet werden.

Rücksetzbare Thread-Prozesse

Aus den im vorherigen Absatz genannten Gründen wird die Designentscheidung getroffen, rücksetzbare CThread-Prozesse in ReChannel durch Thread-Prozesse zu emulieren. Die in SystemC bereits existierende Rücksetzfunktionalität wird durch ReChannel simuliert, indem der Rücksetzmechanismus für die Simulation von DR auch für einen Rück-

setzmechanismus auf Anwenderseite verallgemeinert wird. Die Flexibilität von Thead-Prozessen erlaubt dann darüber hinaus auch die temporäre Änderung der Rücksetzbedingung, was mit echten CThread-Prozessen nicht möglich gewesen wäre.

Darüber hinaus ist gleichzeitig auch eine benutzerdefinierte Rücksetzfunktionalität für reguläre Thread-Prozesse möglich. Da diese Funktionalität nachträglich in die Prozess-API eingefügt worden ist, bestehen bei der Verwendung mit benutzerdefinierten Rücksetzbedingungen noch die auf S. 112 aufgeführten Einschränkungen. Da es sich hierbei um optionale Zusatzfunktionalität handelt, die direkt nichts mit der Simulation von DR zu tun hat, wird eine Behebung dieser Problematik als mögliche, zukünftige Änderung angesehen.

Primäre und sekundäre Rücksetzbedingungen

Jeder mittels ReChannel-Sprachkonstrukten in einem RM erzeugte Prozess besitzt immer als primäre Rücksetzbedingung das Eintreten des Deaktivierungsereignisses des RM. Befindet sich ein Prozess in einem nicht rekonfigurierbaren Modul, so entfällt die primäre Rücksetzbedingung.

Weiterhin können sekundäre Rücksetzbedingungen, die an ein Signal (bzw. dessen Signalpegel, "high" oder „low") gebunden sind, vom Anwender hinzugefügt werden. Dies kann mittels der aus SystemC bekannten Methode `reset_signal_is()` erfolgen. Diese in jedem SystemC-Modul enthaltene Methode wird von ReChannel überladen und kann vom Anwender somit wie gewohnt verwendet werden (siehe dazu auch Abschnitt 4.11.2).

Obwohl einem rücksetzbaren Thread- oder CThread-Prozess beliebig viele sekundäre Rücksetzbedingungen zugewiesen werden können, stehen alle Rücksetzbedingungen mit ein und demselben Rücksetz-Event in Verbindung. Dies ist das Event, auf das der Prozess in den präparierten Wartefunktionen zusätzlich wartet (siehe Listing 4.11).

Rücksetzverhalten

Um das Rücksetzverhalten von CThread-Prozessen korrekt simulieren zu können, wird zwischen synchron und asynchron rücksetzbaren Prozessen unterschieden. Ein zur Simulation eines rücksetzbaren CThread-Prozesses verwendeter Thread-Prozess wird synchron zu seiner Clock zurückgesetzt. Dies geschieht auf dieselbe Weise wie in SystemC. Eine Ausnahme hiervon bildet allerdings die primäre Rücksetzbedingung, da sie auch bei einem synchron rücksetzbaren Prozess zu einem unmittelbaren Abbruch führt.

Ein rücksetzbarer Thread-Prozess wird asynchron zu seiner aktuellen Sensitivity-List zurückgesetzt. Der Unterschied zum synchronen Rücksetzverhalten ist, dass nur das Eintreten einer sekundären Rücksetzbedingung (Signalwertänderung) zu einem Reset führt. Der Prozess wird daher nicht andauernd zurückgesetzt, solange ein Reset-Signal den entsprechenden Wert hat und der Prozess eine `wait()`-Funktion aufruft. Dies verhindert, dass der Prozess ununterbrochen zurückgesetzt wird und die Simulationszeit nicht mehr voranschreiten kann.

Verwendungshinweise und Einschränkungen

Bei Verwendung der simulierten, rücksetzbaren Clocked-Thread-Prozesse sollte auf Anwenderseite darauf geachtet werden, dass – wie es SystemC für diese Art von Prozessen vorschreibt – keine Wartefunktionen aufgerufen werden, die die Sensitivity-List dynamisch ändern. Andernfalls wäre das Rücksetzverhalten vorübergehend asynchron.

Eine Einschränkung im Zusammenhang mit benutzerdefinierten Rücksetzbedingungen besteht bei der Verwendung von asynchron rücksetzbaren Thread-Prozessen, die eine statische Sensitivity-List besitzen. Dieses Problem ist darauf zurückzuführen, dass es in SystemC weder eine Möglichkeit gibt, die in der statischen Sensitivity-List enthaltenen Events zu ermitteln, noch die statische Sensitivity-List dynamisch um ein (Rücksetz-)Event zu ergänzen. Ein Aufruf einer Wartefunktion ohne Angabe eines Parameters bewirkt, dass ein Prozess auf seine voreingestellte, statische Sensitivity-List wartet. Dies hat zur Folge, dass das sekundäre Rücksetzverhalten zwangsläufig vorübergehend auf „synchron“ wechselt. „Synchron“ ist das Verhalten aus dem Grund, dass die Rücksetzbedingung zu dem Zeitpunkt überprüft wird, an dem der Prozess durch seine voreingestellte Sensitivity-List aufgeweckt wird.

Die oben beschriebene Einschränkung hat allerdings nur eine Auswirkung auf die sekundären (d.h. benutzerdefinierten) Rücksetzbedingungen. Im Zusammenhang mit der Deaktivierung eines RM besteht die Beschränkung nicht, da das zu einem RM gehörige Deaktivierungsereignis immer zusätzlich auch in der statischen Sensitivity-List jedes rücksetzbaren Prozesses eingetragen ist. Das Deaktivierungsereignis kann in der statischen Sensitivity-List enthalten sein, da die primäre Rücksetzbedingung im Gegensatz zu den benutzerdefinierten Rücksetzbedingungen nicht dynamisch ein- und ausgeschaltet werden soll.

Reset-Signale

Die Verwendung der Rücksetzfunktionalität in Kombination mit Signalen erfordert einen Signaltyp, der über spezielle Fähigkeiten verfügt. Das Signal muss eine Liste über die Prozesse verwalten, die von diesem zurückgesetzt werden. Wenn sich der Wert des Signals ändert, müssen die Rücksetz-Events derjenigen Prozesse benachrichtigt werden, die bei diesem Signalwert zurückgesetzt werden sollen. Die SystemC-Rücksetzfunktionalität, die ein Signal in SystemC besitzt, kann nicht genutzt werden, da deren Verwendung auf Anwendungsseite vom Sprachstandard nicht vorgesehen ist.

Wenn ein SystemC-Signal verwendet werden soll, um einen mittels ReChannel-Sprachkonstrukten deklarierten Prozess zurückzusetzen, so muss hierfür ein zusätzliches rechannelinternes Hilfssignal (`reset_signal`) erzeugt werden. Dieses neue Signal besitzt die Fähigkeit zum Zurücksetzen der Prozesse und ist mit dem SystemC-Signal verbunden. Für die Verbindung sorgt ein Method-Prozess, der die Events des SystemC-Signals an das ReChannel-Signal übermittelt. Der Benutzer bekommt hiervon nichts mit. Der einzige spürbare Effekt ist, dass ein Reset, das durch ein SystemC-Signal ausgelöst wird, einen Delta-Zyklus später eintritt. Diese Verzögerung scheint im Regelfall vertretbar zu sein. Eine Delta-Zyklus-Verzögerung eines mit einer Clock synchronisierten Reset-Signals

sollte für eine stabile und korrekte Hardwarebeschreibung (z. B. auf RT-Ebene) kein Problem darstellen.

Die von ReChannel bereitgestellten Signale, `rc_signal<bool>` und `rc_buffer<bool>` (siehe Abschnitt 4.12.2), besitzen standardmäßig die Fähigkeit, mit der Prozess-API verwendet zu werden. Wird eins der Rechannel-eigenen Signale verwendet, wird daher kein Hilfssignal benötigt.

`reset_signal_is()` kann auch zusammen mit Ports vom Typ `sc_in<bool>` verwendet werden. Hierbei besteht die Möglichkeit, dass ein solcher Port nicht mit einem Signal, sondern mit einem Signal-Accessor vom Typ `rc_accessor<sc_signal_in_if<bool> >` oder `rc_accessor<sc_signal_inout_if<bool> >` verbunden ist. Diese beiden Accessortypen besitzt zwar ebenfalls die Fähigkeit mit der Prozess-API verwendet zu werden, doch muss hierfür zusätzlich noch ein interner Method-Prozess angelegt werden. Dieser wird benötigt, um den Accessor über eine Signalwertänderung in Kenntnis zu setzen. Der Hilfsprozess wird jedoch nur bei Bedarf erzeugt.

Warten auf Und-verknüpfte Event-Listen

Obwohl auch präparierte Varianten für die vier `wait()`- und `next_trigger()`-Funktionen mit einer Und-verknüpften Event-Liste als Parameter zur Verfügung stehen, sollten diese sparsam verwendet werden und wenn möglich vermieden werden. Wegen der technischen Beschränkung von SystemC, dass Events nicht gleichzeitig mit „&“ und „|“ verknüpft werden können, wird von diesen Funktionen bei jedem Aufruf intern ein temporärer Thread-Prozess und ein Event-Objekt angelegt. Mithilfe derer wird simuliert, dass das Event der Rücksetzbedingung und die Event-Liste mit „|“ verknüpft sind. Durch die Erzeugung des temporären Prozesses besteht hier somit ein extremer Performance-Unterschied im Vergleich zu den anderen Funktionen.

4.11.2 Verfügbare Sprachkonstrukte

Die Prozess-API von ReChannel ist mit eigenen Versionen aller SystemC-Funktionen und -Konstrukte ausgestattet, die mit Prozessen zu tun haben. Diese Versionen besitzen – abgesehen von deren Prefix („rc_“ statt „sc_“) – den gleichen Namen wie ihre SystemC-Pendants.

Bevor in einem Modul rücksetzbare Prozesse verwendet werden können, muss diese Funktionalität erst einmal mittels `RC_HAS_PROCESS()` bereitgestellt werden. Hierdurch wird ein Hilfsobjekt der Klasse `process_support` im Modul angelegt, auf das alle in einem Modul verwendeten Prozessfunktionen und -makros intern zugreifen. In diesem Objekt ist die Verwaltungslogik enthalten, die zur korrekten Funktionsweise der Sprachkonstrukte benötigt wird. `RC_HAS_PROCESS()` überlädt weiterhin alle `wait()`- und `next_trigger()`-Methoden des Moduls. Diese werden daher auf die globalen, präparierten Wartefunktionen der Prozess-API (`rc_wait()` bzw. `rc_next_trigger()`) umgeleitet. Auch werden z. B. die Methoden `dont_intialize()` und `reset_signal_is()` überladen, um alle Aufrufe dieser Methoden mitzubekommen.

Nachdem `RC_HAS_PROCESS()` deklariert worden ist, können im Modul die Prozessdeklarationsmakros von ReChannel (`RC_THREAD()`, `RC_CTHREAD()` und `RC_METHOD()`) sowie `rc_spawn()` verwendet werden. Diese wurden derart implementiert, dass sie auf die selbe Weise wie ihre SystemC-Entsprechungen verwendet werden können. Intern verwenden diese Konstrukte SystemC-Makros und -Funktionen, um einen Prozess zu erzeugen und sie legen für diesen anschließend einen Datensatz mit allen nötigen Informationen in der Prozess-Registrierung von ReChannel an. Die SystemC-Sprachkonstrukte werden hierbei standardkonform eingesetzt. Es gibt entsprechende Logik, die dafür sorgt, dass ReChannel- und SystemC-Makros in einem Modul auch beliebig gemischt werden können. Für die korrekte Zuordnung von Aufrufen (wie z. B. `dont_initialize()`) zum entsprechenden Prozess (des entsprechenden Moduls) ist hierbei gesorgt. Die Verwendung der Prozessdeklarationsmakros, `RC_THREAD()`, `RC_CTHREAD()` und `RC_METHOD()`, wird in Abschnitt 4.12.1 im Zusammenhang mit der expliziten Modellierung von DRHW illustriert.

Erzeugung temporärer Prozesse

Für `rc_spawn()` steht in ReChannel der Datentyp `rc_spawn_options` zur Verfügung, der von `sc_spawn_options` abgeleitet ist. Zusätzlich zu der geerbten Grundfunktionalität bietet `rc_spawn_options` darüber hinaus noch die Verwendung der Methode `set_reset_signal()`, die die Deklaration von Reset-Signalen auch für dynamische Prozesse ermöglicht. Ein mittels `rc_spawn()` während der Simulationszeit erzeugter Thread-Prozess wird als „temporärer Prozess“ angesehen, der bei der Deaktivierung des RM automatisch beendet wird (vgl. Abschnitt 4.11.1). Wenn `rc_spawn()` hingegen zur Konstruktionszeit aufgerufen wird, ist der erzeugte Thread-Prozess nicht temporär.

Da es in SystemC grundsätzlich nicht möglich ist, Method-Prozesse zu terminieren, kann ein Method-Prozess nicht „temporär“ sein. Daher muss bei der Erzeugung von Method-Prozessen mittels `rc_spawn()` speziell auf diese Besonderheit geachtet werden.

Reset noch im selben Delta-Zyklus

Zusätzlich zu der Möglichkeit, Reset-Signale zu verwenden, wird dem Anwender auch eine Simulationskomponente namens `rc_process_reset` zur Verfügung gestellt. Mit dieser kann mittels einer Methode namens `trigger()` ein Rücksetzereignis für asynchron rücksetzbare Thread-Prozesse ausgelöst werden, das noch im selben Delta-Zyklus stattfindet und nicht an einen bestimmten Signalwert gebunden ist.

Abfrage von Prozess-Eigenschaften

Mit der ReChannel-Funktion `rc_get_current_process_handle()` kann (analog zu SystemC) das Prozess-Handle des aktuell ausgeführten Prozesses erfragt werden. Über das zurückgelieferte Prozess-Handle (`rc_process_handle`) sind alle Informationen über einen rücksetzbaren Prozess abfragbar. Das zuletzt ermittelte Handle wird intern zwischengespeichert, damit bei mehrmaligen Aufrufen nicht jedesmal erneut eine Suche in der Prozess-Registrierung durchgeführt werden muss.

```
RC_NO_RESET {
    // in diesem Code-Bereich ist sämtliches
    //  benutzerdefiniertes Rücksetzverhalten abgeschaltet
    [...]
}
```

Listing 4.12: Mit `RC_NO_RESET()` kann ein Codebereich eingeklammert werden, in dem alle benutzerdefinierten Rücksetzbedingungen ausgeschaltet sind.

Änderung des Rücksetzverhaltens

Ein Prozess kann in ReChannel auch temporär sein Rücksetzverhalten ändern. Ein `rc_process_handle`-Objekt besitzt hierzu eine Methode namens `behavior_change()`, die den Datensatz des Prozesses in der Registrierung ändern kann. Der Hauptverwendungszweck der Methode `behavior_change()` ist die vorübergehende Abschaltung der gesamten oder nur der sekundären Rücksetzfunktionalität.

Damit diese Änderung wirklich nur vorübergehend ist, liefert diese Methode ein Objekt vom Typ `rc_process_behavior_change` zurück. Dieses Objekt enthält alle Informationen über den originalen Zustand des Datensatzes (24 Bytes) und darf nur in einer lokalen Variable gespeichert werden. Wird der Gültigkeitsbereicht der lokalen Variable verlassen, so sorgt der Destruktor des Objekts dafür, dass der vorherige Zustand des Registrierungsdatensatzes wiederhergestellt wird.

Um Anwendern eine komfortable Nutzung dieser Funktionalität zu ermöglichen, wird das Makro `RC_NO_RESET` zur Verfügung gestellt (siehe Listing 4.12). Mit `RC_NO_RESET` kann ein Bereich eingeklammert werden, für den alle sekundären Rücksetzbedingungen ausgeschaltet sein sollen. Die Klammerung wird technisch mit einem einmalig ausgeführten `for`-Konstrukt ermöglicht, in dem ein lokales `rc_process_behavior_change`-Objekt angelegt wird.

Wenn zum ersten Mal mittels `rc_get_current_process_handle()` ein Prozess-Handle eines nichtrücksetzbaren Prozesses abgefragt wird, so wird in der Prozessregistrierung ebenfalls ein Datensatz angelegt. Dies dient dem Zweck, nichtrücksetzbare und rücksetzbare Prozesse auf einheitliche Weise verwenden zu können. Dies impliziert, dass auch mittels SystemC-Makros deklarierte Prozesse nachträglich ein temporäres Rücksetzverhalten erhalten können. Dies kann für bestimmte Zwecke sinnvoll sein, in denen ein Prozess einen Code-Bereich verlassen haben muss, sobald ein bestimmtes Ereignis auftritt.

4.12 Explizite Beschreibung von DRHW

Da die Sprache SystemC während der Simlationszeit keine Änderungen an der Modul- oder Verbindungsstruktur eines Designs zulässt, wird in ReChannel die dynamische Rekonfiguration dadurch simuliert, dass zwischen verschiedenen Modulen hin und her geschaltet wird. In Kapitel 2.1.4 und 2.1.5 sind die Probleme diskutiert worden, die sich bei der Verwendung von bestehenden Modulbeschreibungen mit ReChannel ergeben. Auf funkionaler und transaktionaler Abstraktionsebene ist die Wahrscheinlichkeit hoch, dass ein Modul in einer Verhaltensbeschreibung vorliegt, die sich nicht mit der Simulationssemantik von ReChannel vereinbaren lässt.

Um nicht auf die gewohnten Beschreibungsstile verzichten zu müssen, werden in ReChannel-v2 Sprachkonstrukte zur expliziten Modellierung von DRHW hinzugefügt. Mit der Verfügbarkeit von rücksetzbaren Prozessen (siehe Abschnitt 4.11) ist die Hauptvoraussetzung für diese Funktionalität gegeben. Zusätzlich dazu sind auch Konstrukte für die Modellierung von rekonfigurierbaren Strukturen vorhanden, wie z. B. rücksetzbare Module und Channels. Diese Erweiterungen versetzen den Anwender in die Lage, beliebige, rekonfigurierbare Verhaltens- und Strukturbeschreibungen modellieren zu können, die die folgenden Eigenschaften aufweisen:

- implizit rücksetzbar (d.h., ohne dass der Anwender sich hierum kümmern muss)
- intuitive, leicht erlernbare Notation aufgrund starker Ähnlichkeit zu SystemC
- auf allen Abstraktionsstufen modellierbar
- nahtlos in das ReChannel-Konzept integrierbar
- Verwendung auch außerhalb der Simulation von DR möglich
- mit SystemC-Beschreibungen verträglich / gemischte Nutzung möglich
- gewohnter Design-Flow / Refinement wie gewohnt durchführbar
- Synthese mittels Standard-Tools durchführbar

4.12.1 Implizit rücksetzbares Verhalten

Prozesse sind vermutlich die Hauptursache, warum SystemC eine statische Modul- und Verbindungsstruktur besitzt. Da Prozesse beliebig über Modul- und Channel-Grenzen hinweg operieren können, ist es nicht möglich, während der Simulation z. B. einen Channel zwischen zwei Modulen zu entfernen. Wenn eine Datenstruktur gelöscht würde, während auf dieser ein Prozess operiert, hätte dies fatale Folgen für eine Simulation.

In der SystemC-Implementation der OSCI sind intern zwar bereits Prozesskontrollmethoden, wie z. B. `suspend()` und `kill_process()`, vorhanden, doch sind diese den Anwendern nicht zugänglich. Es ist zu vermuten, dass diese Methoden nicht in die Sprache aufgenommen wurden, da diese ebenfalls sehr leicht zu unkontrollierbarem und undefiniertem Verhalten in einer Simulation führen können.

```
// Deklaration eines rücksetzbaren Method-Prozesses
RC_METHOD(method_proc);
sensitive << in1 << in2;

// Deklaration eines rücksetzbaren Threads
RC_THREAD(thread_proc);
dont_initialize();

// Deklaration eines rücksetzbaren CThreads
RC_CTHREAD(cthread_proc, clk.pos());
reset_signal_is(reset);
```

Listing 4.13: Durch Verwendung der Prozessdeklarationsmakros von ReChannel kann rekonfigurierbares Verhalten in zu SystemC analoger Weise modelliert werden.

ReChannel-v2 geht mit seinen Sprachkonstrukten (siehe Listing 4.13) einen Mittelweg, der zwar einigen Einschränkungen unterliegt, aber dafür sicheres Prozessverhalten und die Simulation dynamischer Rekonfiguration ermöglicht:

- Das bestehende Problem mit modulübergreifend operierenden Prozessen und der Simulation von DR ist in ReChannel dadurch gelöst, dass Switches (Portals oder Exportals) zwischen den rekonfigurierbaren und statischen Design-Bereichen eingezogen werden. Die Switches verhinden durch das Zählen von Transaktionen, dass eine Modulverbindung gelöst wird, während noch externe Zugriffe bestehen. Die primäre Rücksetzbedingung (=Deaktivierung des RM) eines rücksetzbaren Prozesses ist somit bei Verlassen eines RM inaktiv.

