Kapitel 1

Einleitung

VLSI-Bausteine der heutigen Zeit und die aus ihnen aufgebauten digitalen Systeme können zu den komplexesten Konstruktionen gezählt werden, die bisher technisch realisiert wurden. Ihre Komponenten bestehen teilweise aus mehreren hunderttausend Objekten, die selber nichttriviale Funktionalitäten zur Verfügung stellen. Der Entwurf dieser Hardware-Systeme steht unter dem steten Zwang, in immer kürzeren Zeitzyklen fehlerfreie Designs marktreif zu erstellen. Etwaige Entwurfsfehler im Design müssen deshalb frühzeitig aufgespürt werden, um die Höhe der Folgekosten für ein Re-Design als auch den Zeitverlust bis zur Markteinführung minimal zu halten. Neben diesen ökonomischen Gründen steht der steigende Einsatz von komplexen Hardware Systemen in sicherheitskritischen Bereichen wie z.B. Luft-/ Raumfahrt und nukleartechnischen Anlagen. Ein Versagen bzw. fehlerhaftes Verhalten in diesen Bereichen kann zu nicht mehr überschaubaren katastrophalen Folgen führen.

Es ist ersichtlich, daß eine frühzeitige Simulation bzw. formale Verifikation eines Gesamtentwurfs notwendig ist, um sicherzustellen, daß Systeme dieser Komplexitätsklasse von Beginn an korrekt entworfen werden können. Hierbei stellt die Simulation von Hardware-Designs nur eine Fehlersuchtechnik dar, die nur eine Aussage in der Form erlaubt, daß noch kein Fehler gefunden werden konnte. Dies ist in frühen Prototyp-Phasen akzeptabel und kann ein Grundverständnis der Funktionalität der Realisierung des neu entwickelten Systems dem Designer vermitteln. Die generelle Korrektheit des Entwurfs kann jedoch nur durch eine formale Verifikation nachgewiesen werden. Hierbei wird Korrektheit in ihrer exakten mathematischen Bedeutung verstanden, d.h. es wird formal, basierend auf einem mathematischen Kalkül, sichergestellt, daß der Entwurf frei von Fehlern in Bezug auf eine gegebene formale Spezifikation ist.

Die Anwendung mathematischer Methoden im Entwurf von Hardware wurde schon früh durch den Einsatz von booleschen Algebren und der Automatentheorie auf der Abstraktionsebene von Gate-Level Designs eingeführt. Die Anwendung dieser Methodik im Kontext des VLSI Designs als auch die für die Systemebene notwendigen höheren Abstraktionsebenen führen zu neuen Problemstellungen für die abstrakte Simulation bzw. formale Verifikation, da u.a. komplexe Datentypen und Protokolleigenschaften behandelt werden müssen. Bei der Analyse von komplexen Entwürfen konnte jedoch auf die schon erlangten Erfahrungen aus der Programmverifikation zurückgegriffen werden. Abbildung 1.1 gibt einen Eindruck von der Komplexität des Einflußspektrums auf den Entwurf korrekter, hardwarenaher Systeme [CP87, CP89, Yoe90].

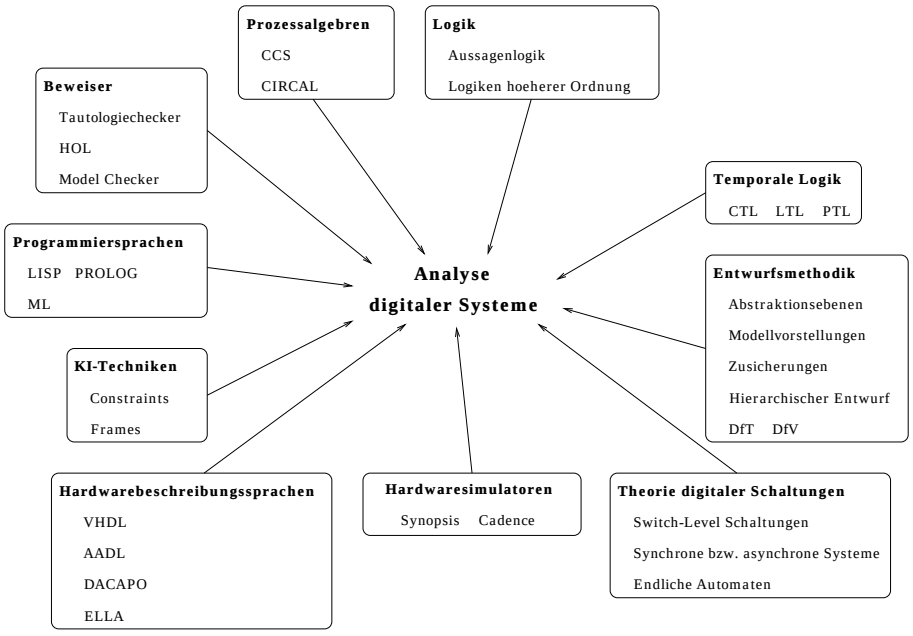


Abbildung 1.1: Einflüsse auf die Analyse des Hardwaredesigns

Mit Hilfe temporaler Logik kann eine formale Beschreibung eines Designs erfolgen und somit eine rigide mathematische Analyse ermöglicht werden. Eine vollständige Spezifikation des Gesamtsystems in temporaler Logik kann nur sehr schwer angegeben werden und ist dann nicht prägnant, d.h. für einen Hardwaredesigner intuitiv verständlich.[1] Dies kann effektiver durch eine Hardwarebeschreibungssprache (Architecture Definition Language ADL) geschehen, deren Konstrukte viel problemorientierter als ein allgemeines Kalkül sind. Die Semantik der Konstrukte der

[1] Eigenschaften über das Verhalten der Hardware lassen sich hingegen leicht in temporaler Logik spezifizieren.

ADL muß jedoch durch eine Axiomatisierung mathematisch faßbar gemacht werden [DDG+89, DJS93c], um formale Aussagen über Entwürfe in einer ADL erstellen und beweisen zu können.

Das COMDES(*COM*puter architecture *DES*ign tool)-System basiert auf der Hardwarebeschreibungssprache *AADL* (*A*xiomatic *ADL* [DDG+89]), eine Sprache, deren atomare Aktionen/Transformationen axiomatisch mittels pre/post Bedingungen spezifiziert werden können. AADL wurde im Zusammenhang eines ESPRIT-Projektes und im Kontext industrieller Multiprozessor-Fallstudien entwickelt und formalisiert. AADL erlaubt eine Unterstützung des Designers in vielen Phasen/Ebenen des Hardware Entwurfs (siehe Tabelle 1.1). In dieser Übersicht sind den Abstraktionsebenen der Entwurfshierarchie ihre ihnen zugeordneten Strukturen und Verhaltensbeschreibungen gegenübergestellt. Auf der Stufe der Gatter-Ebene kann das Verhalten der Objekte noch durch boolesche Operationen beschrieben werden. Durch höhere Abstraktionsebenen nimmt zwar die relative Anzahl der Objekte ab, die ein Gesamtsystem beschreiben, jedoch werden ihre Strukturen und die von ihnen verarbeitbaren Daten immer differenzierter und komplexer. Eine Entwicklungsumgebung für den Hardware Entwurf, wie z.B. das COMDES-System, welches diese höheren Designebenen unterstützen will, muß sowohl komplexe hierarchische Strukturen als auch komplexe, abstrakte Kontroll- und Datenstrukturen unterstützen.

Abstraktionsebene	Strukturen	Verhalten
Architektur		spezifizierte Verhalten
System	**CPU, Speicher, Cache**	**abstrakte Mikroprogramme**
Register-Transfer	ALU, Register, MUX	arithmetische Operationen
Gatter	NAND, NOR, XOR	boolesche Operationen
Transistor	Transistoren, Widerstände	

Tabelle 1.1: Ebenen der Entwurfsdarstellung im Hardware-Design

AADL erlaubt eine modulare und kompakte Spezifikation komplexer Hardware Entwürfe von der Gatterebene bis hin zur Architekturebene. Sowohl die theoretischen Konzepte als auch die syntaktischen Konstruktoren von AADL sind an dieses breite Spektrum von Abstraktionsebenen angepaßt, basierend auf dem uniformen Konzept, daß Module mit ihrer Umgebung Dienstleistungen/Daten nur durch *Ports* austauschen können und somit der interne Aufbau der Module der Umgebung oberhalb dieser Schnittstelle verborgen bleibt. Hierbei können sowohl reine strukturelle Module auf Gate Level als auch Module mit einer Verhaltenbeschreibung in einer CSP ähnlichen Notation zur Anwendung kommen.

Durch die hardwarenahe Spezifikationsmöglichkeit von AADL ergibt sich die Notwendigkeit, parallelen Zugriff auf shared memory Strukturen und parallele, nichtprozedurale Aktivierung von atomaren Aktionen abhängig vom Zustand der Architektur zu unterstützen. Eine adäquate Behandlung dieser beiden komplexen Interaktionen auch im Konfliktfall, z.B. gleichzeitig schreibender Zugriff zweier Prozesse auf eine kritische Ressource des Designs, kann nur durch eine auf partieller Ordnung basierenden Semantik repräsentiert werden. Eine kompositionelle Semantik, die dieser Anforderung genügt, kann in zwei Schritten aus AADL entwickelt werden: Ein AADL-Programm kann im ersten Schritt in ein 1-sicheres Petri-Netz abgebildet werden, in dem die Parallelität explizit sichtbar wird. Ausgehend von einem Prozeß (eine azyklische, konfliktfreie Abwicklung) des Netzes kann die Semantik als Abbildung der Stellen des Prozesses auf Zustände des AADL Moduls angesehen werden, wobei Zustandstransformationen der atomaren Aktionen der Architektur auf Transitionen des Prozesses abgebildet werden.

Wie stellt sich jedoch die Entwurfsmethodik des Hardware Designs in diesem Kontext dem Benutzer dar, die das COMDES-System transparent unterstützen muß?

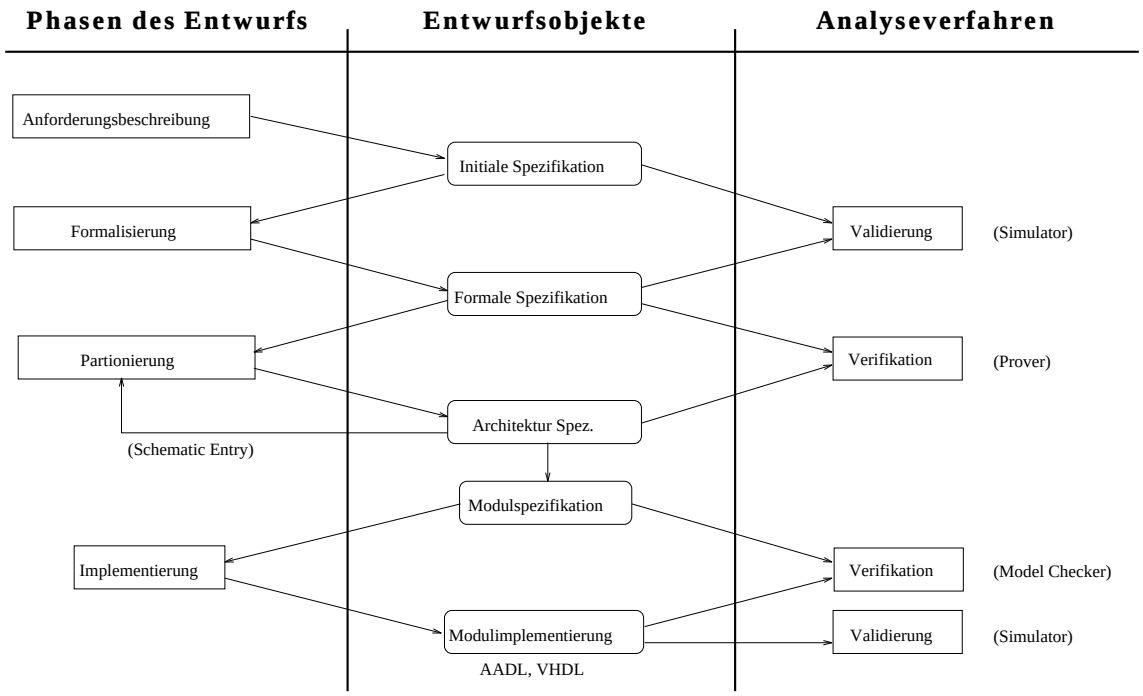


Abbildung 1.2: COMDES - Entwurfsmethodik

Abbildung 1.2 versucht, den Phasen des Designprozesses ihre Entwurfsobjekte und mögliche Analyseverfahren gegenüber zu stellen. Hierbei wird davon ausgegangen, daß der Ausgangspunkt eines Entwurfs eine allgemeine, informelle Verhaltensbe-

schreibung des Moduls als Anforderungskatalog gegeben ist, die als initiale Spezifikation angesehen werden kann. In einem Formalisierungsprozeß wird dann eine formale Spezifikation entwickelt, die schon durch Simulation im Vergleich mit der Anfangsspezifikation evaluiert werden kann. Da dieser Entwurf auf einer sehr abstrakten Ebene angesiedelt sein kann, der noch alle Eigenschaften des Systems vereinigt, schließt sich hieran ein Partitionierungsprozeß an. Als Ergebnis wird eine modularisierte und ggf. hierarchisch aufgebaute Spezifikation abgeleitet. Mögliche Designentscheidungen der Partitionierung können z.B. eine Verteilung von Funktionalitäten auf mehrere Komponenten oder eine Abbildung von abstrakten Datentypen auf hardwarenähere Strukturen sein. Ist dieser Prozeß abgeschlossen, so kann die neue Spezifikation dahin verifiziert werden, ob die Komposition aller Teilspezifikationen der Anfangsspezifikation formal entspricht. Als letzte Phase der Realisierung des Entwurfs findet eine Implementierung des Verhaltens der Modulspezifikationen statt. Bei diesem Konkretisierungsprozeß kann sowohl auf schon vorhandene und erfolgreich verifizierte Module zurückgegriffen werden, als auch auf neu zu realisierende Module aufgebaut werden, die dann in einer sich anschließenden Analysephase gegen ihre formale Spezifikation verifiziert werden müssen. In frühen Phasen der Implementierung eines Moduls kann durch Simulation schon frühzeitig eine Validierung erfolgen, um grobe Fehler rechtzeitig ausschließen zu können.

Der Systemumfang des COMDES-Systems (Abbildung 1.3) trägt diesen Anforderungen Rechnung. COMDES stellt Spezifikations-Tools zur Verfügung, die sowohl die formale Spezifikation von Entwürfen in textueller als auch in graphischer Repräsentation unterstützen. Eine Dekomposition der Entwürfe wird wiederum auf beiden Medienebenen durch Tools unterstützt, die automatisch statische Semantik-Checks durchführen und auf etwaige Inkonsistenzen hinweisen. Zur Validierung erstellter Entwürfe, sowohl der Spezifikationen als auch der Implementierungen, können komplexe Simulatoren angewandt werden. Sie dienen zur graphischen Animation, Visualisierung von erzeugten Datenmengen oder als Erklärungskomponente von Widerlegungsequenzen, die als Ergebnis der Verifikation von Modulimplementierungen gegen ihre formale Spezifikation erzeugt wurden. Um möglichst effizient komplexe Zugriffe auf große Datenmengen schnell durchführen zu können, wurde das Gesamtsystem auf einer globalen Datenbank aufgebaut, die als abstrakter Datentyp realisiert ist. Die Client/Server-Architektur von COMDES ermöglicht den einzelnen Tools durch ihre integrierte Netzwerkschnittstelle einen transparenten Zugriff auf alle Betriebssystemressourcen des logischen Netzwerkes. Dies wird durch die Realisierung der auf dem X11-System beruhenden Userinterface homogen weitergeführt.

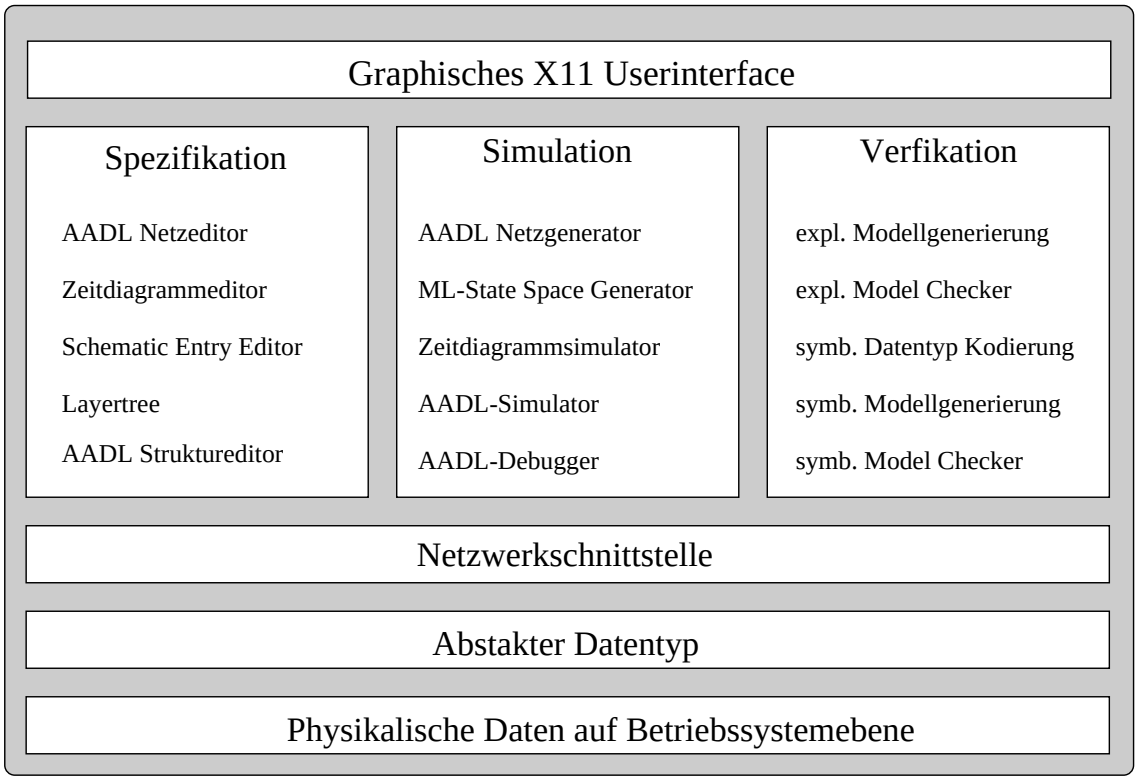


Abbildung 1.3: COMDES-Systemumfang

Um diesen funktionalen Anforderungen gerecht zu werden, d.h. ein Tool-Set zu entwickeln, welches den Designer elegant während des Entwurfsprozesses unterstützt, mußte neben einer Vielzahl von ingenieurmäßigen Problemstellungen (z.B. Software-Engineering, Datenhaltung, Client/Server-Architektur) eine groe Anzahl theoretischer Problemstellungen gelöst werden:

Im Kontext der Simulation von AADL Designs wurde eine effiziente modulare Simulation auf Petri-Netz-Ebene entwickelt, die es erlaubt, selbst komplexe Multiprozessor-Fallstudien schnell compilieren, konfigurieren und simulieren zu können. Desweiteren mußten Konzepte für High-Level Debugging eines parallelen, verteilten Systems entwickelt und realisiert werden. Hierbei wurden graphische Tools entwickelt, die ein Navigieren zwischen Hierarchieebenen eines Entwurfs anhand der Verbindungsstrukturen erlauben. Hierbei erfolgte eine transparente Einbettung in die Simulationsumgebung dahingehend, daß auf graphischer Ebene Breakpoints definiert und visualisiert werden können.

Die formale Verifikation von Modulimplementierungen gegen ihre gegebene formale Spezifikation wird im COMDES-Systems mittels automatischem Model Checking

[BCMD90, BCL91] realisiert. Da dieses Verfahren ein endliches Modell des zu verifizierenden AADL Moduls voraussetzt, wurden Kodierungsverfahren entwickelt, die explizit Kontroll- und Typinformationen eines AADL Moduls in einem einzigen Modell repräsentieren. Da explizite Repräsentationen komplexer Entwürfe auf Systemebene durch eine Zustandsexplosion jedoch schnell Größenordnungen erreichen, die nicht mehr im Computer darstellbar sind, wurden symbolische Kodierungsverfahren entwickelt, um diesem Problem ausweichen zu können. Für diese durch ROBDDs [BCM$^+$90, McM93] repräsentierten Modelle wurden symbolische Reduktionstechniken entwickelt, die die Menge der quantitativ handhabbaren Module noch weiter vergrößern. Hierbei wird ausgenutzt, daß die Umgebung nur eine eingeschränkte Sicht auf das zu verifizierende Modul hat und somit interne Berechnungsschritte entfernt werden können, die bzgl. des Interfaces nicht unterschieden werden können. In diesem Zusammenhang wurde auch ein ROBDD basierter, funktionaler Model Checker entwickelt, der für die Verifikation von deterministischen Modellen optimiert wurde.

Das Arbeiten mit dem COMDES-System bzw. mit der inhärenten Entwurfsmethodik hat gezeigt, daß der Entwurfschritt von der initialen, informellen Beschreibung hin zu einer formalen Spezifikation ein komplexer, oft unterschätzter Entwurfsschritt ist. In vielen Fällen ist eine durch Model Checking aufgedeckt Inkonsistenz zwischen der formalen Spezifikation und der Implementierung nicht immer durch eine fehlerhafte Modulimplementierung gegeben, sondern durch eine ungenügende Spezifikation des Moduls. Um den Prozeß der Formalisierung zu unterstützen, wurde ein symbolisches Constraint-System entworfen, welches eine graphische Animation von möglichen Abläufen des spezifizierten Verhaltens inkrementell entwickelt und dem Designer graphisch darstellt.

Um die Akzeptanz des Systems im Hardware Entwurf zu erhöhen, wurde das COMDES System im Rahmen der ESPRIT-Projektes 6128 „FORMAT“ um VHDL/S erweitert, welches einen Zusammenschluß folgender Komponeten darstellt: Zur Beschreibung von Architekturen stehen die standardisierten Hardwarebeschreibungssprache VHDL [IEE87] und VHDL/S StateChart, eine graphische, zustandsbasierte ADL, zur Verfügung. Desweiteren wurde die Spezifikation der temporalen Eigenschaften mittels temporaler Logik und symbolischer Zeitdiagramme weiter verfeinert.

Ausgehend von einer AADL basierten Einführung in den Hardware Entwurf wird eine Einführung in die Simulations- und Verifikationstechniken auf Systemebene

anhand des COMDES-Systems zu gebe. Von dieser Beschreibung ausgehend wird ein Einblick in die theoretischen Problemstellungen zur Realisierung präsentiert.

Die Arbeit ist wie folgt in fünf Bereiche gegliedert:

1. *Comdes aus Sicht des Designers*

   Ausgehend von einer Einführung in die Verhaltens- und Modularisierungskonzepte von AADL wird das COMDES-System in seiner Systemarchitektur beschreiben. Hierbei werden effektive Realisierungs- und Integrationkonzepte des Toolsets bzgl. Datenhaltung, Kommunikation und Userinterface vorgestellt und diskutiert.

2. *Theoretische Grundlagen*

   Neben der Einführung der mathematischen Notation und einer näheren Betrachtung der Booleschen Algebra werden BDDs eingeführt. Desweiteren werden als grundlegende Strukturen AADL Petrinetze und deren Semantik definiert.

3. *Simulationstechniken für den Hardware Entwurf auf Systemebene*

   In diesem Abschnitt wird die effiziente Simulation von AADL Modulen auf der Basis der von ihnen abgeleiteten AADL Petri-Netze vorgestellt. Hieran schließt sich als weitere Anwendung der Simulation von AADL Petri-Netzen eine symbolische Animation von symbolischen Zeitdiagrammen an.

4. *Symbolische Verifikationstechniken für den Hardware Entwurf auf Systemeben*

   Ausgehend von der Generierung symbolischer Modelle für Bedingungs/-Ereignis-Systeme wird die Modellgenerierung für AADL Petri-Netze vorgestellt. Um die industrielle Relevanz dieser Arbeit zu zeigen, werden die für AADL vorgestellten Konzepte und Verfahren der symbolischen Modellgenerierung für VHDL portiert. Als Modelloptimierungen werden automatische Reduktions- und Abstraktionstechniken präsentiert. Abschließend wird in das funktionale symbolische Model Checking eingeführt, welches effizient die für VHDL erzeugten Modelle gegen ihre temporallogische Spezifikation verifizieren kann.

5. *Zusammenfassung und Ausblick*

Teil I

COMDES aus Sicht des Designers

Kapitel 2

Die Hardware Beschreibungssprache AADL

Der Entwurf komplexer Hardwarestrukturen im heutigen VLSI-Design eröffnet mindestens zwei orthogonale Problemstellungen für den Designer: Einerseits muß ausgehend von einer abstrakten, globalen Spezifikation eine Konkretisierung bishin zu einer VLSI-Realisierung durchgeführt werden. Andererseits muß die Komplexität einer verteilten Realisierung des Entwurfs über ggf. mehrere Module beherrscht werden. Der Entwurfsraum spaltet sich somit in folgende Dimensionen auf:

- *Vertikal:* Beherrschung der Komplexität durch Unterstützung verschiedener Abstraktionsebenen vom System- bis zum Gate-Level Entwurf
- *Horizontal:* Verteilung der Funktionalität über mehrere eng verbundene, aber nicht unbedingt synchronisierte Komponenten des Entwurfs einer Hierarchieebene

AADL [DDG+89] wurde bezüglich konkreter Syntax und formaler Semantik so konzipiert, daß sie den o.g. Problemstellungen beim Hardwareentwurf vollständig und effizient gerecht wird [Dö97]. Zur Unterstützung der Strukturierung eines Designs kann eine AADL-Spezifikation aus einer Vielzahl von *Refinement-* und *Basis-Architekturen* aufgebaut werden. Hierbei ermöglichen Refinement-Architekturen eine hierarchisch strukturierte Komposition von Basis-Architekturen, die eine Verhaltensbeschreibung des Moduls mittels einer parallelen Kontroll- und einer CSP [Hoa78] ähnlichen Kommunikationsstruktur realisieren. Ein AADL-Modul kann jedoch nur entweder Konstrukte zur Verhaltens- oder Strukturbeschreibung beinhalten.

Exemplarisch sollen in diesem Zusammenhang sowohl Konstrukte zur Verhaltensbeschreibung und zur Datentypdefinition von Basis-Architekturen, als auch das Modularisierungskonzept von AADL zum strukturierten Aufbau von Refinement-Architekturen vorgestellt werden [DDG+89, DD88, DD89a, DD89b].

2.1 Das Verhaltenskonzept von AADL

Die Abstraktionsebene einer Architektur ist durch den Abstraktionsgrad der *atomaren Aktionen* und den Aufbau der *Kontrollstruktur* der Architektur gekennzeichnet, welche das funktionale Verhalten charakterisieren. Die Kontrollflußspezifikation einer ADL, welche mehrere Abstraktionsebenen unterstützen will, muß besonderen Wert auf eine adäquate Realisierung von Parallelitäts- und Kommunikationskonzepten legen:

- **Parallelität** wird in AADL direkt in einer CSP [Hoa78] ähnlichen Notation im Kontrollbereich einer Basisarchitektur definiert. Im Gegensatz zu CSP können jedoch parallel aktivierte Aktionen in Konflikt stehen. In diesem Fall wird nur eine Aktion zufällig ausgewählt und ausgeführt. Alle anderen Aktionen werden derweil suspendiert, bis der Selektionsprozeß erneut initiiert wird. Hierdurch werden alle im Konflikt stehenden Aktionen in einem nichtdeterministischen, fairen Verfahren serialisiert.
  Durch das „`on` *trigger* `do` *statement* `od`“-Konstrukt innerhalb der Kontrollsektion einer AADL Spezifikation wird die Spezifikation einer nicht prozeduralen Aktivierung von atomaren Aktionen in Abhängigkeit des Zustandes der Architektur erlaubt. Mehrere nicht im Konflikt stehende Aktionen, die durch einen Zustand simultan aktiviert wurden, laufen dann parallel ab.
- **Kommunikation** zwischen AADL-Modulen ist entweder implizit über die Modifikation von *shared storages (variables)* oder direkt über verbundene *synchrone* bzw. *asynchrone Ports* möglich. Kommunikation über synchrone Ports erlaubt dabei ein Verbergen der konkreten Realisierung des Protokolls, wie es für hohe Abstraktionsebenen notwendig ist. Asynchrone Ports garantieren im Gegensatz dazu nur den „physikalischen“ Transport der ihnen übergebenen Nachricht, wobei die eigentlichen Protokolleigenschaft noch zusätzlich auf einer höheren Abstraktionsebene sichergestellt werden muß.

Abbildung 2.1 gibt eine Basisarchitektur an, anhand derer exemplarisch einige Konstrukte von AADL diskutiert werden sollen:

```
architecture A1
   sync outport p1: tuple
is
  type     tuple = {con(boolean; cardinal), cardinal}
  storage  S1,S2,S3,S4 : cardinal
  behaviour
     action A1 (v : cardinal) is
        effect post S3 = v
     end
     action A2 is
        condition S1 = 0
        effect post S4 = 0
     end
     action A3 (storage v : cardinal) is
        effect post v = 17
     end
     control
        par A1(S1) || A3(S1) rap;
        par A1(S2) || A2     rap;
        p1 ! con(true,S1)
     end
  end   /* behaviour part */
end  /* architecture A1 */
```

Abbildung 2.1: AADL Basis-Architektur

- *Interface-Spezifikation*
  Im Deklarationsteils des Modulkopfes wird die Spezifikation der Interface-Objekte angegeben. Das AADL Modul `A1` besitzt einen synchronen Outport `p1`, mit dessen Hilfe der Umgebung ein Objekt vom Typ `tupel` übergeben werden kann. Die Deklaration des Typs `tupel` erfolgt dabei gesondert im Rumpf der AADL Architektur.
- *Speicherstrukturen*
  Sie werden im Rumpf des Moduls definiert und spezifizieren lokale Speicherstrukturen der Basis-Architektur (`S1,S2,S3,S4`).
- *Verhaltenbeschreibung*
  Das mögliche Verhalten des Moduls, d.h. die Abfolge von atomaren Aktionen (`actions`), wird im `behaviour`-Teil festgelegt. Hierzu stehen mehrere Konstrukte

zur Iteration und parallelen Programmausführung zur Verfügung. In dem Beispielmodul werden drei Anweisungen sequentiell ausgeführt:
In dem ersten Parallel-Konstrukt sollen `A1` und `A3` parallel ausgeführt werden. Durch den impliziten Konflikt auf dem aktuellen Storage-Parameter `S1` wird eine zufällige Serialisierung erzwungen.
Das letzte Statement beschreibt eine synchrone Kommunikation über den Port `p1`, wobei hier das Pattern `con(true,S1)` des Typs `tupel` übertragen werden soll. Die Kommunikation kann jedoch erst dann erfolgreich abgewickelt und ggf. folgende Aktionen angestoßen werden, wenn auch die Umgebung des Moduls bereit ist, genau dieses Pattern zu empfangen.

AADL unterstützt im Kontext der parallen Programmausführung nur statische Prozeßgenerierung mit einer statisch festgelegten Kommunikationsstruktur zwischen den Modulen einer Architektur. Konstrukte zur dynamischen Prozeßgenerierung sind auf hohen, betriebssystemnahen Abstraktionsebenen eine sicherlich wünschenswerte Eigenschaft, jedoch können alle aus Rechnerarchitektursicht relevanten Problemstellungen in dieses statische Modell eingebettet werden. Desweiteren erlaubt diese Restriktion das Anwenden von auf endlichen Modellen agierenden, automatischen Verifikationsverfahren.

2.2 AADL Modulkonzept

Die bisher eingeführten Konzepte erlauben nur die Beschreibung des Verhaltens eines einzelnen Moduls. Um große Systeme verteilt entwerfen zu können, werden Basis-Architekturen als modulare Blöcke aufgefaßt, mit deren Hilfe komplexere Systeme aufgebaut werden können. Hierbei können einige Architekturen zur reinen Definition von Datentypen und Speicherstrukturen benutzt werden, um ein *package*-Konzept, wie es in VHDL [IEE87] oder ADA [MH92] anzutreffen ist, zu realisieren. Strukturen, die als Komposition von Architekturen aufgebaut sind, werden als Refinement-Architekturen bezeichnet. Es ist dabei irrelevant, ob es sich bei den beteiligten Modulen um Basis- oder Refinement-Architekturen handelt, da ihre Modulschnittstellen diese Struktureigenschaft verdecken.

Mittels dieser Methodik läßt sich ein modulares Design-Szenario ableiten, wie es durch Abb. 2.2 beschrieben wird. Ausgehend von einem Fundus bereits erstellten Spezifikationen innerhalb einer Design-Datenbasis, die sich in Architektur- und

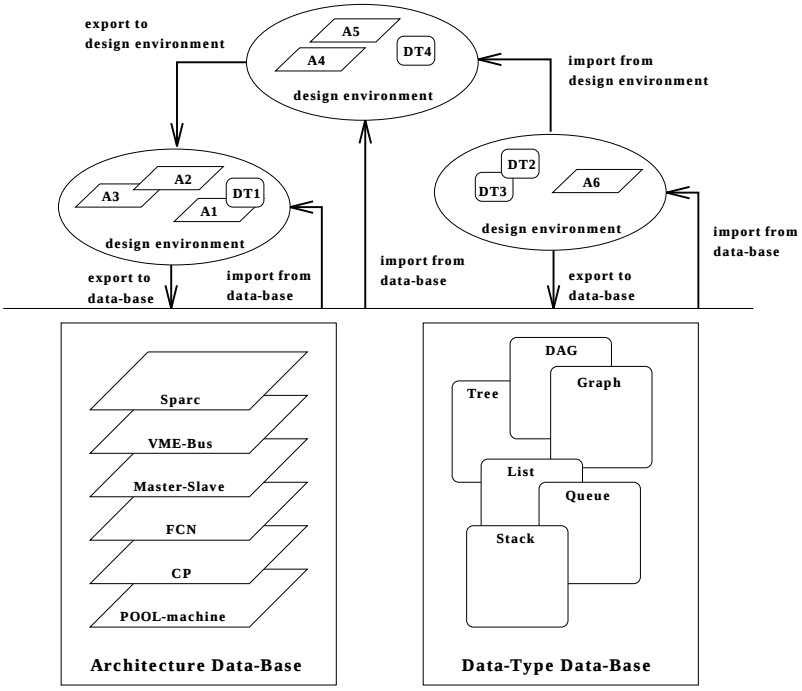


Abbildung 2.2: Design-Szenario

Datentyp-Spezifikationen differenzieren läßt, können in einer lokalen Entwicklungsumgebung neue Architekturen entwickelt werden, die sich auf schon realisierte Module abstützen können. Hierbei kann der Ex- und Import von Modulen transparent über mehrere Ebenen hinweg durchgeführt werden. Um die Austauschbarkeit bzw. Skalierbarkeit von Modulen noch weiter zu steigern, können AADL Module innerhalb ihrer Interface Spezifikation neben der Deklaration von Ports auch Architekturparameter aufführen, welche interne Strukturen, z.B. Größe von Adreß- und Datenregistern, skalieren können.

Neben dieser Komposition von Modulen aus einer sehr abstrakten Sicht, muß noch die konkrete Realisierung der Komposition betrachtet werden. Ein natürlicher Weg zur Konstruktion von Refinement-Architekturen ist eine Verbindung der beteiligten Architekturen durch eine explizite Verbindungs- bzw. Kommunikationsstruktur. Dies wird realisiert, indem die Ports der jeweiligen Inkarnationen[1] der Architekturen miteinander anhand einer spezifizierten Kommunikationsstruktur verbunden werden. Das Verhalten einer Architektur, welche in dieser Weise aufgebaut wurde, ist durch ihre strukturelle Dekomposition charakterisiert, d.h. ihr Verhalten ist vollständig durch das Verhalten der Sub-Module und durch die Verbindungsstruktur

[1] Instanziierung einer AADL-Architekturspezifikation

determiniert.

Die interne Kommunikationsstruktur zwischen einzelnen Komponenten einer Refinement-Architektur ist jedoch für die Umgebung der Architektur nicht sichtbar, da die Interface Beschreibung eine Einsicht in Subkomponenten einer Architektur nicht ermöglicht. Ob das Verhalten eines Moduls funktional oder strukturell spezifiziert ist, ist daher für die Umgebung nicht erkennbar und irrelevant.

```
architecture A1
  async outport p : integer
is
  subarchitecture
    S1 : A2;
    S4 : A3
  topology
    S1.p1 is S4.p2 is p,
    .p2 is .p1
end
```

```
architecture A2
  async inport p1 : integer
  async outport p2 : integer
is
  subarchitecture
    S2 : A4;
    S3 : A3
  topology
    S2.pX is p1,
    S2.pY is S3.p1
    p2 is S3.p2
end
```

```
architecture A3
  async inport p1 : integer
  async outport p2 : integer
is
  subarchitecture
    S5 : A5;
  topology
    S5.pK is p1,
    S5.pL is p2
end
```

```
architecture A4
  async inport pX : integer
  async outport pY : integer
is
end

architecture A5
  async inport pK : integer
  async outport pL : integer
is
end
```

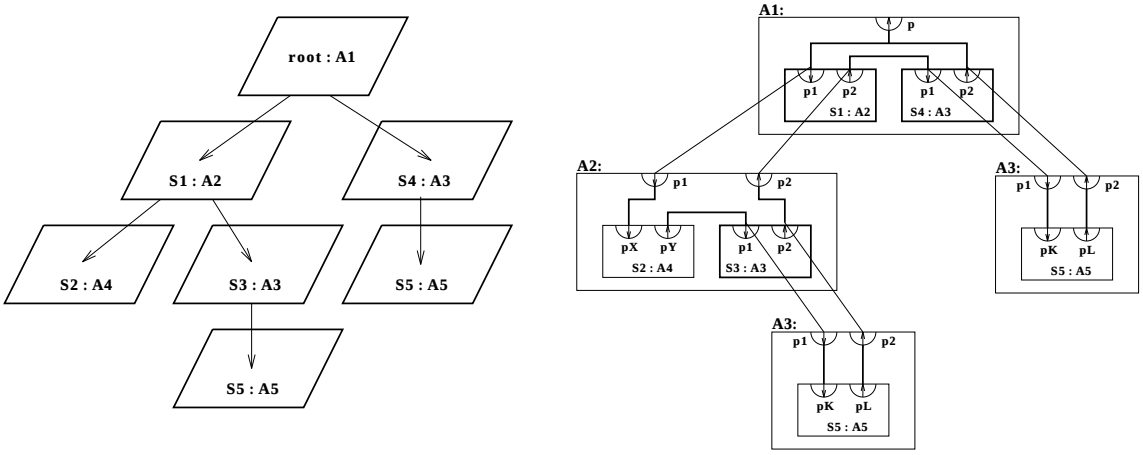


Abbildung 2.3: Layertree und Schematic Structure einer AADL Refinementarchitektur

AADL unterstützt zwei Arten der Informationsaustauschmechanismen zwischen Architekturen:

1. Synchrone bzw. asynchrone Kommunikation zwischen Komponenten eines Moduls durch Verbindung ihrer Ports anhand einer explizit gegebenen Topologiebeschreibung.
2. Kommunikation über gemeinsam nutzbare Speicherstrukturen.
   Da Subarchitekturen nur lokale Speicherstrukturen und ihre Outputports zur

Kommunikation verändern dürfen, kann eine spezielle (Speicher-) Architektur definiert werden, welche die benötigten Komponenten über Ports der Umgebung bereitstellt und deren Modifikation zum Informationsaustausch zwischen den Submodulen genutzt werden kann.

Abb. 2.3 zeigt einen AADL-Design bestehend aus den Refinement-Architekturen `A1`, `A2` und `A3` und den nur durch ihrer Schnittstellendefinition gegebene Basis-Architekturen `A4` und `A5`. Die Kommunikation in diesem Design ist über synchrone Ports realisiert. Der *Layertree* gibt baumartig den hierarchischen Aufbau wieder. Die Verbindungsstruktur, d.h. die Visualisierung der Topologiebeschreibung der Refinement-Architekturen wird durch die *Schematic Structure* angegeben. Es ist zu erkennen, daß zwei Inkarnationen der Architektur `A3` ohne Umstrukturierung der Interna von `A3` und `A4` in verschiedenen Ebenen bzw. Umgebungen des Designs eingebettet werden können.

Kapitel 3

Das COMDES-System

Ziel des COMDES-Systems ist es, dem Designer ein integriertes Framework zur Spezifikation, Simulation und Verifikation von AADL basierten Entwürfen zur Verfügung zu stellen, um die AADL-Designmethodik möglichst umfassend toolmäßig zu unterstützen.

Dieses Kapitel soll dem Leser eine Betrachtung des COMDES-Systems aus einer softwaretechnologischen und benutzerorientierten Perspektive darlegen. Ausgehend von der COMDES-Systemarchitektur wird die globale Integration der Tools vorgestellt. Aufbauend auf der Erläuterung der Struktur der globalen COMDES-Datenbasis und des COMDES Design-Managers werden exemplarisch einzelne zentrale Tools der drei Teilbereiche des Systems vorstellt, die einen Eindruck von der Leistungsfähigkeit sowie der Komplexität der von COMDES gelösten Problemstellungen vermitteln sollen.

3.1 Systemarchitektur und Integration

COMDES hat in seiner Gesamtstruktur eine Vielzahl von komplexen Tools zu integrieren, die sich, trotz starker Unterschiede in Benutzerinteraktion und Betriebsmittelauslastung, bezüglich einer gemeinsamen Datenhaltung und einer homogenen Benutzungsoberfläche dem Benutzer als ein integriertes Toolset zu präsentieren hat.

Der Entwurf der COMDES-Systemarchitektur ist von folgenden Randbedingungen beeinflußt:

- Durch die Vielzahl der Tools, die z.T. von verschiedenen Softwareentwicklern be-

treut wurden, mußte eine adäquate Kapselung der Interfaces zur Datenbasis auf verschiedenen Abstraktionsebenen bereitgestellt werden.

- Da die Betriebsmittel eines CAD-Arbeitsplatzes nicht in allen Entwurfsphasen eines komplexen Designs ausreichend sind, muß eine transparente Netzwerkunterstützung angeboten werden.
- Durch einen ggf. parallelen Zugriff auf die Datenbasis muß die Konsistenz der Datenbasis jederzeit sichergestellt und der Status von generischen Objekten abgeleiteten Objekten erkennbar sein.
- Da bei der Entwicklung eines Designs auf verschiedene Designbibliotheken zugegriffen werden kann, muß das System einen Designkontext verwalten können, aus dem heraus aktivierte Tools selektiv operieren können.

Bevor jedoch die realisierte Systemarchitektur motiviert werden kann, muß noch eine Darstellung der Struktur des COMDES Tool-Sets erfolgen, wie sie in Abbildung 3.1 veranschaulicht wird. Zur Spezifikation von AADL Designs stehen ein Struktu-

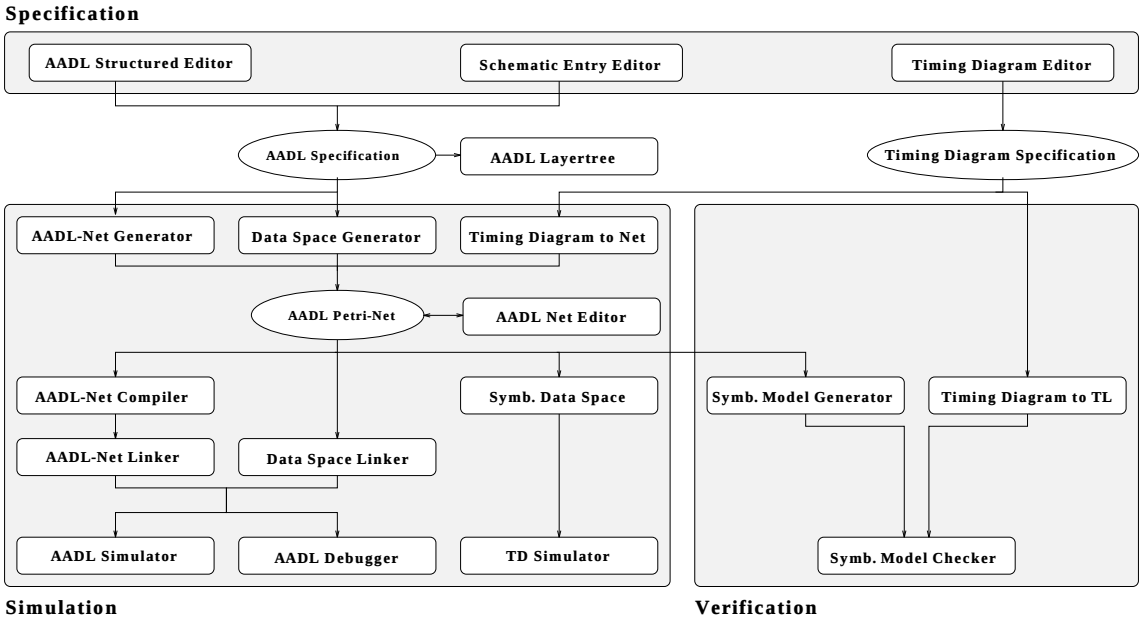


Abbildung 3.1: Das COMDES Toolset

reditor und ein auf die Erstellung von Refinementarchitekturen spezialisierter Schematic Entry Editor zur Verfügung. Zur graphischen Spezifikation des temporalen Verhaltens eines AADL-Moduls bezüglich der Wertverläufe der Port-Variablen seines Interfaces steht ein Zeitdiagrammeditor zur Verfügung. Als generische Objekte für die Simulation und für die formale Verifikation werden in diesem Kontext die

AADL-Spezifikationen und die Zeitdiagrammspezifikationen dem Benutzer sichtbar gemacht.

Die Simulationskomponente von COMDES erlaubt eine Evaluation der erstellten AADL- und Zeitdiagrammspezifikationen auf Basis der Simulation von AADL Petri-Netzen, die die jeweilige Semantik charakterisieren. In der ersten Phase erfolgt eine Übersetzung des Quellcodes in AADL Petri-Netze sowie die Generierung eines Datenraums, auf dem die Datentransformationen und booleschen Bedingungen des Petri-Netzes ausführt bzw. evaluiert werden. Die Existenz und die Struktur des AADL Petri-Netzes sind dem Designer verborgen. Die Struktur kann jedoch durch Freigabe des AADL Petri-Netzeditors visualisiert, modifiziert und unter Einschränkungen interpretativ simuliert werden. Für eine effiziente Simulation der AADL Spezifikationen schließt sich eine Compilationsphase des AADL Petri-Netzes in Code für eine Netzmaschine an (Kapitel 5.1.2). Zum Abschluß dieser Phase wird der erzeugte Netz-Code mit dem bereits erzeugten Datenraum verbunden und sowohl ein eigenständig lauffähiger Simulator als auch ein Debugger generiert, die eine Analyse des erstellten Designs erlauben (Kapitel 5.1.4.1).

Die Simulation von Zeitdiagrammspezifikationen (Kapitel 5.2) weicht von diesem Schema ab. Durch eine symbolische, auf *Reduced Ordered Binary Decision Diagrams (ROBDDs)* (siehe Kapitel 4.3 und [Ake78, BRB90]) Kodierung des zur Spezifikation gehörenden Datenraums und den sich dynamisch verändernden Netzstrukturen wird ein interpretativer Simulationsansatz unterstützt. Als Ergebnis der Simulation wird graphisch auf Zeitdiagrammebene eine Menge von gültigen Traces der beobachtbaren Wertverläufe der Ports erzeugt, welche durch die Zeitdiagrammspezifikation charakterisiert werden. Somit kann der Designer seine Spezifikation frühzeitig auf Plausibilität hin überprüfen.

Die formalen Verifikation eines AADL-Moduls, d.h. die Verifikation seines durch eine Implementierung gegebenen Verhaltens gegenüber seiner formalen Zeitdiagrammspezifikation, erfolgt in COMDES mittels symbolischem Model Checkings. Für diesen Prozeß wird aus dem von dem AADL-Modul abgeleiteten AADL Petri-Netz ein endliches, symbolisches Transitionssystem generiert und die zu testende Zeitdiagrammspezifikation in temporale Logik übersetzt. Der Model Checker kann die formale Korrektheit des Entwurfs aufzeigen oder im Fehlerfall eine Widerlegungssequenz generieren, die dem Designer eine Inkonsistenz zwischen Implementierung und Spezifikation offenbart.

Abbildung 3.2 stellt die realisierte Softwarearchitektur des COMDES Systems dar.

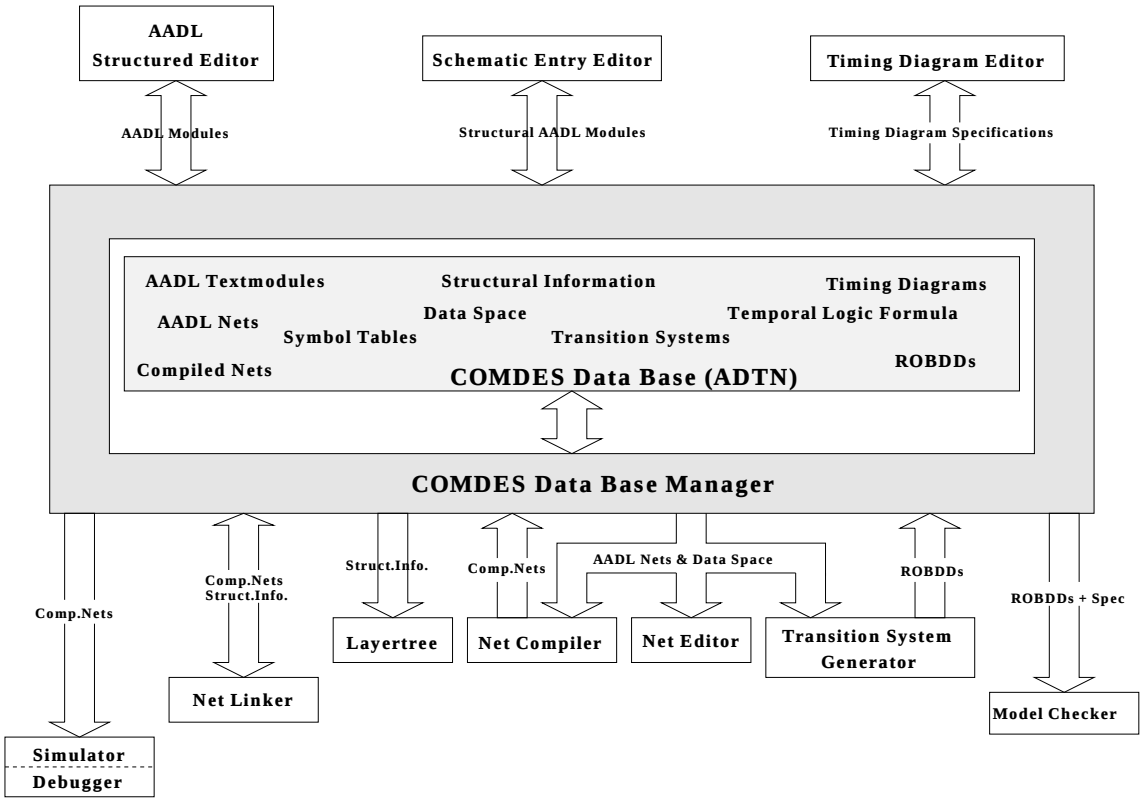


Abbildung 3.2: COMDES Systemarchitektur

Zentrale Komponente dieser Struktur ist die *COMDES Data Base*. In ihr werden alle Daten des Toolsets abgelegt und verwaltet. Ein Zugriff auf die hier abgelegt Daten erfolgt durch den *COMDES Data Base Manager*. Diese Schicht stellt zwei Funktionalitäten für das Gesamtsystem zur Verfügung: Einerseits wird hierdurch eine Schnittstelle zwischen den Datenanforderungen der einzelnen Tools und der Datenbasis bereitgestellt und somit kontrollierbar. Andererseits stellt der Manager auch eine Funktionalität bereit, welche die Verwaltung der Betriebsmittel des Netzwerks zuläßt und eine Verwaltung eines COMDES-Designkontextes erlaubt. Über dieser logischen Schicht sind die einzelnen Tools angesiedelt, die Daten aus einer Datenbasis anfordern können und ggf. neu berechnete Informationen in ihr ablegen können. Zudem wird ein Netzwerk-Nachrichtendienst zur Verfügung gestellt, mit dessen Hilfe die Tools Statusinformationen austauschen können.

3.2 COMDES Datenbank

COMDES basiert auf einer als abstrakter Datentyp realisierten Datenbasis, in der sowohl generische Design-Objekte als auch deren Derivate effizient abgelegt sind, um einen schnellen und direkten Zugriff für weitere Transaktionen durch andere Tools bereitzustellen und einen Zugriff auf verschiedenen Abstraktionsgraden zu ermöglichen. Die Klasse der zu verwaltenden Objekte ist durch ihre besondere strukturelle Inhomogenität gekennzeichnet. Neben „kleinen“ Objekten, die selber typreich strukturiert sind, z.B. Zeitdiagrammspezifikationen, gibt es „große“ Strukturen, z.B. ROBDDs, die eine einfache Struktur aufweisen. Auch konnte durch die Integration von *externen*-Tools keine homogene Datenrepräsentation der Datenbasis realisiert werden, sondern das System erlaubt eine Verarbeitung von Daten sowohl in binär- als auch in ASCII-Repräsentation. Jedoch wurde sichergestellt, daß große, reguläre Datenbestände in kompakter Binärdarstellung gehalten werden, die eine schnellere Datenein- bzw. Datenauslagerung erlauben.

Abbildung 3.1 verdeutlicht, daß der Verwaltung von AADL Petri-Netzen eine zentrale Rolle im System zufällt. Aus diesem Grund sei anhand dieser Struktur die generelle Realisierung der Datenhaltung in COMDES exemplarisch dargestellt. In analoger Weise werden z.B. ROBDDs als abstrakter Datentyp dem Toolset zugänglich gemacht.

3.2.1 ADTN

Der ADTN (Abstract Data Type Net) hat die Aufgabe, die bei der Übersetzung von AADL-Basisarchitekturen und Zeitdiagramm-Spezifikationen in AADL Petri-Netze erzeugten Netzstrukturen aufzunehmen und zu verwalten. Aus diesem Grunde wurde für diese Objektklasse in der COMDES Datenbank eine Datenstruktur zur Verfügung gestellt, in der ein uneingeschränktes Navigieren auf Netzebene unterstützt wird und dem Benutzer ein differenziertes Operieren auf verschiedene Abstraktionsebenen der Datenbeständen zugänglich macht. Der ADTN hat folgenden Anforderungen gerecht zu werden:

- einfache Modifizierbarkeit der zu speichernden Informationen,
- schnelle Berechnungen auf den Laufzeitdatenstrukturen,
- geringer Hauptspeicherbedarf der Laufzeitdatenstrukturen,
- schneller Zugriff auf Daten des Hintergrundspeichers,

- einfache Integration in schon bestehende Tools,
- Zugriffsoperationen auf verschiedenen Abstraktionsebenen für die Laufzeitdatenstrukturen,
- Bereitstellung von Analysefunktionen auf Netzebene.

In den folgenden Paragraphen soll kurz auf einzelne Aspekte eingegangen werden, die für die Designentscheidungen relevant waren.