- Weiterhin sorgen die Switches dafür, dass auch die sekundären Rücksetzbedingungen ausgeschaltet werden, sobald die Ausführung eines Prozesses einen Switch passiert. Dies ist notwendig, um unvorhergesehenes Verhalten und undefinierte Zustände in statischen SystemC-Modulen zu vermeiden.

- Prozesskontrolloperationen, wie z. B. ein Aufruf von `kill_process()` für einen beliebigen Prozess, sind in ReChannel grundsätzlich nicht möglich, da die Bibliothek auf Änderungen am SystemC-Kernel verzichtet. Doch dafür ist es mit ReChannel möglich, einen Prozess in einem lokal kontrollierten Bereich mit einer Rücksetzfähigkeit auszustatten.

  Nur in den präparierten `wait()`- und `next_trigger()`-Methoden führt eine erfüllte Rücksetzbedingung zu einem Abbruch des Prozesses mittels einer Exception. Da Rücksetzbedingungen somit nur innerhalb von Komponenten mit präparierten Wartemethoden gelten, die auf rücksetzbare Prozesse ausgerichtet und dementsprechend exceptionsicher programmiert sind, können viele Probleme vermieden werden.

```
class rc_resettable
{
   // fordert das Objekt zum Zurücksetzen seines Zustands auf
   virtual void rc_on_reset() = 0;
   // wird aufgerufen, wenn das Objekt seinen Zustand sichern soll
   virtual void rc_on_init_resettable() = 0;
};
```

Listing 4.14: Rücksetzbare Objekte implementieren das ABI `rc_resettable`

4.12.2 Implizit rücksetzbare Objekte

Wenn ein RM deaktiviert wird, müssen nicht nur dessen Prozesse, sondern auch dessen Struktur implizit zurückgesetzt werden können. Zudem ist es wünschenswert, wenn alle potentiell blockierenden Channels, die ein rücksetzbarer Prozess aufruft, ebenfalls wie das RM über präparierte Wartefunktionen verfügen würden. Zu diesem Zweck werden in ReChannel die so genannten **rücksetzbaren Objekte** eingeführt.

Ein Objekt gilt als „rücksetzbar“, wenn dieses das ABI `rc_resettable` (siehe Listing 4.14) implementiert. Solche Objekte werden beim rekonfigurierbaren Kontext mittels der Methode `rc_register_resettable()` registriert, um im Deaktivierungsfall implizit zurückgesetzt zu werden.

Der **rekonfigurierbare Kontext** ist ein rekonfigurierbares Objekt, d.h. ein Objekt vom Typ `rc_reconfigurable`. Das nächste Modul in der Modulhierarchie oberhalb eines Simulationsobjektes X, dass von `rc_reconfigurable` abgleitet ist, wird im Folgenden als der rekonfigurierbare Kontext des Objektes X bezeichnet. Wenn der rekonfigurierbare Kontext deaktiviert wird, soll die gesamte darin enthaltene Objekt- und Modulstruktur zurückgesetzt werden. Der rekonfigurierbare Kontext kann mittels der Methode `rc_get_reconfigurable_context()` für jedes Objekt ermittelt werden.

Die Methode `rc_on_reset()` wird von ReChannel unmittelbar vor der Aktivierung und unmittelbar nach der Deaktivierung eines rekonfigurierbaren Kontextes aufgerufen, um das Objekt zu einem sofortigen Rücksetzvorgang aufzufordern. Das Objekt wird zu diesen zwei Zeitpunkten zurückgesetzt, da bei gemischten Beschreibungen (siehe Abschnitt 4.12.4) zwischenzeitliche Änderungen möglich sind.

Zu Beginn der Simulation (*start_ of_ simulation*) wird ein Mal auf jedem registrierten, rücksetzbaren Objekt die Methode `rc_on_init_resettable()` aufgerufen. Dies dient dem Zweck, den Zeitpunkt von außen festlegen zu können, an dem das Objekt seinen Zustand sichern soll, auf den es später zurückgesetzt werden soll.

ReChannel besitzt rücksetzbare Varianten aller SystemC-Channel und -Komponenten, wie z. B. `rc_module`, `rc_signal`, `rc_event`, usw. Um eine konsistente Notation mit dem „rc_“-Präfix zu gewährleisten, gibt es z. B. auch Typdefs und Makros für die Verwendung aller SystemC-Ports. Als Vorlage für die Implementation der Channels diente der Code der in der SystemC-Implementation enthaltenen Channels. Auch für die Basisklassen `sc_prim_channel` und `sc_channel` gibt es eine Entsprechung in ReChannel mit „rc_“-Präfix. Diese sind von den SystemC-Klassen abgeleitet und besitzen neben

einigen Hilfsmethoden u. a. auch die Fähigkeit sich automatisch beim rekonfigurierbaren Kontext zu registrieren. Eine Liste der von ReChannel zur Verfügung gestellten Komponenten findet sich in Anhang B.

Bei jedem der von ReChannel vordefinierten Channels wird bei einem Reset sichergestellt, dass dieser exakt denselben Zustand besitzt wie zu Beginn der Simulation. Bei einer FIFO bedeutet dies, dass deren initialer Inhalt wiederhergestellt wird. Im Falle von Resolved-Signalen wird exakt dieselbe Wert- und Treiberprozesszuordnung wiederhergestellt. Die in ReChannel enthaltenen Resolved-Signal-Ports sind von den jeweils entsprechenden SystemC-Ports abgeleitet, doch können jene auch mit Accessoren verbunden werden. Somit besteht bei der expliziten Modellierung von DRHW nicht die in Abschnitt 2.1.7.e beschriebene Einschränkung. Für sämtliche SystemC-Makros stehen auch ReChannel-Varianten zur Verfügung. So kann ein Modul z. B. auch mittels der Makros `RC_MODULE()` und `RC_CTOR()` definiert werden.

Mit den Sprachkonstrukten zur expliziten Beschreibung von DRHW können sowohl allgemein verwendbare, rücksetzbare Module von Grund auf neu modelliert werden, als auch bestehende, rekonfigurierbare Module mit rücksetzbarem Verhalten und rücksetzbaren Strukturen erweitert werden. Dies ist über die Trennung von `rc_module` und `rc_reconfigurable` möglich.

Rekonfigurierbare Module

Ein `rc_module` ist nicht automatisch von `rc_reconfigurable` abgeleitet. Der Grund hierfür ist, dass diese Art von Modul auch im nichtrekonfigurierbaren Kontext (d.h. im statischen Fall) verwendet werden soll. Um ein rücksetzbares Modul bei der Simulation von DR zu verwenden, gelten in ReChannel dieselben Regeln wie für reguläre SystemC-Module, d.h. auch hier ist die Verwendung von `rc_reconfigurable_module` notwendig. `rc_reconfigurable_module` besitzt die Funktionalität, die notwendig ist, um den rekonfigurierbaren Kontext für alle enthaltenen Objekte effizient zu ermitteln. Hier wird ein Mechanismus verwendet, der auch einen rekonfigurierbaren Kontext erkennt, obwohl dessen Konstruktion noch im Gange ist.

Synchronisation

Um eine Transaktion zu definieren, die ausgeführt werden muss, bevor das RM deaktiviert werden kann, steht das Makro `RC_TRANSACTION` zur Verfügung. Mit diesem kann der Codebereich eingeklammert werden, der zu einer Transaktion zusammengefasst werden soll. Das Makro ist durch eine (einmalig ausgeführte) `for`-Schleife realisiert, die zu Beginn den Transaktionszähler des RM um eins erhöht und bei Verlassen der Schleife diesen wieder um eins erniedrigt. Am Schluss wird implizit ein Aufruf der Methode `rc_possible_deactivation()` durchgeführt. Die Methode dient dem Zweck, einen Thread-Prozess anzuhalten, falls das Modul den Befehl zur Deaktivierung bekommen hat und alle Transaktionen beendet sind. Weiterhin steht auch eine „weichere“ Variante namens `rc_possible_deactivation_delta()` zur Verfügung. Diese hält den Prozess nur bis zum nächsten Delta-Zyklus an. Sollte am Ende des aktuellen Delta-Zyklus eine

```
RC_MODULE(MyModule)
{
    rc_in<bool> clk;
    rc_in<bool> reset;
    rc_in<int>  in;
    rc_out<int> out;

    rc_signal<bool> sig1;
    rc_signal<int>  sig2;

    rc_event e;

    RC_CTOR(MyModule)
    {
        RC_METHOD(method_proc);
        sensitive << in << e;

        RC_THREAD(thread_proc);
        sensitive << clk;

        RC_CTHREAD(cthread_proc, clk.pos());
        reset_signal_is(reset);
    }

    void method_proc();
    void thread_proc();
    void cthread_proc();
};
```

Listing 4.15: Beispiel: Ein mit ReChannel-Konstrukten modelliertes Modul

```
while(true) {
  a = portA.read();     // Eingabe von Port A lesen
  RC_TRANSACTION {      // Transaktion beginnen
    b = portB.read();     // Eingabe von Port B lesen
    c = berechnen(a, b); // Berechnungen durchführen
    portC.write(c);       // Ausgabe Ergebnisses an Port C
  }                     // Transaktion beenden
  // <-- implizites rc_possible_deactivation()
}
```

Listing 4.16: Der Prozess des rekonfigurierbaren ABC-FIFO-Moduls aus Abbildung 2.1 (S. 33), versehen mit einer `RC_TRANSACTION`-Klammer zur Verhinderung von Dateninkonsistenzen

Deaktivierung nicht möglich sein, wird der Prozess an dieser Stelle weiter ausgeführt. Ferner können auch die in Kapitel 4.16 beschriebenen Transaktionszähler-Komponenten für Synchronisierungszwecke verwendet werden.

Für explizit beschriebene RM besteht daher die Möglichkeit, dass sie sich selbständig mit der Rekonfigurationskontrolle synchronisieren, wie in Listing 4.16 demonstriert.

Benutzerdefinierte Komponenten

Ausgehend von dem Code einer bestehenden Channel-Klasse ist lediglich das Interface `rc_resettable` zu implementieren und die Basisklasse von `sc_prim_channel` (bzw. `sc_channel`) auf `rc_prim_channel` (bzw. `rc_channel`) zu ändern. Hierbei ist darauf zu achten, dass im Channel ausschließlich die in der Basisklasse überladenen `wait()`- und `next_trigger()`-Methoden aufgerufen werden und nicht z. B. `sc_core::wait()`. Da die Methode `rc_on_init_resettable()` den initialen Zustand des Channels sichern soll, sind für die Speicherung entsprechende Datenelemente zu deklarieren. Die Methode `rc_on_reset()` muss den Code enthalten, der den Rücksetzvorgang durchführt, indem der initiale Zustand wiederhergestellt wird.

4.12.3 Rücksetzbare Variablen

In ReChannel-v1 kann mittels der Methode `rc_reset()` ein Wert registriert werden, auf den eine Variable in einem rekonfigurierbaren Modul zurückgesetzt werden soll. Der Nachteil der hierbei verwendeten Technik ist, dass eine zusätzliche Syntax für die Zuweisung eines Initialwertes eingeführt wird. Einerseits kann eine Variable auf die in C++ übliche Weise initialisiert werden, andererseits muss noch einmal `rc_reset()` verwendet werden, um den Initialwert für die Rekonfiguration zu setzen. Auch eine Mehrfachregistrierung ist bei dieser Implementierungsvariante leicht möglich, da die Methode an mehreren Stellen im Code (z. B. Konstruktor und `rc_setup()`) aufgerufen werden kann. Wenn ein Benutzer die Rücksetzwerte der Variable eines bestehenden Moduls nachträglich ändern möchte, dann muss dieser die Methode `rc_setup()` überladen. Wenn hierbei die überladene Methode aufgerufen wird, muss eine Mehrfachregistrierung in Kauf genommen werden. Wird diese nicht aufgerufen, ist zusätzlicher Aufwand notwendig, da alle restlichen in der Basisklasse vorgenommenen Initialisierungen und Deklarationen wiederholt werden müssen. Dabei besteht die Möglichkeit, dass die Rücksetzbarkeit einer Variable absichtlich oder unabsichtlich wieder zurückgenommen wird. Abgesehen davon stellt die Einführung der Methode `rc_setup()` aus Benutzersicht eine zusätzliche Konvention dar, die beachtet werden muss.

Für die Sicherung und Wiederherstellung des Wertes primitiver Datentypen, Structs sowie einiger Objekte kann der Kopierkonstruktor und der Zuweisungsoperator ("`=`") verwendet werden. Aufgrund dessen kann eine generische Lösung gefunden werden, die eine Variable eines solchen Typs kapselt und das `rc_resettable`-Interface implementiert.

Die Variable in einem Objekt zu kapseln, kann zu Unterschieden bei der Verwendung bzw. dem Zugriff auf den Wert dieser Variablen führen. Ein Grund hierfür ist, dass alle Operatoren überladen werden müssen, die für einen primitiven oder zusammengesetzten Datentyp definiert sind. Einige Operatoren wie z. B. der „.“-Operator können in C++ jedoch nicht überladen werden. Somit sind bei dieser Lösung Einschränkungen bei der Benutzung unvermeidbar.

Es stellt sich daher die Frage, wie eine Lösung aussehen könnte, die am wenigsten Aufwand für den Anwender bedeutet und gleichzeitig die Variable nicht in einem Objekt kapselt. Am Vorteilhaftesten wäre, wenn die rücksetzbare Variable einfach deklariert wird und daraufhin wie gewohnt im Konstruktor oder per Zuweisung initialisiert werden kann. Die zuletzt erfolgte Zuweisung vor dem Beginn der Simulation sollte automatisch derjenige Wert sein, auf den die Variable zurückgesetzt wird. Dabei wären folgende Eigenschaften wünschenswert:

- Auf die Variable soll wie bisher ein direkter Zugriff (d.h. ohne Verwendung eines kapselnden Objekts) möglich sein. Dies schließt auch die normale Verwendung der Konstruktor-Initialisierung sowie des Zuweisungs- und des Elementzugriffs-Operators („.“) mit ein.

- Die Registrierung der Variable beim rekonfigurierbaren Kontext muss für den Anwender vollständig unsichtbar durchgeführt werden.

- Zur Deklaration einer rücksetzbaren Variablen soll lediglich eine deklarierende Zeile im Klassenscope notwendig sein. Dabei soll kein Rücksetzwert spezifiziert werden müssen.
- Der Rücksetzwert einer Variable ist der zuletzt zugewiesene Wert vor Beginn der Simulation.
- Die Syntax der eingeführten Sprachkonstrukte soll ähnlich der von C++ und SystemC her gewohnten sein.

Die Schwierigkeit besteht darin, dass eine zurückzusetzende Variable mit dem ABI `rc_resettable` in Verbindung stehen muss. Aus diesem Grund ist es unvermeidbar, dass ein zusätzliches Objekt erzeugt werden muss. Die normale Vorgehensweise wäre, dass diesem Objekt eine Referenz auf die Variable im Konstruktor übergeben wird. Dies soll aber nicht nötig sein, da der Benutzer dies angeben müsste und es somit der Anforderung widerspräche. Dieses Problem wird in ReChannel-v2 mittels Makros und Templatetechniken gelöst.

Ein Makro `rc_var()` wird implementiert (siehe Listing B.5 auf S. 173), das als Deklarationsmakro für rücksetzbare Variablen dient. Die Aufgabe von `rc_var()` ist, die Variable und das zusätzliche Objekt zu deklarieren. Das Makro verlangt den Typ und Namen der Variablen als Parameter, wie in folgender Codezeile illustriert:

```
rc_var(int, myVar);
```

Das Makro definiert intern eine Hilfsmethode, die eine Referenz auf die deklarierte Variable zurückliefert. Die Klasse des zusätzlich deklarierten Objekts implementiert das ABI `rc_resettable` und trägt den Namen `resettable_var<>`. Bei `resettable_var<>` handelt es sich um ein Klassentemplate, das mit dem Typ des umgebenden Moduls, dem Typ der Variable und dem Member-Pointer auf die Methode parametrisiert wird. Über den Klassentyp des Moduls und den Pointer auf die Methode erhält das Objekt Zugriff auf die Variable. Der Vorteil dieser Technik liegt darin, dass keine explizite Konstruktorinitialisierung durch den Benutzer erforderlich ist. Zusätzlich wird auch die Registrierung des `rc_resettable`-ABIs beim rekonfigurierbaren Kontext automatisch durchgeführt.

Die Implementation von `resettable_var` verwendet zwei Basisklassen und ist darauf optimiert, möglichst wenig Speicherplatz zu verbrauchen. Die Basisklassen dienen speziell dem Zweck, die Codegröße zu verringern, die durch die Verwendung dreier Templateparamter hervorgerufen wird. Generell gilt die Regel, dass von einem Compiler pro Templateklasse der Code aller Methoden des Klassentemplates erzeugt wird. Für ein einzelnes Modul gilt, dass der Overhead von `resettable_var` nicht direkt proportional zur Anzahl der deklarierten Variablen ist, sondern proportional zur Anzahl der unterschiedlichen Variablentypen ist. Daher wird einer vereinfachten Verwendung bzw. Syntax der Vorzug gegenüber einer einfacheren Implementierung gegeben und der durch den Template-Code verursachte Speicher-Overhead in Kauf genommen.

Da `resettable_var<>` mit der Klasse des umgebenden Moduls parametrisiert wird, muss dem Makro `rc_var` dieser Typ bekannt sein. Dies erfordert die einmalige Angabe von `RC_HAS_VAR()` im Klassenscope, wodurch ein entsprechendes `typedef` gesetzt

Deklarationsmakro	Beschreibung
rc_var()	Deklariert eine Variable (rücksetzbar)
rc_declare_var()	Deklariert eine existierende Variable der Basisklasse als „rücksetzbar“
rc_resettable_var()	Synonym für rc_var()
rc_preservable_var()	Deklariert eine Variable (nicht rücksetzbar)

Tabelle 4.5: Die in ReChannel-v2 zur Verfügung stehenden Makros zur Deklaration rücksetzbarer Variablen im Klassenscope. Um optional auch die in ReChannel-v1 verwendete, explizite Unterscheidung von rücksetzbar und nichtrücksetzbar anhand der Syntax zu ermöglichen, können optional auch die Makronamen rc_preservable_var() und rc_preservable_var() verwendet werden. Zu jedem der obenstehenden Makros existiert auch eine Variante in Großschreibung.

wird. Dies funktioniert analog zu dem Verfahren, das hinter dem SystemC-Makro `SC_HAS_PROCESS()` steht. Da es bei Deklarationen im Klassenscope auf die Reihenfolge ankommt, muss `RC_HAS_VAR()` vor sämtlichen `rc_var`-Deklarationen notiert werden. Die Compilerkonstante, die in `RC_HAS_PROCESS()` und `RC_HAS_CTOR()` deklariert wird, kann nicht wiederverwendet werden, da diese Makros an jeder beliebigen Position stehen könnten. Andernfalls wäre diese zusätzliche Bedingung auch an die Verwendung dieser Makros gebunden.

Listing 4.17 zeigt ein Beispiel für die Verwendung von rücksetzbaren Variablen in einem `RC_MODULE`.

```
RC_MODULE(myModule)
{
    RC_CTOR(myModule) : j(false) {
        i = 0;
    }
protected:
    RC_HAS_VAR(myModule);
    rc_var(int, i);
    rc_var(bool, j);
};
```

Listing 4.17: Der Codeausschnitt illustriert die Verwendung von rücksetzbaren Variablen in einem `RC_MODULE`. Die beiden als „rücksetzbar“ deklarierten Variablen, `i` und `j`, können wie gewöhnliche Variablen verwendet werden. Der Rücksetzwert während der Simulation entspricht dem Wert, den die Variablen am Ende der Konstruktionsphase besitzen. Wie in obigem Beispiel ersichtlich, können die Makros `RC_HAS_VAR()` und `rc_var()` trotz der Verwendung von externen Hilfsklassen auch im nichtöffentlichen Bereich des Moduls (d.h. „protected“ und „private“) notiert werden.

4.12.4 Gemischte Beschreibungen

Wenn in einem Design ausschließlich rücksetzbare Komponenten verwendet werden, gibt es keine Einschränkungen bezüglich des Rücksetzverhaltens der Prozesse. Das Problem bei einer gemeinsamen Verwendung von ReChannel-Prozessen mit statischen SystemC-Komponenten ist, dass in den statischen Komponenten keine präparierten Wartefunktionen vorhanden sind und Prozesse somit temporär ihr Rücksetzverhalten verlieren.