Konzept für optimales Laufzeitverhalten

Als Standarddarstellungen für die Petri-Netze sind zwei Grundstrukturen auszumachen. Die einfachste Darstellung von Bedingungs/Ereignis-Systemen (B/E-Systeme, siehe auch Definition 4.4) und Stellen/Transitions-Netzen (S/T-Netze) ist durch die Inzidenzmatrixdarstellung gegeben [Rei85]. Bei dieser Repräsentation liegt eine $n \times m$-Matrix M vor, in der n und m jeweils die Kardinalität der Stellen und Transitionen des Netzes wiederspiegeln. M ist mit Null-Elementen initialisiert. Existiert eine Kante von Stelle i zu einer Transition j, so wird M wie folgt aktualisiert:

- $M[i, j] = 1$, bei B/E-System
- $M[i, j] = k$, bei S/T-Netzen mit k als Wichtung der Kante

Für kleine Netze, die einen hohen Vermaschungsgrad besitzen, ist diese Darstellung als optimal anzusehen, da basierend auf dieser Struktur auch numerische Verfahren zur Berechnung von Invarianten und Erreichbarkeitsaussagen anwendbar sind. Ist das Netz jedoch sehr dünn vermascht, so ist die Speicherplatzausnutzung als nicht optimal anzusehen, da sehr viele Null-Elemente, die bei einer Berechnung nicht benötigt werden, ungenutzten Speicherplatz belegen. Als Abschätzung für diesen Ansatz sei ein AADL Petri-Netz, wie es z.B. bei der Multiprozessornetzwerk-Fallstudie [Dö97] erzeugt wird, mit ca. 10.000 Transitionen und Stellen gegeben. Bei einer zu verwaltenden Kanteninformationsgröße von 20 Byte ergibt sich ein benötigter Speicherplatz von ca. 2 GByte, wodurch dieser Ansatz zu verwerfen ist. Wird für die Darstellung der Matrix eine Realisierung gewählt, in der nur die Elemente verwaltet werden, die ungleich dem Null-Element sind, wird die Adjazenzlistendarstellung erlangt. Hierbei wird eine Listenstruktur generiert, in der alle Kanten verwaltet werden, die entweder von Stellen oder Transition ausgehen. Der Vorteil dieser Methode liegt klar in der besseren Ausnutzung des Speicherplatzes, wenn von dem geringen Overhead zur Verwaltung der Pointerinformationen abgesehen wird. Nachteilig an

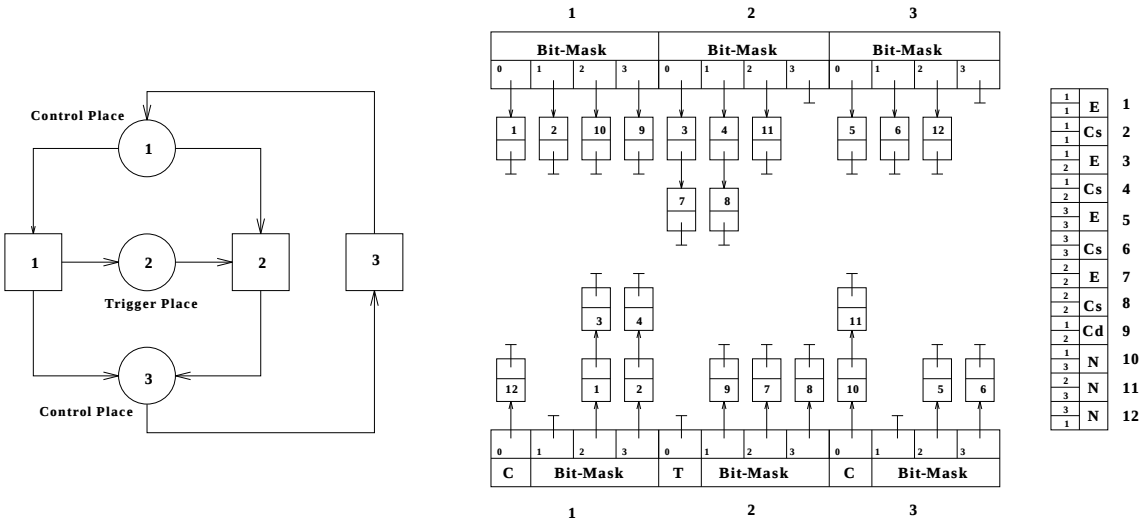


Abbildung 3.3: ADTN Laufzeitdatenstrukturen der Graph-Informationen

diesem Ansatz ist nun die sequentielle Suche nach einer Kante, die zwei Netzobjekte verbindet, da im Mittel $O(n/2)$ Kanten untersucht werden müssen, wobei n der Anzahl der durchschnittlich von einem Netzobjekt ausgehenden Kanten entspricht. Um eine Navigation auf der Adjazenzlistendarstellung zu beschleunigen, ist die Netzdarstellung um virtuelle, inverse Kanten angereichert, d.h. jede real existierende Kante im Netz erhält eine zu ihr in umgekehrter Richtung weisende Kante (siehe Abbildung 3.3). Die Haltung dieser redundanten Information wird aber vollständig durch die Beschleunigung der zum Einsatz kommenden Graphalgorithmen gerechtfertigt.

Konzept zur Minimierung des Hauptspeicherbedarfes

Um eine Minimierung des zur Laufzeit benötigten Speicherplatzes zu gewährleisten, wird die zu verwaltenden Information klassifiziert und nach statischer bzw. dynamischer Größe differenziert. Folgende vier Arten von Informationen werden verwaltet:

- Graphspezifische Informationen, die das reine Netzgerüst darstellen,
- Systembezogene Informationen, die z.B. die Markierung einer Stelle mit einem Token, beschreiben,
- Simulationsspezifische Informationen, die zur Compilation des Netzes benötigt werden und
- Verifikationsbezogene Informationen, die zur Generierung eines symbolischen Modells für das Netz temporär benötigt werden.

Für Berechnungen, die nur die Struktur des Netzes benötigen, werden auch nur

die hierfür benötigten Daten geladen. Algorithmen, die applikationsbedingt weiterführende Daten benötigen, können diese dann *on demand* anfordern. Analog hierzu wird die Verwaltung der Informationen mit dynamischer Größe durchgeführt. So können z.B. Texte, welche Netzobjekten zugeordnet sind, on demand geladen werden, um einen möglichst geringen Teil des Hauptspeichers zu belegen.

Konzepte zum File Input/Output

Eine externe Repräsentation der Daten des ADTNs auf der Basis einer ASCII-Repräsentation führt nicht zu befriedigenden Laufzeiten für große Petri-Netze, da ein inkrementelles Parsen der Files inklusive Aufbau der internen Datenstrukturen zu lange dauert. Der ADTN repräsentiert deshalb seine Laufzeitdatenstruktur in einer Form, die in einem Block[1] geladen bzw. geschrieben werden kann. Zum Aufbau der Laufzeitdatenstrukturen der (inversen) Kanten ist abschließend nur noch ein einmaliger, linearer Durchlauf über die Kantengraphinformation notwendig.

3.2.2 Softwaretechnologische Integration

Für die Realisierung des Softwaretools ADTN wurde eine maximale Modularität angestrebt, die es ermöglicht, auf alle Veränderungen der Schnittstellen zu existierenden Tools, hier sei die Netzsimulation und symbolische Transitionssystemgenerierung exemplarisch genannt, flexibel zu reagieren, ohne daß eine vollstände Reimplementierung von Submodulen notwendig wird. Erste Ansätze zur Lösung dieser Problemstellung hatten ein Generatorsystem vorgesehen, welches anhand einer abstrakten Spezifikation alle Datenstrukturen und Zugriffsfunktionen generiert und so immer flexibel auf die Modifikationen der Datenrepräsentation reagieren kann[2]. Aus Gründen der Komplexität der Struktur der zu verwaltenden Designobjekte und der notwendigen Effizienz der Zugriffsalgorithmen erschien dieser allgemeine Generierungsansatz nicht optimal. Es wurde ein ähnliches Konzept realisiert, welches sich auf die Funktionalitäten eines Präprozessors abstützt. Alle Informationen werden so variabel gehalten, daß sie mit Hilfe von Präprozessordirektiven verwaltet werden können. Dies schließt mit ein, daß eine Veränderung von schon bestehenden Funktionalitäten des Gesamtsystems nicht gegeben ist. Nach einer Recompilation ist das

[1] mit einem einzigen UNIX system call

[2] Dieser Vorgehensweise ist z.B. im Leda VHDL-Frontend [Led92] teilweise realisiert worden; jedoch können nur die vom Benutzer generierten Schemaerweiterungen nach diesem Ansatz verwaltet werden, der strukturelle Kern der Datenbasis ist nicht modifizierbar.

gesamte System sofort wieder einsatzbereit und kann auf der neuen Struktur allen geforderten Funktionalitäten gerecht werden.

Wie sieht die Entwicklung von Software aus, die sich auf den ADTN abstützen soll? Hierbei wurde der Ansatz aller großen, schon bestehenden UNIX-Softwaresysteme verfolgt, z.B. dem X11-Window System. Globale Datenstrukturen und benötigte Funktionalitäten werden via `include`-Dateien in den Quellcode integriert. Desweiteren sind alle Funktionen auf globalen Variablen in zentralen Bibliotheken hinterlegt, die nur zur Linkzeit an die Applikation gebunden werden. Hieraus folgt ein sehr kurzer, komprimierter Quellcode mit guten Analyseeigenschaften für den Codereview. Desweiteren wird durch das Information-Hiding des abstrakten Datentyps die modulare Entwicklung der Applikation unterstützt. Die Funktionalitäten des ADTNs lassen sich in folgende Aufgabengebiete unterteilen:

1. Basiszugriffsoperationen auf den Datenstrukturen
2. abstrakte Navigationsfunktionen auf der Netzebene
3. Operationen für die Analyse der statischen Semantik
4. File Input/Output

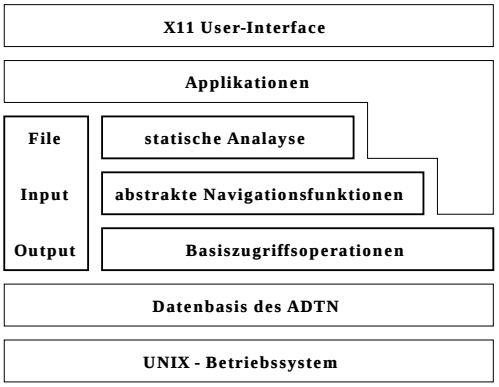


Abbildung 3.4: Struktur der funktionalen Hierarchie im ADTN

Ihre Abhängigkeiten sind in in Abbildung 3.4 dargestellt. An dieser Darstellung kann leicht der hierarchische Aufbau des Systems erkannt werden. Es gibt mehrere Abstraktion, die es dem Softwareentwickler ermöglichen, auf verschiedenen Abstraktionsebenen mit dem ADTN zu kommunizieren. Aus diesem Konzept fällt das System der File-Funktionen heraus. Sie sind aus Effizienzgründen so organisiert, daß eine Zuordnung in das bestehende Abstraktionsschema nicht möglich ist.

3.3 COMDES Manager

Ein wichtiger Indikator für die Güte eines CAD Systems ist neben einer umfangreichen Auswahl von funktionsstarken Design-Tools und einer optimalen Designdatenbasis (DDB) durch die Verwaltung bzw. globale Organisationsstrukur des CAD Systems gegeben.

Das COMDES-System unterstützt die in Kapitel 2.2 vorgestellte Designmethodik für strukturierte AADL Entwürfe in folgender Hinsicht:

- Verschiedene DDB können parallel verwaltet werden, wobei zwischen Refinement- und Basis-Architektur-DDB unterschieden wird.
- Ein neues Design kann nur im Kontext einer schon existenten *Designumgebung* erzeugt und administriert werden.
- Eine statische Semantikanalyse anhand der aktuell zugeordneten DDB wird kontinuierlich sichergestellt.

Neben der Umsetzung dieser methodischen Konzepte zur Designerstellung wurde bei der Realisierung auf eine optimale Ausnutzung der Betriebsmittel der Arbeitsumgebung des Designers hoher Wert gegeben. Wenn das lokale Potential einer Arbeitsstation in einem Netzwerk genutzt werden soll, gleichzeitig aber Daten des COMDES Systems nicht bzw. minimal redundant gehalten werden sollen, ergibt sich zwingend eine Realisierung des Systems als Client/Server-Architektur, in der verschiedene Module ihr Wissen auf Anfrage bereitstellen. Aufgrund der Vielzahl und Verschiedenheit der zu integrierenden Tools und der hieraus folgenden Komplexität der Datenhaltung wird im folgenden kurz die Prozeßarchitektur vorgestellt.

3.3.1 Verteilungsansätze

Unter der Voraussetzung, daß die Kopplung zwischen Client- und Server-Prozessen möglichst lose sein soll, um eine hohe Modularität unterstützen zu können, ensteht in einer applikationsorientierten Betrachtungsweise ein logisches Drei-Ebenen Modell, welches sich wie folgt gliedert:

- Dienstanforderungs-Ebene
- Isolations- und Vermittlungs-Ebene
- Diensterbringungs- und Ressourcen-Ebene

Die erste Ebene führt Applikationen aus und fordert im Verlauf der Ausführung typischerweise Ressourcen (z.B. Daten) an. Die Isolations- und Vermittlungs-Ebene

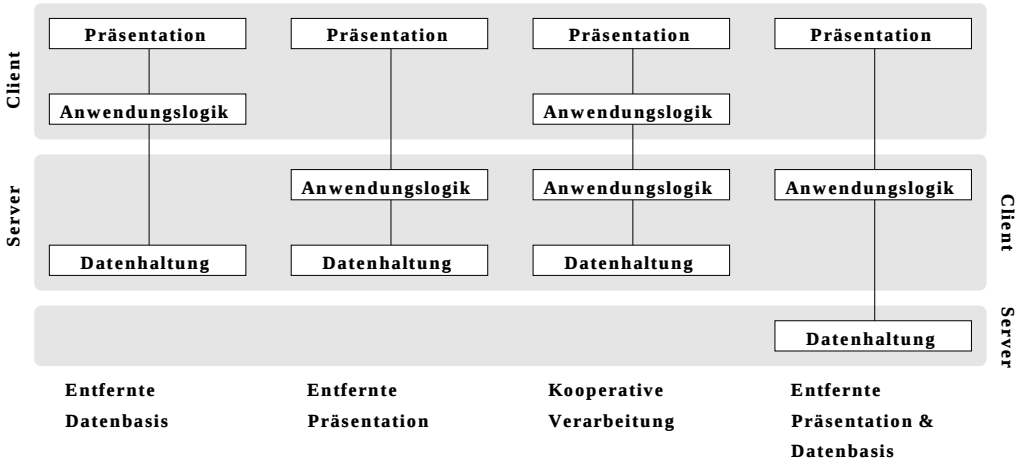


Abbildung 3.5: Verteilungsmodelle logischer Client/Server-Architekturen

hat die Aufgabe, Ort und Struktur der Diensterbringer und Ressourcen vor der Applikation zu verbergen. Die Diensterbringungs- und Ressourcen-Ebene verwaltet hingegen nur Applikationen, Daten und Betriebsmittel. Für das logische Modell ist es hierbei zunächst unerheblich, ob ein paralleler Zugriff auf die Objekte stattfindet oder nicht. Dieses logische Modell macht jedoch keine Aussagen über die physikalische Lokalisierung der drei Ebenen.

In einer auf die Aufgaben der Applikation gerichtete Betrachtungsweise (Abbildung 3.5) lassen sich ebenfalls drei logische Komponenten bestimmen, die sich wie folgt gliedern:

- *Präsentation:* User-Interface zur Applikation
- *Anwendungslogik:* Realisierung der Kernfunktionalitäten der Applikation
- *Datenhaltung:* Zugriff der Applikation auf externe Speicherobjekte

Durch Verteilung der logischen Komponenten auf unterschiedliche Systeme lassen sich drei Grundformen darstellen: Entfernte Repräsentation, entfernte Datenhaltung und kooperative Verarbeitung bzw. verteilte Funktionalität. Bei dieser Strukturierung kann auch eine mehrstufige Client/Server-Beziehung aufgebaut werden. Ein Server kann seinerseits als Client Dienste eines anderen Servers in Anspruch nehmen. Diese Strukturierung findet u.a. im COMDES-Simulationsszenario Anwendung (siehe Kapitel 3.5.1). Durch die Differenzierung einer Applikation in einzelne auto-

nome Prozesse ist ihre jeweilige Allokation auf verschieden Knoten eines Netzwerkes, die das jeweilige Anforderungsprofil optimal unterstützen, leicht zu realisieren.

3.3.2 Kommunikationsstruktur

Das Konzept der Realisierung einer Applikation mittels einer Client/Server-Architektur kann als Vorgehensweise auch auf das gesamte Framework zur Anwendung kommen. Abbildung 3.6 gibt eine schematische Darstellung der logischen Kommunikationsstruktur aller in das COMDES System integrierten Applikationen wieder. Hierbei kann keinem Prozeß eindeutig eine Client- bzw. Server-Rolle zugewiesen werden, sondern alle (Client-)Prozesse können Anforderungen bzw. Anfragen stellen, die dann von entsprechen anderen (Server-)Prozessen beantwortet werden. Als ein zentraler Prozeß des COMDES Prozeß-Systems kann der *Network Resour-*

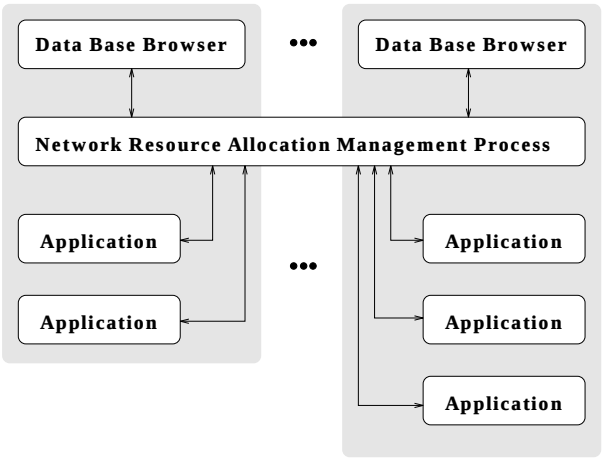


Abbildung 3.6: Logischer Aufbau der Interprozeßkommunikation

ce Allocation Management Process (NRAMP) betrachtet werden. Über ihn werden zentral alle Kommunikationswünsche zwischen den einzelnen Applikationen abgewickelt. Durch diese Architektur ist es ihm möglich, alle Zugriffe auf die DDB zu erkennen und verschiedene Designumgebungen disjunkt und parallel den einzelnen Usern zuzuordnen und zu verwalten. Desweiteren hat er Wissen über die Knoten des Netzwerkes, die ihre Ressourcen freigegeben haben.

Ausgehend von einem *DDB-Browser* Prozeß, der durch seine Sicht auf eine DDB einen einzelnen Designkontext repräsentiert, können ihm zugeordnet Applikationen für existierende bzw. neu zu generierende Designobjekte gestartet werden. Kommunikationswünsche einer Applikation werden hierbei vom NRAMP ihren jeweiligen

Kontexten zugeordnet, ohne daß sie einen anderen Kontext beeinflussen. Diese Eigenschaft ist besonders in einer Multi-User Umgebung notwendig, um ungewollte Seiteneffekte zu verhindern.

Der Datentransfer zwischen der jeweiligen DDB und einer Applikationen wird hierbei jedoch nicht vollständig über den NRAMP abgewickelt. Die zu transportierenden Informationen werden innerhalb von COMDES nach Kontroll- und Designdaten unterschieden. Alle Kontroll- und Statusinformationen werden vom NRAMP administriert. Sie stellen im Gegensatz zu den anfallenden Designdaten eine fast zu vernachlässigen Größe dar. Um diese großen Datenmengen effizient zu transportieren, wird von COMDES der Datentransfer zwischen verschiedenen Knoten des Netzwerkes via *NFS* [3] vom Betriebssystem realisiert.

User-Interface 3.1 COMDES-DDB-Manager.

Der COMDES Manager ist aus zwei miteinander kommunizierenden Prozessen aufgebaut, die jeweils als Client bzw. Server agieren: dem zum Netzwerk globalen NRAMP-Prozeß und dem dem Designer zugeordneten DDB-Browser, der das User-Interface zum COMDES Systen darstellt. Abbildung 3.11 zeigt einen typischen Bildschirmaufbau, in dem eine lokale Designumgebung durch den DDB-Browser aufgebaut wurde. Durch ihn stehen zur Konfiguration der Designumgebung vier Kategorien zur Verfügung:

1. *AADL Manager*
   Er dient als Architecture Browser *für die selektierten Designdatenbasen. Für selektierte Designobjekte einer Datenbasis können via PopUp-Windows ihnen zugeordnete Applikationen gestartet werden, die ggf. neue Designobjekte in den Datenbasen ablegen.*
2. *Environment*
   Diese Konfigurationskomponente legt die Designumgebung des Benutzers fest, die durch den Netzwerkkommunikationsserver, der Menge von anwendbaren Applikationen und den selektierten Designdatenbasen für Basis- und Refinementarchitekturen gegeben ist.
3. *Applications*
   Mittels dieser Komponente kann der Designer eine Zuordnung der einzelnen Applikationen zu ihren Compute- und Display-Servern vornehmen.
4. *Communications*

[3] Network File System

Diese Umgebung stellt einen Netzwerkkommunikationsmonitor dar, mit dessen Hilfe eine Überwachung der Kommunikationsaktivitäten und eine Administration der Kommunikationskanäle ermöglicht wird.

Alle möglichen Ressourcen des Netzwerks und des COMDES Systems[4] *werden in speziellen Konfigurationsdateien verwaltet, so daß der COMDES-Manager über PopUp-Menüs benutzerfreundlich eine Konfiguration der Designumgebung ermöglicht.*

3.4 Spezifikationswerkzeuge

Die Hardwarebeschreibungssprache AADL [DDG+89] stellt sowohl Sprachkonstrukte zur Darstellung von Struktur- und Verhaltensbeschreibungen als auch Konstrukte zur Spezifikation des temporalen Verhaltens der implementierten AADL Module zur Verfügung. Um die Spezifikation von AADL-Entwürfen dem Designer zu erleichtern, werden vom COMDES System drei unterschiedliche Editoren zur Verfügung gestellt:

- Der *AADL-Strukturediteor*, welcher syntaxgesteuert eine Entwicklung von AADL-Modulen auf Quellcode-Ebene erlaubt. Er wurde erweitert um Funktionalitäten zur Generierung einer auf AADL Petri-Netzen basierenden Semantikrepräsentation inklusive expliziter und symbolischer Repräsentation des AADL-Datenraums.
- Ein *AADL-Schematic Entry Editor* zur einfachen graphischen Erstellung bzw. Modifikation von AADL-Refinementarchitekturen.
- Ein *Zeitdiagrammeditor*, welcher die Spezifikation bzgl. des temporalen Kommunikationsverhaltens des AADL-Moduls auf graphischer Ebene erlaubt (siehe [SD93])

Im folgenden wird auf den AADL-Strukturediteor und den Schematic Entry Editor soweit eingegangen, wie es für das Verständnis der AADL Codeentwicklung und der Simulation notwendig erscheint.

3.4.1 AADL Strukturediteor

AADL unterstützt neben Struktur- und Verhaltensbeschreibungen noch die Möglichkeit der Spezifikation von komplexen abstrakten Datentypen bishin zur Spezifika-

[4] Zuordnung von Designobjekten zu ihren auf sie anwendbaren Applikationen

tion von temporalen Eigenschaften der entworfenen AADL Module. Dieser universelle Ansatz der Sprache spiegelt sich in einer komplexen Struktur der konkreten Syntax von AADL wieder. Um umfangreiche Spezifikationen mit AADL zu unterstützen, wurde ein interaktiver Editor entworfen, der den Designer in allen Phasen des Programmentwurfs mit AADL unterstützt. Um die Entwicklungszeit des Editors für AADL zu minimieren, wurde der Cornel Program Synthesizer Generator [RT88a, RT88b], ein System zur Generierung von sprachbasierten Editoren, eingesetzt.

Ausgehend von einer Spezifikation der Sprache AADL, welche die abstrakte Syntax, kontextsensitive Beziehungen, Anzeigeformate, konkrete Syntax und Transformationsregeln zur Restrukturierung von AADL-Sprachkonstrukten festlegt, wird automatisch ein sprachspezifischen Editor generiert. Der Cornel Program Synthesizer [RT88a, RT88b] kreiert einen Bildschirmeditor zur Manipulation von Sprachkonstrukten anhand der spezifizierten Regeln der Sprache.

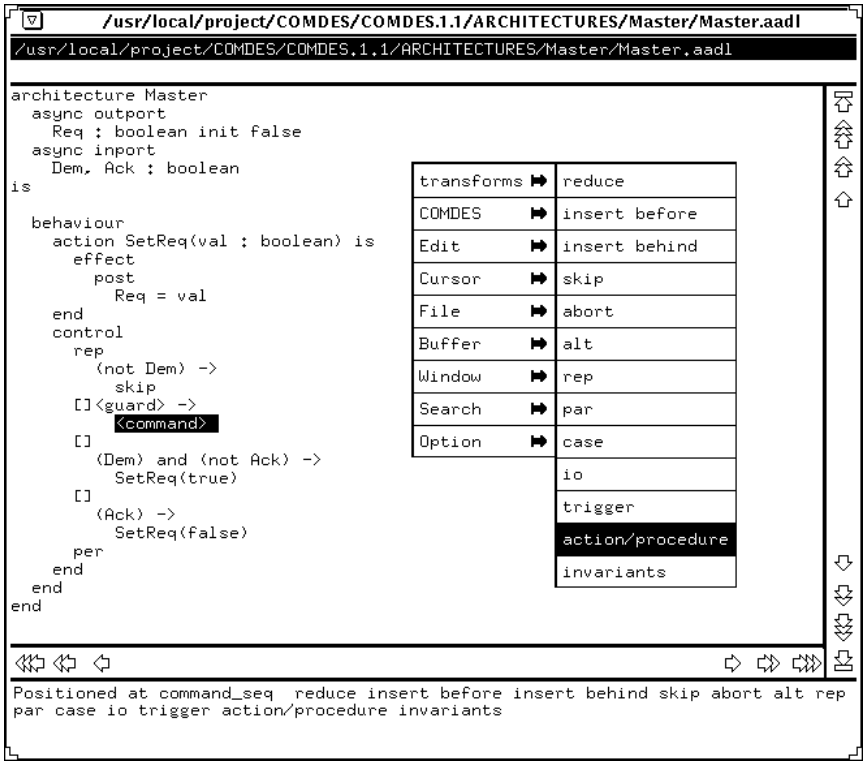


Abbildung 3.7: AADL Strukturediteor

Die Syntax-gesteuerte Behandlung von AADL durch den Generator ist von zentraler Bedeutung. Die Spezifikation der Sprache behandelt nicht nur die kontextfreien Aspekte der Syntax, sondern postuliert auch kontextsensitive Eigenschaften, wie

sie z.B. durch die statische Semantik von AADL (z.B. Type-Checking) festgelegt wird. Während der Benutzer AADL-Konstrukte im Editor eingibt bzw. modifiziert, werden vom System inkrementell alle spezifizierten und zu erfüllenden Kontextbedingungen berechnet und Verletzungen sofort dem Benutzer mitgeteilt.

Kontextbedingungen werden durch die Einführung spezieller Attribute und deren zugeordneten Attributgleichungen ausgedrückt. Durch sie wird abgeprüft, ob eine festgelegte Bedingung erfüllt ist oder nicht. Die Art, in der Objekte mit Informationen über etwaige Verletzungen von Kontextbedingungen annotiert werden, ist durch die *Unparsing*-Spezifikation des Editors festgelegt, welche den Bildschirmdarstellung der Konstrukte des abstrakten Syntaxbaumes beschreibt. Nach jeder Eingabe bzw. Modifikation des Benutzers werden die Attribute des Syntaxbaumes neu evaluiert und die Bildschirmdarstellung restrukturiert. Eventuelle neue Verletzungen bzw. Erfüllungen von Kontextbedingungen werden sofort sichtbar. Der Benutzer hat während des gesamten Entwurfs durch das sofortige Feedback einen Überblick über den Status seiner gerade in der Entwicklung befindenden AADL-Spezifikation.

3.4.1.1 AADL-Quellcodeentwicklung

User-Interface 3.2 COMDES-AADL-Strukturediteor.