Ein Anwender hat entsprechend darauf zu achten, dass sich rücksetzbare Prozesse bei der gemischten Verwendung von SystemC und ReChannel-Komponenten in einem RM korrekt verhalten. Z. B. sollte vermieden werden, dass ein rücksetzbarer Prozess in einem statischen Channel blockiert.

Auch muss beachtet werden, dass das Rücksetzverhalten wieder aktiv ist, wenn ein präparierter Bereich verlassen wurde und über den Umweg durch eine statische Beschreibung wieder betreten wird. In einem solchen Fall sollten die Rücksetzbedingungen mittels `RC_NO_RESET()` (siehe Abschnitt 4.11.2) vorübergehend ausgeschaltet werden, wenn ein Zugriff auf einem Bereich ohne präparierte Wartefunktionen durchgeführt wird.

4.12.5 Refinement und Synthese

Das Refinement ist bei explizit beschriebener DRHW auf dieselbe Weise möglich, wie dies bei SystemC-Beschreibungen der Fall ist. Die Sprachkonstrukte von ReChannel decken den vollen Sprachumfang von SystemC ab, so dass auf jeder Abstraktionsebene modelliert werden kann.

Die Synthese von ReChannel-Beschreibungen ist ohne ein spezielles Tool möglich. Hier können die verfügbaren Standard-Synthese-Tools verwendet werden. ReChannel stellt eine Header-Datei namens „synthesis_header.h“ zur Verfügung, die eingebunden werden soll, wenn die Synthese durchgeführt wird. Die Header-Datei hat den Effekt, dass alle ReChannel-Sprachkonstrukte, die eine Entsprechung in SystemC besitzen, mittels Präprozessormakros in ihre SystemC-Pendants umbenannt werden. Wenn die normale ReChannel-Header-Datei eingebunden ist, dann kann alternativ auch einfach die Compilerkonstante `RC_SYNTHESIS_MODE` definiert werden. Daraufhin wird dann automatisch ausschließlich die Synthese-Header-Datei inkludiert.

4.13 Synchronisation der DR

Die Analyse der Synchronisationsproblematik bei der Simulation von DR (siehe Kapitel 2.1.1) hat ergeben, dass eine mögliche Lösung des Problems die folgenden drei Schlüsselfähigkeiten beinhalten muss:

a) die Bestimmung des inneren Modulzustands anhand externer Zugriffe,

b) die Möglichkeit zur Abblockung von beliebigen Zugriffen sowie

c) die Definition von Transaktionen für Eingabe- und Ausgabezugriffe.

Im Folgenden soll erörtert werden, wie diese drei Anforderungen in einem allgemeinen Lösungskonzept vereinigt werden können und welche Sprachkonstrukte und Synchronisierungsfunktionalität dem Anwender von ReChannel bereitgestellt werden müssen.

In ReChannel kann aus einem beliebigen statischen SystemC-Modul mittels Ableitungstechniken ein rekonfigurierbares Modul (RM) erzeugt werden. Da die Modulbeschreibungen auf beliebigen Abstraktionsebenen vorliegen können, darf nicht davon ausgegangen werden, dass diese immer über ein `busy`-Signal oder ähnliches verfügen. Daher muss eine Lösung gefunden werden, die unabhängig von derartigen Voraussetzungen ist. Die einzige Gemeinsamkeit, die alle rekonfigurierbaren Module teilen, ist, dass die Kommunikation in Form von Interface Method Calls (IMCs) erfolgt. Die Begriffe „IMC“ und „Zugriff“ werden im Folgenden synonym verwendet.

Eine Hauptanforderung an ReChannel ist, dass die Definition eines rekonfigurierbaren Moduls ohne Änderung oder Kenntnis des Codes des statischen Moduls erfolgen kann. Die Konsequenz daraus ist, dass keine direkten Kenntnisse über den inneren Zustand des statischen Moduls vorausgesetzt werden dürfen. Hinsichtlich dessen könnte es z. B. auch möglich sein, die Synchronisation von extern verbundenen Channels übernehmen zu lassen.

Daher ist die erste Frage, die beantwortet werden muss, ob Synchronisationsfunktionalität innerhalb oder außerhalb des Moduls anzusiedeln ist. „Außerhalb“ des Moduls würde bedeuten, dass eine Kontrolle sämtliche Kenntnis über alle möglichen Modulbeschreibungen besitzen müsste. Jedoch ist außerhalb des Moduls meist nur die Spezifikation der Kommunikationsschnittstelle und des verwendeten Protokolls bekannt. Ob ein Modul mit seiner Berechnung fertig ist oder ob noch Daten über einen Bus zu lesen sind, kann eine äußere Kontrolle im allgemeinen Fall nicht für alle Module wissen. Diese Entscheidung ist daher im Modul anzusiedeln. Jedem Modul muss somit seine eigene Synchronisationsfunktionalität hinzugefügt werden können.

Die nächste Frage ist die nach der genauen Position, an dem die Kommunikation abgegriffen werden soll, um den inneren Zustand des Moduls bestimmen zu können. Da sich die Synchronisationslogik innerhalb des rekonfigurierbaren Moduls befinden muss, scheint es von Vorteil zu sein, die Kommunikation möglichst ortsnah zu analysieren. Die nächstmöglichen, in Frage kommenden Stellen sind der Accessor oder ein aktuell mit diesem verbundener Interface-Wrapper (siehe Abschnitt 4.7). Die Kommunikation

direkt im externen Channel zu überwachen, scheint nicht sinnvoll zu sein, da hierfür eine zusätzliche Schnittstelle für den Informationsaustausch existieren müsste. Zudem würde dies Änderungen am Channel notwendig machen, die die ReChannel-Bibliothek von einem Anwender nicht verlangen kann.

Eine naheliegende Idee ist, unmittelbar vor oder nach externen Zugriffen bestimmte **Callback-Funktionen** aufzurufen. Dies könnte z. B. durch den Interface-Wrapper geschehen, da dieser bereits über Callback-Funktionalität verfügt. Mittels Callback-Funktionen wäre es dann möglich, zum Zeitpunkt eines Zugriffs bestimmte Aufgaben durchzuführen, wie z. B. das Zählen von Transaktionen oder die sofortige Deaktivierung eines Moduls.

Die Verwendung von **vordefinierten Callback-Funktionen** hat jedoch den entscheidenden Nachteil, dass hierüber nur sehr bedingt Informationen über den Zugriff selbst übergeben werden können. Vor seiner Ausführung ist ein Zugriff charakterisiert durch eine Interfacemethode und beliebig viele Übergabeparameter. Nachdem der Zugriff auf dem Channel durchgeführt worden ist, kann ein Rückgabewert eines beliebigen Typs vorliegen oder es sind Werte von Referenzparametern geändert worden. In allen diesen Fällen scheint es schwierig bis unmöglich zu sein, die entsprechenden Informationen an eine vordefinierte Callback-Funktion zu übergeben und dort anschließend analysieren zu lassen. Beispielsweise gehen durch das Einbetten eines IMCs in ein generisches Objekt von außen betrachtet entweder alle Information über die Typen verloren oder das Objekt selbst besitzt wiederum einen variablen Templatetyp.

Als Lösung für dieses Problem scheint das Konzept des **Filters** geeignet zu sein. Ein Synchronisations-Filter wird als zusätzliches Interface zwischen die Kommunikation eingeschleust und „filtert“ somit unmittelbar die Kommunikation. Die Callback-Funktionen sind bei einem Filter identisch mit den Interfacemethoden. Der einzige Nachteil eines Filter-Interfaces ist, dass dieses vom Anwender definiert werden muss. Speziell die Implementierung einer korrekten IMC-Weiterleitung ist hierbei problematisch. In jedem Filter müsste zusätzliche Verwaltungslogik zur Verfügung stehen, die für die Zuweisung des Kommunikationsziels zuständig wäre.

4.13.1 Der Accessor als Synchronisations-Filter

Ein Filter-Interface steht in ReChannel bereits in Form des Accessors (siehe Abschnitt 4.8) zur Verfügung. Die Interfacemethoden des Accessors leiten einen Zugriff an ein Kommunikationsziel weiter, wie es auch die Aufgabe eines Filters ist. Der Benutzer kann einen Accessor überladen und diesen mit beliebiger Synchronisationsfunktionalität ergänzen. Sowohl vor als auch nach dem Zugriff kann benutzerdefinierter Code ausgeführt werden. Diesem Code stehen alle Informationen über die IMCs unmittelbar zur Verfügung und es kann entsprechend darauf reagiert werden. Da für die Verwendung von Portal und Exportal ein Accessor definiert sein muss, steht somit immer automatisch auch ein Filter zur Verfügung.

Folgende Vorteile ergeben sich durch die Verwendung des Accessors als Grundlage für den Synchronisations-Filter:

- Die vollständige Weiterleitungsfunktionalität des Filters ist bereits vorhanden.

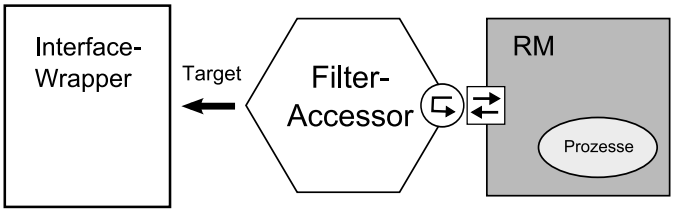

Abbildung 4.18: Der Accessor als Synchronisations-Filter

- Nicht alle Interfacemethoden müssen für die Defintion eines Filters überladen oder mit zusätzlichem Code ausgestattet werden.
- Die wait()-Methoden sind standardmäßig mit deren präparierten Varianten überladen. Die Verwendung von benutzerdefinierter Wartefunktionalität zusammen mit rücksetzbaren ReChannel-Prozessen ist somit gewährleistet.
- Eventmethoden und die dazugehörigen Events sind bereits definiert. Der Anwender kann diese Events dazu verwenden, einen Prozess warten zu lassen. Wenn der Event-Forwarder eines Switches die Events des Filter-Accessors benachrichtigt, werden diese aufgeweckt.
- Ein solcher Filter ist vor einen Interface-Wrapper schaltbar.

Die Position des Filter-Accessors stellt sich als günstig heraus (siehe Abbildung 4.18), da sich dieser aus Sicht eines zugreifenden Prozesses noch vor dem Interface-Wrapper befindet. Daher ist der Transaktionszähler im Switch noch nicht automatisch erhöht worden, wenn der Filter-Code ausgeführt wird. Der Transaktionszähler des Moduls kann somit frei für die Erzeugung von Transaktionen für die Eingabe- und Ausgabezugriffe verwendet werden.

Um eine korrekte Funktionsweise des Filter-Mechanismus sicherstellen zu können, sollen für die Definition von Filter-Accessoren die folgenden Regeln gelten:

1. Zur Weiterleitung von Zugriffen muss die überladene Interfacemethode des Accessors aufgerufen werden.
2. Der Filter sollte ausschließlich Exception-sicheren Code enthalten.
3. Nicht erlaubt sind Änderungen des Accessor-Targets.
4. Ebenfalls nicht erlaubt sind Änderungen am Rücksetzverhalten der Prozesse (siehe dazu Abschnitt 4.13.3).

Die Frage, die sich noch stellt, ist, wie die zu verwendenden Filter-Accessoren bei den Switches registriert werden sollen. Die Switch-Implementationen haben bisher selbständig eine Accessor-Implementation für sich wählen können. Diese Möglichkeit müsste z. B. im Fall des Portals entfallen, wenn der von einem Modul bereitgestellte Accessor-Filter den Platz des regulären Accessors einnehmen soll. Das Portal wird somit angepasst, dass

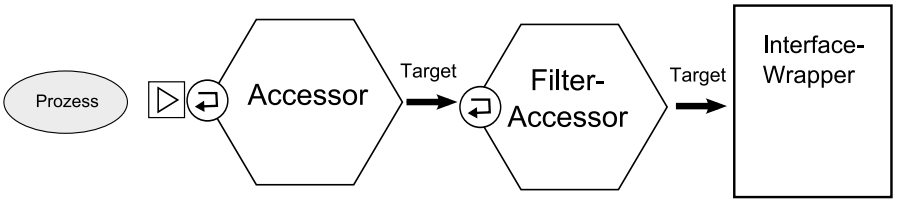

Abbildung 4.19: Filter-Kette beim Exportal

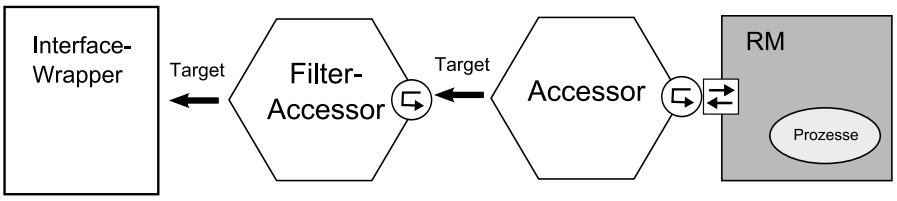

Abbildung 4.20: Filter-Kette beim Portal

es auch einen Accessor akzeptiert, der bereits an einem Modulport gebunden ist. In diesem Fall erzeugt das Portal keinen neuen Accessor, sondern übernimmt einfach den bestehenden.

Im Fall des Exportals besteht bezüglich dessen jedoch ein gravierendes, konzeptionelles Problem, wie im folgenden Abschnitt erörtert wird.

4.13.2 Filterkette

Wie beim Portal muss auch die Kommunikation eines Exportals gefiltert werden können. Dateninkonsistenzen, Deadlocks und Timing-Probleme müssen auch dort verhindert werden können. Da das Exportal jedoch einen einzelnen, festen Accessor auf der statischen Seite verwendet, kann dieser nicht einfach durch einen zu einem Modul gehörigen Filter-Accessor ausgetauscht werden.

In diesem Zusammenhang wäre es von Vorteil, wenn Accessoren untereinander verkettet werden könnten (siehe Abbildung 4.19 und 4.20). Ein Filter-Accessor kann so bequem als Kommunikationsziel eines beliebigen, bereits vorhandenen Accessors eingesetzt werden. Dazu müssen einem Accessor nun zwei Typen von Targets zugewiesen werden können: Ein Interface-Wrapper oder ein beliebiges, kompatibles Interface. Diese Erweiterung des Accessor-Konzepts ermöglicht es nun, Filterketten zu bilden und darüber hinaus den Accessor im allgemeinen Anwendungsfall noch flexibler einsetzen zu können. Da der Accessor von ReChannel-v2 im Gegensatz zum Accessor von ReChannel-v1 unabhängig von einem bestimmten Verwendungszweck einsetzbar ist, kann dieser auch als Filter-Interface in einer Kette verwendet werden.

Prozesse können nun nicht mehr nur in einem einzigen Accessor, sondern theoretisch an jedem beliebigen Glied in der Filter-Kette auf ein Event des Interfaces warten. Im Falle des Portals hat dies z. B. zur Folge, dass die Events des statischen Channels an die gesamte Filterkette weitergeleitet werden müssen. Somit wird der in Portal und Exportal

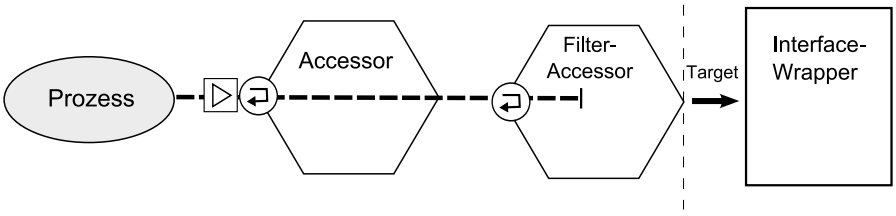

Abbildung 4.21: Ein Prozess, der in einem Filter blockiert

verwendete Event-Forwarder dahin gehend geändert, dass dieser nun auch eine ganze Filterkette mit Events versorgen kann.

4.13.3 Filterketten und wartende Prozesse

Durch den Einsatz von Filterketten wird jedoch ein neues Problem in Verbindung mit Exportals hervorgerufen. Wenn ein Accessor-Filter zwischen den statischen Accessor und den Interface-Wrapper geschaltet wird, befindet sich dieser Filter ebenfalls auf der statischen Seite. Die Prozesse dieser Seite greifen über den Filter zu und können dort beliebig blockiert werden (siehe Abbildung 4.21). Während der Filter einen statischen Prozess mittels `wait()` warten lässt, kann in der Zwischenzeit das Modul deaktiviert werden. Ein externer Zugriff, der die Deaktivierung eines Moduls verhindert, läge erst dann vor, wenn der Prozess den Interface-Wrapper erreicht hätte.

Spätestens wenn das Modul entladen wird, muss die bestehende Filterkette abgehängt werden. Um alle Filter unabhängig von ihrer jeweiligen Position in der Kette gleich zu behandeln, werden die Targets aller Filter entfernt und die Filterkette somit vollständig aufgelöst. Die Frage hierbei ist, was mit dem Prozess geschieht, der noch innerhalb des Filters wartet. Dieser Filter ist nun nicht mehr Teil der Filterkette. Der Prozess befindet sich aufgrund dessen in einem unverbundenen Filter eines entladenen Moduls. Um undefiniertes Verhalten zu vermeiden, muss der Prozess daher in den statischen Accessor zurückfallen und dort blockiert werden. Für einen solchen Mechanismus scheint nur die Verwendung von C++-Exceptions geeignet zu sein. Dies impliziert, dass ein Accessor-Filter beim Entladen (bzw. Deaktivieren) eines Moduls eine Exception in allen wartenden Prozessen auslösen können muss.

Hierbei kann die Funktionalität der Prozess-API von ReChannel verwendet werden, die es erlaubt, eine Rücksetzbedingung für einen beliebigen Prozess mittels der Methode `rc_process_handle::behavior_change()` zu ändern. Der Accessor wird um die Fähigkeit erweitert, das Rücksetzverhalten eines zugreifenden Prozesses dynamisch zu ändern. Welche Rücksetzbedingung hier zu nehmen ist, kann direkt bei der Zuweisung eines Accessor-Targets mittels `rc_set_target()` festgelegt werden. Da die Zuweisung einer Rücksetzbedingung optional ist, gibt es auch weiterhin noch die `rc_set_target()`-Methode ohne den zusätzlichen Übergabeparameter.

In den blockierenden Weiterleitungsmethoden des Accessors kann einem Prozess nun unmittelbar vor dem Aufruf des Targets eine bestimmte Rücksetzbedingung zugewiesen werden. Hierzu wird `behavior_change()` aufgerufen. Das von `behavior_change()`

zurückgelieferte, temporäre `rc_process_behavior_change`-Objekt wird in einer lokalen Variable gespeichert, denn solange dieses temporäre Objekt existiert, ist die neue Rücksetzbedingung gültig. In demselben Gültigkeitsbereich findet daraufhin der Aufruf auf dem Target-Interface statt. Wenn der aktuelle Gültigkeitsbereich anschließend verlassen wird, wird der Destruktor des temporären Objekts aufgerufen und die Rücksetzbedingung wird wieder zurückgenommen. Damit der Mechanismus wie gewünscht funktioniert, ist in jeder blockierenden Weiterleitungsmethode des Accessors ein Try-Catch-Block vorhanden, der die Cancel-Exceptions der Prozess-API abfangen kann. Der Catch-Block ist derart angelegt, dass eine Exception erneut ausgelöst wird, wenn die Rücksetzbedingung eines Prozesses erfüllt ist und andernfalls konsumiert wird.

Damit dies beim gegebenen Problem mit dem Exportal weiterhilft, wird dem statischen Accessor das Deaktivierungsereignis des rekonfigurierbaren Moduls als die zu verwendende Rücksetzbedingung zugewiesen. Wartet ein Prozess gerade in einem `wait()` innerhalb eines Accessor-Filters und wird währenddessen das Modul deaktiviert, dann ist die diesem Prozess zugewiesene Rücksetzbedingung damit erfüllt und in `wait()` wird infolgedessen eine Exception ausgelöst. Da bei Erreichen des Catch-Blocks des statischen Accessors die Rücksetzbedingung aufgrund der automatischen Zerstörung des temporären `rc_process_behavior_change`-Objekts wieder ausgeschaltet ist, wird der Prozess erst an dieser Stelle abgefangen. Der Prozess wartet schließlich wie gewünscht in einem `wait()` innerhalb des statischen Accessors.

Sollte der Prozess bereits vorher schon rücksetzbar gewesen sein, verlässt der Prozess gegebenenfalls dennoch den Accessor, da nun wieder seine ursprünglichen Rücksetzbedingungen gelten (die möglicherweise erfüllt sein könnten). Die Verwendung von rücksetzbaren Prozessen zusammen mit Filter-Ketten ist somit problemlos möglich.

Auflösen der Filterkette

Es wird festgelegt, dass bei allen Switches automatisch die Filterkette aufgelöst werden soll, wenn das zugehörige Modul deaktiviert wird. Der im vorangegangen Abschnitt beschriebene Mechanismus wird somit nicht nur beim Exportal, sondern auch beim Portal angewendet.

Der hauptsächliche Grund für diese Designentscheidung ist, dass sich alle Switchtypen in Bezug auf Filter gleich verhalten sollten. Andernfalls scheint eine allgemeine, vereinheitlichte Verwendung von Filtern nicht möglich zu sein.

4.13.4 Interface-Filter-ABI

Da Accessoren aus Sicht des Rekonfigurationsalgorithmus unbekannte Implementationsdetails darstellen, muss für den Accessor-Filter eine abstraktere Repräsentation gefunden werden. Um das Filter-Konzept auf allgemeine Weise in das Switch-ABI und die Klasse `rc_reconfigurable` integrieren zu können, wird ein ABI namens `rc_interface_filter` entworfen (siehe Listing 4.18). Dieses ABI besitzt eine Schnittstelle für das Zuweisen und Entfernen eines Kommunikationsziels (Target) und ist zudem von `sc_interface` abgeleitet. Um einen solchen abstrakten Filter einsetzen zu können, muss dieser zuvor auf einen

```
// Der Interface-Filter
class rc_interface_filter : virtual public sc_interface {
public:
    virtual sc_interface* rc_get_target_interface() const = 0;
    virtual bool rc_set_target(sc_interface& target) = 0;
    virtual void rc_clear_target() = 0;
    virtual rc_event_filter* rc_get_event_filter() = 0;
};

// Der Event-Filter
class rc_event_filter : virtual public sc_interface {
public:
    virtual bool rc_on_event(const sc_event& e) = 0;
    virtual void rc_set_event_trigger(
        const sc_event& e, rc_event_trigger& t) = 0;
    virtual void rc_clear_event_trigger(const sc_event& e) = 0;
};

// Der Event-Trigger (wird vom Event-Forwarder zur Verfügung gestellt)
class rc_event_trigger {
public:
    virtual void rc_trigger_event(const sc_event& e) = 0;
};
```

Listing 4.18: Die Filter-ABIs von ReChannel

bestimmten Interfacetyp gecastet werden. Sämtliche Typüberprüfungen sind in dieser Sichtweise den jeweiligen Implementationen vorbehalten.

Weiterhin besitzt das Interface-Filter-ABI eine Methode, die einen Pointer auf ein Event-Filter-ABI (siehe Abschnitt 4.13.5) zurückgeben soll, wenn der Filter auch Events filtern möchte. Das Event-Filter-ABI ist optional. Wenn eine Filter-Implementation eine solche Funktionalität nicht benötigt, soll die Methode den NULL-Pointer zurückliefern.

Das Accessor-ABI (und somit jeder Accessor) ist von `rc_interface_filter` abgeleitet.

4.13.5 Event-Filter-ABI

Filter müssen die Möglichkeit besitzen, auch Events filtern zu können, d.h. zu bestimmen, ob diese benachrichtigt werden sollen oder nicht. Nur wenn die Möglichkeit zur Filterung von Events vorhanden ist, können nichtblockierende Zugriffe effektiv abgeblockt werden. Ein Beispiel für eine Anwendung ist die Vortäuschung einer leeren FIFO. Dies ist nur auf korrekte Weise möglich, wenn auch die entsprechenden Events zur Ankündigung neuer Daten ausbleiben. Andernfalls läge ein Widerspruch zwischen der Event-Benachrichtigung und der von `nb_read()` zurückgelieferten Statusmeldung vor, der potentiell zu undefiniertem Verhalten führen kann.

Da es sehr ineffizient ist, sämtliche Events durch die Filter selbst weiterreichen (und somit „filtern“) zu lassen, soll anstelle dessen die Event-Filterung durch einen äußeren Mechanismus ermöglicht werden.

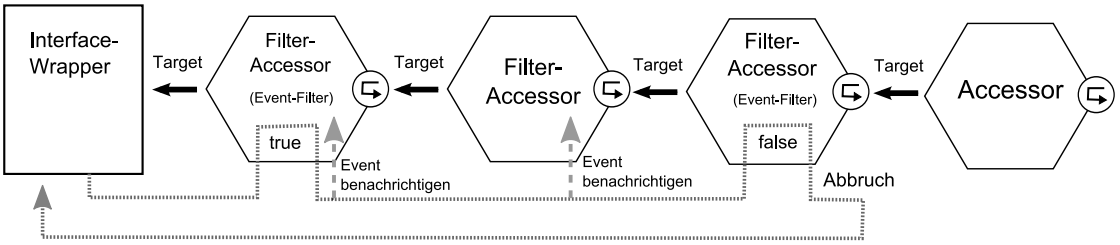

Abbildung 4.22: Der Event-Forwarder eines Interface-Wrappers benachrichtigt die Filter-Kette über Events. Der dritte Filter hat hier das Event geblockt.

Ein Event-Filter wird durch ein ABI namens `rc_event_filter` repräsentiert (siehe Listing 4.18). Dieses ist als spezielle Schnittstelle für die Filterung von Events konzipiert. Ein externer Event-Forwarder soll hierbei die Aufgabe übernehmen, die Event-Filter über alle stattfindenden Event-Benachrichtigungen zu informieren (siehe Abbildung 4.22). Hierzu soll die Callback-Methode `rc_on_event()` aufgerufen werden. Über den booleschen Rückgabewert dieser Methode hat der Event-Filter die Möglichkeit, die Weiterleitung des Events in der Filterkette entweder zu verhindern oder zuzulassen. Dies ist deshalb möglich, da die Methode aufgerufen wird, bevor das entsprechende Event benachrichtigt worden ist.

Auch ist es dem Event-Filter möglich, für jedes im Interface definierte Event einen so genannten **Event-Trigger** zu erlangen. Mit diesem Event-Trigger wird der Filter in die Lage versetzt in der Filterkette selbst aktiv Events auszulösen. Der Event-Trigger ist direkt mit dem Event-Forwarder verbunden, der für die Weiterleitung in dieser Filterkette zuständig ist. Ein auf diese Weise ausgelöstes Event wird ab der Position des auftraggebenden Filters beginnend in der Kette weitergeleitet. Die Weiterleitung von Events wird in der umgekehrten Reihenfolge zur Weiterleitung von IMCs durchgeführt.

Übernimmt ein Event-Forwarder die Aufgabe der Event-Weiterleitung für eine bestimmte Filterkette, so muss initial ein Mal jedem Event-Filter in der Kette der entsprechende Event-Trigger mittels `rc_set_event_trigger()` zugewiesen werden. Sobald der Event-Forwarder die Aufgabe wieder abgibt (z. B. wenn die Filterkette aufgelöst wird), soll der vormals zugewiesene Event-Trigger mittels `rc_clear_event_trigger()` wieder entfernt werden.

Die Events, die den Methoden des Event-Filter-ABIs sowie `rc_trigger_event()` übergeben werden, sind identisch mit den Events, die das Interface des Filters besitzt. Somit ist dem Filter sowie dem Event-Forwarder immer unmittelbar bekannt, um welches Event es sich handelt.

4.13.6 Verwaltung

Jedem Port und jedem Export muss individuell ein Interface-Filter zugewiesen werden können. Dadurch, dass ein Modul selbst ein Channel sein kann, muss auch diesem ein Interface-Filter zugewiesen werden können. Da Filterketten technisch möglich sind, ist es wünschenswert, auch den Anwender von dieser Fähigkeit profitieren zu lassen. Daher ist

es möglich, einer beliebigen Kommunikationskomponente beliebig viele Interface-Filter zuzuweisen.

Die Filterketten werden einem rekonfigurierbaren Objekt (RO) zugewiesen und dort verwaltet. Die Klasse `rc_reconfigurable` (die Basisklasse aller rekonfigurierbaren Objekte) besitzt zu diesem Zweck eine Methode namens `rc_add_filter()`. Der Anwender kann diese Methode aus einem Modul heraus aufrufen, um dem Modul einen Filter hinzuzufügen:

```
// fügt einem Port zwei Filter-Accessoren hinzu
rc_add_filter(din, *new myFilter(this));
rc_add_filter(din, *new myFilter2(this, data));

// fügt einem Export einen Filter-Accessor hinzu
rc_add_filter(exportA, *new myFilter(this));
```

Der Methode wird ein Interface-Filter-Objekt und ein beliebiges Kommunikationsobjekt (Port, Export oder Channel) übergeben, an das dieser Filter angehängt werden soll. Zur Speicherung der zu Kommunikationsobjekten gehörigen Filterketten verwendet `rc_reconfigurable` intern einen Map-Container.

Die Filterketten werden jeweils durch einen Vektor von Interface-Filter-Objekten repräsentiert. Für die Speicherung von Kommunikationsobjekten werden Objekte der Klasse `rc_object_handle` (siehe Abschnitt 4.14.3) verwendet, durch die die Verwaltung der unterschiedlichen Typen von Ports, Exports und Channels vereinheitlicht wird. Durch diese vereinheitlichte Sichtweise ist es möglich, alle diese Kommunikationsobjekte auf die gleiche Weise zu verwenden und das mit diesen in Verbindung stehende Interface einfach durch die Methode `rc_object_handle::get_interface()` abzufragen.

Das RO übergibt die Filterkette dem zugehörigen Switch jedesmal bei dessen Öffnung (d.h. mittels der Methode `open()`). Der Switch verbindet die Filterkette und löst diese später bei seiner Schließung wieder auf. Die Filterkette wird in Form eines Vektors übergeben. Die Reihenfolge ist hierbei fest vorgegeben. Der Interface-Filter mit dem Index 0 stellt dabei den Anfang der Filterkette dar (aus Sicht der IMCs). Beim Portal beispielsweise würde der Accessor den 0-ten Interface-Filter als Target zugewiesen bekommen. Das Target des letzten Interface-Filters in der Kette wäre schließlich der Interface-Wrapper.

Einer Filterkette muss ein neuer Filter auf der Seite des Kommunikationsziels angefügt werden, um zu verhindern, dass dieser Filter die bestehende Filterkette einfach überschreibt. Andernfalls könnten bestimmte Teile eines Synchronisationsmechanismus ausgeblendet werden. Dies kann leicht zu undefiniertem Verhalten in einem Modul führen. Die geltende Sichtweise ist daher, dass jeder neu eingefügte Filter nur das Verhalten des Kommunikationsziels modifiziert.

Methoden zur Entfernung eines Filters oder der Filterkette sind nicht verfügbar. Der Grund hierfür ist, dass Synchronisations-Filter dann auch nachträglich z. B. von einer höheren Ableitungsinstanz entfernt werden können. Die Gefahr, dass die Verwendung solcher Methoden zu undefiniertem Verhalten führt, wird als sehr hoch eingeschätzt.

4.13.7 Transaktionszähler

In ReChannel wird ein vordefinierter Transaktionszähler (`rc_transaction_counter`) für die komfortable Verwendung in Modulen bereitgestellt.

Ein Objekt der Klasse `rc_transaction_counter` ist mit der internen Transaktionszählvariable des rekonfigurierbaren Kontextes verbunden, in dem es erzeugt wurde. Jede Änderung des Zählers wirkt sich direkt auf die Transaktionszählvariable des entsprechenden Kontextes aus. Dennoch ist der Transaktionszähler ein unabhängiger Zähler. Wenn für einen rekonfigurierbaren Kontext N Transaktionszähler verwendet werden, ist die Regel, dass nur dann rekonfiguriert werden kann, wenn alle N den Wert 0 haben. Ein Transaktionszähler kann auch in den negativen Bereich zählen. Wenn c der momentane Wert eines Transaktionszählers ist, dann hat dieser im rekonfigurierbaren Kontext genau $|c|$ Transaktionen erzeugt. Die durch den Transaktionszähler repräsentierte Funktionalität kann als „implizites `busy`-Signal“ eines Moduls bezeichnet werden.

Ein Transaktionszähler kann z. B. Filtern übergeben werden, zusammen mit weiteren Informationen, wie diese Lese- und Schreibzugriffe zu zählen haben. Aber auch die direkte Verwendung innerhalb der Modulbeschreibung ist möglich.

Ein `rc_transaction_counter` kann in zwei Modi betrieben werden. Der Erste ist die bereits erwähnte Verwendung als eigenständiger Zähler. Der Zweite ist die gewichtete Zählung von Transaktionen in anderen `rc_transaction_counter`-Objekten. Dies kann zur Verteilung von Transaktionen auf mehrere Zähler verwendet werden. Dies ist z. B. sinnvoll, wenn einem Filter nur ein Transaktionszähler übergeben werden kann, aber dennoch in mehreren unterschiedlichen Transaktionszählern gezählt werden soll.

4.13.8 Verwendung von Filtern für die Synchronisation

In ReChannel stehen dem Anwender durch Filter und Transaktionszähler die grundlegenden Mittel zur Verfügung, um funktionale und transaktionale Modulbeschreibungen nachträglich mit beliebiger Synchronisationsfunktionalität ausstatten zu können:

1. Der Filter besitzt bereits alle Voraussetzungen, um feststellen zu können, ob und welche Daten gelesen oder geschrieben wurden. Dies ist hilfreich bei der Verhinderung von Dateninkonsistenzen bei Rekonfigurationsvorgängen (siehe Kapitel 2.1.1), da aufgrund dieser Informationen Eingabe- und Ausgabezugriffe in Transaktionen zusammengefasst werden können.

   Mittels Transaktionszählern kann einem Modul ein implizites `busy`-Signal für die Rekonfigurationskontrolle hinzugefügt werden. Nur wenn alle verwendeten Transaktionszähler 0 sind, kann das Modul deaktiviert werden. Zum Beispiel könnte ein Filter den Transaktionszähler nach dem Lesen eines Datenpaketes erhöhen und nach dem Schreiben eines berechneten Ergebnisses wieder erniedrigen. Im Verbund sorgen die Filter eines Moduls dann dafür, dass es nicht zu Dateninkonsistenzen kommen kann und ein Modul nur dann deaktiviert werden kann, wenn alle zu berechnenden Daten das Modul verlassen haben.

2. Das Timingproblem bei der Deaktivierung eines Moduls (siehe Kapitel 2.1.1) kann

dadurch gelöst werden, dass das erneute Einlesen von Eingabedaten verhindert wird, nachdem der Deaktivierungsbefehl gegeben wurde. Der exakte Zeitpunkt eines Deaktivierungsbefehls ist nicht entscheidend, da mittels Filtern ein Modulzustand künstlich herbeigeführt werden kann, in dem alle Transaktionen beendet sind (siehe dazu auch die Beispiele in Abschnitt 4.13.9).

Um Prozesse in blockierenden Zugriffen abblocken zu können, stehen die Methoden `rc_possible_deactivation()` und `rc_possible_deactivation_delta()` (siehe Kapitel 4.16) sowie das normale `wait()` zur Verfügung.

Nichtblockierende Zugriffe werden abgeblockt, indem die Rückgabewerte dieser Zugriffe sowie der dazugehörigen Statusmethoden manipuliert werden. Die Zugriffe werden dabei nicht zum Channel durchgelassen. Zudem kann mit dem Event-Filter dafür gesorgt werden, dass Method-Prozesse z. B. nur dann aktiviert werden, wenn nichtblockierende Zugriffe (wie z. B. das `nb_read()` einer FIFO) zum Erfolg führen. Sollte sich am Filter-Zustand etwas ändern, kann der Filter mittels eines Event-Triggers (siehe Abschnitt 4.13.5) auch selbst Events auslösen, um wartende Prozesse zu aktiveren.

3. Auch die Deadlockproblematik (siehe Kapitel 2.1.1) kann mittels Filtern gelöst werden, da externe Zugriffe beliebig kontrolliert werden können. Zusätzliche Filter sind jedoch nicht erforderlich, wenn der Standard-Accessor bereits mit entsprechender Funktionalität ausgestattet ist, wie dies z. B. bei den FIFO-Accessoren von ReChannel der Fall ist (siehe Abschnitt 4.8.4).

   Deadlocks im Zusammenhang mit Channels ohne Statusmethoden und Events (wie sie in Abschnitt 2.1.7.b beschrieben wurden) können ebenfalls durch `wait()`-Aufrufe im Filter verhindert werden. Allerdings ist der Filter hierbei auf eine äußere Kontrolle angewiesen, die ihm mitteilt, ob ein Zugriff erlaubt sein soll oder nicht, da eine direkte Anfrage an den Channel nicht möglich ist.

Für häufig benutzte Channeltypen können vorgefertigte, allgemein verwendbare Filter-Komponenten zur Verfügung gestellt werden, um Benutzern die Modellierung von Synchronisationsfunktionalität zu erleichtern.

In ReChannel stehen für die FIFO-, Semaphor- und Mutex-Interfaces bereits vordefinierte Filter zur Verfügung. Diese Filter leiten sich alle von einer Klasse namens `rc_abstract_prim_filter` ab, die eine Verwaltungsfunktionalität für Callback-Funktionen enthält und als Basisklasse für primitive Channels dienen soll. Diese Klasse ist wiederum von der Klasse `rc_abstract_filter` abgeleitet. In dieser Klasse sind grundlegende Infrastrukturen für Filter enthalten. Sie ist als Basisklasse für beliebige Filter-Implementationen konzipiert. Zum Funktionsumfang dieser Klasse gehören Hilfs- und Informationsmethoden sowie eine automatische Registrierung beim rekonfigurierbaren Kontext.

4.13.9 Anwendungsbeispiel: FIFO-Filter

Am Beispiel der beiden in ReChannel enthaltenen FIFO-Filter soll im Folgenden die praktische Verwendung von Synchronisations-Filtern illustriert werden.

1. Für einfache Datenflussmodelle auf UTF oder TF kann es ausreichend sein, wenn lediglich die Anzahl der Zugriffe registriert wird. Hierbei können ein oder mehrere Transaktionszähler (siehe Abschnitt 4.13.7) zum Einsatz kommen, um einer abstrakten Beschreibung ein implizites `busy`-Signal hinzuzufügen. Listing 4.19 illustriert dies an einem einfachen Beispiel mit einem einzelnen Transaktionszähler.

   Den FIFO-Filern werden Transaktionszähler und die Anzahl der bei jeder erfolgreichen Lese- bzw. Schreiboperation zu zählenden Transaktionen übergeben. Vor jedem Lesen ruft der FIFO-In-Filter einmal `rc_possible_deactivation()` auf, um das erneute Einlesen von Daten verhindern zu können, wenn das Modul zur Deaktivierung aufgefordert wurde. Der Filter verhindert weitere Lesezugriffe jedoch nur, wenn in diesem Moment die Transaktionszählvariable des Moduls gerade Null ist. Sollte in einer Modulbeschreibung die Reihenfolge von Eingabe und Ausgabe unbestimmt sein (z. B. da nebenläufige Prozesse verwendet werden), besteht die Gefahr, dass der Transaktionszähler niemals von selbst Null wird. In diesem Fall muss ein restriktiveres Synchronisierungsverfahren, wie das in Beispiel 3, verwendet werden.

2. Die Anzahl der von einem Modul gelesenen Datensätze kann kontrolliert werden, indem den FIFO-Filtern ein bestimmtes Lese- bzw. Schreib-Limit gesetzt wird. Mittels dieses Limits kann der Zugriff auf die FIFO beschränkt und somit beliebig kontrolliert werden. Da hierbei Event-Filter zum Einsatz kommen und den Prozessen leere FIFOs vorgetäuscht werden können, ist mittels dieser Technik auch eine Synchronisierung von Modulen mit Method-Prozessen möglich.

   Das Beispiel in Listing 4.20 zeigt ein rekonfigurierbares Modul, dass auf jeder seiner Eingangs-FIFOs maximal 16 Datenpakete lesen kann und danach seine Arbeit einstellt. Bei der nächsten Aktivierung dieses Moduls wird das Read-Limit der FIFO-Filter automatisch wieder auf seinen ursprünglichen Wert zurückgesetzt.

3. In Listing 4.21 wird das vorige Beispiel erweitert, so dass nun die Anzahl der von dem Modul zu lesenden Datenpaare über eine Methode namens `incr_data_count()` dynamisch zugewiesen werden kann. Diese Methode kann von einem beliebigen Mechanismus verwendet werden (z. B. von einem internen Synchronisierungs-Prozess, der mit einer äußeren Kontrolle in Verbindung steht).