Abbildung 3.7 zeigt eine Momentaufnahme des Basisfensters des AADL Strukturediteors bei der inkrementellen Entwicklung eines AADL-Moduls. Der Benutzer hat durch vorgegebene Transformationen auf dem abstrakten Syntaxbaum ein AADL `guarded-command` *in den Syntaxbaum eingeführt. Als Platzhalter für diesen, noch nicht weiter konkretisierten Konstruktor hat der Editor „*`[] <guard> -> <command>`*“ in den Programmtext eingefügt. Der Benutzer kann nun den Platzhalter für ein AADL-Kommando „*`<command>`*“ wiederum durch eine Transformation, in diesem Fall einen* `Action-` *oder* `Procedure-Call`*, verfeinern. Durch diese Formblatt-ähnliche Aufbaustruktur des Programmtextes ist eine syntaktisch korrekte AADL Codeerzeugung sichergestellt. Um den Code wieder auf seine Minimaldarstellung zu reduzieren, sind für alle Objekte der abstrakten Syntax Reduktionsfunktionen erstellt worden (*`reduce-`*Transformationen), die den Syntaxbaum rekursiv nach optionalen und somit eliminierbaren Strukturen absuchen.*

Auf Grund von Verletzungen der statischen Semantik erzeugte Fehlermeldungen werden nicht direkt in dem vom Benutzer erzeugten AADL-Quellcode eingeblendet, sondern in einem extra Fenster, dem Error View, visualisiert (siehe Abbildung 3.8). In

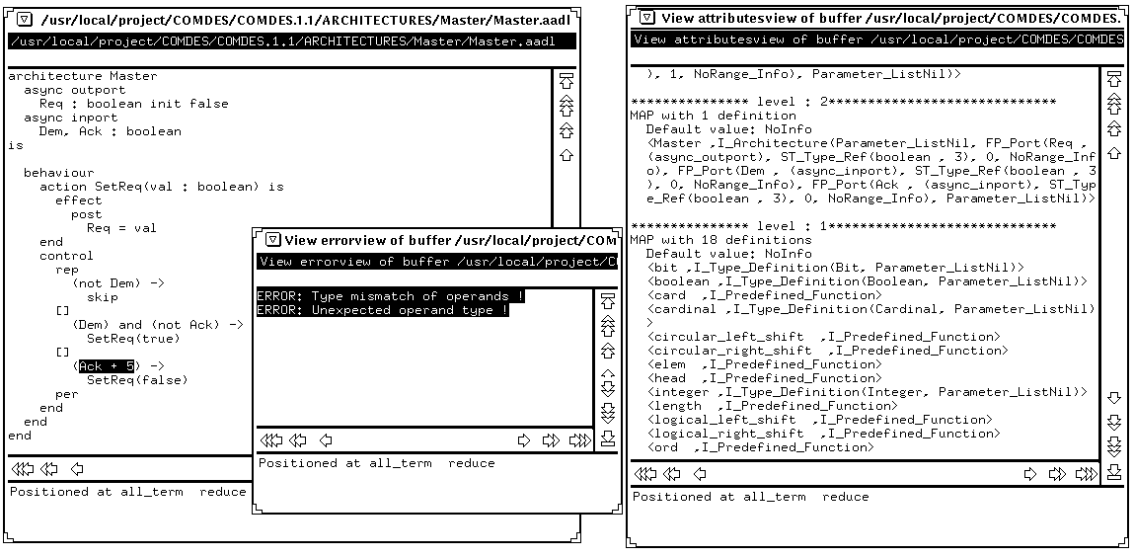


Abbildung 3.8: Sichten auf Attribute des abstrakten Syntaxbaumes

diesem Fenster werden alle im AADL-Programmtext vorhandenen Inkonsistenzen summarisch angezeigt. Die Korrespondenz zwischen Fehler und dem verursachenden AADL-Code wird durch einfache Selektion der Fehlermeldung im Error View hergestellt. Der Strukturediteor selektiert sofort den zugehörigen Code und positioniert ihn sichtbar im AADL-Textfenster.

Neben dieser Sicht werden vom AADL-Strukturediteor noch weitere, detaillierte Informationen bereitgestellt. Die Visualisierung von Attributen der AADL-Syntaxkonstrukte lassen sich in drei Kategorien fassen:

1. *Statische Semantik:*
   Fehlermeldungen und Symboltabellen
2. *AADL Petri-Netzgenerierung:*
   inkrementelle Netzberechnung und Datenflußanalyse
3. *ROBDD Generierung:*
   ROBDD-Kodierung von Datentypen und Variablen, sowie ROBDD-Repräsentationen von booleschen AADL Bedingungen

3.4.1.2 Integration

Die Integration des Cornel Program Synthesizer in das COMDES-System begründet sich durch die Bereitstellung folgender essentieller Funktionalitäten:

- AADL-spezifische Navigationsoperationen auf der AADL-Syntax,
- Speichern von inkonsistenten AADL-Spezifikationen,
- Interface zum ADTN zum Abspeichern von generierten AADL Petri-Netzen,
- Anbindung von BDD-Paketen zur Berechnung und Speicherung von symbolischen Repräsentationen des AADL Datenraums und
- Netzwerkfunktionalitäten zur Kommunikation mit dem NRAMP-Prozeß sowie der petrinetzbasierten AADL (Debug-)Simulationsumgebung.

Eine Realisierung der Intergration wurde auf zwei Ebenen durchgeführt. Durch *SSL*, der Spezifikationssprache des CPS-Systems, konnten alle nichttrivialen Berechnungen auf den Attributen der abstrakten Syntaxdarstellung der AADL-Realisierung durchgeführt werden. Auf dieser Spezifikationsebene steht jedoch keine Erweiterung des Systems bezüglich der Integration fremder, externer Tools zur Verfügung. Erst durch eine Erweiterung des C-Quellcodes des Synthesizers konnte eine transparente Einbettung geschaffen werden, sodaß für den Designer keine Unterschiede zwischen Kernfunktionalitäten und den COMDES Adaptierungen/Erweiterungen des Strukturediteors wahrgenommen werden können.

3.4.2 Schematic Entry Editor

Die Erstellung komplexer hierarchischer AADL Designs mit einer umfangreichen Kommunikationsstruktur in rein textueller Form unter Einsatz des AADL-Strukturediteor ist zwar möglich, doch nicht sehr benutzerfreundlich. Das COMDES-System unterstützt deshalb den Designer mit einer aus dem Hardware-Design bekannten Methodik des *Schematic Entry Editors*. Hierbei wird die Komposition von AADL Architekturen durch einen graphischen Editor unterstützt, in dem die graphischen Objekte (*Box* und *Linie*) AADL Architekturen und Kommunikationsstrukturen repräsentieren.

User-Interface 3.3 Schematic-Entry-Editor.

Abbildung 3.9 zeigt den Editor mit einer geladenen Architektur, welche aus drei Subkomponenten aufgebaut ist. Die besondere Rolle des Schematic Entry Editors wird im Kontext der COMDES-Simulation deutlich. Mit ihm ist es dem Benutzer möglich, Breakpoints innerhalb einer hierarchischen Struktur zu definieren. D.h. sobald ein Kommunikationswunsch über einen selektierten Port etabliert wird, kann der Debugger ein Monitoring bzgl. des aktuellen Simulationskontextes aufbauen und

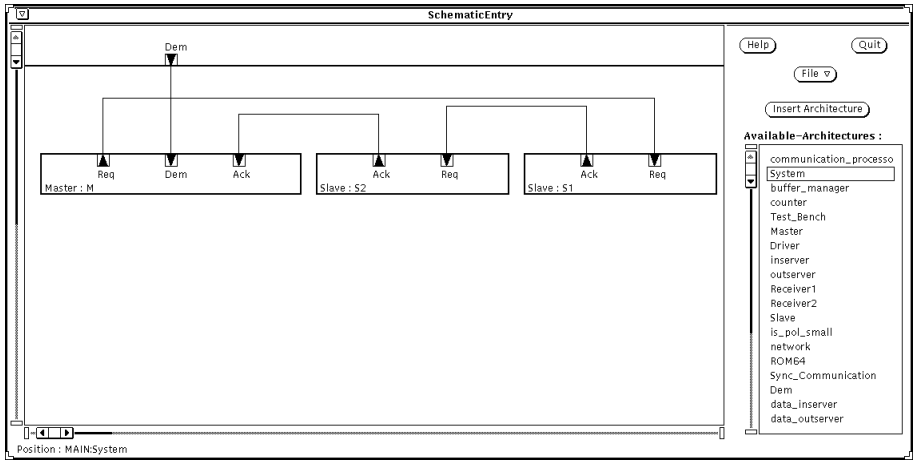


Abbildung 3.9: Schematic-Entry-Editor für AADL Refinement-Architekturen

dem Benutzer eine Navigation im aktuellen Design anbieten.

3.5 Simulation

Wie schon in der Einleitung dieser Arbeit motiviert wurde, kann eine Simulation eines komplexen Designs einen formalen Korrektheitsnachweis nicht erbringen. Im Gegensatz zur formalen Verifikation sind die Verfahren der Simulation jedoch einfacher handhabbar und schneller auf größere zu validierende Fallbeispiele anwendbar. Grobe Fehler bzw. Irrwege lassen sich somit bereits in frühen Designphasen erkennen und kostengünstig beheben.

Ausgehend von der Annahme, daß ein komplexes Design in der Entwurfsphase fast immer mit Fehlern behaftet ist, wurde besonderer Wert auf eine komfortable Debuggingumgebung für die COMDES-Simulation von AADL Designs gelegt. Dies bezieht sich sowohl auf die einfach zu handhabende, graphische Benutzungsoberfläche als auch auf die verschieden Abstraktionsebenen zum Ansatz des Debuggens. Da sich die Simulation der AADL Module auf die Simulation von AADL Petri-Netzen als Semantikrepräsentation abstützt, kann der Benutzer in zwei Kategorien frei durch einen Design navigieren:

1. Navigation auf AADL-Source-Level durch ein hierarchisches AADL-Design anhand des Schematic-Entry-Editors und
2. Navigation durch die Abstraktionsebenen des Debuggers von der AADL-Source-Code Ebene über die AADL Petri-Netzschicht bis hin zum Maschinen-

code der ANM[5], die als unterste Ausführungsschicht der COMDES-Simulation zugänglich ist.

Als Prämisse für die Realisierung des AADL-Debuggers galt es, die Debug-Simulationszeit nicht stark von einer *reinen* Simulation abweichen zu lassen und es nicht zu Seiteneffekten durch Debug-Informationen kommen zu lassen, wie es z.B. in heute existierenden C- und PASCAL-Debuggern der Fall ist. Realisiert wurde ein Ansatz, in dem der generierte ANM-Code für ein AADL-Design uniform für die Simulation und für den Debugger geeignet ist. Nur die Realisierung der ANM des Debuggers unterscheidet sich von der des Simulators dahingehend, daß einzelne ANM-Befehle mit vollständig unabhängigen Debug-Erweiterungen des Simulationskerns versehen sind. Somit ist eine Simulation im Debug-Modus effizient und erzeugt keine unerwünschten Seiteneffekte auf die Ausführung des AADL-Moduls.

3.5.1 Client/Server Architektur der Simulationswerkzeuge

Die vollständige COMDES-Simulationsumgebung ist aus vier eigenständigen Komponenten aufgebaut (siehe Abbildung 3.10), welche ihren Informationsaustausch via Interprozeßkommunikation auf TCP/IP-Protokollebene abwickeln.

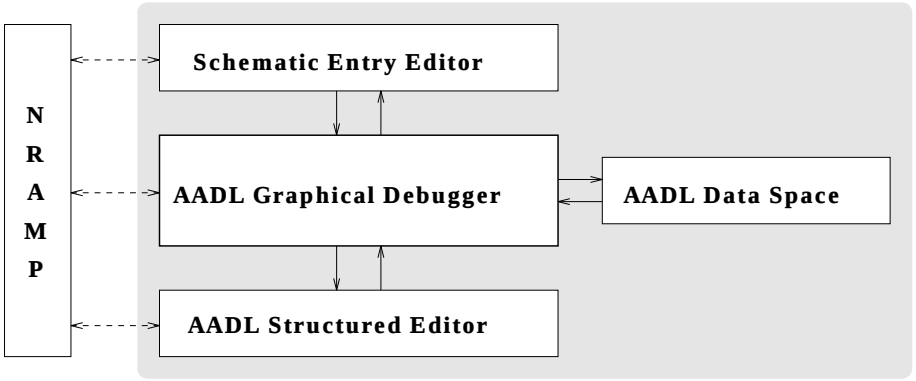


Abbildung 3.10: Interprozeßkommunikation der COMDES-Simulationsumgebung

Als Kern dieses Simulationssystems lassen sich der AADL Debugger und der AADL Datenraum ausweisen. Die Teilung des eigentlichen Simulatorkerns in zwei separate

[5]Als Kern des COMDES-Simulators agiert eine *Abstrakte Netz-Maschine* (ANM). Sie führt speziellen ANM-Code aus, der aus der Übersetzung von AADL Petri-Netzen resultiert, welche als Semantikrepräsentation innerhalb von COMDES eingesetzt werden [Dö97].

Prozesse ist durch die Größe der beiden einzelnen Komponenten motiviert.[6] Desweiteren ist durch diesen Ansatz eine parallele Abarbeitung von Tasks im AADL-Datenraum und im Simulationskern problemfrei möglich. Damit der Datentransfer zwischen beiden Prozessen effizient abgewickelt wird, ist er direkt zwischen beiden Prozessen realisiert worden. Eine Kommunikation via NRAMP hätte keine Informationsvorteile für den AADL-Manager mit sich gebracht.

Als weitere in das Debugger Environment integrierte Tools sind ein speziell adaptierter AADL-Strukturediteor und der Schematic Entry Editor auszumachen. Der Strukturediteor erlaubt eine inkrementelle Visualisierung der erreichten Breakpoints auf AADL-Sprachebene, wodurch ein besonderer Vorteil durch diese Transparenz bei der Analyse von parallel agierenden AADL-Modulen und deren Kommunikationsverhalten mit anderen AADL Modulen erreicht wird. Die Definition von Breakpoints, mit denen die Kommunikation der einzelnen AADL-Module untersucht werden soll, wird durch den Schematic Entry Editor komfortabel möglich. Durch einfache Selektion einer *Verbindungslinie* wird die diesem graphischen Objekt zugeordnete AADL-Struktur berechnet und konsistent in die Liste der zu überwachenden Breakpoints in den Debugger integriert. Gerade bei Designs mit mehreren Inkarnationen einer AADL-Refinementarchitektur hat sich dieses Verfahren als sehr vorteilhaft erwiesen da der Kontext des Gesamtdesigns vom Editor transparent verwaltet wird.

Eine Kommunikation mit dem NRAMP wird außer von dem vom AADL Datenraum repräsentieren Prozeß von den drei weiteren Prozesses etabliert. Dies ist notwendig, um die Verteilung der Simulationsresourcen dem Gesamtsystem und dem Benutzer zu erhalten.

User-Interface 3.4 COMDES-Simulation.

Die Simulation einer hierarchisch aufgebauten AADL-Verhaltensbeschreibung soll anhand eines Beispiels aus Sicht des Designers dargestellt werden. Hierbei handelt es sich ein Master-Slave System, welches das Alternating-Bit-Protokoll darstellt. Das Master-*Modul sendet, nachdem es von einer externen Komponente getriggert wurde (dem* Dem-*Modul), ein REQUEST-Signal an ein* Slave-*Modul, welches daraufhin dem* Master-*Modul ein ACKNOWLEDGE-Signal zurücksendet. In dem aktuellen Beispiel wurde eine Realisierung mit zwei Slave-Modulen und einem Master-Modul gewählt, damit die mehrfache Instanziierung eines Modul gezeigt werden kann.*

[6] Zu dem Zeitpunkt der ersten Realisierung und Konzeption (1992) waren SUN-Workstation mit ca. 8 MB Hauptspeicher und 32 MB Swap-Space die Zielarchitektur des Systems.

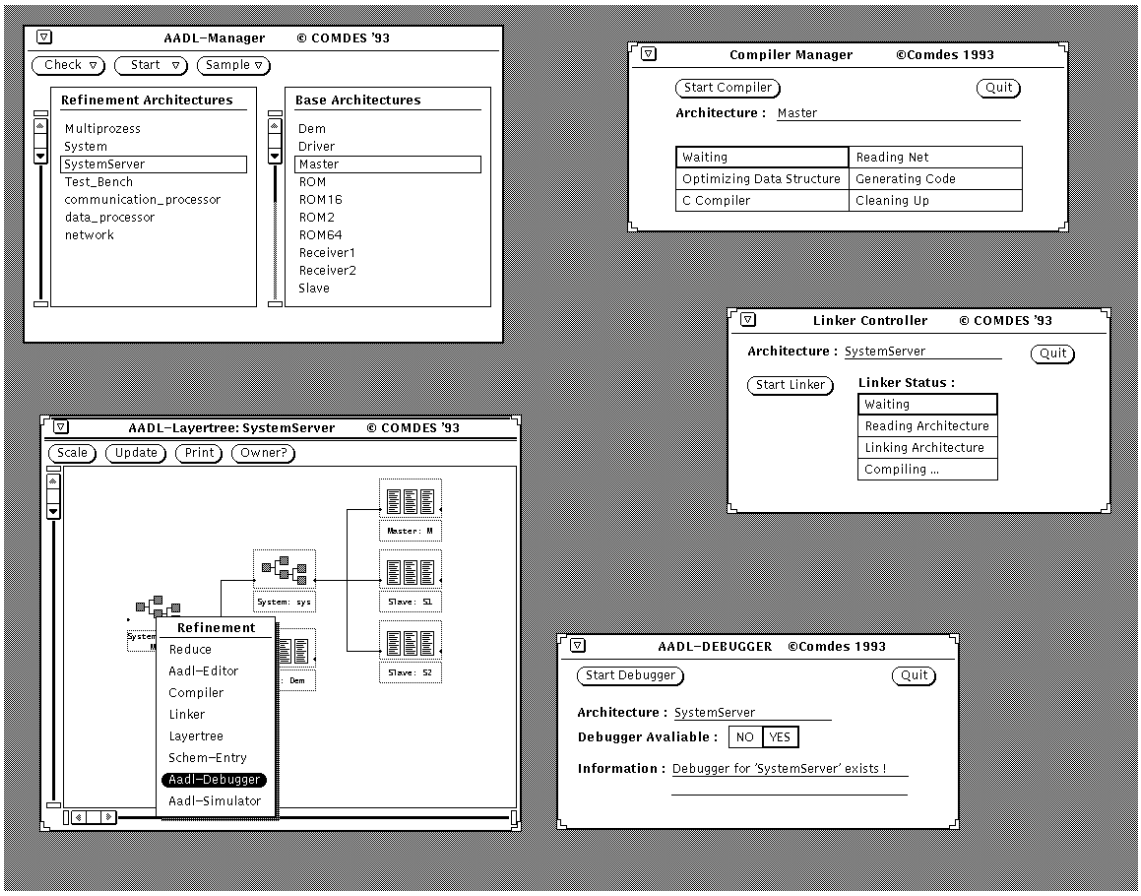


Abbildung 3.11: Startumgebung der Simulation

Nachdem der Designer alle Module, z.B. mittels des AADL-Strukturediteors, in die Designdatenbasis eingetragen hat, wird eine Weiterverarbeitung der Module erlaubt. Abbildung 3.11 zeigt einen typischen Bildschirmaufbau der ersten Phasen der Simulation. Im AADL-Manager (oben links innerhalb der Abbildung) wurde das Modul SystemServer *als Refinement-Architektur selektiert und als Layertree visualisiert. Die oben beschriebene Struktur des* Master-Slave *Systems wird wieder dem Benutzer transparent gemacht. Jedes beteiligte AADL-Modul muß nun in der ersten Phase in Objektcode compiliert werden. Dies kann entweder direkt vom Manager aus geschehen oder via Pop-Up-Menüs innerhalb der Layertree Darstellung angestoßen werden. Die verschiedenen Stufen der Compilation können vom Benutzer in den entsprechenden Compiler-Managern nachvollzogen werden. Sind alle Module übersetzt worden, kann mittels eines hierarchischen Linkers ein Gesamtsystem für das SystemServer-Modul erzeugt werden. Der gesamte Prozeß wird ständig vom System überwacht. Ggf. weisen Benutzerhinweise mittels Pop-Up-Fenster auf Fehlersituationen hin, wie z.B. Linken eines noch nicht vollständig compilierten AADL-Entwurfs durch Modifikation*

eines Moduls. Am Ende der ersten Phase existieren für die Refinement-Architektur sowohl ein Simulator als auch ein Debugger als Runtime-System, die nun benutzergesteuert selektiv gestartet werden können. Da die Simulation auf reine Geschwindigkeit ausgelegt ist und nur eine Statistik und Traceinformationen erzeugt, soll hier auf das Debuggen einer AADL-Architektur beispielhaft eingegangen werden. Abbildung 3.12 zeigt einen Bildschirmaufbau, wie er als typisch für eine Debugging-Sitzung angesehen werden kann. Im AADL-Debugger (im unteren rechten Bildschirmbereich) sind alle Architekturen des Moduls SystemServer aufgelistet und dem Benutzer zur Definition von Breakpoints, Visualisierung von Variablenbelegungen, Navigation und Aktionsverwaltung zugänglich.

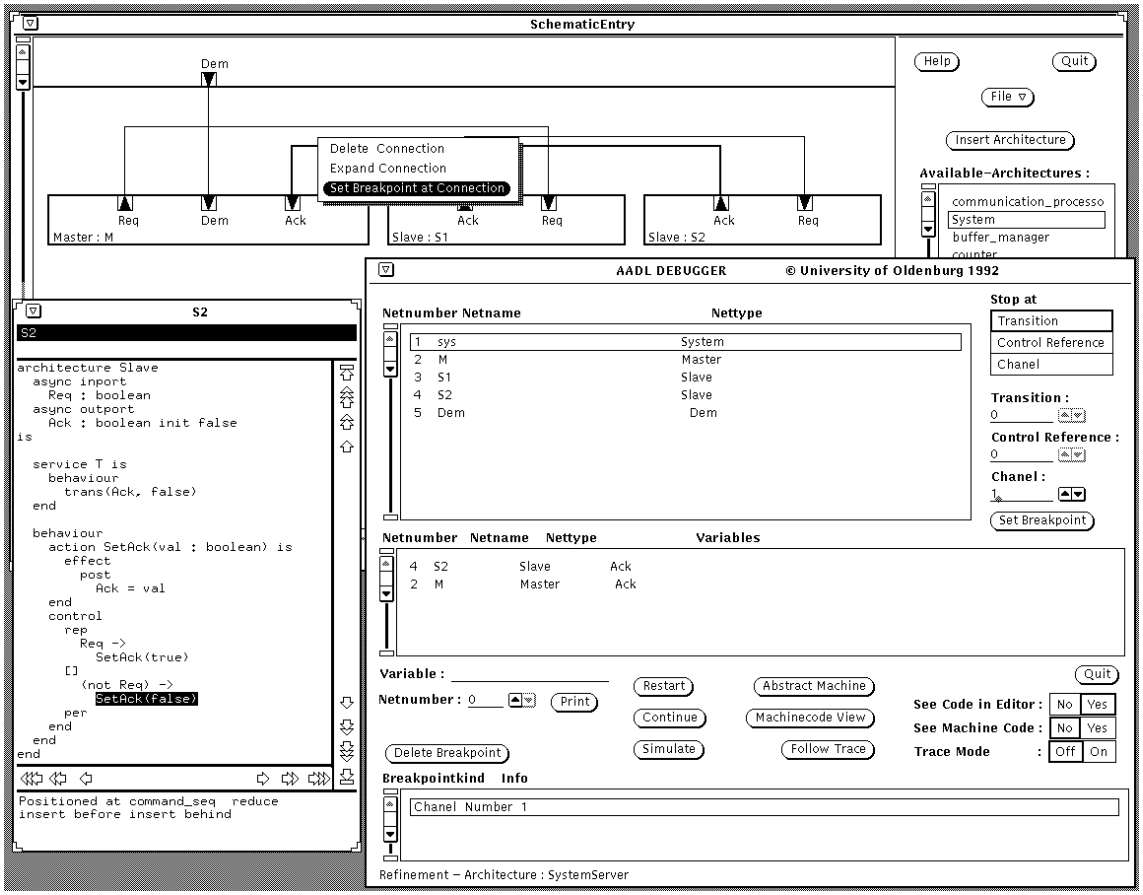


Abbildung 3.12: Teilansicht auf die Benutzungsoberfläche des COMDES-Debuggers

Mit Hilfe des Schematic-Entry-Editors können durch Selektion von Verbindungslinien *zwischen Ports der aktuellen Architektur auf graphischer Ebene Breakpoints dem Debugger mitgeteilt werden. Sie werden automatisch mit den (entsprechenden) zu überwachenden AADL-Objekten der Architektur assoziiert. Im konkreten Fall wird sowohl die Variable* Ack *der Inkarnation* S2 *der Architektur* Slave *als auch die Va-*

riable Ack *der Inkarnation* M *der Architektur* Master *vom Debugger überwacht. Der Debugger wird im aktuellen Beispiel seine Ausführung der Simulation beenden, bevor versucht wird, ein Datum zwischen den beiden beteiligten Architekturen zu transferieren. Um dem Benutzer eine detailliertere Sicht in seine AADL-Implementierung zu erlauben, wird im zugeordneten Debugging-AADL-Strukturediteor das AADL-Modul der* „aktiven" *Architektur geladen und die zu schaltende Action als selektiert visualisiert. Weitere Debugmöglichkeiten, auf die hier nicht weiter eingegangen werden soll, bestehen z.B. aus in einer Low-Level Analyse der zugrunde liegenden abstrakten Netzmaschine und der Visualisierung des AADL-Datenraumes.*

Eine Performance-Evaluation hat ergeben, daß sich die Kommunikation zwischen dem AADL-Debugger und dem zugeordneten AADL-Datenraum als einzig möglicher Engpaß der Simulation erweist. Messungen ergaben auf einem SUN Sparc2-System eine obere Schranke von ca. 10.000 Kommunikationen zwischen beiden Prozessen [Par91]. Dies führt bei großen AADL-Fallstudien zu einer akzeptablen durchschnittlichen Schaltrate von ca 15.000 Transitionen pro Sekunde.

3.6 Verifikation

Durch die in [Dö97] gegebene Übersetzung von AADL in Petri-Netze wird eine formale Semantik von AADL Modulen definiert. Eine Analyse der abgeleiteten AADL Petri-Netze kann durch Konstruktion und Analyse von Fallgraphen der Petri-Netze basierend auf Finite State Model Checking erfolgen. Dieses automatische Verfahren setzt eine endliche Kodierung des AADL-Moduls in AADL Petri-Netze voraus, sodaß nur endliche Datentypen zur AADL-Modulspezifikation zum Einsatz gelangen dürfen.

Die Verifikation von Moduleigenschaften basiert auf den aus AADL Petri-Netzen abgeleiteten endlicher Automaten; z.B. die des Moore-Typs [HS66]. Für die Verifikationskomponente des COMDES-Systems ergeben sich zwei zu erfüllende Funktionalitäten:

1. Erzeugung der endlichen Automatenrepräsentation einer AADL-Basisarchitektur und
2. Verifikation des abgeleiteten Automaten gegen die Spezifikation des Verhaltens der AADL-Basisarchitektur, welche in Symbolischen Zeitdiagrammen bzw. temporaler Logik gegeben ist.

Ein kritischer Engpaß bei diesem Verifikationsansatz stellt die Repräsentation des Fallgraphen dar. Für kleine AADL-Basismodule mit weniger als 10^6 Zuständen ist eine explizite Repräsentation des Graphen in einer Adjazenzlistendarstellung im Hauptspeicher eines heutigen Computers mit ca. 1 GB noch möglich. Für industrielle Fallstudien ist dieser Ansatz jedoch nicht adäquat und erzwingt den Einsatz von symbolischen, ROBDD-basierten Methoden. Hierbei wird der Fallgraph bzw. die Transitionsrelation des Automaten durch boolesche Relationen (Funktionen) charakterisiert (siehe Kapitel 6.2.3). Durch den Einsatz dieser symbolischen Methoden sind Automaten mit mehr als 10^{500} Zuständen handhabbar.

Das COMDES-System unterstützt eine vollständige Verifikationsschiene sowohl für die explizite als auch für die symbolische Verifikationsmethodik. Da die Realisierung der expliziten Verfahren [EC83, EC86, Jos90] kein Neuigkeitspotential für diese Arbeit darstellt, soll im folgenden nur auf die symbolische Verifikationsschiene von COMDES näher eingegangen werden. Die theoretischen Grundlagen werden ausführlich in Kapitel 6.2 diskutiert, so daß in diesem Kapitel nur die COMDES-Verifikationskomponente aus Sicht eines Designers vorgestellt wird. Er muß hierbei vier Berechnungsschritte durchführen:

1. Erzeugung der AADL Petri-Netzrepräsentation der AADL-Basisarchitektur
2. Erzeugung einer ROBDD-Repräsentation des AADL-Datenraums
3. Erzeugung eines symbolischen Modells, welche sowohl die Kontroll- als auch die Datentransformationen symbolisch kodiert
4. Verifikation der zu überprüfenden Eigenschaften durch Einsatz eines symbolischen Modell Checkers

User-Interface 3.5 COMDES-Modellgenerierung.

Die Erzeugung des AADL Petri-Netzes wird automatisch beim Einlesen der AADL-Basisarchitektur in die COMDES-Datenbasis generiert. Die symbolische Darstellung des AADL-Datenraums benötigt durch seine Parametrisierungsmöglichkeit noch eine weitere Benutzerinteraktion, welche explizit vom Designer initiiert werden muß.