4. Wenn komplexere Synchronisationsfunktionalität erforderlich ist, können vom Anwender Callback-Methoden definiert werden, die von Filtern aufgerufen werden. Die FIFO-Filter können sowohl vor als auch nach einem Lese-/Schreib-Zugriff eine beliebige Callback-Methode aufrufen. Diese können beliebige Aufgaben erfüllen. Von der Zählung von Transaktionen bis zur Synchronisation mit einer internen oder externen Kontrolle kann hierdurch alles modelliert werden.

```
RC_RECONFIGURABLE_MODULE_DERIVED(ABC_fifo_rc, ABC_fifo) {
    RC_RECONFIGURABLE_CTOR_DERIVED(ABC_fifo_rc, ABC_fifo) {
        rc_add_filter(A, *new rc_fifo_in_filter<int> (tc, +1));
        rc_add_filter(B, *new rc_fifo_in_filter<int> (tc, +1));
        rc_add_filter(C, *new rc_fifo_out_filter<int>(tc, -2));
    }
    rc_transaction_counter tc; // Transaktionszähler
};
```

Listing 4.19: Beispiel 1: Das ABC-FIFO-Modul aus Abbildung 2.1 (S. 33) mit implizitem `busy`-Signal zur Verhinderung von Dateninkonsistenzen

```
RC_RECONFIGURABLE_MODULE_DERIVED(ABC_fifo_rc, ABC_fifo) {
    RC_RECONFIGURABLE_CTOR_DERIVED(ABC_fifo_rc, ABC_fifo) {
        // voreingestelltes Lese-Limit von 16
        rc_add_filter(A, *new rc_fifo_in_filter<int> (tc, +1, 16));
        rc_add_filter(B, *new rc_fifo_in_filter<int> (tc, +1, 16));
        rc_add_filter(C, *new rc_fifo_out_filter<int>(tc, -2));
    }
    rc_transaction_counter tc; // Transaktionszähler
};
```

Listing 4.20: Beispiel 2: Das ABC-FIFO-Modul, versehen mit einem festen Lese-Limit, das ein Lesen von maximal 16 Datenpaaren zulässt

```
RC_RECONFIGURABLE_MODULE_DERIVED(ABC_fifo_rc, ABC_fifo) {
    RC_RECONFIGURABLE_CTOR_DERIVED(ABC_fifo_rc, ABC_fifo)
      : // Lese-Limit voreingestellt auf 0 ('FIFO leer')
        filterA(tc, +1, 0), filterB(tc, +1, 0),
        filterC(tc, -2) {
        rc_add_filter(A, filterA); // filterA filtert Port A
        rc_add_filter(B, filterB); // filterB filtert Port B
        rc_add_filter(C, filterC); // filterC filtert Port C
        [...]
    }
    rc_transaction_counter  tc;       // Transaktionszähler
    rc_fifo_in_filter<int>  filterA; // FIFO-In-Filter
    rc_fifo_in_filter<int>  filterB; // FIFO-In-Filter
    rc_fifo_out_filter<int> filterC; // FIFO-Out-Filter
    void incr_data_count(int count) { // erhöht das Lese-Limit um count
        filterA.incr_read_limit(count);
        filterB.incr_read_limit(count);
    }
    [...]
};
```

Listing 4.21: Beispiel 3: Synchronisierung eines rekonfigurierbaren Moduls mittels der Lese-Limits des FIFO-Filters

```
RC_RECONFIGURABLE_MODULE_DERIVED(ABC_fifo_rc, ABC_fifo) {
    RC_RECONFIGURABLE_CTOR_DERIVED(ABC_fifo_rc, ABC_fifo)
    : // Lese-Limit voreingestellt auf 1, damit Modul anlaufen kann
      filterA(tc, +1, 1), filterB(tc, +1, 1),
      // filterC soll Callback-Methode nach erfolgtem Schreiben aufrufen
      filterC(NULL, RC_SYNC_CALLBACK(on_data_written)),
      data_to_read(16) { // 16 Datenpaare dürfen gelesen werden
        rc_add_filter(A, filterA); // filterA filtert Port A
        rc_add_filter(B, filterB); // filterB filtert Port B
        rc_add_filter(C, filterC); // filterC filtert Port C
    }
    rc_transaction_counter  tc;       // Transaktionszähler
    rc_fifo_in_filter<int>  filterA; // FIFO-In-Filter
    rc_fifo_in_filter<int>  filterB; // FIFO-In-Filter
    rc_fifo_out_filter<int> filterC; // FIFO-Out-Filter
    // Callback-Methode wird von filterC nach dem Schreiben aufgerufen
    void on_data_written(bool is_nb_access) {
        tc -= 2;                      // Transaktionszähler verringern
        --data_to_read;               // Datenzählvariable erniedrigen
        if (data_to_read > 0) {       // sind noch Daten zu lesen?
            filterA.incr_read_limit(1); // Lese-Limit um eins erhöhen
            filterB.incr_read_limit(1);
        }
    }
    RC_HAS_VAR(ABC_fifo_rc);  // Modul verwendet rücksetzbare Variablen
    rc_var(int, data_to_read);// Anzahl der noch zu lesenden Daten
};
```

Listing 4.22: Beispiel 4: Verwendung von Callback-Methoden in FIFO-Filtern

Das Beispiel in Listing 4.22 zeigt, wie eine zu Listing 4.20 äquivalente Synchronisierungsfunktionaltität mittels Callback-Methoden modelliert werden kann. Ein etwas komplexeres Beispiel für die Verwendung von Callback-Methoden findet sich in Listing B.6 auf S. 174.

In Fällen, in denen das für die Synchronisierung relevante Verhalten des Moduls zusätzlich auch von den eingelesen Datenwerten abhängt, können FIFO-Filter nach ihrem zuletzt gelesenen oder geschriebenen Wert befragt werden. Hierzu stehen die Methoden `get_last_read()` bzw. `get_last_written()` zur Verfügung.

Um die Anwendbarkeit des Filter-Konzepts für die Synchronisierung von Rekonfigurationsvorgängen zu demonstrieren, sind die Beispiel-Projekte „fifo_filter_dataflow“, „fifo_filter_sync“ und „fifo_filter_sync2“ implementiert worden. Sie befinden sich im Ordner „examples“ der ReChannel-Bibliothek. In „fifo_filter_dataflow“ werden FIFO-Filter und Transaktionszähler dazu verwendet, Datenflussbeschreibungen mit verschiedenen Datenflussraten und Reihenfolgen zu rekonfigurieren. Die beiden anderen Filter-Beispiele demonstrieren, wie gleichartigen, rekonfigurierbaren Modulen auf generische Weise ein Synchronisationsmechanismus hinzugefügt werden kann, der sich mit einer äußeren Kontrolle synchronisiert. Hierbei kommen ReChannel-Sprachkonstrukte zur expliziten Modellierung von DRHW zum Einsatz.

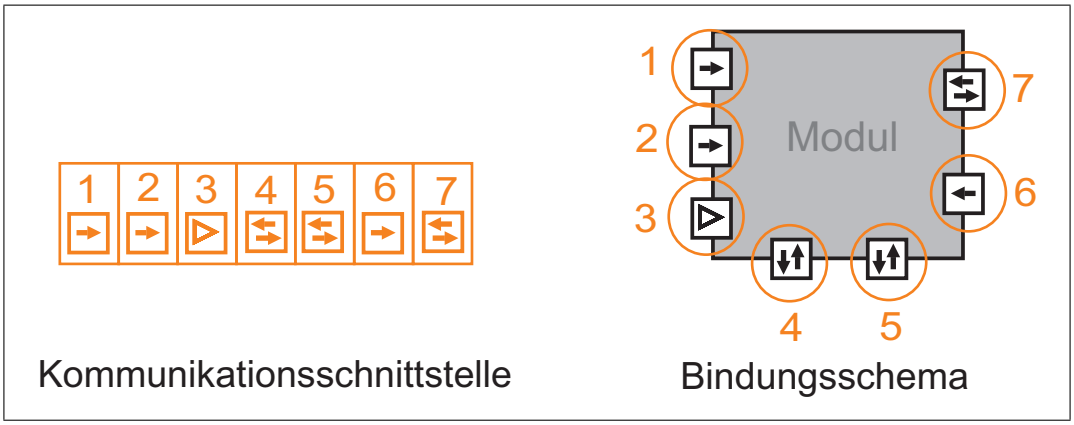

Abbildung 4.23: Schematische Darstellung einer Kommunikationsschnittstelle und eines dazu passenden Bindungsschemas für ein Beispielmodul.

4.14 Automatisierung der Switch-Bindung

Bei der Verwendung von vielen rekonfigurierbaren Modulen kann es mühsam sein, „per Hand“ jeden einzelnen Port bzw. Export dieser Module mit einem Switch verbinden zu müssen. Dasselbe Problem besteht auch für die statische Bindung der Portals, da alle Portals jeweils einzeln mit einem statischen Channel verbunden werden müssen.

Für einen jeweiligen rekonfigurierbaren Bereich muss bei der Switch-Bindung in den meisten Fällen exakt derselbe Vorgang mehrmals wiederholt werden. Es ist somit offensichtlich, dass bezüglich des Bedienungskomforts von ReChannel, sowohl bei der Bindung der statischen als auch der dynamischen Seite, noch Möglichkeiten zur Optimierung bestehen.

Zur Vereinfachung des bisher in ReChannel manuell vorzunehmenden Switch-Bindungsverfahrens wäre es hilfreich, wenn eine **Kommunikationsschnittstelle** definiert werden könnte, welche die Typen beliebiger SystemC-**Kommunikationsobjekte** (d.h. Ports, Exports und Channels) auf eindeutige Weise zu einer Schnittstelle zusammenfasst, wie in Abbildung 4.23 illustriert. Passend zu dieser Kommunikationsschnittstelle können dann **Bindungsschemata** definiert werden. Ein Bindungsschema repräsentiert eine bijektive Abbildung zwischen den Elementen einer Kommunikationsschnittstelle und einer Menge von in einem Design vorhandenen Kommunikationsobjekten jeweils übereinstimmenden Typs. Für zwei Bindungsschemata desselben Typs gilt, dass ihre Kommunikationsobjekte aufgrund der bijektiven Abbildung grundsätzlich eine direkte Zuordnung zueinander besitzen.

Einem rekonfigurierbaren Modul muss ein Bindungsschema zugewiesen werden können, das alle seine externen Ports und Exports umfasst und somit seine Kommunikationsschnittstelle eindeutig festlegt. Ferner müssen die Switches, die zu demselben rekonfigurierbaren Bereich gehören, gruppiert werden können. Wenn nun noch zwischen einer solchen Gruppe von Switches (der sogenannten **Switch-Gruppe**) und der dazu kompatiblen Kommunikationsschnittstelle eine logische Zuordnung der einzelnen Elemente

definiert wäre, dann könnten sämtliche Bindungen automatisiert durchgeführt werden.

Durch die Existenz eines automatisierten Bindungs-Mechanismus hätte man nicht nur die Benutzung von ReChannel deutlich vereinfacht, sondern gleichzeitig auch die wichtigste Voraussetzung für die Verschiebbarkeit von Modulen zur Simulationszeit erfüllt (siehe Abschnitt 4.15).

4.14.1 Anforderungen

Ein eventueller, automatischer Bindungs-Mechanismus sollte in der Lage sein, eine Switch-Gruppe sowohl auf der statischen als auch auf der dynamischen Seite automatisiert mit beliebigen Kommunikationsobjekten verbinden zu können. Obwohl es sich auf statischer und dynamischer Seite technisch gesehen um verschiedene Bindungsarten handelt, soll dies für den Benutzer dieses Mechanismus keinen sichtbaren Unterschied machen.

Des Weiteren sollte der Bindungs-Mechanismus flexibel genug sein, um auch mit sich gegenseitig beliebig überlappenden, rekonfigurierbaren Bereichen zusammen verwendet werden zu können.

Ein solcher Fall liegt z. B. vor, wenn modelliert wird, dass auf einem rekonfigurierbaren Bereich entweder ein einzelnes großes oder alternativ zwei kleine Module nebeneinander Platz finden können. Für die Simulation bedeutet dies, dass die hierbei beteiligten Switches sich mit unterschiedlichen Modultypen automatisiert verbinden können müssen. Dadurch dass es im Endeffekt drei unterschiedliche, sich überlappende rekonfigurierbare Bereiche gibt, müsste man für die vorhandenen Switches drei verschieden zusammengesetzte Switch-Gruppen definieren können.

Ein Modul sollte ferner eine Menge paarweise verschiedener Bindungsschemata besitzen dürfen, um z. B. eine Permutation der Port-Reihenfolgen o. ä. modellieren zu können.

Es sollte möglich sein, eine Kommunikationsschnittstelle global im gesamten Design verfügbar zu machen. Dazu gehört z. B. auch, dass die Eindeutigkeit ihrer Definition bei der Verwendung mehrerer getrennter Modul-Bibliotheken gewährleistet werden kann.

Zusammengefasst stellen sich folgende Anforderungen an die Umsetzung eines automatisierten Bindungs-Mechanismus:

1. Inkompatibilitäten zwischen Kommunikationsobjekt-Typen sollten bereits bei der Definition eines Bindungsschemas erkannt werden können und nicht erst bei einer tatsächlich durchgeführten Bindung durch SystemC gemeldet werden.

2. Port, Export und Channel – alles Komponenten mit unterschiedlichen Basisklassen und Bindungsverfahren – müssen auf einheitliche Weise verwaltet werden können, da sie alle Teil eines Bindungsschemas sein können.

3. Kommunikationsschnittstellen müssen in einem Design global bekannt gemacht werden können und daher eindeutig sein. Die Bindungsschemata müssen wohldefiniert sein und sollten untereinander durch Vergleich ihres zugrunde liegenden Typs auf Gleichheit überprüft werden können.

4. Switch-Gruppen müssen gebildet werden können, die einem rekonfigurierbaren Bereich entsprechen und sich somit gegenseitig beliebig überlappen dürfen.

5. Eine Switch-Gruppe muss einem eindeutigen Bindungsschema zugeordnet werden können. Zwischen dem einer Switch-Gruppe zugewiesenen Bindungsschema und den Elementen dieser Gruppe muss eine Abbildung existieren, durch die eine automatische Bindung möglich ist.

6. Rekonfigurierbaren Modulen müssen eine Menge paarweise verschiedener Bindungsschemata zugewiesen werden können, die jeweils alle externen Ports und Exports dieser Module miteinbeziehen.

7. Die Bindung einer Switch-Gruppe sollte sowohl auf der statischen Seite als auch auf der dynamischen Seite möglich sein, und dies, sowohl konzeptionell als auch syntaktisch, auf gleichartige Weise beschrieben werden können.

4.14.2 Entwurf

Bei einem möglichen nichtgenerischen Ansatz könnte zur Implementation einer Kommunikationsschnittstellen-Klasse entweder ein Vektor oder eine Map verwendet werden. Die erste Variante entspräche einer Positionsbindung und die zweite einer Namensbindung. Passend dazu müsste für die Implementation einer Klasse für das Bindungsschema entweder eine Vektor- oder eine Map-Datenstruktur verwendet werden. Die Verbindung zwischen diesen beiden Klassen müsste über einen ID-Wert hergestellt werden. Diese ID könnte entweder durch einen Datenwert repräsentiert werden oder möglicherweise durch die Angabe eines Templateparameters festgelegt werden. Um die Eindeutigkeit der IDs in einem Design zu gewährleisten, müsste eine globale Registrierung angelegt werden, die alle IDs verwaltet. Ein Problem stellt hierbei die Typüberprüfung bei der Definition der Bindungsschemata dar. Es ist umständlich, den Typ eines beliebigen SystemC-Kommunikationsobjekts zur Laufzeit zu überprüfen, da dies über Vergleiche des Namens dieses Typs zu erfolgen hätte. Eine solche Typüberprüfung wäre auch nur imstande, die Identität exakter Typen zu erkennen, aber nicht eine etwaige Übereinstimmung von Basistypen. Des Weiteren fehlt die Möglichkeit, Fehler bei der Definition bereits zur Kompilationszeit zu erkennen.

Sinnvoller scheint daher eine generische, streng typisierte Lösung zu sein, bei der die Schnittstelle über die Angabe eines Datentyps, anstatt der Erzeugung einer Datenstruktur, definiert wird. Dies bedeutet, dass für jeden Port bzw. Export ein Templateparameter verwendet wird. Eine Kommunikationsschnittstelle in Form einer C++-Templateklasse wäre im gesamten System verfügbar und des Weiteren würde der Compiler die Typüberprüfungen bei der Definition der Bindungsschemata übernehmen.

Die Idee ist, ein Klassentemplate zu entwerfen, dessen Templateklassen jeweils eine Kommunikationsschnittstelle repräsentieren und dessen Objekt-Exemplare jeweils einem Bindungsschema entsprechen. Die Objekt-Exemplare stellen demnach eine Abbildung dar,

welche die Kommunikationsschnittstelle auf existierende Kommunikationsobjekte abbildet. Im Folgenden soll dieses Klassentemplate auch synonym als **Portmap** bezeichnet werden, da es eine Abbildung auf eine Menge von Kommunikationsobjekten definiert. Hierbei steht das am häufigsten verwendete Kommunikationsobjekt, der „Port“, stellvertretend für alle anderen möglichen Kommunikationsobjekt-Typen.

Weiterhin wird ein zweites Klassentemplate entworfen, das als Switch-Verbinder bzw. **Switch-Connector** bezeichnet wird. Die Objekt-Exemplare dieser Klasse repräsentieren eine Switch-Gruppe, die zu einer bestimmten Portmap kompatibel ist. Die Aufgabe eines Switch-Connectors ist, die Switches in der Gruppe mit einer kompatiblen Portmap, sowohl auf statischer als auch auf dynamischer Seite, zu binden.

4.14.3 Implementation

Die generische Variante wird umgesetzt, da sie die beschriebenen Vorteile besitzt, und sich darüber hinaus auch bequemer verwenden lässt.

Das einzige Problem bei einem generischen Ansatz resultiert daraus, dass die Kommunikationsschnittstellen von Modulen mitunter auch recht groß sein können. Für jede mögliche Schnittstellengröße müsste ein Klassentemplate mit der jeweiligen Parameter-Anzahl zur Verfügung gestellt werden. Es wird daher auf eine unterschiedliche Anzahl von Templateparametern verzichtet, um eine unverhältnismäßig aufwändige Implementierung zu vermeiden.

In der nächsten C++-Version (siehe [5]) wird es die Möglichkeit zur Definition von Klassen mit einer variablen Templateparameter-Anzahl geben, so dass sich das Problem einer Beschränkung nicht mehr stellt. Die gewählte Implementierungsart mit fester Parameteranzahl wäre beim Umstieg auf die neue Version sehr leicht zu portieren.

Kommunikationsobjekte

Um die Kommunikationsobjekte von SystemC (Ports, Exports und Channels) in einheitlicher Weise verwalten und abspeichern zu können, wird die Klasse `rc_object_handle` implementiert. Diese kann wie ein Zeiger-Objekt („Handle“) verwendet werden. Diese Klasse ermöglicht es, ohne Kenntnis irgendwelcher Templateparameter, auf das Interface des referenzierten Objektes zugreifen zu können. `rc_object_handle` kann daher als Abstraktion der unterschiedlichen Kommunikationsobjekte angesehen werden.

Die Klasse verwendet intern die beiden ReChannel-Handles, `rc_port_handle` und `rc_export_handle`. Ihr Speicherbedarf ist daher sechsmal größer als der eines einfachen Pointers. Die deutliche Reduktion der Code-Komplexität, die durch die Abstraktion der unterschiedlichen SystemC-Kommunikationsobjekte erreicht wird, rechtfertigt aber die Inkaufnahme eines erhöhten Speicherbedarfes.

Portmap

Zur Implementation der Portmap wird ein Klassentemplate namens `rc_portmap` definiert, dessen Templateklassen jeweils eine eindeutige Kommunikationsschnittstelle repräsentie-

ren. Für jeden Port bzw. Export, der in einer Kommunikationsschnittstelle enthalten ist, steht ein Templateparameter zur Verfügung.

Das Klassentemplate verfügt über eine Templateparameter-Anzahl von N_{max}. Der erste Templateparameter wird als ein „tag“ (dt.: „Namensschild“, „Kennzeichnung“) verwendet, der es ermöglicht, dem Bindungsschema zusätzlich zu der rein strukturellen Beschreibung auch noch ein Unterscheidungsmerkmal hinsichtlich seiner funktionalen Eigenschaften (wie z. B. dem Verwendungszweck als Audiocodec, etc.) zuzuweisen. Hier kann irgendein sinnvoller Datentyp übergeben werden, der dann als der Name dieses Bindungsschemas angesehen werden kann. Die restlichen Parameter – implementationsbedingt[1] sind dies exakt 63 – legen den Typ eines Kommunikationsobjektes fest. Da die SystemC-Positionsbindung von Modulen ebenfalls auf maximal 64 beschränkt ist, sollte dies in den meisten Fällen ausreichend sein.

Die Templateparameter haben den Standard-Wert `void`, um eine variable Parameteranzahl bei der Verwendung dieser Klasse zu erlauben. Der Typ `void` dient als Platzhalter und hat die Bedeutung von *„Kein Kommunikationsobjekt“*.

Ein internes **enum** `portmap_size` enthält die Anzahl der verwendeten Kommunikationsobjekte. Diese Anzahl wird mittels Boosts *type_ traits*-Klassen (1.2.3) zur Kompilationszeit aus den bei der Instanzierung der Templateklasse angegebenen Templateparametern ermittelt.

Es wird intern ein `rc_object_handle`-Array der Größe `portmap_size` deklariert, über das die Zuordnung der Kommunikationsobjekte nach dem Positional-Binding realisiert wird. D.h., jeder Index eines Array-Elementes entspricht der Position eines Kommunikationsobjektes. Das Array wird initialisiert durch die bei der Konstruktion im Konstruktor übergebenen Kommunikationsobjekte, so dass ein Portmap-Objekt grundsätzlich wohldefiniert ist. Die Initialisierung des Arrays ist im Code für die maximale Anzahl $N_{max}-1$ ausgelegt, doch ist dies mittels Templatetechniken derartig implementiert, dass die überflüssigen Initialisierungsanweisungen keine Auswirkung haben und aufgrund dessen vom Compiler wegoptimiert werden können.

In der Klasse `rc_portmap` ist nur ein einziger Konstruktor definiert, der über $N_{max}-1$ Übergabeparameter verfügt. Jeder dieser Parameter hat einen speziellen generischen Datentyp namens `ReChannel::internals::arg`, der die Inkompatibilitäten zwischen den deklarierten und den übergebenen Kommunikationsobjekt-Typen bereits zur Kompilationszeit erkennt. Des Weiteren ermöglicht dieser Parameter, dass ein $N_{max}-1$ großer Konstruktor nur genau `portmap_size` viele Parameter zulässt. Die Reihenfolge der Übergabeparameter entspricht exakt der Reihenfolge, die beim Positional-Binding zur Anwendung kommt.

Damit eine Portmap auch überall dort allgemein verwendet werden kann, wo ihr expliziter Templateklassen-Typ nicht bekannt ist, leitet `rc_portmap` sich von einer nichtgenerischen Basisklasse namens `rc_portmap_base` ab, über die sämtliche Fähigkeiten der Portmap zur Verfügung stehen. Es gibt Methoden zur Abfrage der Größe, für den index-

[1] Aufgrund der Kompatiblität mit *Microsoft Visual C++ 2003 .NET* ist die maximale Templateparameter-Anzahl N_{max} auf 64 beschränkt. Mehr Templateparameter sind mit diesem Compiler nicht möglich.

basierten Zugriff auf die einzelnen Kommunikationsobjekte sowie zur Überprüfung auf Übereinstimmung mit anderen Portmap-Objekten.

Eine Kommunikationsschnittstelle kann mittels eines einfachen `typedef`s deklariert werden, wie folgendes Beispiel mit zwei Eingangs- und einem Ausgangsport zeigt:

```
typedef rc_portmap<
            myPortMapName,
            sc_in<bool>,
            sc_in<int>,
            sc_out<int> > myPortMap;
```

Bei dem hier als ersten Parameter angegebenen Datentyp `myPortMapName` handelt es sich um ein benutzerdefiniertes, extra für Bennungszwecke definiertes, `struct`. In vielen Fällen kann es aber auch Sinn machen, eine bereits existierende Klasse zu verwenden, um z. B. auf einen bestimmten Verwendungszweck hinzuweisen.

Bei der Definition des Typs einer Portmap müssen nicht ausschließlich Ports, sondern können je nach Erfordernis auch Exports verwendet werden. Wenn ein Bindungsschema speziell nur für den Zweck der Bindung mit der statischen Seite definiert wird, dann dürfen in dieser Portmap auch Channels enthalten sein.

Ein Bindungsschema wird durch die Erzeugung eines `rc_portmap`-Exemplars definiert. Passend zu obiger Kommunikationsschnittstellen-Deklaration wird in folgendem Beispiel ein Bindungsschema aus drei kompatiblen Ports gebildet und daran anschließend auf Typ-Gleichheit mit einem anderen Portmap-Objekt überprüft:

```
myPortMap pmap(clk_port, in_port, out_port);
if (pmap.is_compatible(other_pmap)) { [...] }
```

Switch-Connector

Ein Switch-Connector wird durch das Klassentemplate `rc_switch_connector` repräsentiert. Diese Klasse besitzt einen einzelnen Templateparameter `DynPortMap`, über den der Typ der Portmap festgelegt wird, mit der dieser Switch-Connector auf dynamischer Seite verbindbar ist.

Lediglich die dynamische Seite ist verbindlich auf eine bestimmte Portmap festgelegt. Auf der statischen Seite darf ein Switch-Connector mit allen Portmaps verbunden werden, die sich praktisch (d.h. von SystemC aus) auf statischer Seite an die jeweilige Switch-Gruppe binden lassen. Hierbei können demnach also auch Portmaps mit beliebigen Channeln verwendet werden.

Es ist ein einziger Konstruktor definiert, der über eine maximale Parameteranzahl von N_{max} verfügt, aber lediglich `DynPortMap::portmap_size` viele Parameter akzeptiert.

Dies wird, ähnlich wie bei der Definition des Portmap-Konstruktors (s.o.), durch die Verwendung des generischen Übergabedatentyps `ReChannel::internals::arg` erreicht.

Die Switches, die in einer Switch-Gruppe enthalten sein sollen, werden im Konstruktor übergeben. Genau wie bei der Portmap, entspricht die hier verwendete Reihenfolge der Reihenfolge, die beim Positional-Binding verwendet wird. Die übergebenen Switches werden in einem Array abgespeichert, das exakt `DynPortMap::portmap_size` viele Switch-Pointer aufnehmen kann. Der Konstruktor enthält Initialisierungsanweisungen für bis zu $N_{max} - 1$ viele Switch-Elemente, die allerdings, wenn sie überflüssig sind und auf das Array keinen Effekt haben, vom Compiler wegoptimiert werden können. Der erste Parameter des Konstruktors ist ein String, um dem Switch-Connector einen Namen geben zu können.

Die Existenz einer Basisklasse `rc_switch_connector_base` erlaubt die Verwendung von `rc_switch_connector`-Objekten auch überall dort, wo exakte Typen nicht bekannt sein können. `rc_switch_connector_base` verfügt über alle wichtigen Methoden zum Zugriff auf die Array-Elemente (d.h. Switches) sowie zur Kompatibilitätsprüfung mit Portmap-Objekten.

`bind_static()` und `bind_dynamic()` sind die entscheidenden Methoden für die Bindung. Beide sind in der Basisklasse des Switch-Connectors definiert. Mit `bind_static()` wird die Switch-Gruppe an die Kommunikationsobjekte einer gegebenen Portmap gebunden. `bind_dynamic()` dient dem selben Zweck, allerdings für die dynamische Seite. Hier kann entweder eine Referenz auf ein `rc_reconfigurable`-Objekt oder eine Referenz auf eine Portmap übergeben werden.

Notwendige Erweiterungen an anderen ReChannel-Klassen:

- Zu der Klasse `rc_reconfigurable` wurden einige Methoden hinzugefügt, mit denen die zu einem rekonfigurierbaren Modul kompatiblen Portmaps verwaltet werden können. Es können z. B. mittels der Methode `rc_add_portmap()` Portmaps zum Modul hinzugefügt werden, die für eine automatische Bindung durch einen Switch-Connector verwendet werden können. Mittels der Methode `rc_clear_portmaps()` können alle zuvor zugewiesenen Portmaps wieder entfernt werden.

- Damit die Bindung der Switches durchgeführt werden kann, muss das *Abstract Base Interface* (ABI) `rc_switch` um zwei virtuelle Methoden, `bind_static_object()` und `bind_dynamic_object()`, erweitert werden. Diesen beiden Methoden kann ein beliebiges SystemC-Kommunikationsobjekt zur Bindung an den Switch übergeben werden. Eine jeweilige Switch-Implementation muss hierbei selbstständig feststellen, ob eine Bindung möglich ist, und dementsprechend auch wissen, auf welche Art die Bindung zu erfolgen hat.

Wird `bind_dynamic()`, wie in Listing 4.23 demonstriert, unter Angabe eines rekonfigurierbaren Moduls aufgerufen, wird für dieses Modul automatisch in dieser Liste nach einer passenden Portmap (hier `myPortMap`) gesucht. Sollte keine solche gefunden werden können, wird ein Fehler gemeldet.

```
// Erzeugen eines Switch-Connectors für eine Switch-Gruppe
rc_switch_connector<myPortMap> connector(
    "connector", clk_portal, in_portal, out_portal);
[...]
connector.bind_dynamic(m1); // bindet die (Ex)ports von m1
connector.bind_dynamic(m2); // bindet die (Ex)ports von m2
```

Listing 4.23: Beispiel: Verwendung eines Switch-Connectors zur automatischen Bindung von rekonfigurierbaren Modulen an eine Gruppe von Switches. Die Module müssen hierbei zu der zum Switch-Connector gehörigen Portmap kompatibel sein, da es ansonsten zu einem Laufzeitfehler kommt. Ein Modul ist genau dann kompatibel, wenn diesem mittels der Methode `rc_add_portmap()` ein Exemplar einer entsprechenden Portmap hinzugefügt wurde.

4.14.4 Evaluierung

Bezüglich der zuvor aufgestellten Anforderungen (siehe Abschnitt 4.14.1) ergibt sich nun folgendes Bild (in gleicher Nummerierungsreihenfolge):

1. Bei der Deklaration einer Portmap können aufgrund der Implementation des Konstruktors die meisten Probleme bereits zur Kompilationszeit erkannt werden. Bei der Zuweisung einer Portmap zu einem Modul wird überprüft, ob die beteiligten Kommunikationsobjekte zu diesem Modul gehören. Diese Überprüfung kann erst zur Laufzeit erfolgen.

   Für die Klasse `rc_switch_connector` gilt Ähnliches wie für die Implementation des Konstruktors. Hier ist es allerdings nicht möglich, die Typen der Switches bereits zur Kompilationszeit auf Kompatibilität mit der Portmap zu überprüfen. Eine Inkompatibilität kann demnach erst bei einer versuchten Bindung als Fehler erkannt werden.

2. Die einheitliche Verwaltung von Ports, Exports und Channels wird durch die Verwendung der Klassen `rc_object_handle` ermöglicht. Somit können beliebige Kommunikationsobjekte in `rc_portmap` abgespeichert und sogar ohne Kenntnis der exakten Typen in der Basisklasse `rc_portmap_base` verwendet werden.

3. Der Typ eines Bindungsschemas kann auf einfache Weise durch die globale Deklaration eines `typedef`s in einem Design bekannt gemacht werden. Aufgrund der Verwendung von `typedef` sorgt C++ für die Eindeutigkeit dieser Definition. Die Bindungsschemata sind wohldefiniert, dafür sorgt der Konstruktor von `rc_portmap`, der nur exakt die richtige Anzahl und den richtigen Typ der Kommunikationsobjekte zulässt. `rc_portmap` und `rc_portmap_base` werden untereinander durch Vergleich ihres zugrunde liegenden C++-Typs auf Gleichheit überprüft. Der exakte Typ einer Portmap ist also entscheidend.

4. Mittels `rc_switch_connector` ist die Zusammenstellung von beliebigen Switch-Gruppen möglich. Ein gegenseitiges Überlappen der in einem Design existierenden Gruppen stellt kein Problem dar, solange nicht versucht wird, ein rekonfigurierbares Modul mehrmals zu binden. Zur Simulationszeit können die verschiedenen Switch-Gruppen ohne Probleme zum Verschieben von Modulen in einem Design verwendet werden (siehe Kapitel 4.15).

5. Da das Klassentemplate `rc_switch_connector` mit dem Typ einer Portmap parametrisiert wird, ist eine Switch-Gruppe immer einem eindeutigen Bindungsschema zugeordnet. Die automatische Bindung zwischen `rc_switch_connector` und den Kommunikationsobjekten einer kompatiblen Portmap stellt kein Problem dar, da bei beiden die Zuordnung der Elemente durch ihren Index möglich ist ("positional binding").

6. Rekonfigurierbaren Modulen können beliebige Portmaps hinzugefügt werden. Dazu steht die Methode `rc_add_portmap()` zur Verfügung. Es wird als Fehler angesehen, wenn ein und derselbe Portmap-Typ mehr als einmal zugewiesen wird. Ebenso ist es nicht erlaubt, dass eine zugewiesene Portmap Kommunikationsobjekte enthält, die nicht zu diesem Modul gehören.

7. Die Bindung einer Switch-Gruppe ist auf der statischen Seite mit der Methode `bind_static()` und auf der dynamischen Seite mit der Methode `bind_dynamic()` möglich. Die Benutzung gestaltet sich für einen Designer in beiden Fällen gleichartig, da jeweils Portmaps gebunden werden können.

Die Verwendung und Funktionsweise von Switch-Connectoren und Portmaps wird in einem Beispieldesign im ReChannel-Projekt-Ordner „examples/switch_connector" demonstriert. Das Design basiert auf dem Beispieldesign aus „examples/explicit_fifo_sync", das derartig angepasst worden ist, dass alle zuvor noch einzeln vorgenommenen Switch-Bindungen nun auf statischer und dynamischer Seite zusammengefasst und automatisiert durchgeführt werden. Die Bindung hat sich hierbei sichtbar vereinfacht. Bei der Verwendung von einer noch größeren Zahl an rekonfigurierbaren Modulen und/oder Ports sollte dieser Unterschied noch deutlicher ausfallen.

Anmerkung: In diesem Beispieldesign wird für die Deklaration der Kommunikationsschnittstelle ein Makro namens `RC_PORTMAP()` *verwendet. Dabei handelt es sich um eine mögliche alternative Syntax. Bei dieser gestaltet sich die Benennung einer Portmap etwas bequemer, ist aber ansonsten identisch mit der Notation durch ein einzelnes* `typedef`.

4.15 Mobilität von rekonfigurierbaren Modulen

Die Möglichkeit, ein und dasselbe Modul-Exemplar im Laufe einer Simulation an verschiedenen Positionen in einem Design verwenden zu können, wird als *Mobilität* bezeichnet. Für Mobilität gibt es vielfältige Anwendungsmöglichkeiten.

Um Ressourcen zu sparen, könnte z. B. eine begrenzte Menge von rekonfigurierbaren Modulen angelegt und wiederverwendet werden. Dies ist immer dann von Vorteil, wenn in einem Design viele rekonfigurierbare Bereiche modelliert werden sollen, die alle für gleichartige Modulbeschreibungen gedacht sind. Man könnte sich z. B. eine schachbrettartige Aufteilung eines dynamisch rekonfigurierbaren Chips vorstellen, bei der in jedem Feld jeweils nur eine Modulbeschreibung aus einer festen Menge einprogrammiert werden kann. Eine Optimierung hinsichtlich der Anzahl der für die Simulation erzeugten Modul-Exemplare ist möglich, wenn an die Anzahl der gleichzeitig genutzten Modulbeschreibungen auch gewisse Regeln geknüpft sind. Kann beispielsweise bei einer Feldanzahl von 16 ein bestimmter Modultyp nur maximal vier Mal vorkommen, so wären demnach lediglich vier Exemplare dieses Moduls nötig. Wenn diese Module mit ReChannel-Sprachkonstrukten explizit beschrieben wurden, verhalten sie sich wie DRHW. Da derartige Module implizit zurückgesetzt werden, gleichen diese einem Bitstream, der beliebig oft geschrieben (bzw. verschoben) werden kann ohne sich zu ändern.

Aber auch die persistenten Eigenschaften eines Moduls können von Vorteil sein, wenn simuliert werden soll, dass Modulzustände innerhalb eines Systems transportiert werden.

Funktionsweise

Dadurch, dass rekonfigurierbare Module mittels Switch-Connectoren automatisiert an Switches gebunden werden können (siehe Abschnitt 4.14), ist die Grundvoraussetzung für die Verschiebbarkeit von Modulen in ReChannel gegeben. Was noch fehlt, ist eine Möglichkeit für den Anwender diesen Mechanismus zu kontrollieren. Dazu wird die Rekonfigurationskontrolle `rc_control` (bzw. das Interface `rc_control_if`, siehe Abschnitt 4.5.2) um eine Methode namens `move()` erweitert:

```
virtual void move(
        const rc_reconfigurable_set& reconf_set,
        rc_switch_connector_base& target) = 0;
```

Der erste Parameter dieser Methode übergibt die Menge der rekonfigurierbaren Objekte (RO), die verschoben werden sollen. Der zweite Parameter ist ein Switch-Connector-Objekt (siehe Abschnitt 4.14.3), das das Ziel für den Verschiebevorgang repräsentiert. Dieser Switch-Connector muss zu den RO kompatibel sein, d.h. alle müssen die gleiche Portmap (siehe Abschnitt 4.14.3) besitzen. Andernfalls wäre eine Bindung aufgrund der verschiedenen Schnittstellen nicht möglich und es würde ein Fehler gemeldet.

Die Methode `move()` verwendet intern die Lock-Funktionalität von `rc_control`, um einen atomaren Aufruf darzustellen, der nicht von anderen Prozessen unterbrochen werden kann. Nachdem der aufrufende Kontrollprozess den Lock über die RO-Menge erlangt

hat, wird zuerst einmal `unload()` aufgerufen, um sicherzustellen, dass keines der RO noch aktiv oder geladen ist. Dieser implizite Aufruf von `unload()` dient dazu, dem Anwender die Verwendung auf abstrakten Beschreibungsebenen zu vereinfachen. Falls eine Fehlermeldung erforderlich sein sollte, wenn z. B. ein Modul noch aktiv ist, so kann der Anwender dies vor dem Aufruf überprüfen.

Da ein mittels `unload()` entladenes Modul aufgrund gesperrter Accessoren keine externe Kommunikation mehr haben kann, ist ein Verschieben in diesem Zustand ohne Fehlermeldung seitens der Switches durchführbar. Nun kann die private Methode `rc_reconfigurable::move()` nacheinander auf den jeweiligen RO aufgerufen werden. Als Parameter wird der Switch-Connector übergeben, da dieser die Switches enthält, die das Ziel darstellen. Eine Aufspaltung in mehrere temporäre Prozesse wie bei den anderen Kontrollmethoden ist für diese Aufrufe nicht erforderlich, da die Operation weder Simulationszeit noch Delta-Zyklen benötigt.

`rc_reconfigurable::move()` überprüft zuerst, ob das RO eine Portmap besitzt, die mit dem Switch-Connector kompatibel ist. Falls dies der Fall ist, sorgt es dafür, dass das jeweilige RO sich bei allen momentan verbundenen Switches abmeldet und sich bei den neuen Switches zusammen mit dem zugehörigen Interface registriert. Dies geschieht in einer `for`-Schleife über alle in der Portmap enthaltenen Kommunikationskomponenten.

Die Zugehörigkeit, welcher alte Switch mit welchem neuen Switch ausgetauscht werden muss, wird über das Interface der Kommunikationskomponenten aus der Portmap hergestellt. Möglich wird dies, da jedes RO eine Multi-Index-Datenstruktur (siehe Kapitel 1.2.4) mit allen aktuell verbundenen Interface-Switch-Paaren verwaltet. Diese Datenstruktur besitzt Schlüssel zum effizienten Auffinden sowohl von Switches als auch Kommunikationsobjekten.

Der jeweils zu einem Portmap-Elementindex zugehörige neue Switch ist über den gegebenen Switch-Connector direkt abfragbar, da Anzahl und Reihenfolge der im Switch-Connector gruppierten Switches mit der Anzahl und Reihenfolge der Kommunikationskomponenten in der Portmap übereinstimmen.

Der Switch-Connector, der das Ziel darstellte, wird im RO als der aktuelle Switch-Connector gespeichert. Über die Methode `rc_get_current_switch_connector()` kann der aktuelle Switch-Connector abgefragt werden. Hierdurch wird es einem Anwender ermöglicht, die Information über den aktuellen Ort eines Moduls erfragen zu können. Dies könnte bei der Modellierung eines Systems mit Mobility-Aspekten hilfreich sein.

4.16 Hash-Maps

Einige ReChannel-Klassen benötigen effiziente, ungeordnete Map-Container. In den meisten aktuellen Implementationen der C++-Standard-Template-Library (STL) stehen bereits solche Container-Klassen in Form von Hash-Maps und Hash-Multimaps zur Verfügung. Aber da die STL zum jetzigen Zeitpunkt offiziell noch keine Hash-Maps enthält, variieren die Implementationen zum Teil stark. In *Microsoft Visual C++* z. B. werden die Hash-Map-Klassen mittels der Header-Datei „`hash_map`“ eingebunden und befinden sich im Namensraum „`stdext`“, während sie sich in *GNU's GCC 3.4* in der

Datei „`ext/hash_map`„ und im Namensraum „`__gnu_cxx`“ befinden. Ab *GCC 4.x* wurde dies wiederum geändert, da dort der *C++ Library Technical Report 1 (TR1)* [5] als Spezifikation für die Implementation der Hash-Map-Klassen verwendet wird. Hier heißt die Header-Datei daher „`tr1/unordered_map`“ und der Namensraum „`std::tr1`“.