Abbildung 3.13 zeigt auf der rechten Seite den AADL-Strukturediteor mit der über den COMDES-Manager geladenen Slave-Basisarchitektur.Die Erzeugung der symbolischen Darstellung des Datenraum wurde mittels des (erweiterten) Standard-Pop-Up-Menüs gestartet. Bei dieser Aktion werden folgende Teilberechnungen ausgeführt und die resultierenden Informationen in die COMDES Datenbasis integriert:

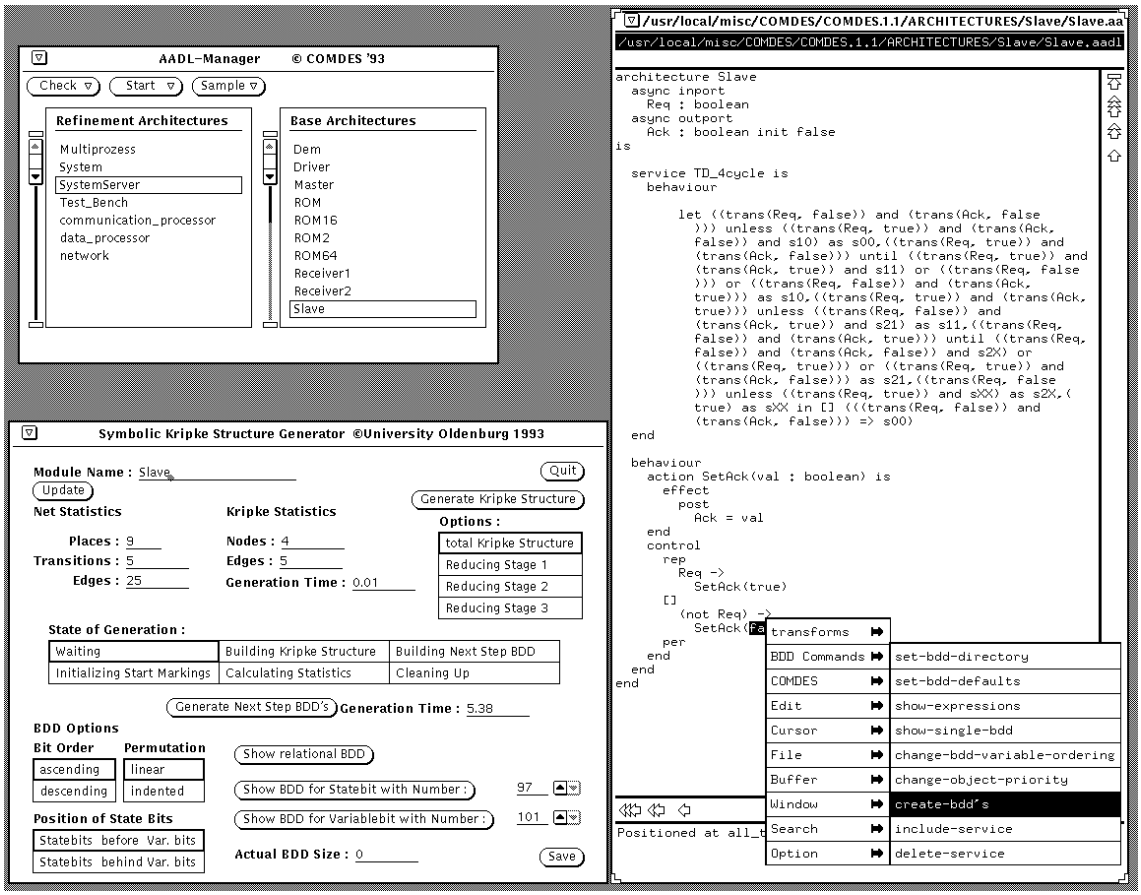


Abbildung 3.13: Symbolische Modellgenerierung für AADL-Basismodule

- *Alle (endlichen) AADL-Datentypen werden in eine binäre Darstellung kodiert*
- *Für alle AADL-Variablen wird anhand der Datentyp-Kodierung eine Menge von BDD-Variablen allokiert*
- *Es wird eine initiale Variablenordnung generiert*
- *Alle referenzierten AADL-Actions und -Conditions der AADL-Basisarchitektur werden in äquivalente ROBDD-Transformationen kodiert*

Für die Erzeugung des symbolischen Modells kann nun hieran anschließend eine Kodierung der Petri-Netzdarstellung angestoßen werden. Für diese Aufgabe steht der symbolische Kripkestrukturgenerator zur Verfügung (siehe Abbildung 3.13 links unten). Um eine effektive Generierung der Zielstruktur zu gewährleisten, steht eine Benutzungsoberfläche des Tools zur Verfügung, die dem Designer eine Modifikation einer Vielzahl von Produktionsparametern, wie z.B. die Einflußnahme auf die Binärkodierung des AADL-Datenraums, erlaubt. Der Ablauf der Modellgenerierung ist in zwei Phasen unterteilt: Zu Beginn der Transformation wird die Kon-

trollstruktur des von der AADL-Spezifikation abgeleiteten Petri-Netzes in eine explizite Kripke-Struktur, de s.g. Kontrollautomaten transfomiert (siehe Kapitel 6.2.2 für eine detaillierte Beschreibung). In einer zweiten Phase wird diese Struktur in ROBDDs kodiert. Der Designer kann sich nun die entstandenen ROBDDs graphisch anzeigen lassen und ggf. die Kodierungsparameter variieren, bis ein hinreichend gutes Ergebnis erzielt wurde und abgespeichert werden soll.

Zur Verifikation eines AADL-Moduls wird die Implementierung des Systems gegen ihre temporallogische Spezifikation validiert. Die hierzu benötigten temporalen Formeln können im COMDES-System auf zwei Arten erzeugt werden:

1. Mittels einer Zeitdiagrammspezifikation, welche durch spezielle Compiler in temporale Logik kodiert und in der COMDES-Datenbasis der AADL-Implementierung zugeordnet abgespeichert wird, oder
2. durch direkte textuelle Eingabe des Designers zu Beginn des Model Checkings.

User-Interface 3.6 COMDES-Model-Checker.

Abbildung 3.14 zeigt die Benutzungsoberfläche des symbolischen COMDES Model Checkers, die auch gleichzeitig als Editor für die zu verifizierenden temporallogischen Spezifikationen dient. Ist eine Spezifikation bzw. ein Service selektiert worden, so werden die Propositionen in ROBDDs transformiert und dem Model Checker übergeben, welcher dann die Formel verifiziert oder widerlegt. Der Designer kann die Berechnungsschritte überwachen und ggf. bei maximaler (virtueller) Speicherauslastung den Model Check Prozeß ggf. von Hand terminieren.

3.7 Abgrenzung zu existierenden Toolsets

Um eine Einordnung zu aktuell existierenden CAD-Systemen für den Hardwaredesign durchzuführen, muß eine Klassifikation der Bewertungskriterien vorgenommen werden, anhand derer die Besonderheiten des COMDES-Systems herausgearbeitet werden kann. Bei der Diskussion der *Referenz*-CAD-Systeme soll in erster Linie auf deren Vielseitigkeit und Leistungsfähigkeit eingegangen werden. Eine Betrachtung der Realisierung der Simulations- und Verifikationskomponenten wird in den Kapitel 5 und 6 vorgenommen, da sie stark von den semantischen Modellen, z.B. der Zwischenrepräsentation als Petri-Netz, abhängig sind.

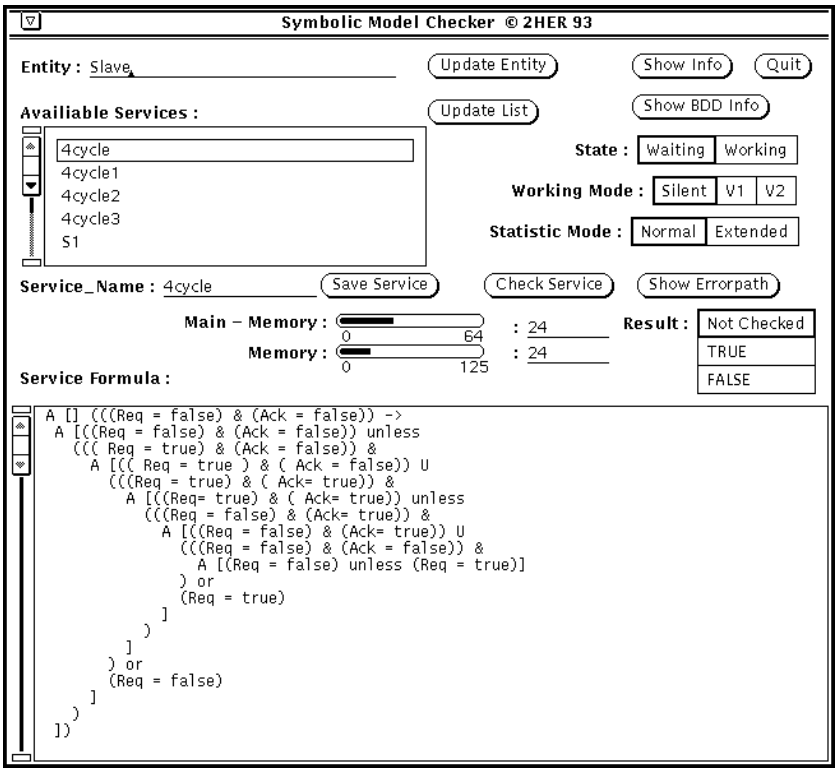


Abbildung 3.14: Symbolischer Model Checker für AADL-Basismodule

Bei der computerunterstützten Entwicklung von Hardwarekomponenten spielt die dem jeweiligenden Toolset zugrundeliegende Hardwarebeschreibungssprache eine zentrale Rolle für die Entwurfsmethodik. Als Vergleichssysteme sollen drei Toolsets herangezogen werden, die auf der standardisierten Hardwarebeschreibungssprache VHDL [IEE87][7] basieren: CADENCE und FORMAT, sowie einem auf SDL als ADL basierendem System.

3.7.1 CADENCE und Synopsys

Diese Systeme können durch ihre weltweite Verbreitung als Vertreter für kommerziellen CAD-Toolsets angesehen werden, welche sowohl Spezifikations- als auch Simulationsumgebungen für den Hardware-Designer zur Verfügung stellen, die eine industrielle Akzeptanz erfahren haben. Im folgenden sollen vier Vergleichskriterien kurz herangezogen werden um eine Differenzierung der Ansätze, basierend auf AADL bzw. VHDL als ADL, zu ermöglichen:

[7] VHDL und AADL wurden etwa zeitgleich Mitte der 80er Jahre entwickelt

Entwurfsebene: COMDES unterstützt durch AADL mit seinen an OCCAM und CSP angelehnten Spezifikationskonzepten höhere Entwurfseben, wie z.B. der Spezifikation abstrakter Mikroprogramme oder komplexe Verhaltensbeschreibungen, wie sie typisch auf der System-Ebene sind. VHDL unterstützt dagegen durch seine Entstehungsgeschichte in sehr hardwarenahen Kontexten optimal die Gatter- und RT-Abstraktionsebenen, wobei sich diese Konzepte nur unzureichend zur Modellierung höhere Entwurfsebenen heranziehen lassen.

Diese Eigenschaften lassen sich explizit an folgenden Punkten belegen:

Datentypen: Die Datentypwelt von AADL ist eng an der Programmiersprache ML[Pau91] orientiert, welche zahlreiche Konzepte und Konstruktoren zur Definition von einfachen, statischen bishin zu komplexen, dynamischer Datenstrukturen anbietet. VHDL bietet dahingegen eine kleinere, an Hardwarestrukturen orientierte Datentypwelt zum Entwurf an.

Kommunikationskonzepte: Für den Entwurf von verteilten und parallel agierenden Komponenten ist die gute Modellierung des Kommunikations- bzw. des Synchronisationsverhaltens des globalen Systems wünschenswert. AADL unterstützt durch seine Nähe zu OCCAM und CSP sowohl *synchrone* als auch *asynchrone* Kommunikation wie auch den Informationsaustausch durch *shared memory*. VHDL unterstützt die Modellierung des Interaktionsverhaltens jedoch nur mit dem Konzept des *wait*-Statements, bei dem alle Prozesse zur gleichen Zeit einen Informationsaustausch durchführen. Aber gerade für höhere Abstraktionsebenen, in dem Komponenten nur sehr lose, z.B. über Protokolle, gekoppelt sind, ist dieser Ansatz als zu regide anzusehen.

Simulation: Durch den langjährigen kommerziellen Einsatz von CADENCE und Synopsys ist ihre Simulationskomponente aus Sicht der Benutzungsoberfläche sehr ausgereift und unterstützt alle vom Designer gewünschten Debugging-Optionen auf VHDL-Source-Level. COMDES kann diesen Standard durch seine im universitären Kontext entwickelte Software aus Gründen der „Man-Power“ nicht gerecht werden. Trotzdem werden von COMDES alle zur vollständigen Entwicklung von AADL-Implementierungen notwendigen Debugging-Optionen vom System her unterstützt, sodaß keine *qualitativen Unterschiede* auszumachen sind (siehe Kapitel 3.5).

Formale Verifikation: Wie schon in der Einleitung motiviert wurde, benötigt die formale Verifikation von Hardwareentwürfen eine semantische Fundierung der

jeweiligen HDL. Für VHDL existierte lange Zeit nur eine informelle Beschreibung der Simulationssemantik [IEE87][8], so daß das CADENCE- als auch das Synopsis-System mit umfangreichen Testumgebungen, jedoch nicht mit einer formalen Verifikationskomponente ausgestattet ist.

Sowohl das Sprachdesign als auch die formale Semantik von AADL wurde unter der Prämisse entwickelt, direkt für die formale Verifikation von AADL-Modulen zugänglich zu sein, was sich u.a. in der axiomatischen Spezifikationsweise der Actions einer Verhaltensbeschreibung ausdrückt. Somit haben die verifizierten Entwürfe des COMDES-Systems einen weitaus höheren qualitativen Standard als die mit mit CADENCE oder Synopsys erstellten Designs, welche nur mittels umfangreicher Simulationen validiert werden können.

Systemarchitektur: Eine Realisierung der beiden kommerziellen Systeme als Client/Server-Architektur ist auf Grund seiner frühen Entstehungsgeschichte nicht gegeben. Die Integration verschiedenster Teilapplikationen ist mittels einer gemeinsamen Benutzungsoberfläche realisiert, wobei die einzelnen Applikationen ihren Informationsaustausch über File-Schnittstellen realisieren. Da COMDES sowohl File- als auch Netzwerk-Schnittstellen bereitstellt, ist eine tiefere Integration von Teilapplikationen in das Gesamtsystem möglich.

3.7.2 FORMAT

Das ESPRIT Projekt FORMAT[9] hatte sich u.a. zur Aufgabe gestellt, Methoden der formalen Verifikation für VHDL industriell zugänglich zu machen. Hierbei flossen die mit dem COMDES-System gemachten Erfahrungen und dessen Ergebnisse bei der Verifikation von AADL-Verhaltensbeschreibungen direkt mit ein und erfuhren somit ihre industrielle Umsetzung.

Die Systemarchitektur ist größtenteils an COMDES angelehnt, d.h. der Datenfluß der Spezifikations- und VHDL-Komponentenverifikations-Tools, die Designdatenbasis und die Verwaltung abgeleiteter Designobjekte wurde analog dem COMDES-System realisiert. Durch die zu gewährleistende Kompatibilität zu existierenden VHDL-Design- und Simulations-Tools wurde die Systemarchitektur jedoch nicht als Client/Server-Architektur realisiert. Dies begründete sich aus der Situation heraus,

[8] Erste Versuche einer formalen Semantikgebung wurde Anfang der 90er Jahre durch z.B. [vT90] und dem ESPRIT Projekt 6128 (FORMAT) vorgenommen.

[9] ESPRIT Projekt 6128 (1992-1995)

daß sich die formale Verifikation von VHDL-Designs problemlos in industrielle Designumgebungen und Designprozesse integrieren sollte.

Zentrale Erweiterungen im Vergleich zum COMDES-System sind die Integration eines *interaktiven Theorembeweisers* sowohl zur Verifikation von Systemeigenschaften zusammengesetzter Module und die Verwaltung eines *Verifikationsstatusses* eines komplexen Designs. Er wird vom *Proof Manager* verwaltet und gibt dem Designer u.a. noch Aufschluß über noch offene Beweisverpflichtungen zur vollständigen Verifikation seines Entwurfs.

3.7.3 PEP

Das PEP[10] Projekt [Gra95] stellt ein *partial oder* Verifikationssystem [Bes96] dar. Es besteht aus einer auf Petri-Netzen basierenden Programmier- und Verifikationsumgebung. Als Eingabe akzeptiert das System eine in der parallelen Programmiersprache $B(PN)^2$ [11] geschriebenen Implementierung und eine temporallogischen Formel, welche Aussagen über Variablen des $B(PN)^2$-Programms als auch über das Erreichen von Kontrollpunkten innerhalb des Programms erlaubt.

Ein Vergleich des PEP- mit dem COMDES-System soll hier kurz anhand ihrer Verifikationskomponente vorgenommen werden. Als Unterschiede lassen sich z.B. folgende Punkte erkennen:

- Die Spezifikation von temprallogischen Eigenschaften kann im COMDES-System durch den Einsatz von symbolischen Zeitdiagrammen kompakt und intuitiv vom Designer vorgenommen werden. Durch den Einsatz von CTL[EC81, Jos93] als weitere Spezifikationssprache ist der Sprachumfang mächtiger als die im PEP-System integrierte temporale Logik zur Beschreibung von Halbordnungseigenschaften. So lassen sich u.a. keine *eventuality properties* als Regeln der branching-time Logik von PEP darstellen, sondern nur eine abgeschwächte Regel als *possibly* beschreiben.
- Die Übersetzung der Eingabesprachen ($AADL$ bzw. $B(PN)^2$) in Petri-Netze unterscheidet sich auch in der Kodierung des Datenraums. Bei der Kodierung von AADL in AADL Petri-Netze werden nur die Kontrollanteile in Netzstrukturen abgebildet. Der Datenraum findet außer in den Triggerstellen keine Repräsentation im Netz. Hierdurch kommt es zu einer sehr kompakten Darstellung, die bei der

[10] *P*rogramming *E*nvironment based on *P*etri Nets

[11] *B*asic *P*etri *N*et *P*rogramming *N*otation

Netzkodierung im PEP-System nicht vorgenommen wird. Bei dem hier verfolgten Ansatz wird der gesamte Datenraum eines $B(PN)^2$-Programms im Netz sichtbar gemacht.

3.7.4 SDL

Als weitere Sprache zur Spezifikation bzw. Implementierung von Hardwarekomponenten, gerade in den Bereichen der Telekommunikation und Signalsteuerung, hat sich SDL [BHS91] weit verbreitet. Im Laufe der langen Entwicklungsgeschichte von SDL wurden – vornehmlich von Anwendern – eine ganze Reihe von entwurfsunterstützenden Werkzeugen entwickelt. Kern solcher Werkzeuge ist in der Regel ein graphikorientierter Editor für SDL-Diagramme. Um diesen gruppieren sich Module zur statischen Überprüfung von SDL-Entwürfen, zur Simulation (hier wurde speziell die Technik der sogenannten Message Sequence Charts zur graphischen Darstellung von Simulationen entwickelt), zur Code-Erzeugung (etwa von C-Code) und evtl. auch zur Analyse spezieller Eigenschaften.

Am weitesten entwickelt und auch am meisten verbreitet ist das System SDT der Telelogic AB [AB97]. Für Analysezwecke greift es auf das von Holzmann (vgl. [AHP96]) entwickelte Werkzeug SPIN zurück, das – ganz oder teilweise – den Zustandsgraphen eines SDL-Diagramms erzeugt, der dann mit Hilfe einer Reihe von Graphalgorithmen weiter untersucht werden kann. Andere Systeme wie etwa SITE [FDT95] berechnen zu einem SDL-Diagramm ein unterliegendes Petri-Netz und verwenden dann ein Analysewerkzeug für Petri-Netze, um bestimmte Eigenschaften wie Lebendigkeit oder Deadlockfreiheit (die dann aber auf der Ebene des unterliegenden Netzes definiert sind) nachzuweisen.

Weitere kommerzielle Tools werden von den Firmen Telelogic und Verilog angeboten. Das Produkt *Object*GEODE von Verilog bietet Funktionalitäten zur Spezifikation und statischen Semantikanalyse. Eine Validierung erstellter SDL-Designs kann aber nur mittels Simulation vom ihm abgeleiteten C++ Code durchgeführt werden. Somit ist für diesen Bereich noch keine umfassende Verifikationsumgebung existent.

Eine formale Semantikgebung und eine darauf aufbauende Verifikation von SDL-Designs wird von der Firma SIEMENS (mit dem SVE-System), die jedoch wiederum keine (guten) integrierten Simulationsumgebungen bereitstellt. Hierdurch ist eine Motivation bzw. Animation für den Designer von aufgedeckten Fehlern in der SDL-Verhaltensbeschreibung nur schwer nachzuvollziehen. Auch fehlt eine transpa-

rente Verwaltung der von den SDL-Implementierungen abgeleiteten Objekten zur Zwischenrepräsentation.

Teil II

Theoretische Grundlagen

Kapitel 4

Notationen und mathematische Grundlagen

4.1 Relationen und Funktionen

Eine binäre Relation R zwischen zwei Mengen X und Y ist eine Teilmenge des Kartesischen Produktes $X \times Y$, d.h. $R \subseteq X \times Y$.

Wenn $X = Y$ ist, dann wird R als Relation auf X bezeichnet. Folgende Eigenschaften von Relationen werden unterschieden:

Reflexivität	*gdw.*	$(x,x) \in R$ *für alle* $x \in X$,
Irreflexivität	*gdw.*	$(x,x) \notin R$ *für alle* $x \in X$,
Symmetrie	*gdw.*	*für alle* $x_0, x_1 \in X$ *gilt :*
		wenn $(x_0,x_1) \in R$ *dann ist auch* $(x_1,x_0) \in R$,
Antisymmetrie	*gdw.*	*für alle* $x_0, x_1 \in X$ *gilt :*
		wenn $(x_0,x_1) \in R$ *und auch* $(x_1,x_0) \in R$,
		dann ist $x_0 = x_1$,
Transitivität	*gdw.*	*für alle* $x_0, x_1, x_2 \in X$*gilt :*
		wann immer $(x_0,x_1) \in R$ *und* $(x_1,x_2) \in R$,
		dann ist auch $(x_0,x_2) \in R$.

Der transitive, reflexive Abschluß R^* einer Relation R auf der Menge X ist die kleinste transitive und reflexive Relation auf X, die R als Teilmenge enthält.

Die relationale Komposition $R_0 \circ R_1$ der Relationen R_0 und R_1 auf der Menge X ist

gegeben durch:

$$R_0 \circ R_1 = \{(x_0, x_2) \mid es\ gibt\ x_1 \in X\ mit\ (x_0, x_1) \in R_0\ und\ (x_1, x_2) \in R_2\}.$$

Die n-fache relationale Komposition R^nder Relation R auf der Menge X ist induktiv wie folgt definiert:

$$\begin{aligned} R^0 &= \{(x_0, x_0) \mid x_0 \in X\}, \\ R^{n+1} &= R^n \circ R. \end{aligned}$$

Für binäre Relationen bietet sich eine infix-Notation als Schreibvereinfachung an. An Stelle von $(x_0, x_1) \in R$ kann auch $x_0 R x_1$ geschrieben werden. Jede binäre Relation $R = X \times Y$ hat eine inverse Relation $R^{-1} \subseteq Y \times X$ gegeben als:

$$x_1 R^{-1} x_0\ wenn\ x_0\ R\ x_1.$$

Seien X und Y als zwei Mengen gegeben. Eine Funktion oder Abbildung von X nach Y ist eine binäre Relation f zwischen X und Y mit folgender Eigenschaft: Für jedes Element $x \in X$ gibt es genau ein Element $y \in Y$ mit xfy ($f(x) = y$ in präfix-Notation). Um die Abbildung f von X nach Y anzugeben, wird folgende Notation verwandt:

$$f : X \to Y$$

Die Menge X wird Definitionsbereich und die Menge Y wird Wertebereich von f genannt. Genau wie bei Relationen sind für Funktionen Eigenschaften speziell charakterisierbar. Eine Funktion $f : X \to Y$ ist:

$$\begin{aligned} &injektiv && gdw.\ f(x_0) \neq f(x_1)\ für\ alle\ x_0, x_1 \in X\ mit\ x_0 \neq x_1, \\ &surjektiv && gdw.\ für\ alle\ y \in Y\ ein\ x\ \in X\ existiert\ mit\ f(x) = y, \\ &bijectiv && gdw.\ f\ ist\ injektiv\ und\ surjektiv, \end{aligned}$$

Für Funktionen deren Definitionsbereich ein Kartesisches Produkt ist, z.B. $f : X_1 \times \ldots \times X_n \to Y$, wird bei präfix-Notation ein Klammerpaar nicht angegeben, d.h.

$$f(x_1, \ldots, x_n) = y.$$

Hierbei wird y als Wert der Funktion f für die Werte $x_1, \ldots, x_n$ betrachtet.

Für Funktionen, in denen Definitionsbereich und Wertebereich überein stimmen ($f : X \to X$), kann ein Wert $x \in X$ als Fixpunkt angesehen werden, wenn

$$f(x) = x$$

gilt.

4.2 Boolesche Algebra

In diesem Kapitel soll eine kurze Einführung in die Basistheorie der Booleschen Algebra gegeben werden, welche für die Theorie der BDD-basierten Codierung endlicher Systeme notwendig ist.

Definition 4.1 Boolesche Algebra.

Eine Boolesche Algebra ist ein fünf-Tupel

$$(B, +, *, 0, 1),$$

in dem B eine endliche Menge (Träger) darstellt, $+$ und $*$ binäre Operatoren auf B, 0 und 1 Elemente aus B sind, so daß $\forall a, b, c \in B$ die folgenden Bedingungen erfüllt sind:

1. *Kommutativgesetz:* $a + b = b + a$; $a * b = b * a$
2. *Distributivgesetz:* $a + (b * c) = (a + b) * (a + c)$, $a * (b + c) = (a * b) + (a * c)$
3. *Identität:* $a + 0 = a$; $a * 1 = a$
4. *Komplement:* $\forall a \in B : \exists \bar{a} : a + \bar{a} = 1$; $a * \bar{a} = 0$ ∘

Das System $(\{false, true\}, +, *, false, true)$ mit den Operatoren $+$ und $*$ als *logisches oder* und *logisches und* stellt bekannter Weise eine boolesche Algebra dar. Im Verlauf dieser Arbeit wird immer davon ausgegangen, daß $B = \{false, true\}$ ist. Eine n-stellige logische Funktion ist dann eine Abbildung

$$f : B^n \to B.$$

Sei $F_n(B)$ die Menge der n-stelligen logischen Funktionen auf B. Dann ist das System

$$(F_n(B), +, *, 0, 1)$$

auch eine boolesche Algebra, in der $+$ und $*$ die Addition und Multiplikation boolescher Funktionen darstellen. 0 und 1 stellen dabei die 0- und 1-Funktionen dar ($f_0(x_1, \ldots, x_n) = 0$ und $f_1(x_1, \ldots, x_n) = 1$).

Die Algebra von Klassen einer Menge S besteht aus der Menge 2^S und zwei Operationen auf 2^S: $\cup$ (Vereinigung) und $\cap$ (Durchschnitt). Diese Algebra erfüllt die Bedingungen der Booleschen Algebra und es ergibt sich, daß das System $(2^S, \cup, \cap, \emptyset, S)$ eine Boolesche Algebra darstellt. Das *Darstellungstheorem* (Stone, 1936) gibt den Ansatz zur Kodierung von Mengen dieser Arbeit wieder:

Satz 4.1 Darstellungstheorem.

Jede endliche Boolesche Algebra ist isomorph zu der Booleschen Algebra auf Teilmengen einer endlichen Menge S.

Hieraus folgt, daß das Argumentieren über Teilmengen, Durchschnitt und Vereinigung bezüglich einer endlichen Menge S isomorph zur Ausführung von logischer Operationen $(+, *)$ auf logischen Funktionen ist. Desweiteren folgt aus dem Theorem, daß die Kardinalität des Trägers jeder Booleschen Algebra eine Potenz von 2 ist. Im Besonderen ist die Algebra von Klassen einer Menge S ($|S| = 2^n$) isomorph zur Booleschen Algebra der n-stelligen logischen Funktionen.

Die Funktionen

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \quad \textit{und} \quad f_{\bar{x}_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

werden als Kofaktor von f bezüglich x_i und $\bar{x}_i$ bezeichnet. Mit Hilfe dieser Schreibweise läßt sich folgender Satz ableiten:

Satz 4.2 Boolesche Expansion.