Die BOOST-Bibliothek bietet die Möglichkeit, für eine Vielzahl von Compilern automatisch feststellen zu lassen, welche Header-Datei und welcher Namensraum verwendet werden. Somit könnte man mittels der von BOOST vordefinierten Macros die entsprechenden Dateien einbinden sowie den korrekten Namensraum adressieren.

Bei dieser Vorgehensweise gibt es aber immer noch das Problem, dass keine verbindliche Spezifikation existiert. Es kann daher nicht garantiert werden, dass die Deklarationen aller Hash-Map-Klassen identisch sind. So können zum Beispiel die Klassennamen der verwendeten, generischen Hash-Funktionen verschieden sein, was dazu führt, dass in RECHANNEL Compiler-spezifische Unterscheidungen getroffen werden müssten. Auch Methoden- oder `typedef`-Deklarationen, die zwischen einzelnen Compiler-Versionen leicht voneinander abweichen, können bereits zu Portabilitätsproblemen führen. Man wäre aufgrund dessen gezwungen, ausschließlich die gemeinsame Teilmenge der unterschiedlichen Implementationen zu nutzen.

Designentscheidung

Um den oben beschriebenen Problemen gänzlich aus dem Weg zu gehen, wird eine Hash-Map- sowie eine Hash-Multi-Map-Klasse aus BOOST-Klassen zusammengestellt. Diese Vorgehensweise erscheint am sinnvollsten, da bereits sowohl RECHANNEL als auch SYSTEMC die BOOST-Bibliothek einbinden und außerdem in BOOST schon fast alle für diesen Zweck benötigten Komponenten enthalten sind.

Die auf diese Weise definierten Hash-Maps verfügen über eine feste und verlässliche Schnittstelle. Dies ist ein deutlicher Vorteil gegenüber der Verwendung von Hash-Map-Klassen, wie sie in den vielfältigen STL-Erweiterungen der Compiler enthalten sind. Des Weiteren wird durch die Verwendung von BOOSTs generischen Multi-Index-Container-Klassentemplates `multi_index_container` (siehe Abschnitt 1.2.4) und deren Indexen automatisch die Korrektheit und Effizienz der darauf aufbauenden Hash-Map-Klassen garantiert. Bei einer etwaigen kompletten Eigenimplementation müsste dies noch gezeigt werden.

Implementation

Mittels des Klassentemplates `boost::multi_index::multi_index_container` werden zwei generische Container-Klassen, `rc_hash_map` und `rc_hash_multimap`, konstruiert, die sich wie eine ungeordnete Map bzw. Multi-Map mit Hashtabellen-Indexierung verhalten.

Die Templateparameter der Klassen `rc_hash_map` und `rc_hash_multimap` entsprechen in Reihenfolge und Typ der `hash_map`-Klasse aus der STL-Erweiterung von [21]:

```
template<
    class key_, class data_,
    class hash_=boost::hash<key_>,
    class pred_=std::equal_to<key_>,
    class alloc_=std::allocator< [...] >
>
class rc_hash_map { [...] };
```

Bei `rc_hash_map` und `rc_hash_multimap` handelt es sich konzeptionell um Wrapper-Klassen, da sämtliche Methodenaufrufe an ein intern gekapseltes Objekt der Klasse `multi_index_container` weitergeleitet werden.

Über die Templateparameter der Klasse `multi_index_container` können sämtliche Eigenschaften und damit die gewünschte Verwendungsart des gekapselten Container-Objektes bestimmt werden. Für die Verwendung als Hash-Map wird lediglich ein einziger Index benötigt. Hierfür existiert in Boost eine vordefinierte Hash-Index-Klasse, die bei der Spezialisierung des Multi-Index-Container-Klassentemplates angegeben werden kann.

Als Hash-Funktion wird die Standard-Hash-Funktion der Boost-Bibliothek, das Funktionsobjekt `boost::hash<key_>`, verwendet, wobei `key_` hierbei den Datentyp des Schlüssels bezeichnet. Für den in ReChannel einzigen Anwendungsfall, in dem `key_` ein Pointer ist, gibt es in Boost bereits eine effiziente Spezialisierung dieses Funktionsobjektes, so dass keine weiteren Klassendefinitionen erforderlich sind.

In Anologie zu den Standard-Maps und -Multimaps wird als Elementtyp ein Paar aus Schlüssel (`key_`) und Wert (`data_`) verwendet. Die Standard-Paar-Klasse `std::pair` kann dabei allerdings nicht als Elementtyp verwendet werden, da der Hash-Index von `multi_index_container` ausschließlich über konstante Iteratoren verfügt, die eine Modifikation des Element-Wertes verbieten. Es wird daher eine eigene Paar-Klasse definiert, die, bis auf die Deklaration des Element-Wertes, mit `std::pair` identisch ist. Damit der Element-Wert auch aus einem konstanten Kontext heraus verändert werden darf, wird er hier unter zusätzlicher Angabe des C++-Modifizierers `mutable` (dt.: „veränderlich“) deklariert.

Ein Hash-Map-Container, bei dem ein Modul effizient mit dem Pointer eines anderen SystemC-Objektes indexiert wird, kann beispielsweise wie folgt deklariert werden:

```
rc_hash_map<sc_object*, sc_module*> myHashMap;
```

4.17 Test und Kompatibilität

Die Implementation wurde anfangs mittels der in ReChannel-v1 bereits vorhandenen Beispiel-Projekte getestet. Hierzu sind zu Beginn der Implementierung die von Andreas Niers programmierten Beispiele „examples/multiple_signals“ und „examples/multi-

ple_fifos“ verwendet worden, die die Verwendung von Signal- bzw. FIFO-Portals demonstrieren.

Später wurden zusätzliche Beispiel-Projekte implementiert, die die Verwendung von Komponenten und Sprachkonstrukten, die in ReChannel-v2 neu hinzugekommen sind, demonstrieren. Diese befinden sich ebenfalls unterhalb des Ordners „examples“ und zeigen u. a. die Verwendung der Filter-Funktionalität und der Switch-Connectoren sowie die Rekonfiguration von Channels mittels Exportals. In den meisten dieser Beispiele kommen auch die Sprachkonstrukte zur expliziten Modellierung von DRHW zum Einsatz.

Für den Test neuer Funktionalität während der Entwicklung wurde ein separates Test-Projekt verwendet. Es befindet sich im Bibliotheks-Order „test/overalltest“. Das hierin implementierte Design dient dem Zweck, einen Großteil des Funktionsumfangs von ReChannel-v2 komfortabel in einem Durchlauf testen zu können.

Die ReChannel-Bibliothek wurde während der gesamten Entwicklung mit den folgenden Compilern übersetzt (und anschließend mit den Beispiel-Projekten getestet), um die Kompatibilität sicherzustellen:

- Microsoft Visual C++ 2005
- Microsoft Visual C++ 2003 .NET (SP1)
- GNU GCC 3.4
- GNU GCC 4.1

Auch mit dem *Intel C++ Compiler 8.0* konnte die ReChannel-Bibliothek erfolgreich kompiliert und verwendet werden. In der Bibliothek stehen für alle oben genannten Compiler entsprechende Solution-, Projekt- und Makefile-Dateien unterhalb der Ordner „dev“ und „src“ zur Verfügung. Die kompilierte ReChannel-Bibliothek („ReChannel.lib“ unter Windows, „rechannel.a“ unter Linux) wird im Ordner „lib“ angelegt.

Als SystemC-Implementation wurde die Referenz-Implementation der OSCI (*SystemC v2.2*) [13] verwendet.

4.17.1 CThread-Prozesse

Die Rücksetzfunktionalität der CThread-Prozesse von SystemC ist an Signale gebunden. Der SystemC-Standard [7] enthält keine Spezifikation bezüglich der Implementationsdetails des dahinter liegenden Mechanismus. Die OSCI-Referenzimplementation implementiert diesen Mechanismus mittels einer nicht standardkonformen Erweiterung. Sie fügt dem Signal-Interface `sc_signal_inout_if<bool>` eine Methode namens `is_reset()` hinzu, die in der Spezifikation dieses Interfaces nicht enthalten ist. `is_reset()` liefert ein Objekt vom Typ `sc_reset` zurück, das ebenfalls nicht im Standard enthalten ist. Die einzige Einflussmöglichkeit eines Anwenders auf diesen Mechanismus ist darauf beschränkt, Reset-Signale mittels der Methode `sc_module::reset_signal_is()` zu definieren. Eine nachträgliche Entfernung oder Änderung ist nicht möglich.

CThread-Prozesse und Signal-Accessoren

Es kann aus obigen Gründen nicht garantiert werden, dass sich CThread-Prozesse einer beliebigen SystemC-Implementation mit einem Accessor verbinden lassen. ReChannel enthält spezielle Anpassungen, um die Rücksetzfunktionalität des CThread-Prozesses auch im Accessor weiterzuleiten. Dies ist allerdings aller Voraussicht nach nur für die Herstellung der Kompatibilität zur OSCI-Referenz-Implementation geeignet. Möglicherweise sind für andere Implementationen anderweitige Spezialanpassungen notwendig. Dies wird davon abhängen, wie ähnlich diese der Referenz-Implementation der OSCI sind.

Als mögliche Lösungsmöglichkeit könnte weiterhin in Betracht gezogen werden, den Accessor direkt von einem Signal abzuleiten. In OSCI-SystemC leitet sich das Signal z. B. vom Signal-Interface ohne das Schlüsselwort „virtual“ ab. Dies hat u. a. zur Folge, dass die gleichzeitige Ableitung vom Accessor-ABI zu zwei verschiedenen Signal-Interfaces in einem Objekt führt. Die Probleme, die sich durch eine solche Verschmelzung ergeben, sind sowohl technischer als auch konzeptioneller Art. Daher ist auf den Versuch einer solchen Implementierung vorerst verzichtet worden.

Mobilität von CThread-Prozessen

Module mit SystemC-CThread-Prozessen können nicht ohne weiteres in einem Design verschoben werden. Der Grund hierfür ist, dass die Rücksetzfunktionalität in SystemC – wie oben bereits erwähnt – eine persistente Bindung darstellt, die ebenso wie Portbindungen nachträglich nicht mehr gelöst werden kann. Wird einem CThread-Prozess innerhalb eines rekonfigurierbaren Moduls mittels `reset_signal_is()` ein externes Reset-Signal zugewiesen, dann gehen diese beiden eine untrennbare Verbindung ein. Wird das Modul innerhalb der Simulation an einen anderen Ort bewegt, so bleibt das ursprüngliche Reset-Signal unweigerlich bestehen und der Prozess wird weiterhin durch ein entferntes Reset-Signal zurückgesetzt. Die Korrektheit der Simulation ist hierdurch nicht mehr gegeben.

ReChannel löst dieses Problem für die OSCI-SystemC-Implementation, indem der Accessor ein nach außen hin unsichtbares Signal erzeugt, wenn die Methode `is_reset()` das erste Mal aufgerufen wird. Das `sc_reset`-Objekt, das diese Methode zurückliefert, stammt von diesem Signal. Zudem wird intern ein Method-Prozess erzeugt, der die Events des Accessors an dieses Signal übermittelt. Über den Umweg über das Signal bekommt der CThread das Rücksetz-Event zwar mit einem Delta-Zyklus Verspätung, doch im Gegenzug ist es möglich, CThreads zusammen mit ihren Accessoren zu verschieben. Wenn der Prozess verschoben wird, ändert sich aufgrund der durch den Accessor gebildeten Zwischenschicht somit auch das Reset-Signal. Der beschriebene Mechanismus funktioniert ebenfalls zusammen mit dem Accessor des Exportals und somit auch mit exportierten Signalen. Wenn sicher ist, dass Module mit CThread-Prozessen in einer Simulation niemals verschoben werden, kann zur Optimierung die Compilerkonstante `RC_DISABLE_CTHREAD_MOBILITY` definiert werden. Diese bewirkt, dass auf den hier beschriebenen Spezialmechanismus im Accessor verzichtet wird.

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein vollständiges Re-Design der SystemC-Erweiterungsbibliothek ReChannel durchgeführt.

Als Einführung in die Thematik wurden in Kapitel 1 die grundlegenden Konzepte und Techniken beschrieben, die in dieser Arbeit Verwendung finden. Hierzu wurden die Sprache C++vorgestellt und Einblicke in einige Aspekte der Quellcode-Bibliothek Boost gegeben. Die Hardwarebeschreibungssprache SystemC wurde vorgestellt und ein Überblick über die unterschiedlichen mit SystemC beschreibbaren Berechnungsmodelle und Abstraktionsebenen gegeben. Ebenso wurden die Simulationssemantik von ReChannel sowie die Konzepte der zentralen ReChannel-Komponenten – Portal und Accessor – beschrieben. In diesem Zusammenhang ist auch der Funktionsumfang von ReChannel vor dem Re-Design beleuchtet worden.

In Kapitel 2 wurde systematisch analysiert, welche Probleme bei der Simulation von DR auf funktionaler bzw. transaktionaler Abstraktionsebene mit der ursprünglichen ReChannel-Bibliothek bestehen. Zu welchen Synchronisationsproblemen es bei der Rekonfiguration von abstrakten Beschreibungen kommen kann, wurde anhand eines Datenfluss-Beispiels veranschaulicht (vgl. Kapitel 2.1.1). Die hier beobachteten Probleme wurden als Dateninkonsistenz-, Deadlock- und Timing-Problematiken klassifiziert und dienten als Grundlage zur Formulierung der Anforderungen, die an eine mögliche allgemeine Lösung der Synchronisationsproblematik gestellt werden. Auch der Kontrollmechanismus von ReChannel wurde auf die Existenz von implementationsbedingten Synchronisationsproblemen überprüft. Das Ergebnis hierbei war, dass der Kontrollmechanismus einer tief gehenden Überarbeitung unterzogen werden musste. Die Funktionsweise des Accessors und des endlichen Automaten des rekonfigurierbaren Moduls ist für einige Synchronisationsprobleme direkt verantwortlich. Ferner wurde erörtert, worin die Ursache einer prinzipiellen Inkompatibilität, mit dem SystemC-Sprachstandard begründet ist (vgl. Kapitel 2.1.2), die sich durch Treiberkonflikte in Signalen äußert. Des Weiteren wurde aufgeführt, welche SystemC-Funktionalität zurzeit noch nicht bei der Simulation von DR mit ReChannel verwendet werden kann (vgl. Kapitel 2.1.3). Hierzu zählen u. a. SystemC-Exports, für die eine mögliche Lösung mittels einer Komponente namens Exportal diskutiert wurde. Weiterhin war aufgefallen, dass Signale, die mit Portals verbunden sind, in einer Mischform von Register und einfacher Datenleitung betrieben werden. Dies führte zu der Erkenntnis, dass das Portal die Möglichkeit zur Definition eines Schalterverhaltens benötigt (vgl. Kapitel 2.1.6).

Während der Analyse der Funktionsweise von ReChannel waren auch einige neue Problemstellungen aufgefallen. Es zeigte sich, dass eine Fülle von Beschreibungsarten und Konstrukten innerhalb von rekonfigurierbaren Modulen zu einem Verhalten führt, das sich mit der Deaktivierung durch eine äußere Kontrolle nicht in Einklang bringen

lässt (vgl. Kapitel 2.1.4 und 2.1.5). Weiterhin wurde festgestellt, dass einige Channeltypen momentan nicht auf die gewünschte Weise bei der Simulation von DR in ReChannel verwendet werden können. Die hiervon betroffenen Channels wurden anhand der Ursache für deren Inkompatibilität mit ReChannel in verschiedene Kategorien eingeteilt und die jeweils bestehende Problematik beschrieben (vgl. Kapitel 2.1.7). Ferner wurde die interne Verwaltung der ReChannel-Klassen hinsichtlich möglicher Optimierungen untersucht (vgl. Kapitel 2.1.8). An der ursprünglichen ReChannel-Bibliothek wurden die in Kapitel 2.2 aufgeführten Änderungen durchgeführt.

In Kapitel 3 wurde erörtert, aus welchen Gründen ein vollständiges Re-Design der ReChannel-Bibliothek erforderlich war. Dass ein Re-Design mit der angestrebten Zielsetzung auch technisch durchführbar ist, konnte mittels zweier experimenteller Implementierungen gezeigt werden. Aufgrund dessen wurde die Entscheidung getroffen, ein komplettes Re-Design von ReChannel-v1 durchzuführen. Die Zielsetzungen dieses Re-Designs wurden in Kapitel 4.1 beschrieben.

Nachdem die Abhängigkeiten veranschaulicht wurden, die Probleme bei der Erweiterung von ReChannel-v1 bereiteten, wurden aus den hieraus gewonnenen Erkenntnissen entsprechende Lösungskonzepte für den Entwurf der Basisimplementation der neuen Bibliothek (ReChannel-v2) entwickelt (vgl. Kapitel 4.2) und anschließend das Re-Design durchgeführt.

Es wurde eine neue Klassenhierarchie entwickelt, bei der Portal (vgl. Kapitel 4.9.2) und Accessor (vgl. Kapitel 4.8) lediglich Implementationsdetails darstellen und dem zentralen Rekonfigurationsalgorithmus nicht bekannt sind. Dies wurde durch die Einführung des Switch-Konzeptes (vgl. Kapitel 4.4) und der Interface-Wrapper- und Accessor-ABIs (vgl. Kapitel 4.7) erreicht. Die Konzepte der bisher in ReChannel verwendeten Algorithmen und Datenstrukturen wurden dahin gehend abstrahiert, dass die Erweiterung um beliebige, zusätzliche Switch-Implementationen keine Änderungen am Rekonfigurationsalgorithmus erforderlich machten. Die gegenseitigen Abhängigkeiten der Klassen wurden minimiert und die Wiederverwendbarkeit von Code dadurch erhöht. So konnte z. B. zusätzlich zum Portal auch das Exportal implementiert werden, das für die Rekonfiguration von Exports benötigt wird (vgl. Kapitel 4.9.5). Auch wurde ermöglicht, dass der Accessor vielseitig verwendbar ist, anstatt fest an den Rekonfigurationsalgorithmus gebunden zu sein. Der Kontrollmechanismus und sämtliche Zustandsmodellierungen wurden neu entworfen, wodurch die in Kapitel 2 festgestellten Probleme gelöst werden konnten.

ReChannel-v2 wurde um Sprachkonstrukte zur Modellierung grundlegender Synchronisationsfunktionalität erweitert. Mittels Filtern und Transaktionszählern werden einem Anwender in ReChannel-v2 die grundlegenden Mittel zur Verfügung gestellt, um funktionale und transaktionale Modulbeschreibungen nachträglich mit beliebiger Synchronisationsfunktionalität ausstatten zu können. Der Entwurf des Filter-Konzepts wurde in Kapitel 4.13 beschrieben. Die praktische Anwendbarkeit von Filtern wurde anhand von FIFO-Filtern demonstriert.

Bei Überlegungen während der Vorbereitung zu dieser Arbeit ist ein Konzept entstanden, wie sich implizit rücksetzbare Prozesse und implizit rücksetzbare Komponenten als Spracherweiterung in SystemC integrieren lassen, ohne dass Änderungen am Simulationskernel von SystemC vorgenommen werden müssen. Die ReChannel-Bibliothek

wurde daher um eine eigene Prozess-API mit Prozessregistrierung (vgl. Kapitel 4.11) und einer Infrastruktur für rücksetzbare Komponenten (vgl. Kapitel 4.12.2) erweitert. Für jedes SystemC-Sprachkontrukt wird eine rücksetzbare Entsprechung mit „rc_“-Präfix zur Verfügung gestellt, so dass ReChannel nun auch zur expliziten Modellierung von DRHW verwendet werden kann. Diese Erweiterung fügt sich nahtlos in das bestehende ReChannel-Konzept ein, woraufhin es nun z. B. möglich ist, ein Modul von Grund auf neu zu modellieren oder eine bestehende Beschreibung nachträglich zu erweitern.

Um weitere Eigenschaften eines Rekonfigurationsstreams auch in der Simulation abbilden zu können, wurde die Möglichkeit zur Mobilität von Modulen in einem Design implementiert (vgl. Kapitel 4.15). Da eine Voraussetzung hierfür ist, dass Switch-Bindungen automatisch gelöst und eingegangen werden können, wurden zu diesem Zweck die Switch-Connector- und die Portmap-Komponenten entworfen und implementiert (vgl. Kapitel 4.14).