Sei $f : B^n \to B$ eine Boolesche Funktion. Für alle $(x_1, \ldots, x_n) \in B^n$ gilt:

$$f(x_1, \ldots, x_n) = x_i * f_{x_i} + \bar{x}_i * f_{\bar{x}_i}$$

Durch die Darstellung von f mittels Kofaktorisierung läßt sich das Konzept der Abstraktion in das System logischer Funktionen integrieren; so sind z.B. f_{x_i} und $f_{\bar{x}_i}$ unabhängig von x_i. Abstraktionen haben eine direkte Korrespondenz zur All- und Existenzquantifizierung boolescher Prädikate. Die existentielle und universelle Abstraktion von f in Bezug zu x_i wird definiert als:

$$\exists_{x_i} f = f_{x_i} + f_{\bar{x}_i}; \quad \forall_{x_i} f = f_{x_i} * f_{\bar{x}_i}$$

Als Beispiel sei die Funktion $f = a\bar{b}\bar{c} + \bar{a}c + bc$ betrachtet. Die Kofaktorzerlegung bezüglich a und $\bar{a}$ ergibt $f_a = bc + \bar{b}\bar{c}$ und $f_{\bar{a}} = c + bc$. Die Abstraktionen bezüglich a $\exists_a f = f_a + f_{\bar{a}} = \bar{b}\bar{c} + c$ und $\forall_a f = f_a * f_{\bar{a}} = bc$. $\exists_a$ ist die Funktion, welche für Werte von b und c zu *true* evaluiert, für die es einen Wert von a gibt, für die f zu *true* evaluiert. $\forall_a f$ ist die Funktion, die zu *true* für $b = 1$ und $c = 1$ evaluiert, so daß f für jeden Wert von a zu *true* evaluiert.

4.3 Binäre Entscheidungsgraphen

Eine optimale Darstellung von booleschen Termen im Kontext der technischen Informatik ist eine notwendige Voraussetzung zur Handhabung industrieller Problemstellungen, wie. z.B. die Schaltkreissynthese und die Verifikation digitaler Systeme. Die bekanntesten Darstellungsformen boolescher Terme sind sicherlich die *Veitch-* oder *Karnaugh-Diagramme* [McC86, Pil88].

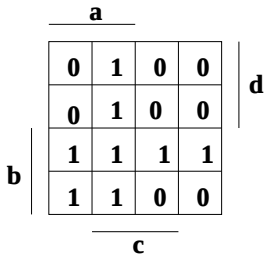


Abbildung 4.1: Karnaugh-Diagramm der Funktion $ab \vee db \vee ac$

Sie haben jedoch den Nachteil, daß *jede* boolesche Funktion in n Variablen ein Diagramm mit 2^n Feldern benötigt, siehe Abbildung 4.1. Sie stellen somit ein gutes visuelles Hilfsmittel dar, sind aber ungeeignet für die Darstellung von Funktionen in vielen Variablen.

Auf der Basis von *Mintermen (Maxterme)* können *kanonische DNF (kanonische KNF)* einer Funktion in n Variablen angegeben werden. Da die Anzahl der Minterme bzw. Maxterme bereits bei sehr einfachen Funktionen exponentiell anwachsen kann sind sie keine optimale Darstellungsform. Auch die Anwendung der *Reed-Muller-Form* [McC86] auf boolesche Terme führt zu ähnlichen Ergebnissen und scheidet aus. Die Darstellung von booleschen Termen in Form von *Termbäumen* stellt eine sehr kompakte Darstellung dar. Jedoch sind sie nur begrenzt einsetzbar, da die Überprüfung der Gleichheit zweier boolescher Terme nicht ohne weiteres wegen der fehlenden kanonischen Darstellung möglich ist.

Während *Termbäume* zu Darstellungen führen, bei denen eine Variable über viele Stellen des Baumes *verteilt* vorkommen kann, ist die Shannonsche Normal Form faktorisiert. Nach dem Shannonschen Entwicklungssatz kann jeder boolescher Term nach jeder Variablen orthogonal zerlegt wereden. Die Entwicklung eines Terms nach allen in ihm vorkommenden Variablen führt zur Shannonschen Normalform, die durch einen Entscheidungsbaum repräsentiert werden kann, siehe Abbildung 4.2.

Das Beispiel verdeutlicht, daß die Blätter des Entscheidungsbaumes nur die boole-

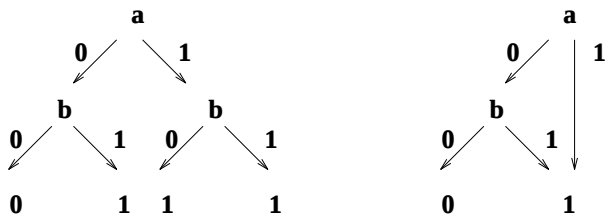


Abbildung 4.2: Entscheidungsgraph und -baum der Funktion $(\neg a \wedge b) \vee a$

schen Konstanten enthalten. Da die Anzahl der Blätter mit der Anzahl der Variablen exponentiell steigt, ist eine kompaktere Darstellung durch Mehrfachnutzung von Teilgraphen notwendig und führt zu einem binärem Entscheidungsgraphen (Binary Decision Diagram, BDD). Ein Entscheidungsgraph ist ein gerichteter, zyklenfreier und zusammenhängender Graph mit einem Wurzelknoten, wobei die Blätter die booleschen Terme $true$ und $false$ bzw. 1 und 0 tragen. Mit den restlichen, inneren Knoten k wird

- eine Variable $v(k)$,
- zwei Nachfolgerknoten $true(k)$ und $false(k)$ und
- ein boolescher Term b_k assoziiert. Er wird wie folgt bestimmt:

$$b_k := (v(k) \wedge b_{true(k)}) \vee (\neg v(k) \wedge b_{false(k)}).$$

  $b_{true(k)}$ und $b_{false(k)}$ sind die mit den Nachfolgerknoten $true(k)$ und $false(k)$ assoziierten booleschen Terme.

Formal läßt sich ein BDD wie folgt charakterisieren:

Definition 4.2 Binary Decision Diagram (BDD).

Sei X eine Menge von Booleschen Variablen. Ein gerichteter, azyklischer Graph (V, E, l, r) heißt BDD, gdw.

- *$V = T \dot{\cup} N$ ist eine endliche Menge von Knoten, disjunkt zerlegbar in eine Menge von Terminalknoten (T) und Nichtterminalknoten (N).*
- *$E \subseteq N \times ((N \cup T) \times (N \cup T))$ ist eine Menge von Kanten, die jeden Nichtterminalknoten mit zwei Nachfolgern aus V verbinden, einem sogenannten 1- bzw. 0-Nachfolger. Im folgenden soll gelten, daß der erste Nachfolger der 1-Nachfolger ist, während der zweite der 0-Nachfolger ist.*
- *$l : V \to X \cup B$ ist eine Beschriftungsfunktion, für die gilt: $l(T) \subseteq B$ und $l(N) \subseteq X$, d.h. Terminalknoten sind mit Booleschen Werten beschriftet, Nichtterminalknoten mit Variablen.*

- *$r \in V$ ist die Wurzel des Graphen. Es muß gelten, daß r nicht Nachfolger eines anderen Knoten ist.* ◦

Diese Darstellungsform von BDDs wurde erstmals von Lee und Akers [Ake78] eingeführt. Um jedoch effizient Berechnungen auf BDDs ausführen zu können, muß eine kanonische Darstellung erfolgen. Hierzu wird eine globale, totale Ordnung „<" auf den Variablen definiert. Für jeden Knoten k und Sohnknoten l von k gilt:

$$v(k) < v(l)$$

Desweiteren werden drei Transformationsregeln angegeben, deren Anwendung auf einen BDD seine Darstellung reduziert, jedoch seine berechnende Funktion erhält:

1. Löschen doppelter Terminalknoten
   Eliminiere alle Knoten außer einem Terminalknoten mit einem speziellem Label und *setze* alle alten Kanten auf dieses Blatt um.
2. Löschen doppelter Knoten
   Für zwei nichtterminale Knoten k und l mit $v(k) = v(l)$, $true(k) = true(l)$ und $false(k) = false(l)$ kann ein Knoten gelöscht werden und seine ein- und ausgehenden Kanten auf den verbleibenden Knoten umgesetzt werden.
3. Löschen redundanter Tests
   Nichtterminale Knoten k mit $true(k) = false(k)$ werden gelöscht und alle eingehenden Kanten werden auf $true(k)$ gesetzt.

Ist ein BDD mit Hilfe der o.g. Regeln reduziert worden, so befindet er sich in einer kanonischen Form bezüglich der globalen Ordnung und wird als ROBDD bezeichnet. Ein ROBDD kann dann durch folgende Definition charakterisiert werden:

Definition 4.3 Reduced Ordered BDD (ROBDD).
Sei $(X, <)$ eine total geordnete Menge von Booleschen Variablen. Ein gerichteter, azyklischer Graph (V, E, l, r) heißt ROBDD, gdw.

- *(V, E, l, r) ist ein BDD*
- $\forall v \in N, v_1, v_2 \in V : \forall (v, (v_1, v_2)) \in E : (v_1 \in N \Rightarrow l(v) < l(v_1)) \wedge (v_2 \in N \Rightarrow l(v) < l(v_2))$
- *$\forall v, v' \in N : v \neq v' \Rightarrow (V, E, l, v)$ ist nicht isomorph zu (V,E,l,v')*
- $\forall v \in N : \forall (v, (v_0, v_1)) \in E : v_0 \neq v_1$ ◦

ROBDDs haben u.a. zwei positive Eigenschaften: Einerseits sind sie kanonisch in ihrer Darstellung und anderseits können boolesche Standardoperationen effizient mit ihnen ausgeführt werden können. Eine erste effiziente Implementierung eines (RO)BDD-Paketes zur Manipulation und Berechnung von ROBDDs wurde von Bryant vorgestellt [BRB90, Bry86].

Eine Kodierung höherer Datentypen, wie sie bei der Spezifikation von AADL-Systementwürfen zum Einsatz kommen, kann durch ihre Transformation in eine adäquate Binärkodierung erfolgen. Eine detailierte Beschreibung der hierfür notwendigen Verfahren wird in [Her92, HP94] beschrieben. In diesen Quellen wird die Kodierung höherer Datentypen und getypter Variablen eingeführt. Desweiteren wird eine Motivation für eine Heuristik der zur Kodierung notwendigen binären ROBDD-Variablenordnung eingeführt.

4.4 AADL Petri-Netze

Petri-Netze stellen ein Modell zur Beschreibung und Analyse von Abläufen mit nebenläufigen und nichtdeterministischen Vorgängen dar und wurde erstmals von C.A. Petri 1962 vorgeschlagen. Für eine detaillierte Einführung in die Petri-Netz-Thematik wird auf [Rei85, BF88] verwiesen.

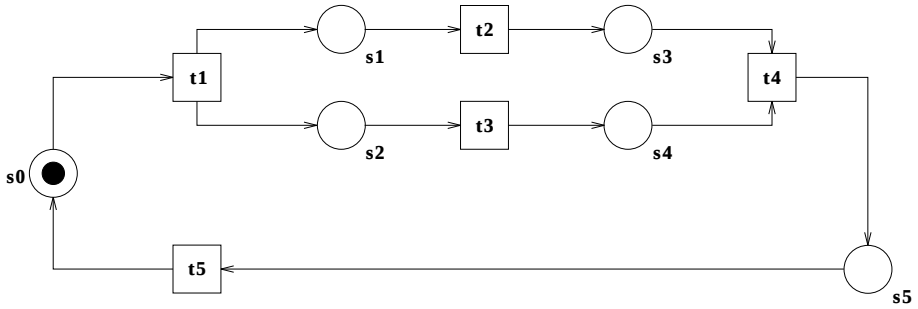


Abbildung 4.3: Petri-Netz

Als Motivation soll im folgenden kurz auf das Beispielnetz aus Abbildung 4.3 eingegangen werden. Der Graph beschreibt die Ablaufstruktur. Um dynamische Vorgänge aufzuzeigen, werden die Stellen mit Objekten (Token) belegt, die durch die Transitionen weitergereicht werden. Sie werden als Punkte in den Stellen dargestellt. Eine Belegung der Stellen mit Token wird als *Markierung* des Netzes bezeichnet und als Tupel geschrieben. Die Kodierung ergibt sich wie folgt: Ist eine Stelle (un-)markiert,

so ist die entsprechende Komponente des Tupels mit einer 1 (bzw. 0) belegt. Für das Beispielnetz aus Abbildung 4.3 ergibt sich als aktuelle Markierung

$$(1, 0, 0, 0, 0, 0).$$

Der Bewegungsablauf der Marken wird durch die *Schaltregel* der Transitionen festgelegt: Eine Transition kann schalten und erzeugt somit eine Folgemarkierung, wenn alle Stellen ihres Vorbereichs mit mindestens einem Token belegt sind. Schaltet die Transition, dann wird von den Stellen im Vorbereich jeweils ein Token abgezogen und auf jede Stelle im Nachbereich ein Token gelegt. Ein B/E-System kann formal wie folgt angegeben werden:

Definition 4.4 B/E-System. *Eine Struktur $N = (S, T, F, m_0)$ heißt B/E-System, falls*

- S *endliche Menge von Stellen,*
- T *endliche Menge von Transitionen,*
- $F \subseteq (S \times T) \cup (T \times S)$ *eine zweistelle* Flußrelation *von N und*
- $m_0 : S \to \{true, false\}$ *die Startmarkierung des Netzes ist.* ◦

Um die Beschreibung komplexer Systeme, wie sie z.B. im systemnahen Hardware-Entwurf üblich und notwendig sind, noch kompakt repräsentieren zu können, reichen BE-Systeme als Strukturen nicht mehr aus. Die Transformation von komplexen Datenstrukturen und die Handhabung von Kontrollstrukturen verlangen eine Erweiterung der Netzstruktur, um eine kompakte Darstellung des Petri-Netzes zu erhalten.

Eine effiziente Realisierung stellen für diese Problematik *AADL Petri-Netze* dar, welche im Kontext der Formalisierung der Semantik von AADL [DD88, Dö97] entwickelt wurden. Eine detaillierte Einführung in die diese Netzklasse stellt [DJS93a] dar. Die Netzklasse ist eine Erweiterung der B/E-Systeme. Transitionen sind mit Zustandstransformationen auf einem getypten Datenraum annotiert. Stellen können mit booleschen Ausdrücken über diesen Datenraum annotiert sein und werden dann als Trigger Stellen bezeichnet. Sind sie mit einem Token belegt, so symbolisiert sies, daß die zugehörige Bedingung über den Datenraum gültig ist. Das Schalten einer Transition führt neben dem Tokenspiel auch eine Transformation auf dem Datenraum aus. Abbildung 4.4 zeigt als Veranschaulichung ein AADL Petri-Netz, welches zyklisch eine boolesche Variable b negiert.

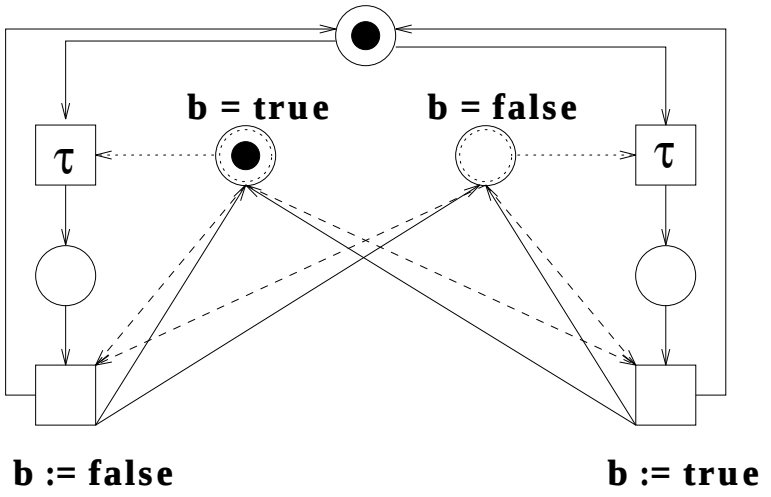


Abbildung 4.4: AADL Petri-Netz eines Negators

Die Kantenrelation der AADL Petri-Netze unterscheidet vier Arten von Kanten: *Enabling, Consuming, Delivering und Conditional* Kanten. Die beiden ersten Kantenarten führen von Stellen zu Transitionen. Eine Enabling Kante charakterisiert nur die Testeigenschaft, ob die Stelle im Vorbereich der Transition belegt ist, damit die Transition schalten kann. Beim Schalten der Transition wird jedoch kein Token abgezogen. Consuming Kanten dagegen ziehen beim Schalten der Transition ein ggf. vorhandenes Token aus dem Vorbereich ab, d.h. Stellen im Vorbereich der Transition, die nicht mit einem Token markiert sind und nur über eine Consuming Kante mit der Transition verbunden sind, hindern die Transition nicht am Schalten. Delivering Kanten führen von Transitionen zu Kontrollfluß oder Semaphor Stellen und belegen nach dem Schalten der Transition die Stellen im Nachbereich mit einem Token. Conditional Kanten führen nur zu Trigger Stellen. Sie legen in Abhängigkeit der aktuellen Gültigkeit der Bedingung der Triggerstelle über den Datenraum ein Token auf sie. Abbildung 4.5 zeigt die graphische Repräsentation der verschiedenen AADL Petri-Netzobjekte.

Bevor eine formale Definition der AADL Petri-Netze erfolgen kann, soll noch kurz auf die Notation der Zustandstransformation des Datenraumes eingegangen werden. Die Konstruktion eines AADL Petri-Netzes ist im Rahmen dieser Arbeit immer an eine AADL Architektur gebunden, von der der Datenraum, bestehend aus Typen und getypten Variablen, abgeleitet wird. Der Zustand eines Netzes wird neben der Markierung der Stellen des Netzes durch die Werte der Inports I, Outports O und lokalen Variablen V charakterisiert. Die Menge der Input-(Output- und lokalen Variablen-)Werte ist durch die Funktion $inputs(I) = I \rightarrow type_I$

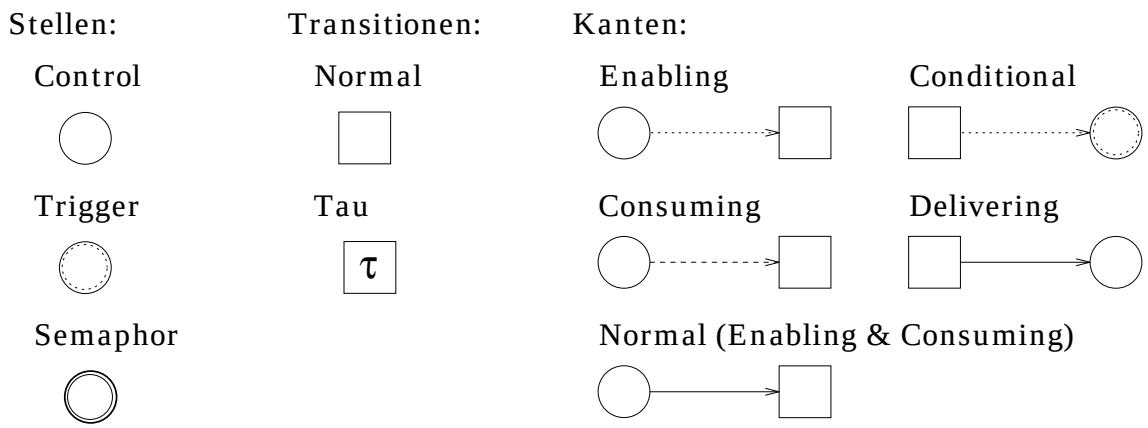


Abbildung 4.5: AADL Petri-Netzobjekte

($outputs(O) = O \to type_O$, $states(V) = V \to type_V$) bestimmt. Die Menge der Zustandstransformationen oder *continuation* $Cont(I, V, O)$ ist durch $Cont(I, V, O) = inputs(I) \times states(V) \to states(V) \times outputs(O)$ gegeben. Tau-Transitionen sind mit der Identität als Datentransformation annotiert.

Eine formale Definition von AADL Petri-Netzen kann nun wie folgt angegeben werden:

Definition 4.5 AADL Petri-Netz (Basisnetz).

Ein AADL Petri-Netz ist eine Struktur $N = (ed, V, init_V, S, T, F, \Gamma, \pi, \mathcal{M}_0)$, *mit*

- $ed = (I, O, init)$;
- V *bezeichnet eine endliche Menge von internen, getypten (Speicher-)Variablen, welche disjunkt von* $I \cup O$ *ist;*
- $init_V : V \to Type_V$ *beschreibt die Initialisierung der lokalen Variablen;*
- $S \equiv$ *endliche Menge der Stellen. Sie zerfällt disjunkt in folgende Teilmengen:*
  - $S_{Semaphor}$ *: Menge der Semaphorstellen,*
  - $S_{Control}$ *: Menge der Kontrollflußstellen,*
  - $S_{Trigger}$ *: Menge der Triggerstellen.*
    Sie zerfällt disjunkt in die Teilmengen:
    - $S_{Trigger_{no\_cond}}$ *: ohne Bedingung über dem Datenraum,*
    - $S_{Trigger_{pm}}$ *: Pattern, welches bei der synchronen Kommunikation, erkannt wird*
    - $S_{Trigger_{normal}}$ *: mit Prädikat über dem Datenraum;*
- $T \equiv$ *endliche Menge der Transitionen. Sie zerfällt disjunkt in folgende Teilmengen:*
  - T_{normal} *: Menge der Transitionen mit Datentransformation,*

T_τ : *Menge der Transitionen ohne Datentransformation,*
T_{input} : *Menge der Input-Transitionen (synchrone Kommunikation),*
T_{output} : *Menge der Output-Transitionen (synchrone Kommunikation);*

- $F \equiv$ *die Flußrelation von* N*, mit disjunkten Teilmengen*
  $F_{enabling} \subseteq S \times T$,
  $F_{consuming} \subseteq S \times T$,
  $F_{delivering} \subseteq T \times S$ *und*
  $F_{conditional} \subseteq T \times S_{Trigger}$;
- $\Gamma : T \setminus T_\tau \to Cont(I, V, O)$ *ordnet den nicht Tau-Transitionen eine Zustandstransformation zu;*
- $\pi : S_{Trigger} \to Bexpr(I, V)$ *ordnet jeder Triggerstelle eine boolesche Bedingung Bexpr in Abhängigkeit von* I *und* V *zu;*
- $\mathcal{M}_0 \subseteq 2^S$ *ist die Klasse der initialen Markierungen, wobei jedes* $M_0 \in \mathcal{M}_0$ *die Eigenschaft* $S_{Semaphor} \subseteq M_0$ *erfüllt.* ∘

Definition 4.6 Vorbereich, Nachbereich.

Für alle $s \in S$ *und* $t \in T$ *gilt:*

$$
\begin{array}{lrcl}
\textit{Enabling Vorbereich} & {}^{\cdot enb}t & := & \{s \mid (s,t) \in F_{enabling}\} \\
\textit{Consuming Vorbereich} & {}^{\cdot cons}t & := & \{s \mid (s,t) \in F_{consuming}\} \\
\textit{Delivering Vorbereich} & {}^{\cdot del}s & := & \{t \mid (t,s) \in F_{delivering}\} \\
\textit{Conditional Vorbereich} & {}^{\cdot cond}s & := & \{t \mid (t,s) \in F_{conditional}\} \\
 & & & \\
\textit{Enabling Nachberich} & s^{\cdot enb} & := & \{t \mid (t,s) \in F_{enabling}\} \\
\textit{Consuming Nachberich} & s^{\cdot cons} & := & \{t \mid (t,s) \in F_{consuming}\} \\
\textit{Delivering Nachberich} & t^{\cdot del} & := & \{s \mid (s,t) \in F_{delivering}\} \\
\textit{Conditional Nachberich} & t^{\cdot cond} & := & \{s \mid (s,t) \in F_{conditional}\}
\end{array}
$$

∘

Desweiteren werden folgende Notationen eingeführt:

- Für jede Transition t kann die Menge der durch sie potentiell veränderten (gelesenen) Variablen festgelegt werden. Diese Information wird durch die Funktion $write : T \to (V \cup O)$ $(read : T \to (I \cup V))$ repräsentiert.
- $free(b)$ beschreibt für einen booleschen Ausdruck b die in b vorkommenden Variablen

Definition 4.7 Wohlgeformte AADL Petri-Netze.
Sei N ein AADL Petri-Netz. N wird als wohlgeformt (wf(N)) bezeichnet, gdw.

- $\forall t \in T, s \in S_{Trigger} : free(cond(s)) \cap write(t) \neq \emptyset \Rightarrow s \in t^{\cdot del} \cap^{\cdot cons} t$
  Wenn eine Transformation einer Transition t die Belegung einer Triggerstelle beeinflußt, dann existiert eine Consuming-Kante von der Stelle zur Transition und eine Conditional Kante von ihr zur Triggerstelle.
- $\forall s \in S_{control} \wedge \forall t \in T : (s,t) \in F_{enabling} \Leftrightarrow (s,t) \in F_{consuming}$
  Kontrollflußstellen sind immer über eine Enabling- und eine Consuming-Kante mit einer Transition verbunden.
- $\forall s \in S_{semaphor} \wedge \forall t \in T : (s,t) \in F_{consuming} \Rightarrow (s,t) \in F_{enabling}$
  Semaphorstellen sind entweder über eine Enabling- und eine Consuming-Kante oder nur mit einer Enabling-Kante mit einer Transition verbunden. ∘

Mit den bisherigen Definitionen sind Syntax und strukturelle Eigenschaften von AADL Petri-Netzen festgelegt worden. Durch die Definition von Transitionssystemen wird den AADL Petri-Netzen eine Interleaving-Semantik zugeordnet:

Definition 4.8 Transitionssystem

Sei Σ eine endliche Menge von Zuständen. Ein Transitionssystem Θ über Σ ist ein Tripel $\Theta = (\Sigma, \rightarrow, \Sigma_0)$, mit

- $\rightarrow \subseteq \Sigma \times \Sigma$ *(Transitionsrelation)*
- $\Sigma_0 \subseteq \Sigma$ *(Menge der initialen Zustände)* ∘

Die Semantik von AADL Petri-Netzen soll formal über eine Transitionsrelation von sogenannten Netzzuständen definiert werden. Ein Netzzustand kombiniert Kontrollinformationen und Zustandsinformationen über dem zugeordneten Datenraum. Alle Kontrollflußaspekte werden durch das Tokenspiel modelliert, wobei Zustandsraumsinformationen durch die Werte von Input, Output und lokalen Variablen gegeben sind. Eine „Überlagerung“ dieser verschieden Informationen wird nur bei der Markierung der Triggerstellen sichtbar. Im folgenden werden nur *konsistente* Netzzustände betrachtet, welche garantieren, daß eine Triggerstelle nur genau dann markiert ist, wenn die assoziierte Bedingung sich zu *true* bei den aktuellen Werten der Input und lokalen Variablen evaluieren läßt.

Definition 4.9 Netzzustand *Sei N ein AADL Petri-Netz.*

- *Die Menge von Netzzuständen von N, $\Sigma(N)$, mit typischem Element $ns \in \Sigma(N)$ ist ein 4-Tupel*

$$\Sigma(N) = \{(M, inp, \sigma, out) \quad | \quad M \subseteq S,$$

$$inp \in inputs(I),$$
$$\sigma \in states(V),$$
$$out \in outputs(O)\}$$

- *Sei* $ns = (M, inp, \sigma, out) \in \Sigma(N)$. *ns wird als konsistent bezeichnet, gdw.* $\forall s \in S_{Trigger} : [\![\pi(s)]\!] inp\ \sigma$[1] $= true$ *gdw.* $s \in M$
- *Die Menge von initialen Netzzuständen von N,* $\Sigma_0(N)$*, ist die Menge aller Netzzustände* $ns = (M_0, inp_0, \sigma_0, out_0)$*, die folgende Eigenschaften erfüllen:*
  - $M_0 \in \mathcal{M}_0$
  - *ns ist konsistent*
  - $\forall o \in O : out_0(o) = init(o)$
  - $\forall v \in V : \sigma_0(v) = init_V(v)$ ∘

Die Semantik des AADL Petri-Netzes ist durch die Schaltregel des Netzes festgelegt. Das Schalten einer Transition hängt von der aktuellen Markierung des Netzes und dem Zustand des Datenraums ab. Eine Transition t kann ausgeführt werden, wenn alle Stellen im Enabling Vorbereich mit einem Token markiert sind. Der Zustand des Datenraums wird dann anhand der Transformation der Transition geändert, und die neue Markierung des Netzes wird durch das Entfernen aller Token aus dem Consuming Vorbereich und das Belegen aller Stellen im Nachbereich der Transition herbeigeführt. Hierbei ist zu berücksichtigen, daß Triggerstellen im Nachbereich nur genau dann markiert werden dürfen, wenn ihre boolesche Bedingung im neuen Zustand sich zu true evaluieren läßt.

Definition 4.10 Netzsemantik.