Ausblick

Bei der Implementierung der ReChannel-Bibliothek wurde das Ziel verfolgt, sämtliche SystemC-Sprachkonstrukte möglichst in Übereinstimmung mit dem SystemC-Standard zu verwenden. Hierdurch sollte erreicht werden, dass die ReChannel-Bibliothek zwischen verschiedenen SystemC-Implementation portabel ist. Diese Eigenschaft konnte jedoch nicht hinreichend getestet werden, da als Einzige die SystemC-Implementation der OSCI zur Verfügung stand. Hinsichtlich dessen könnte es sinnvoll sein, ReChannel in Zukunft auch noch mit einigen anderen SystemC-Implementationen zu testen.

Über die genauen Ursachen für die Einschränkungen, die noch in der Prozess-API bezüglich der Verwendung der Wartefunktionen und des asynchronen Rücksetzverhaltens bestehen (siehe Kapitel 4.11.1), könnten noch einmal genauere Nachforschungen angestellt werden. Die Einschränkungen könnten möglicherweise auch dadurch bedingt sein, dass benutzerdefinierte Rücksetzbedingungen erst nachträglich hinzugefügt worden sind.

Wenn in einem Design sehr häufige Rekonfigurationen durchgeführt werden müssen, kann das Schalterverhalten einiger Portals und Exportals möglicherweise einen Flaschenhals darstellen. Die Schalterlogik benachrichtigt bei Öffnung die entsprechenden Events, um wartende Prozesse zu aktivieren. Die Events innerhalb der Switches werden jedoch anhand ihres Namens identifiziert, der durch einen String repräsentiert wird (vgl. Listing 4.6, S. 99). Der Anwender kann so zwar komfortabel die Events per Namen ansprechen, doch führt dies bei Suchoperationen in den internen Map-Datenstrukturen zu einem erhöhten Aufwand. Hier besteht folglich noch Optimierungsbedarf.

Bei der Weiterentwicklung von ReChannel könnte ein nächster Schritt sein, auch vordefinierte Accessoren, Portals und Exportals für die Verwendung der TLM-Bibliothek der OSCI [14] hinzuzufügen. Die Grundvoraussetzungen müssten hierfür in ReChannel-v2 gegeben sein. Da die TLM-Bibliothek speziell dazu entwickelt wurde, eine allgemeine Methodologie für transaktionale Beschreibungen zu definieren, wäre es sehr interessant zu überprüfen, ob ReChannel tatsächlich mit den Interfaces und Konzepten der TLM-Bibliothek verwendet werden kann.

Zur Evaluierung des Filter-Konzepts sollte ein Einsatz in einem realen Projekt durchgeführt werden. Hierbei wäre speziell interessant, ob die praktische Verwendbarkeit von Filtern auch noch bei komplexen Bus-Systemen gegeben ist.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Literaturverzeichnis

[1] Andreas H. Niers. Evaluierung des ReChannel-Workflows zur Synthese von dynamisch rekonfigurierbaren SystemC-Modellen, Dezember 2007.

[2] Black, D. *SystemC From The Ground Up.* Springer, Berlin, 2005.

[3] Boost. *Boost C++ Libraries.* `http://www.boost.org`.

[4] Bruce Eckel and Chuck D. Allison and Chuck Allison. *Thinking in C++, Vol. 2.* Pearson Education, 2003.

[5] C++ Standards Committee. *C++.* `http://www.open-std.org/jtc1/sc22/wg21`.

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Entwurfsmuster.* Addison-Wesley, 1996.

[7] IEEE Standards Association ("IEEE-SA") Standards Board. *IEEE Std 1666-2005 Open SystemC Language Reference Manual*, 2005. `http://www.systemc.org`.

[8] John Aynsley, Doulos, Ringwood, UK. *Here's Exactly What You Can Do with the New SystemC Standard!*, 2006. `http://www.doulos.com/knowhow/systemc/new_standard/`.

[9] Lee, E. A., and Parks, T. M. Dataflow process networks. In *Proceedings of the IEEE, Volume 83* (1995), pp. 773–799.

[10] Lysaght, P., and Stockwood, J. A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 4*, 3 (1996), 381–390.

[11] Marshall Cline. *C++ FAQ LITE – Frequently Asked Questions.* `http://www.parashift.com/c++-faq-lite/`.

[12] Open SystemC Initiative (OSCI). *An Introduction to System-Level Modeling in SystemC 2.0.* `http://www.systemc.org`.

[13] Open SystemC Initiative (OSCI). *SystemC.* `http://www.systemc.org`.

[14] Open SystemC Initiative (OSCI). *TLM Transaction Level Modeling Library, Release 2.0 Draft 1*, 2006. `http://www.systemc.org`.

[15] Pasricha, S. Transaction Level Modeling of SoC with SystemC 2.0, 2002.

[16] Pelkonen, A., Masselos, K., and Cupak, M. System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. *Proceedings of International Symposium on Parallel and Distributed Processing (Reconfigurable Architecturs Workshop)* (Apr. 2003).

[17] Projektgruppe "Electronic Design Automation". ReChannel-v1 (Doxygen Dokumentation), 2005/2006.

[18] Raabe, A., and Anlauf, J. K. A SystemC Reconfiguration Simulation Library. *(unpublished)* (2005).

[19] Raabe, A., Hartmann, P. A., and Anlauf, J. K. ReChannel: Describing and Simulating Reconfigurable Hardware in SystemC. *ACM Transactions on Design Automation of Electronic Systems (accepted).*

[20] Schallenberg, A., Oppenheimer, F., and Nebel, W. Designing for dynamic partially reconfigurable FPGAs with SystemC and OSSS. In *Forum on Specification and Design Languages* (Lille, France, Sept. 2004).

[21] Silicon Graphics (sgi). *Standard Template Library Programmer's Guide.* `http://www.sgi.com/tech/stl/`.

[22] Stephen Brown and Jonathan Rose. Architecture of FPGAs and CPLDs: A Tutorial, 1996.

[23] Xilinx (Application Note 290). Two Flows for Partial Reconfiguration: Module Based or Difference Based, September 2004.

Anhang A – ReChannel-v1

Die ursprüngliche Idee und „Proof of Concept“ des Projekts ist in [18] beschrieben. An der Implementierung der ReChannel-Bibliothek im Rahmen der Projektgruppe “Electronic Design Automation“ (im Wintersemester 2005/2006) waren die folgenden Personen beteiligt:

Leitung:
Prof. Dr. Joachim K. Anlauf,
Dipl.-Inform. Andreas Raabe,
Dipl.-Inform. Philipp A. Hartmann

Studenten (in alphabetischer Reihenfolge):
Bjoern Bales,
Thomas Becker,
Thomas Loraing,
Michael Nolden,
Rebecca Reiffenheuser,
Uwe Schuster,
Ralph Thesen,
Jennifer Wolf.

Anhang B – ReChannel-v2

Die Dokumentation (engl.) der ReChannel-v2-Bibliothek befindet sich auf der dieser Arbeit beliegenden CD. Die Dokumentation wurde mit *Doxygen* aus den Quellcode-Dateien des ReChannel-v2-Projekts extrahiert und enthält eine vollständige Auflistung aller in der Bibliothek enthaltenen Klassen, Methoden und Funktionen. Für die Erstellung der Dokumentation mittels Doxygen existiert im Makefile des „src“-Ordners auch ein separates *make*-Target namens „doc“.

make-Targets

***make*-Target**	**Beschreibung**
ReChannel *(default)*	Kompiliert die ReChannel-Bibliothek
doc	Erzeugt die ReChannel-Dokumentation
doc-systemc	Erzeugt die Dokumentation für SystemC
doc-all	Erzeugt die Dokumentation für SystemC und ReChannel
all	Erzeugt die Bibliothek und die Dokumentationen
clean	Entfernt u. a. die Compilerobjektdateien
distclean	Entfernt die Compilerdateien und die Dokumentation

ReChannel-Sprachkonstrukte mit SystemC-Entsprechung

```
RC_REPORT_INFO     (macro)
RC_REPORT_WARNING  (macro)
RC_REPORT_ERROR    (macro)
RC_REPORT_FATAL    (macro)
rc_assert          (macro)
RC_MODULE          (macro)
rc_module          (class)
rc_behavior        (typedef)
rc_channel         (typedef)
RC_CTOR            (macro)
RC_HAS_PROCESS     (macro)
RC_METHOD          (macro)
RC_THREAD          (macro)
RC_CTHREAD         (macro)
rc_module_name     (typedef)
rc_spawn_options   (class)
rc_spawn           (method)
rc_bind            (macro)
rc_ref             (macro)
rc_cref            (macro)
```

```
RC_FORK            (macro)
RC_JOIN            (macro)
rc_process_handle  (class)
rc_port            (macro)
rc_in              (typedef)
rc_inout           (typedef)
rc_out             (typedef)
rc_in_resolved     (class)
rc_inout_resolved  (class)
rc_out_resolved    (class)
rc_in_rv           (class)
rc_inout_rv        (class)
rc_out_rv          (class)
rc_fifo_in         (typedef)
rc_fifo_out        (typedef)
rc_export          (macro)
rc_next_trigger    (function)
rc_wait            (function)
rc_get_current_process_handle (function)
rc_prim_channel    (class)
rc_signal          (class)
rc_buffer          (class)
rc_signal_resolved (class)
rc_signal_rv       (class)
rc_fifo            (class)
rc_sc_event        (class)
rc_sc_signal       (class)
rc_sc_buffer       (class)
rc_sc_signal_resolved (class)
rc_sc_signal_rv    (class)
rc_sc_fifo         (class)
rc_mutex           (class)
rc_semaphore       (class)
```

```
class rc_switch
{
  friend class rc_reconfigurable;
  friend class rc_switch_connector_base;
public:
  enum state_type { UNDEF=0, OPEN, CLOSED };
  enum { STATE_COUNT=3 };
  typedef std::vector<rc_interface_filter*> filter_chain;
public:
  virtual std::string get_switch_kind() const = 0;
  virtual std::string get_switch_name() const = 0;
  virtual state_type get_switch_state() const = 0;
  virtual unsigned int get_transaction_count() const = 0;
  virtual sc_interface* get_static_interface() const = 0;
  virtual sc_interface* get_dynamic_interface() const = 0;
  virtual rc_reconfigurable* get_current_reconfigurable() const = 0;
  virtual bool is_locked() const = 0;
  virtual bool is_registered(const rc_reconfigurable& reconf) const = 0;
  virtual bool is_registered(const sc_interface& dyn_if) const = 0;
protected:
  virtual void bind_static_object(
      const rc_object_handle& obj_to_bind) = 0;
  virtual void bind_dynamic_object(
      const rc_object_handle& obj_to_bind) = 0;
  virtual void open() = 0;
  virtual void open(
      rc_reconfigurable& reconf,
      const filter_chain& filters = filter_chain()) = 0;
  virtual void close() = 0;
  virtual void set_undefined() = 0;
  virtual void refresh_notify() = 0;
  virtual void register_reconfigurable(
      rc_reconfigurable& reconf, sc_interface& dyn_if) = 0;
  virtual void unregister_reconfigurable(
      rc_reconfigurable& reconf) = 0;
  virtual sc_interface* get_registered_interface(
      rc_reconfigurable& reconf) const = 0;
  virtual bool is_lock_owner(const rc_reconfigurable& reconf) const = 0;
  virtual bool set_locked(rc_reconfigurable& lock_owner, bool lock) = 0;
};
```

Listing B.1: Das Switch-ABI (siehe Kapitel 4.4)

```
template<
    class R, class A1, class A2, class A3, class A4,
    class A1_, class A2_, class A3_, class A4_, class IF_>
inline R rc_nb_forward(
    R (IF_::*method)(A1_, A2_, A3_, A4_) const,
    A1 a1, A2 a2, A3 a3, A4 a4) const
{
  // Prozess-Handle des aktuellen Prozesses abfragen
  rc_process_handle hproc = rc_get_current_process_handle();
  while(true) {
    try {
      // Fallunterscheidung: Welche Art von Target?
      if (p_target_if != NULL) { // Target ist ein Interface
        // Rücksetzbedingung für den Aufruf ändern (falls eingestellt)
        rc_process_behavior_change temp =
          this->_rc_process_behavior_change(hproc);
        // IMC an das Target-Interface weiterleiten
        return (p_target_if->*method)(a1, a2, a3, a4);
      } else if (p_target_wrapper != NULL) { // Target ist ein IFW
        // IMC an den Interface-Wrapper weiterleiten
        return (p_target_wrapper->get_interface_access()->*method)
                                                    (a1, a2, a3, a4);
      } // else: kein Target verfügbar
    } catch(rc_process_cancel_exception* e) {//Rücksetz-Events abfangen
      // ist die RÜcksetzbedingung erfüllt?
      if (hproc.is_canceled()) {
        // Rücksetz-Event erneut auslösen
        ::ReChannel::rc_throw(e);
      } else {
        // Rücksetz-Event konsumieren
        delete e;
      }
    }
    // warten, bis wieder ein Target verfügbar ist
    this->_rc_wait_activation();
    // dann Aufruf wiederholen
  }
}
```

Listing B.2: Code einer Weiterleitungsmethode des Accessors für einen blockierenden (const) IMC mit vier Parametern und einem Rückgabewert vom Typ R [Anm.: Der Code im Listing liegt bereits in substituierter Form vor. In der Bibliothek werden Makros verwendet, um den Code der Methodentemplates zu bilden.]

```
template<class R, class A1, class A1_, class IF_>
inline R rc_nb_forward(R (IF_::*method)(A1_), A1 a1) const
{
  try {
    // Fallunterscheidung: Welche Art von Target?
    if (p_target_if != NULL) {
      // IMC an das Target-Interface weiterleiten
      return (p_target_if->*method)(a1);
    } else if (p_target_wrapper != NULL) {
      // IMC an den Interface-Wrapper weiterleiten
      return (p_target_wrapper->get_nb_interface_access()->*method)(a1);
    } // else: kein Target verfügbar
  } catch(rc_process_cancel_exception* e) { //Rücksetz-Events abfangen
    // Prozess-Handle des aktuellen Prozesses abfragen
    rc_process_handle hproc = rc_get_current_process_handle();
    if (hproc.is_canceled()) {
      // Rücksetz-Event erneut auslösen
      ::ReChannel::rc_throw(e);
    } else {
      // Rücksetz-Event konsumieren
      delete e;
    }
  }
  // IMC auf dem Fallback-Interface ausführen
  return (rc_get_fallback_if().*method)(a1);
}
```

Listing B.3: Code einer Weiterleitungsmethode des Accessors für einen nichtblockierenden IMC mit einem Parameter und einem Rückgabewert vom Typ R [Anm.: Der Code im Listing liegt bereits in substituierter Form vor. In der Bibliothek werden Makros verwendet, um den Code der Methodentemplates zu bilden.]

```
#define RC_HAS_VAR(user_module_name)                                    \
  template<                                                             \
    class _rc_var_P,                                                    \
    class _rc_var_T,                                                    \
    _rc_var_T& (_rc_var_P::*_rc_var_get_ref)(void)                      \
  > friend class                                                        \
    ::ReChannel::internals::resettable_var::resettable_var; \
  typedef user_module_name RC_CURRENT_VAR_CONTAINER

#define RC_DECLARE_VAR(var_type, var_name)                              \
  var_type& _rc_var_get_##var_name() { return this->var_name; } \
  ::ReChannel::internals::resettable_var::resettable_var<               \
    RC_CURRENT_VAR_CONTAINER, var_type,                                 \
    &RC_CURRENT_VAR_CONTAINER::_rc_var_get_##var_name>                  \
      _rc_var_##var_name

#define RC_VAR(var_type, var_name)    \
  var_type var_name;                  \
  RC_DECLARE_VAR(var_type, var_name)

#define rc_var(var_type, var_name) \
  RC_VAR(var_type, var_name)
```

Listing B.4: Code der Deklarationsmakros für rücksetzbare Variablen (siehe Kapitel 4.17)

```
#define RC_METHOD(func) \
  if (true) { \
    if (this->_rc_process_support.has_reconfigurable_context()) { \
      this->_rc_process_support.declare_process( \
        *this, \
      boost::bind( \
        &RC_CURRENT_USER_MODULE::_rc_declare_method, this), \
      boost::bind( \
        &RC_CURRENT_USER_MODULE::func, this), \
      this->sensitive); \
    } else { \
      SC_METHOD(func); \
    } \
  } else (void)0
```

Listing B.5: Code des Deklarationsmakros für einen rücksetzbaren Method-Prozess (siehe Kapitel 4.11.2)

```
template<class M> // M may be the type of an existing encryption module
RC_RECONFIGURABLE_MODULE_DERIVED(CryptoModule, M) {
    // sync. handshake to communicate with an external control
    rc_fifo_in<bool>  sync_in;
    rc_fifo_out<bool> sync_out;

    // constructor
    RC_RECONFIGURABLE_CTOR_DERIVED(CryptoModule, M),
        key_in_filter  (NULL, NULL, 0),
        data_in_filter (NULL, NULL, 0),
        data_out_filter(NULL, RC_SYNC_CALLBACK(on_data_written))
    {
        // assign filters to the ports
        rc_add_filter(this->key_in,   key_in_filter);
        rc_add_filter(this->data_in,  data_in_filter);
        rc_add_filter(this->data_out, data_out_filter);
        // process responsible for synchronization
        RC_THREAD(sync_proc);
    }
    void on_data_written(bool nb);
    void sync_proc();

    // FIFO-filters
    rc_fifo_in_filter<sc_bv<192> >  key_in_filter;
    rc_fifo_in_filter<sc_bv<192> >  data_in_filter;
    rc_fifo_out_filter<sc_bv<192> > data_out_filter;
    // notified when output data has been written:
    sc_event data_written_event;
};
// called directly after an output word is written:
template<class M> void CryptoModule<M>::on_data_written(bool nb) {
    data_written_event.notify(); // notify data_written_event
}
// synchronisation process
template<class M> void CryptoModule<M>::sync_proc() {
    RC_TRANSACTION { // begin transaction
        while(true) {
            // read the next sync command
            bool sync_cmd = sync_in.read();
            if (sync_cmd == false) { // deactivation command received
                sync_out.write(true); // send ack
                return; // "terminate" resettable process
            }
            // increase read limit of the input filters by one
            key_in_filter.incr_read_limit(1);
            data_in_filter.incr_read_limit(1);
            // wait for output data has been written
            wait(data_written_event);
            sync_out.write(true); // send ack
        }
    } // end transaction
}
```

Listing B.6: Beispiel: Generisches RM mit Synchronisationsfunktionalität für beliebige, gleichartige Kryptographie-Module

Anhang C – Legende

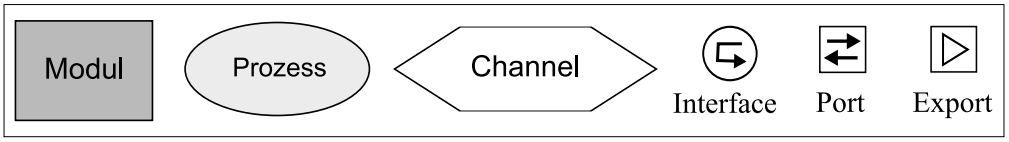

Abbildung C.1: Legende der Symbole, die in dieser Arbeit für die Darstellung von SystemC-Komponenten verwendet werden

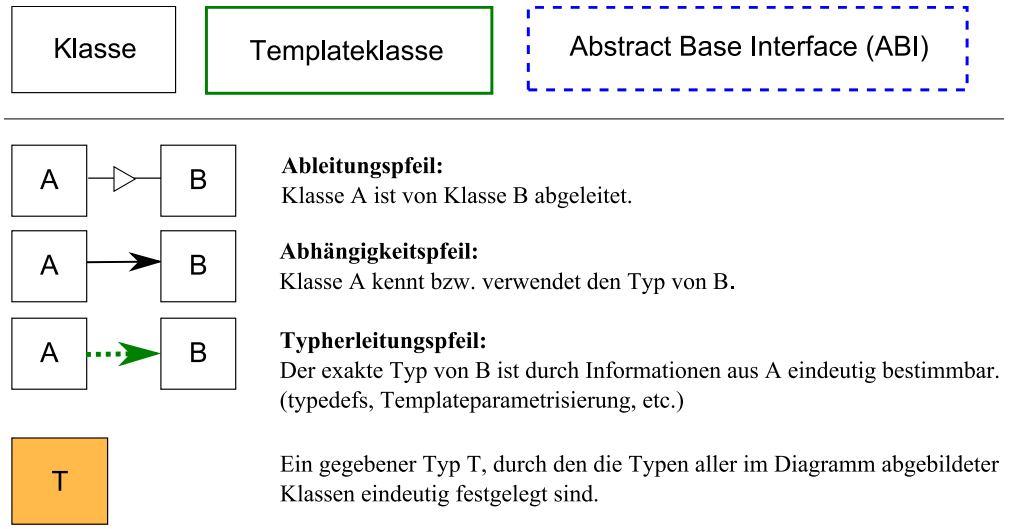

Abbildung C.2: Symbollegende für die in Kapitel 4 verwendeten Klassendiagramme