1. *Das Schalten einer Transition t ist definiert durch:*
   $(M, inp, \sigma, out)\ [t>\ (M', inp, \sigma', out')$ *gdw.*
   - $^{\cdot enb}t \subseteq M$
   - $(inp, \sigma', out') = \Gamma(t)(inp, \sigma, out)$
   - $M' = (M -^{\cdot cons} t) \cup t^{\cdot del} \cup \{s \in t^{\cdot cond} \mid [\![\pi(s)]\!]\sigma' = true\}$
2. *Ein Lauf eines Netzes ist durch eine unendliche Sequenz von Netzzuständen* $(M_i, inp_i, \sigma_i, out_i)$ *gegeben, so daß*

[1] In dieser Definition wird der Ausdruck $[\![b]\!] inp\ \sigma$ zur Bezeichnung der Semantik des booleschen Ausdrucks b unter der Input Belegung inp und Zustand σ benutzt.

- *für einige* t_i, $(M_i, (inp_i, \sigma_i, out_i))\ [t_i >\ (M_{i+1}, (inp_{i+1}, \sigma_{i+1}, out_{i+1}))$ *und*
- $(M_0, (inp_0, \sigma_0, out_0))$ *ein initialer Zustand ist, d.h.* $M_0 \in \mathcal{M}_0, out_0(O) = init(o), \sigma_0(v) = init_V(v)$ *gilt.*

3. *Die Semantik* $[\![N]\!]$ *ist dann durch alle Läufe des Netzes gegeben:*

$$[\![N]\!] := \{r \mid r \text{ ist ein } Lauf \text{ von } N\}$$

$\circ$

Teil III

Simulationstechniken für den Hardware Entwurf auf Systemebene

Kapitel 5

Simulationswerkzeuge

Wie schon in der Einleitung beschrieben wurde, stellt die Simulation eine wichtige Unterstützung bei der Evaluierung von komplexen Designs in frühen Entwicklungsphasen dar. Im Kontext des *COMDES* Projektes werden Simulationsverfahren zur Validierung von AADL-Implementierungen und zur Analyse von Zeitdiagrammspezifikationen eingesetzt.

Um eine effiziente Validierung zu gewährleisten, welche jeweils eine optimale Adaption an die gewünschten Analyseanforderungen darstellt, werden zwei unterschiedliche Verfahren zur Simulation der Petri-Netzzwischenrepräsentation beider Strukturen vorgestellt.

Bei der Simulation von AADL-Implementierungen wird ein compilativer Simulationsansatz der Petri-Netze verfolgt, welcher eine hohe Geschwindigkeit der Simulation der zur Laufzeit statischen Netzstruktur garantiert.

Die Simulation der aus Zeitdiagrammen abgeleiteten Petri-Netze stützt sich hingegen auf ein interpretatives Verfahren ab, da es bei ihrer Analyse nicht primär auf die Schaltgeschwindigkeit der Transitionen ankommt. Bei dem vorgestellten Simulationsverfahren gehören die Handhabung dynamischer erzeugter Petri-Netzinkarnationen und die Unterstützung von ROBDD basierter Methoden zur symbolischen Kodierung des Datenraums zur wichtigsten Aufgabe.

In den beiden folgenden Kapiteln wird in die Simulation von AADL basierten Systementwürfen eingeführt und die Simulation symbolischer Zeitdiagrammen motiviert.

5.1 Simulation von AADL-Spezifikationen

Die Spezifikationssprache AADL stellt eine Vielzahl von Konstruktoren zur Verhaltensbeschreibung und zum hierarchischen Aufbau von komplexen Designs zur Verfügung (siehe Kapitel 2). Um dem Benutzer frühzeitig eine Evaluation seines Entwurfs zu ermöglichen, soll dem Entwickler eine Analyse mittels eines AADL-Simulators und eines AADL-Debuggers zur Verfügung gestellt werden. Eine Simulation eines AADL Designs soll hierzu folgende Eigenschaften erfüllen:

- Schnelle Compilation einzelner Module,
- Verwendung von compilierten Modulen aus einer Design-Bibliothek,
- Bei Modifikation eines Moduls soll keine Recompilation des Gesamtsystems erfolgen, sondern nur dessen Teilübersetzung inklusive *Linken*,
- Die Erstellung eines Gesamtdesigns soll nur durch *Linken* compilierter Teilmodule erfolgen,
- Eine Validierung der Implementierung mit Hilfe umfangreicher Eingabedaten muß in kürzester Zeit durchführbar sein.

Wie eingangs erläutert, basiert die Simulation eines AADL Designs auf der Simulation einer Petri-Netzzwischenrepräsentation. Deren Erzeugung bzw. die Übersetzung von AADL in AADL Petri-Netze wird detailliert in [DD88, Dö97] vorgestellt. Bei dem zu Grunde liegenden Verfahren werden die AADL-Sprachkonstrukte inkrementell in AADL Petri-Netze transformiert und zu einem einzigen Gesamtnetz zusammengefaßt. Da der eigentliche Transformationsprozeß keinen Einfluß auf die Simulation der AADL Petri-Netze hat, wird er an dieser Stelle nicht weiter ausgeführt.

Damit aber die gewählte Realisierung des Simulationskonzeptes, welche die o.g. Anforderungen erfüllt, verständlich wird, erfolgt eine Gegenüberstellung zweier möglicher Simulationsansätze für Petri-Netze:

1. Simulation durch Interpretation:
   Bei diesem Verfahren wird nach jedem Schalten einer Menge von aktiven, konfliktfreien Transitionen, d.h. nach dem Ausführen des Tokenspiels, das Netz nach neuen, aktiven Transitionen abgesucht, und aus der so gewonnenen Menge mit Hilfe einer Auswahlstrategie eine neue Menge konfliktfrei schaltbarer Transitionen ausgewählt. Alle hierzu notwendigen Berechnungen werden dabei auf der Netzdatenstruktur durchgeführt.

Dieser Ansatz wurde erstmals von dem DACAPO II System als industrielles Produkt realisiert. Es simuliert Computer Systeme auf der Hardware-Ebene und benutzt dabei konfliktfreie, zeitbezogene Petri-Netze mit verschiedenen Auswahlstrategien [Ram78, Ram89, Pau86].

2. Simulation durch Compilation:
   Bei diesem Ansatz wird die Menge der aktiven Transitionen nicht nach jedem Schaltvorgang neu durch vollständige Analyse der Netzdatenstruktur berechnet. Jede Transition trägt bei diesem Ansatz die statische Information, welche Transitionen durch ihr Schalten aktiviert bzw. deaktiviert werden können. Das Schalten einer Transition führt somit das Tokenspiel und ein automatisches Update auf der Menge der aktiven Transitionen aus. Die Auswahlstrategie muß somit die Menge der gerade aktiven Transitionen nicht neu berechnen.
   Dieser effizientere Ansatz wurde u.a. von dem Nachfolgersystem DACAPO III realisiert[Ram89, Pau86]. Hierdurch wird eine signifikante Beschleunigung der Simulation erzielt, wodurch eine Handhabung industrieller Fallstudien erst ermöglicht wurde.

Im Gegensatz zu den im DACAPO System zum Einsatz kommenden Petri-Netze sind AADL Petri-Netze nicht zeitbezogen, und die Menge der aktiven Transitionen ist auch nicht immer konfliktfrei. Somit können die Ansätze von [Pau86] nicht vollständig übernommen werden, sondern es muß ein neuer, an AADL Petri-Netze angelehnter Ansatz entwickelt werden. Um eine effiziente Simulation von AADL Petri-Netzen durchzuführen, ist eine Anlehnung an die Struktur von AADL Spezifikationen [DDG+89] zwingend notwendig. Sie zerfallen disjunkt (siehe Kapitel 2) in

1. Spezifikation von Basisarchitekturen und
2. Spezifikation von Refinementarchitekturen.

Basisarchitekturen definieren das Verhalten von AADL Modulen, welche durch AADL Petri-Netze charakterisiert werden. Refinementarchitekturen legen die Topologie fest, anhand derer die Teilmodule verknüpft werden. Sie wird als Beschreibung für das *Linken* der Basisnetze auf AADL-Ebene benötigt. Dies bedeutet für die Simulation eine Unterstützung der folgenden Schritte:

1. Compilation der AADL Petri-Netze der Basisarchitekturen,
2. Linken der Basisnetze mit Hilfe der Topologiebeschreibung.

Das Linken der Basisnetze führt ein Verkleben von Netzobjekten durch. Bei diesem Vorgang werden Stellen und Transitionen unifiziert und zusätzliche Netzkanten eingeführt. Um Zeit für eine etwaige Recompilation zu sparen und eine modulare Compilation zu ermöglichen, kann der Linker den compilierten Objektcode der Basisarchitekturen ohne spätere Modifikation verarbeiten. Dies bedeutet, daß der compilierte C-Code der Petri-Netze der Basisarchitekturen so gekapselt ist, daß er unabhängig von seiner späteren Umgebung ist. Ein besonderes Linkverfahren, in dem nur noch die Topologie anhand der Refinementarchitekturen relevant ist, unterstützt diese komplexen Modifikationsmöglichkeiten auf AADL Petri-Netzebene.

5.1.1 Schaltregel und Simulationsalgorithmus

Eingangs wurde schon erläutert, daß eine effiziente Simulation von AADL Petri-Netzen anhand der Compilation der Netzstruktur in ein ausführbares Programm durchgeführt wird. Jeder Transition wird dabei eine Funktion zugeordnet, die neben dem Tokenspiel und ggf. einer Transformation auf dem Zustandsraum auch ein Update auf der Menge der aktiven Transitionen durchführt. D.h. daß die Information, welche Transitionen durch das Schalten aktiviert bzw. deaktiviert werden können, statisch kodiert werden kann. Um diese Codierung des Netzes zu unterstützen, müssen die Netzobjekte in adäquate Speicherstrukturen einer Zielsprache übersetzt werden, auf denen dann die Transformationen effizient durchgeführt werden können.

Die Menge der Transitionen zerfällt dabei während der Simulation dynamisch in die disjunkten Menge der aktiven (T_{active}) und nicht aktiven Transitionen ($T_{inactive}$). Da Stellen entweder mit einem Token belegt sein können oder nicht, ergibt sich die analoge Zerlegung der Menge der Stellen in S_{Token} und $S_{noToken}$. Um den o.g. Simulationsansatz zu unterstützen, werden für jede Transition $t \in T$ zwei Funktionen benötigt, die das Schaltverhalten beschreiben:

- $Check_t$ ist charakteristische Funktion für die Transition t und hat die Aufgabe, festzustellen, ob die Transition t zur Aufrufzeit unter der aktuellen Markierung M aktiviert ist. Dies ist immer dann der Fall, wenn ${}^{\cdot enb}t \subseteq M$ gilt.

- $Fire_t$ führt sowohl alle Transformationen auf Netzebene, die durch das Schalten von t angestoßen werden, als auch die Transformation auf dem Zustandsraum aus. Dies sind die in Definition 4.10 vorgestellten Aktionen:
  - Entferne alle Token aus dem Consuming-Vorbereich von T.
  - Transformiere den Datenraum bezüglich der Transformation von t.

- Lege ein Token auf alle Semaphor- und Kontrollflußstellen im Nachbereich von t.
- Lege ein Token auf alle Triggerstellen im Nachbereich von t, wenn ihre jeweilige Bedingung im aktuellen Zustand des Datenraums erfüllt ist.

Neben diesen durch die Definition vorgegebenen Netztransformationen muß noch eine Modifikation der Menge der aktuell aktiven Transitionen vorgenommen werden.

Die Menge von Transitionen, welche durch das Schalten der Transition t deaktiviert bzw. potentiell aktiviert werden können, werden mit $DeactBy_t$ bzw. $MayBeActBy_t$ bezeichnet und sind durch

$$DeactBy_t := ({}^{\cdot cons}t)^{\cdot enb}$$

$$MayBeActBy_t := (t^{\cdot del} \cup t^{\cdot cond})^{\cdot enb}$$

gegeben. Da beide Mengen nicht zwangsläufig disjunkt sein müssen, wird zuerst T_{actice} nach dem Schalten von t um die Menge $DeactBy_t$ verkleinert, um danach um die Menge $MayBeActBy_t$ vergrößert zu werden:

$$T_{actice} := T_{active} - DeactBy_t$$

$$T_{actice} := T_{active} \cup \{t' \in MayBeActBy_t \mid Check_t = true\}$$

5.1.1.1 Lazy Evaluation

Alle Informationen über das Belegen und Entfernen von Token auf Kontrollflußstellen und Semaphorstellen sind zur Compilezeit des Netzes bekannt und können fest in die Struktur der Funktion $Fire_t$ eingebettet werden. Die Behandlung der Triggerstellen weicht hiervon ab, da ihre Markierung dynamisch vom Zustand des Netzes (aktueller Zustand des Datenraums) abhängt. Das bedingte Belegen von Triggerstellen mit einem Token während der Ausführung von $Fire_t$ setzt immer eine zeitaufwendige Evaluation der zugehörigen Bedingungen über den Datenraum voraus. Die Netzstruktur kann aber so gebaut sein, daß diese berechnete Information durch das Schalten einer anderen Transitionen invalidiert wird, ohne daß ein Schalten der eigentlich vom Kontrollfluß her zugeordneten Transitionen durchgeführt wird.

Eine Behebung dieser Ineffizienz ist die Berechnung der Gültigkeit der Bedingung über dem Datenraum und somit der Markierung der Triggerstellen erst dann, wenn

sie definitiv das Schalten einer Transition beeinflußt. Diese Berechnung *on demand* wird als *Lazy Evaluation* bezeichnet und verschiebt den Test der Bedingung von der $Fire_t$-Funktion in die $Check_t$-Funktion. Dies wird durch die Einführung eines weiteren Token-Typs, dem *grauen* Token auf Netzebene ermöglicht. Es verdeutlicht, daß die Bedingung der Triggerstelle potentiell erfüllt ist, aber noch einer Evaluation bedarf. Triggerstellen können somit wie folgt belegt sein:

- *ohne* Token, gdw. $[\![b]\!]inp\ \sigma = false$,
- *graues* Token, gdw. $[\![b]\!]inp\ \sigma$ kann $true$ oder $false$, sein
- *schwarzes* Token, gdw. $[\![b]\!]inp\ \sigma = true$.

Die Einführung dieser weiteren Sorte von Token durch [Kor91] hat folgenden Zeitvorteil für die Simulation: Immer, wenn ein graues Token von einer Trigger Stelle durch eine Consuming Kante abgezogen wird, wird der Zeitaufwand für die Berechnung auf dem Datenraum eingespart.

Damit die Auswahlstrategie keine Verlangsamung hierdurch erfährt, muß die $Check_t$-Funktion an die neue Situation angepaßt werden. Sie zerfällt deswegen in folgende zwei Funktionen:

- $Check_t^{PA} = true \Rightarrow {}^{\cdot enb}t \cap (S_{semaphor} \cup S_{control}) \subseteq M$
  Die Funktion $Check_t^{PA}$ überprüft die potentielle Aktiviertheit einer Transition t, welche durch eine Markierung ihre Semaphor- und Kontrollflußstellen im Enabling-Vorbereich gegeben ist.
- $Check_t^{A} = true \Rightarrow {}^{\cdot enb}t \cap S_{trigger} \subseteq M$
  Die Funktion $Check_t^{A}$ überprüft, ob alle Triggerstellen im Vorbereich der Transition belegt sind. Unter der Prämisse, daß die Funktion $Check_t^{PA}$ bereits die potentielle Aktiviertheit sichergestellt hat, wird durch diese Funktion die sichere Aktiviertheit abgeprüft.

Aus dieser Struktur folgen für die Schaltregel vier zu unterscheidende Zustandsmodi für Transitionen [Kor91]:

0. nicht potentielle Aktiviertheit
   $T_0 := \{t \mid t \in T \wedge Check_t^{PA} = false\}$
1. potentielle Aktiviertheit ohne sichere Aktiviertheit
   $T_1 \subseteq \{t \mid t \in T \wedge Check_t^{PA} = true \wedge Check_t^{A} = false\}$

2. potentielle Aktiviertheit
   $T_2 \subseteq \{t \mid t \in T \wedge Check_t^{PA} = true\}$
3. sichere Aktiviertheit
   $T_3 \subseteq \{t \mid t \in T \wedge Check_t^{A} = true \wedge Check_t^{PA} = true\}$

Zustand 0 bedeutet, daß Semaphor- und Kontrollflußstellen aus dem Vorbereich der Transition, welche mit einer Enablingkante mit der Transition verbunden sind, nicht mit einem Token belegt sind. Im Zustand 1 sind alle Semaphor- und Kontrollflußstellen mit Token belegt, nur Triggerstellen sind noch unmarkiert. Im Zustand 2 befinden sich Transitionen, die Zustand 1 erfüllen und eine Belegung ihrer Triggerstellen im Enabling-Vorbereich besitzen. Es ist dabei irrelevant, welche Markierung (grau oder schwarz) sie tragen.

Diese Definition ist abweichend von der Definition in [Kor91]. Da in diesem Ansatz eine Interleaving Schaltregel realisiert wird, unterbleibt eine weitere Differenzierung des Zustands 2.

5.1.1.2 Konzept der Codegenerierung und statischen Netzanalyse

Wie schon erläutert, wird in diesem Ansatz ein compilatives Verfahren zur Simulation von AADL Petri-Netzen verfolgt. Für jede Transition wird Code einer noch zu definierenden Zielmaschine generiert, in der implizit die Netzstruktur kodiert vorliegt. Dieser generierte Code kann automatisch das Tokenspiel und die Transformation auf dem Datenraum ausführen und für alle Transitionen, die durch das Feuern der Transition aktiviert bzw. deaktiviert werden können, deren Aktiviertheitsstufe neu charakterisieren. Diese Berechnung wird im weiteren als *Administration* bezeichnet; der Code, welcher die Analyse durchführt, wird als *Administrations Code* bezeichnet.

Das Tokenspiel und die Datentransformation können statisch zur Compilezeit des Netzes festgelegt werden. Eine Ausnahme stellt dagegen der Administrations Code dar. Er zerfällt in zwei Arten:

1. Code, der fest eine neue Aktiviertheitsstufe einer Transition zuordnen kann und
2. Code, der abhängig von der vorherigen Aktiviertheitsstufe der Transition und der Markierung des Netzes ausgeführt wird.

Die zu der Kategorie 1 gehörenden Befehle können fest zu einem einzigen komplexen Maschinenbefehl zusammengefaßt werden. Die Befehle der Kategorie 2 müssen mit

einer Kontrollstruktur umgeben werden, welche die Aktiviertheitsstufe neu evaluieren. Für alle Transitionen, die entweder in der Menge $DeactBy_t$ oder $MayBeActBy_t$ liegen, muß nach dem Schalten der Transition t Administrations Code ausgeführt werden. Da er abhängig von der lokalen Netzstruktur um t ist, muß eine Fallunterscheidung anhand der gegebenen Struktur durchgeführt werden. Definition 5.1 beschreibt Basisstrukturen, die sich in wohlgeformten AADL Petri-Netzen ausmachen lassen. Folgende Notationen des Wechsels der Zustandsmodi der einzelnen Transitionen sind zu beachten, mit $T_{i\in\{0,1,2,3\}}$:

- $Ai(t,t') : T_m \rightarrow T_n$
  Durch das Schalten einer Transition t wird der Aktiviertheitszustand einer Transition $t' \in DeactBy_t \cup MayBeActBy_t$ vom Aktiviertheitszustand T_m in den Aktiviertheitszustand T_n transformiert.
- $Ai(t,t') : T_m \rightarrow -$
  Diese Schreibweise deutet an, daß bei wohlgeformten AADL Petri-Netzen es strukturell nicht vorkommen kann, daß sich t' im Aktiviertheitszustand T_m befindet (siehe [Kor91, Kor90]).

In Definition 5.1 werden formal alle in wohlgeformten AADL Petri-Netzen möglichen Netzstrukturen zwischen zwei Transitionen aufgeführt, siehe hierzu auch Abbildung 5.1. Desweiteren wird jeder Netzstruktur der entsprecheden Administrationscode zugeordnet, der angibt, in welchen Aktiviertheitszustand die durch das Schalten beeinflußte Transition wechseln muß.

Definition 5.1 Lokale Netzstrukturattribute.

Für jede Transition $t \in T$, $t' \in DeactBy_t \cup MayBeActBy_t$ und gültigem Netzzustand $(M,(inp,\sigma,out))$ werden folgende Netzstrukturattribute definiert:

- $A1(t,t') = true$ *gdw.* $\{s \in S_{Control} \cup S_{Semaphor} | s \in {}^{\cdot enb}t' \wedge s \in t^{\cdot del} \wedge s \in {}^{\cdot cons}t \wedge s \in {}^{\cdot enb}t\} \neq \emptyset$

  Administrationscode: $A1(t,t') :$ $0 \rightarrow 0$ $1 \rightarrow 1$ $2 \rightarrow 2$ $3 \rightarrow 3$

  Begründung:

  Da $s \in t^{del}$ liegt, verändert das Schalten von t nicht die Markierung von s. Somit verbleibt t' im alten Aktiviertheitszustand.

- $A2(t,t') = true$ *gdw.* $\{s \in S_{Control} \cup S_{Semaphor} | s \in {}^{\cdot enb}t' \wedge s \notin t^{\cdot del} \wedge s \in {}^{\cdot cons}t \wedge s \in {}^{\cdot enb}t\} \neq \emptyset$

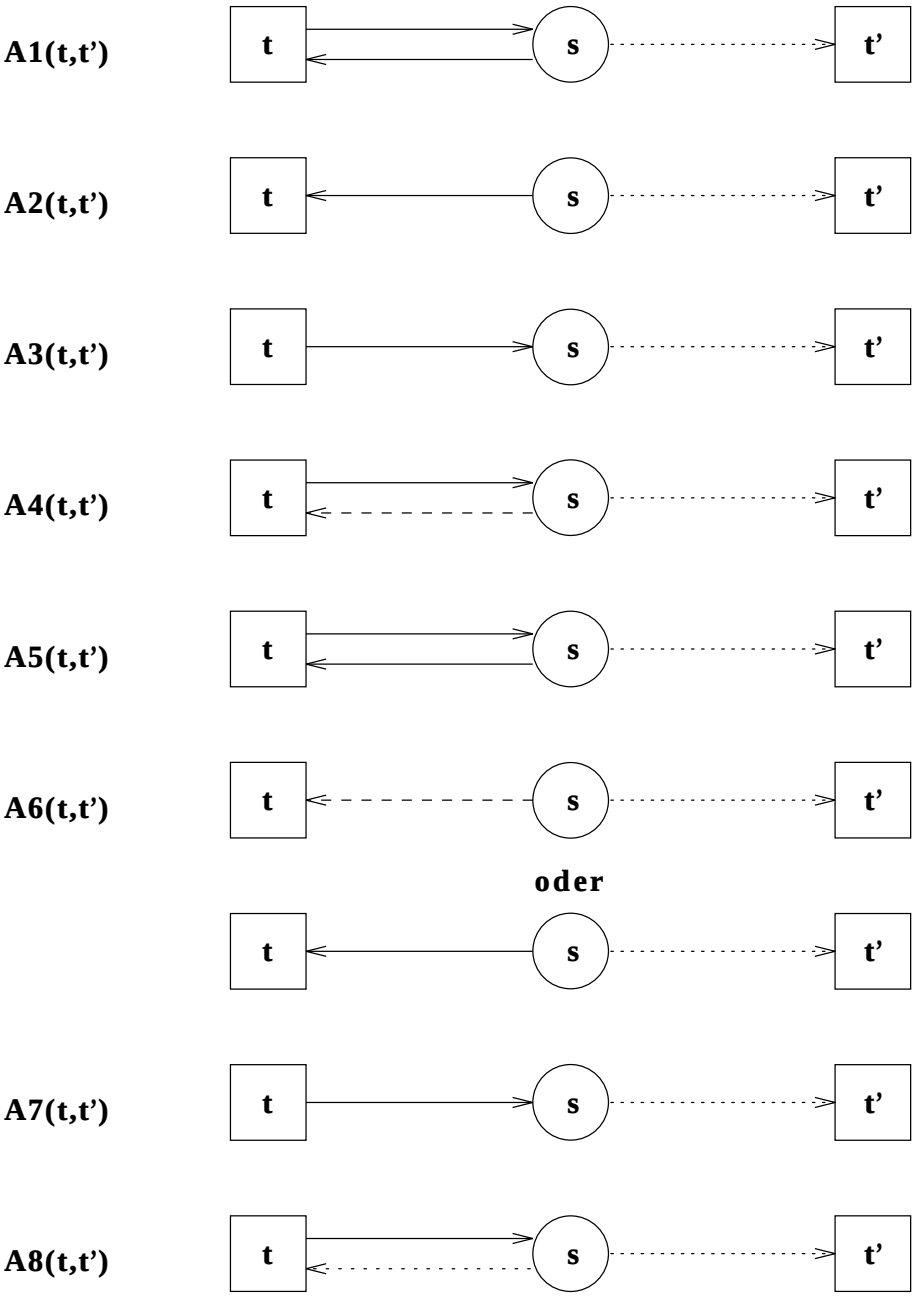


Abbildung 5.1: Netzstrukturattribute

Administrationscode: $A2(t,t'):$ $0 \to 0$ $1 \to 0$ $2 \to 0$ $3 \to 0$

Begründung:

Das Schalten von t zieht das Token von s ab. Da $s \notin t^{\cdot del}$ liegt, wird t' auch nicht wieder aktiviert.

- $A3(t,t') = true$ *gdw.* $\{s \in S_{Control} \cup S_{Semaphor} | s \in {}^{\cdot enb}t' \wedge s \in t^{\cdot del} \wedge s \notin$

${}^{\cdot cons}t \wedge s \notin {}^{\cdot enb}t\} \neq \emptyset$

Administrationscode: $A3(t,t'):$ $0 \rightarrow (if\ Check_t^{PA}\ then\ 2\ else\ 0)$ $1 \rightarrow -$
$2 \rightarrow -$ $3 \rightarrow -$

Begründung:

Das Schalten von t markiert eine Stelle von ${}^{\cdot enb}t'$*. Somit muß die potentielle Aktiviertheit von* t' *überprüft werden.*

- $A4(t,t') = true$ *gdw.* $\{s \in S_{Trigger} | s \in {}^{\cdot enb}t' \wedge s \in t^{\cdot del} \wedge s \in {}^{\cdot cons}t \wedge s \notin {}^{\cdot enb}t\} \neq \emptyset$

  Administrationscode: $A4(t,t'):$ $0 \rightarrow 0$ $1 \rightarrow 2$ $2 \rightarrow 2$ $3 \rightarrow 2$

  Begründung:

  Durch das Schalten von t wird eine Trigger Stelle aus ${}^{enb}t$ *mit einem grauen Token belegt. Somit folgert* $3 \rightarrow 2$ *und* $2 \rightarrow 2$*. Es ist nicht bekannt, ob s mit einem Token belegt war oder nicht. Hieraus folgert* $1 \rightarrow 2$*. Da das Markieren von s nicht die Menge* ${}^{enb}t' \cap (S_{semaphor} \cap S_{Control})$ *beeinflußt, wird durch das Schalten von t auch nicht* $t' \in T_0$ *verändert:* $0 \rightarrow 0$*.*

- $A5(t,t') = true$ *gdw.* $\{s \in S_{Trigger} | s \in {}^{\cdot enb}t' \wedge s \in t^{\cdot del} \wedge s \in {}^{\cdot cons}t \wedge s \in {}^{\cdot enb}t\} \neq \emptyset$

  Administrationscode: $A5(t,t'):$ $0 \rightarrow 0$ $1 \rightarrow 1$ $2 \rightarrow 2$ $3 \rightarrow 2$

  Begründung:

  Durch das Schalten von t folgert, daß s mit einem schwarzem Token belegt war und nun mit einem grauen Token belegt ist. Daraus folgert: $3 \rightarrow 2$ *und* $2 \rightarrow 2$*.*

- $A6(t,t') = true$ *gdw.* $\{s \in S_{Trigger} | s \in {}^{\cdot enb}t' \wedge s \notin t^{\cdot del} \wedge s \in {}^{\cdot cons}t\} \neq \emptyset$

  Administrationscode: $A6(t,t'):$ $0 \rightarrow 0$ $1 \rightarrow 1$ $2 \rightarrow 1$ $3 \rightarrow 1$

  Begründung:

  Analog zu A2 mit $s \in S_{Trigger}$*.*

- $A7(t,t') = true$ *gdw.* $\{s \in S_{Trigger} | s \in {}^{\cdot enb}t' \wedge s \in t^{\cdot del} \wedge s \notin {}^{\cdot cons}t \wedge s \notin {}^{\cdot enb}t\} \neq \emptyset$

  Administrationscode: $A7(t,t'):$ $0 \rightarrow 0$ $1 \rightarrow 2$ $2 \rightarrow 2$ $3 \rightarrow 3$

  Begründung:

  Wenn $t' \in T_3$*, dann ist s mit einem schwarzen Token markiert. In diesem Fall verändert das Schalten von t nicht die Aktiviertheit von* t'*:* $3 \rightarrow 3$*. Für die Fälle* $t' \in T_2$ *oder* $t' \in T_1$ *ist es nicht ableitbar, welche Markierung s trägt. Ist s unmarkiert, dann wird ein graues Token auf die Stelle gelegt:* $2 \rightarrow 2$ *und* $1 \rightarrow 2$*.*

- $A8(t,t') = true$ *gdw.* $\{s \in S_{Trigger} \wedge s \in {}^{\cdot enb}t' \wedge s \in t^{\cdot del} \wedge s \notin {}^{\cdot cons}t \wedge s \in {}^{\cdot enb}t\} \neq \emptyset$

  Administrationscode: $A8(t,t')$: $0 \to 0$ $1 \to 1$ $2 \to 2$ $3 \to 3$

  Begründung:

  Das Schalten von t verändert nicht die Markierung von s. ◦

Mit Hilfe dieser lokalen Attribute läßt sich für jede Transition innerhalb eines wohlgeformten AADL Petri-Netzes die lokale Netzstruktur als Komposition der einzelnen Attribute A1 bis A8 beschreiben. Für die Komposition der Attribute gibt es Attributkombinationsregeln, die sich in Tabelle 5.1 zusammenfassen lassen.[1]. Sie stellt den möglichen Attributkombinationen ihren Administrationscode gegenüber. Die Tabel-

Attributkombinationen									Administrationscode von t' in $Fire_{t'}$			
Fall	A1	A2	A3	A4	A5	A6	A7	A8				
1	*	t	t	*	*	*	*	*	$0 \to 0$	$1 \to -$	$2 \to -$	$3 \to -$
2	*	t	f	*	*	*	*	*	$0 \to 0$	$1 \to 0$	$2 \to 0$	$3 \to 0$
3	*	f	t	*	*	t	*	*	$0 \to Check_{t'}^{PA} : 1$	$1 \to -$	$2 \to -$	$3 \to -$
4	*	f	t	*	*	f	*	*	$0 \to Check_{t'}^{PA} : 2$	$1 \to -$	$2 \to -$	$3 \to -$
5	*	f	f	*	*	t	*	*	$0 \to 0$	$1 \to 1$	$2 \to 1$	$3 \to 1$
6	*	f	f	t	f	f	*	*	$0 \to 0$	$1 \to 2$	$2 \to 2$	$3 \to 2$
	*	f	f	t	t	f	f	*				
	*	f	f	*	t	f	t	*				
7	*	f	f	f	t	f	f	*	$0 \to 0$	$1 \to 1$	$2 \to 2$	$3 \to 2$
8	*	f	f	f	f	f	t	*	$0 \to 0$	$1 \to 2$	$2 \to 2$	$3 \to 3$
9	*	f	f	f	f	f	f	*	$0 \to 0$	$1 \to 1$	$2 \to 2$	$3 \to 3$
$t : AX(t,t') = true$						$f : AX(t,t') = false$			$* : don't\ care$			

Tabelle 5.1: Administrationscode der Netzattributkombinationen

le beschreibt die 256 möglichen Attributkombinationen. Diese Anzahl von Kombinationen läßt sich jedoch auf nur neun zu unterscheidende Fälle zusammenfassen[2].

Da bei der Interleaving Schaltregel in Verbindung mit der Lazy Evaluation der Triggerstellen die Mengen T_2 und T_3 zu einer einzigen Menge T_{23} zusammenfaßbar sind, ergibt sich eine neue Tabelle, die sich an der obigen Aufzählung der einzelnen Fälle orientiert. Nach dem Schalten der Transition ist die Semantik des Administrationscodes für die Transition laut Tabelle 5.2 gegeben.

[1] Wenn $Check_{t'}^{PA}$ sich im aktuellen Netzzustand zu *true* evaluieren läßt, dann wechselt t' den Aktiviertheitszustand von 0 nach 1. Im anderen Fall bleibt der Aktiviertheitszustand von t' gleich 0.

[2] Zur Berechnung dieser Daten wurde eine PROLOG-Implementierung herangezogen, die die möglichen Attributkombination analysierte.

Fall	Aktion	Administrationscode
1		Identität $\Rightarrow$ Kein Code notwendig
2	I	füge t' in T_0 ein
3	II	wenn $Check_{t'}^{PA} = true$ dann füge t' in T_1 ein
4	III	wenn $Check_{t'}^{PA} = true$ dann füge t' in T_{23} ein
5	IV	wenn $t' \in T_{23}$ dann füge t' in T_1 ein
6,8	V	wenn $t' \in T_1$ dann füge t' in T_{23} ein
7,9		Identität $\Rightarrow$ Kein Code notwendig

Tabelle 5.2: Administrationscode

Es lassen sich nach dieser Beschreibung die Aktionen I, IV und V mit Hilfe eines einzigen Maschinenbefehls kodieren, da sie nur eine konstante Operation ausführen. Die Aktionen II und III müssen dagegen mittels einer Kontrollstruktur kodiert werden, da die potentielle Aktiviertheit, abgesehen von eventuellen Invarianten des gegebenen Netzes, nicht statisch ermittelbar ist.

5.1.2 Die abstrakte Netz-Maschine (ANM)

Bei dem Konzept der Compilation ergeben sich verschiedene Ansätze: Bei einem Ansatz, der erstmals von dem *DACAPO III* System realisiert wurde, wird das Netz in C-Code übersetzt. Jeder Transition wird dabei genau eine Funktion zugeordnet, die alle Aktionen durchführen kann. Der Vorteil liegt in der schnellen Codeausführungszeit und in der einfachen Compilationstechnik. Nachteilig wirkt sich bei großen Netzen das Problem des Compilierens und Linkens der verschiedenen C-Funktionen aus. Es ist sehr zeitaufwendig und durch das jeweilige Betriebssystem noch nach oben begrenzt. Auch ist kein Debugging der Netzstruktur möglich, außer es wird mit Source-Code-Debuggern auf dem generierten C-Quellcode agiert.

Der im COMDES System gewählte Ansatz geht davon aus, daß das Netz in eine abstrakte Zwischensprache übersetzt wird. Diese stellt abstrakte Spezialbefehle auf Petri-Netzebene zur Verfügung, die von einer abstrakten Netzmaschine interpretiert werden. Aus diesem Ansatz ergeben sich folgende Vorteile für die Simulation:

1. Durch die Entwicklung einer Maschine, welche mit einem speziell für sie entwickelten Code angesteuert wird, ergibt sich die Möglichkeit der schnellen modularen Compilation von Code als Eingabespezifikation des Netzes. Schon erzeugter Netzcode kann beim Integrieren in ein anderes Netz eingefügt werden, ohne neu

compiliert zu werden, da spezielle Informationen, z.B. Adressen, nur reorganisiert werden brauchen. Desweiteren gibt es nur eine kleine, konstant bleibende Anzahl von Maschinenbefehlen, die schon optimiert compiliert vorliegen. Die Compilation des C-Codes, welcher letztlich das ausführbare Simulationsprogramm erzeugt, beschränkt sich somit nur auf das Initialisieren eines Arrays mit Referenzen auf compilierte C-Funktionen. Dieses Verfahren ist etwa um den Faktor 100 schneller als das Linken der gleichen Anzahl von C-Funktionen.

2. Durch das Maschinenkalkül wird das Aufsetzen eines Debuggers auf Petri-Netzebene unterstützt, da es einem Monitor gleich auf den Ressourcen der Maschine aufsetzen kann, ohne dabei den Lauf der Maschine zu beeinflussen. Auf verschiedenen Abstraktionsebenen kann der Benutzer seine Anfragen spezifizieren und erhält *online* seine gewünschten Informationen. Nebeneffekte, verursacht durch bedingtes Eincompilieren von *Debug-Code* entfallen, was eine bedeutende Reduktion des Codes und somit eine Verringerung von eventuellen Fehlerquellen darstellt. Desweiteren ist es durch diesen Ansatz möglich, ohne Zeitverlust, d.h. ohne eine erneute Compilation, direkt von einem Simulationslauf in einen Debuglauf umzuschwenken und umgekehrt.
3. Die Abstraktion des Petri-Netzkalküls auf das Agieren einer abstrakten Maschine hat den Vorteil der formalen Spezifikation des Ablaufverhaltens bezüglich der Komplexität von Laufzeitverhalten und Speicherplatznutzung.
4. Der auf dem abstrakten Maschinenkonzept beruhende Ansatz hat auch Vorteile bezüglich der Portabilität der Software bzw. der Simulation der Petri-Netze. Sollte das System nicht mehr in einer Uniprozessor UNIX-SUN-Umgebung eingesetzt werden, so ist eine Portierung der Software mit der Adaptierung der Maschine an die neue Zielumgebung abgeschlossen.

Nicht ganz ohne Nachteile kann dieser Ansatz hingenommen werden. Durch diese Realisation wird eine Verschlechterung des Laufzeitverhaltens der Simulation hingenommen. Durch den Code für die abstrakte Netzmaschine muß ein geringer konstanter Faktor an Zeitverlust für Befehlshol- und Dekodierphase jedes Maschinenbefehls als Overhead in Kauf genommen werden, der sich jedoch unter Berücksichtigung der o.g. Vorteile und der immer größer werdenden CPU-Leistungen der Computer vernachlässigen läßt.

5.1.2.1 Aufbau der ANM

Im vorigen Kapitel wurde der Einsatz einer Petri-Netzmaschine zur Simulation von Netzen motiviert und begründet. Im folgenden gilt es, eine Maschine zu entwickeln, welche optimal den Anforderungen der Simulation gerecht wird.

Anforderungen an die abstrakte Netzmaschine

1. Größe der verarbeitbaren Petri-Netze

   Als eine äußerst wichtige Randbedingung ist die Größe der verarbeitbaren Petri-Netze anzusehen. Als behandelbare Fallstudie wurde die AADL-Spezifikation eines Multiprozessorsystems vorgegeben [Dö97]. Hieraus folgerte, daß das System in der Lage sein muß, Netze mit jeweils ca. 50.000 Stellen und Transitionen und einem hohen Vermaschungsgrad im Hauptspeicher zu repräsentieren und effizient zu simulieren. Dabei sollte die Anzahl der Kanten nicht stärker als linear in den Simulationsablauf eingehen. Bei dem realisierten Entwurf kann gezeigt werden, daß im *average-case* die Anzahl der Kanten als Konstante in das Laufzeitverhalten des Simulators eingeht.

2. Modularität des Codes

   Einzelne Module eines gegebenen AADL-Designs, bzw. deren abgeleitete AADL Petri-Netze, müssen separat compilierbar sein - in Hinblick auf die Entwicklung von Netzmodulen. Dies hat insofern Bedeutung, als ein Petri-Netz nur einmal compiliert werden muß und dann in einer Netzdatenbank abrufbar hinterlegt wird. Dabei soll es möglich sein, die Startmarkierung zu variieren, ohne daß dabei eine Recompilation notwendig ist. Desweiteren muß es möglich sein, mehrere Inkarnationen eines Netzes in der Maschine zu verwalten.

3. Debug-Anforderungen:

   (a) Die Maschine muß so konstruiert sein, daß ein *High-Level* Debugging auf mehreren Ebenen möglich ist, ohne daß dies zu einer eklatanten Verringerung des Durchsatzes der Netzmaschine führt. Es muß möglich sein, auf Netzebene und Maschinenebene Breakpoints zu definieren.

   (b) Eine Veränderung der Debugmodi muß zur Laufzeit der Simulation möglich sein, ohne daß eine Recompilation notwendig wird.

   (c) Ein fertig getestetes Modul muß ohne Recompilation in ein Netz ohne Debuginformation transformierbar sein.

Logischer Aufbau der abstrakten Netzmaschine

Die abstrakte Netzmaschine hat einen ähnlichen internen Aufbau wie die klassische *von Neuman* Rechnerarchitektur. Es gibt einen Befehlscode, der in einem steten Befehlszyklus abgearbeitet und interpretiert wird. Ein Halten der Netzmaschine wird jedoch nicht durch einen *Stop*-Befehl initiiert, sondern durch einen besonderen Status eines Betriebsmittels der Maschine. Um das logische Arbeiten der Netzmaschine besser erläutern zu können, sei hier der Aufbau durch Abbildung 5.2[3] veranschaulicht.

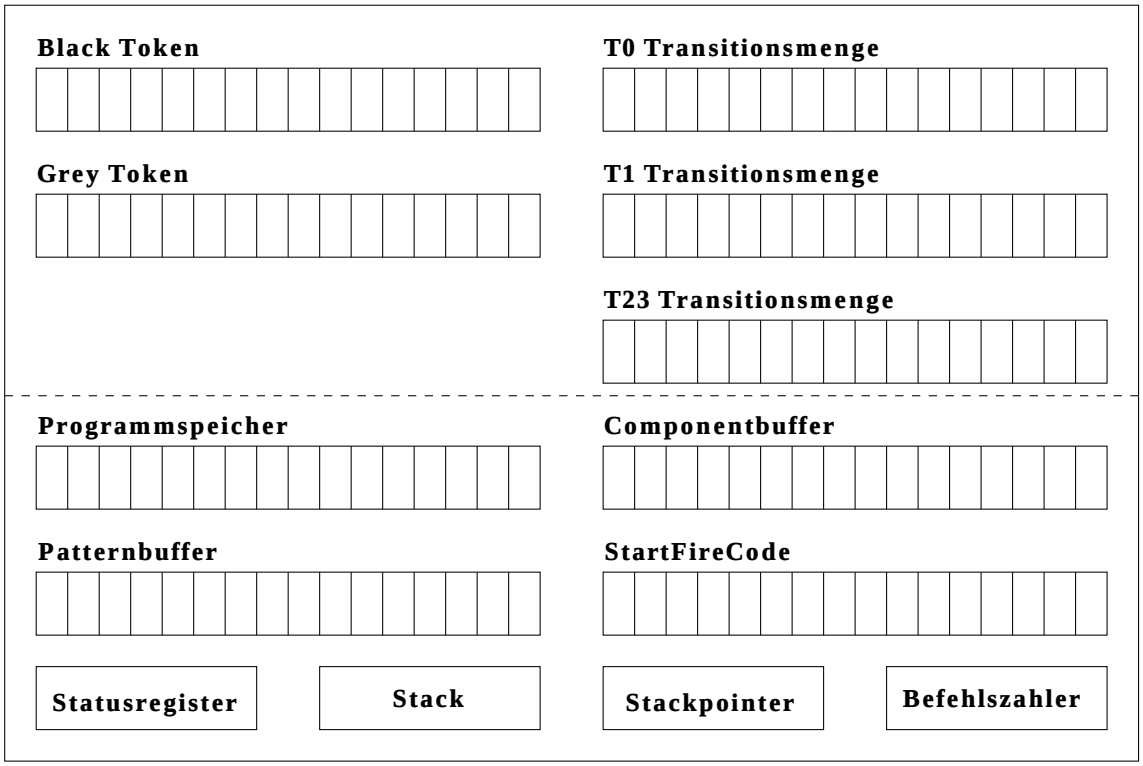


Abbildung 5.2: Aufbau der Netzmaschine

Die Maschine besteht aus folgenden Komponenten, die semantisch in zwei Bereiche eingeteilt werden können:

1. Speicherplätze für die Belegung der Stellen mit schwarzen und grauen Token und Speicherplätze für die Repräsentation der Mengen T_0, T_1 und T_{23}, in welchem der Zustand der Transitionen bezüglich ihrer Aktiviertheit verwaltet werden kann.
2. Für die Realisierung der Funktionalität der Maschinenbefehle werden folgende Strukturen benötigt:

[3] In dieser Abbildung sind Arraystrukturen als Raster dargestellt, um den Gegensatz zu normalen Registern visualisieren zu können. Alle Mengen sind als Bitvektorregister realisiert.

- Programmspeicher für den Code der Netzmaschine
- Befehlszähler
- Stack und Stackpointer für Subroutine Calls
- Component und Pattern Buffer als Register für die Operanden des selektierten Befehls
- Statusregister
- Speicherstrukturen für die Startadressen des Fire-Codes der einzelnen Transitionen

Die ANM kann wie folgt als AADL-Architektur angegeben werden:

```
architecture ANM ( MAXTransitions, MAXPlaces, MAXStatements, TupelSize,
                   StackSize : cardinal)
type
    SizeOfTransitions  = 0..(MAXTransitions mod TupelSize)+1,
    SizeOfPlaces       = 0..(MAXPlaces mod TupelSize)+1,
    BitSize            = 0..TupelSize,
    StackSize          = 0..StackSize,
    ProgramAddress     = 0..MAXStatements,
    AmountTransitions  = 0..MAXTransitions,
    pattern            = array[BitSize] of bit,
    flag               = record
                             ReachedEndOfTransition : boolean;
                             LastBoolOperation      : boolean;
                             ERROR                  : boolean
                         end,
    address            = cardinal,
    command            = { pending }
storage
       Programmspeicher = array [ProgramAddress] of command,
       Patternbuffer    = array [ProgramAddress] of pattern,
       Componentbuffer  = array [ProgramAddress] of cardinal,
       Stack            = array [StackSize] of ProgramAddress,
       Stackpointer     = StackSize,
       Befehlszähler    = ProgramAddress,
       Statusregister   = flag,
       StartFireCode    = array [MAXTransitions] of ProgramAddress,
```

```
        T0, T1, T23      = array [SizeOfTransitions] of pattern,
        GreyToken        = array [SizeOfPlaces] of pattern,
        BlackToken       = array [SizeOfPlaces] of pattern,
end
```

Um den Aktiviertheitszustand einer Transition innerhalb eines Schaltvorganges überprüfen zu können, muß die Information über die Belegung der Stellen in ihrem Vorbereich für die Netzmaschine zugänglich sein. Ob der Vorbereich einer Transition mit Token belegt ist, kann nicht immer statisch ermittelt und somit konstant in den Code eingebettet werden. Diese Information muß zur Laufzeit des Systems jeweils neu berechnet werden und muß schnell zugreifbar sein. Dies ist mit Hilfe von Bitvektor-Operationen gewährleistet.

Da der Code für die Netzmaschine verwaltet werden muß, wird eine Struktur benötigt, in der die Maschinenbefehle abgelegt werden können. Durch den Befehlszähler kann die Maschine auf die aktuelle Instruktion des Programmcodes zugreifen und Operanden in die Register *Component Buffer* und *Pattern Buffer* laden. Alle Mengen sind auf Netzmaschinenebene als Bitvektoren realisiert. Somit können immer 32 Objekte mittels Bitmasken auf eine Eigenschaft hin überprüft bzw. modifiziert werden. Hieraus ergibt sich auch die Semantik der Operanden der Maschinenbefehle:

1. Operand: Index der Array-Komponente
2. Operand: verwendete Bitmaskierung

Die Netzmaschine muß auch in der Lage sein, Subroutine Calls auszuführen. Aus diesem Grunde ist es notwendig, eine stackartige Verwaltung der Rücksprungadressen für den Programmcode zu verwalten. Da der aus der Compilation von AADL Spezifikationen abgeleitet Netzmaschinencode nur eine konstante Anzahl von geschachtelten Subroutine Calls enthält (zur Zeit maximale Tiefe 3, da nur ein Unterprogramm z.Z. vom generierten Code angesprungen wird und von ihm nur ggf. ein weiterer Sprung in die ANM zur Realisierung der synchronen Kommunikation etabliert wird.), kann der Stack als Array realisiert werden mit dem Index *Stackpointer* als aktuell gültige Adresse.

Die Konfliktlösungsstrategie der ANM basiert auf der Interleaving Schaltregel und agiert nur auf der Menge T_{23}, in der die potentiell aktivierten Transitionen verwaltet werden. Sie selektiert nach der folgenden Strategie zufällig eine Transition zum Schalten aus:

1. Selektiere durch Zufall ein 32-Bit Tupel aus der Struktur T_{23}, welche nicht leer ist,
2. selektiere durch Zufall ein gesetztes Bit aus dem Tupel,
3. setze das Befehlsregister auf die Startadresse der zu schaltenden Transition mit Hilfe der Struktur *StartFireCode*.

5.1.2.2 Befehlssatz der ANM

Beschreibung	ANM-Befehl	
Update Netzobjekte	`SetGreyToken`	`UnsetGreyToken`
	`SetBlackToken`	`UnsetBlackToken`
	`SetT0`	`UnsetT0`
	`SetT1`	`UnsetT1`
	`SetT23`	`UnsetT23`
Administrationscode	`MoveT0toT23`	`MoveT23toT0`
	`MoveT0to1`	`Case2`
	`Case5`	`Case68`
Kontrollstrukturen	`SetLastBoolOperationFlag`	`UnsetLastBoolOperationFlag`
	`CheckBlackToken`	`CheckGreyToken`
	`EOT`	`ReturnFromSubroutine`
	`JumpOnTrue`	`JumpOnFalse`
	`JumpToSubroutine`	
Kommunikation mit dem Datenraum	`EvaluateTrigger`	`OperateOnStateSpace`

Tabelle 5.3: Befehlssatz der ANM

Der Befehlssatz hat die Aufgabe, Funktionalitäten bereitzustellen, die das Arbeiten des Simulators effizient realisieren, d.h. sowohl Befehle, die zur Simulation benötigt werden, als auch Befehle für das abstrakte Debuggen der Maschine vorrätig zu halten. Die *nichtmodulare* Netzmaschine besitzt in ihrer derzeitigen Version 27 Maschinenbefehle, die sich nach ihrem Aufgabengebiet (siehe Tabelle 5.3) differenzieren lassen:

Die Befehle für den Update von Netzobjekten und die Administration von Transitionen führen Transaktionen auf mehreren Betriebsmitteln der Maschine durch und sind speziell an die Simulation angepaßt worden.

Die Befehle, welche die Kontrollstrukturen realisieren, basieren auf folgendem Kalkül: Das Ergebnis eines ausgeführten Tests wird mit einem logischen *und* mit

dem Wert des Statusregisters verknüpft. So ist es möglich, mehrere Tests aufeinanderfolgend auszuführen. Dies ist z.B. notwendig, wenn die Stellen aus dem Vorbereich einer Transition nicht vollständig in einem 32-Bit Tupel allokiert werden konnten und somit von zwei Befehlen abgetestet werden müssen. So setzt beispielsweise der Befehl *SetLastBoolOperationFlag* das Statusregister, damit alle Testoperationen einen definierten Startzustand besitzen.

Die Befehle, welche die Kommunikation mit dem Datenraum realisieren, sind mit einer eindeutigen Referenz parametrisiert. Der Datenraum erkennt eigenständig, ob eine boolesche Bedingung abgetestet oder eine Datentransformation angestoßen werden soll. Die Vergabe der eindeutigen Nummer ist bereits vor der Übersetzung der AADL Spezifikation in die Netzebene realisiert.

5.1.3 Compilation von nichtmodularen AADL Petri-Netzen in Code für die ANM

Die Codegenerierung für AADL Petri-Netze wird in drei Schritten durchgeführt:

1. Um ein optimales Laufzeitverhalten der abstrakten Netzmaschine für ein gegebenes Netz gewährleisten zu können, muß eine Reorganisation der Netzobjekte vorgenommen werden, um die irreguläre Graphstruktur zu optimieren. Durch eine Umsortierung der Netzobjekte, basierend auf einer topologischen Analyse des Netzes, soll erreicht werden, daß der gesamte Vorbereich bzw. Nachbereich einer Transition in einem einzigen 32-Bit Tupel allokiert und parallel abgetestet werden kann. Zur Konstruktion dieses Abhängigkeitsgraphen wird die Flußrelation des Petri-Netzes benutzt. Dieser Graph kann jedoch mit Zyklen durchsetzt sein, so daß das normale topologische Sortierverfahren [Wir86] nicht angewendet werden kann. Die Umstrukturierung muß der ANM als bijektive Funktion zur Laufzeit zugänglich sein, um in einer Debugphase die Referenz zu den originalen Netzobjekten wieder herstellen zu können.
2. In der nachfolgenden Phase wird die Generierung des Maschinencodes für alle Transitionen im Petri-Netz vorgenommen, die in weitere vier Phasen zerfällt:
   (a) Erzeugung des $Check_t^{PA}$-Codes Diese Funktion hat die Aufgabe, die Aktiviertheit einer Transition bezüglich der Belegung Ihrer Semaphor- und Kontrollflußstellen im Vorbereich zu überprüfen. Es werden somit Bitmasken generiert, die an der Position ein Bit tragen, deren referenzierte Stellen abgetestet werden müssen. Die Realisation der Funktionalität des

	Codesegment	Beschreibung
(1)	`SetLastBoolOperationFlag`	Setze für den Test das Test-Flag auf *true*
(2)	*for all related 32-Bit Tupel do*	Setze alle Bits in dem Testpattern auf 1,
	`CheckBlackToken` *index pattern*	die eine wie o.g. Stelle referenzieren
	done	
(3)	`ReturnFromSubroutine`	Springe zur aufrufenden Funktion zurück

Tabelle 5.4: Codesequenz $Check_t^{PA}$-Funktion

Codes wird in Tabelle 5.4 erläutert. Der Status der Maschine nach Abarbeitung des Codesegmentes gibt Auskunft über die potentielle Aktiviertheit der Transition t.

(b) Erzeugung des $Check_t^A$-Codes

	Codesegment	Beschreibung
(1)	`SetLastBoolOperationFlag`	Setze für den Test das Test-Flag auf *true*
	for all $s \in {}^{\cdot enb} t$ *do*	
(2)	`CheckBlackToken` *index pattern*	Triggerstelle mit schwarzem Token belegt?
(3)	`JumpOnTrue 10`	True: Springe zum ersten Befehl nach der Schleife
(4)	`SetLastBoolOperationFlag`	Setze für neuen Test das Test-Flag auf true
(5)	`CheckGreyToken` *index pattern*	Triggerstelle mit grauem Token belegt?
(6)	`UnsetGreyToken` *index pattern*	Graues Token weg und ggf. schwarzes Token setzen
(7)	`JumpOnTrue 2`	Wenn kein graues Token $\Rightarrow Check_t^A$ = false
(8)	`ReturnFromSubroutine`	Springe zur aufrufenden Funktion zurück
(9)	`EvaluateTrigger` *refNo*	Evaluiere die Bedingung über den Datenraum
(10)	`JumpOnTrue 2`	false: kein schwarzes Token $\Rightarrow Check_t^A$ = false
(11)	`ReturnFromSubroutine`	Springe zur aufrufenden Funktion zurück
(12)	`SetBlackToken` *index pattern*	Bedingung war true $\Rightarrow$ schwarzes Token für t
	done	
(13)	`ReturnFromSubroutine`	Springe zur aufrufenden Funktion zurück.

Tabelle 5.5: Codesequenz $Check_t^A$-Funktion

Diese Funktion testet die eigentliche Aktivierung der Transition t ab. Aufbauend auf der Voraussetzung, daß die potentielle Aktiviertheit von t gegeben ist, braucht in dieser Funktion nur die Markierung der Triggerstellen im Vorbereich von t überprüft werden. Sollte eine Triggerstelle mit einem grauen Token belegt sein, so ist zu überprüfen, ob die Bedingung über dem Datenraum wahr ist. Dies kann aber nur sequentiell für jede Stelle erfolgen. Sollte sich bei einer Evaluation der Bedingung herausstellen, daß sie nicht erfüllt war, so ist die Transition nicht aktiviert, unabhängig von der Evaluation der Bedingung der anderen Triggerstel-

len. Im erfolgreichen Fall wird das graue Token durch ein schwarzes Token ersetzt. Somit ergibt sich die Befehlsstruktur auf Codeebene laut Tabelle 5.5.

Als Optimierung für den Code der ANM ist eine Verknüpfung der Befehle 2-12 zu einem einzigen Maschinenbefehl möglicherweise schneller in der Ausführung, da Zyklen der ANM für die Befehlsinterpretation eingespart werden. Dieser Ansatz wurde jedoch nicht realisiert, da eine Erweiterung der Operandenzahl von 2 auf 4 für die ANM die Folge wäre, was einen großen Overhead nach sich ziehen würde.

(c) Erzeugung des $Fire_t$-Codes

Die Überprüfung, ob eine Transition nicht nur potentiell, sondern auch wirklich aktiviert ist, d.h. auch alle Bedingungen über den Datenraum der Triggerstellen erfüllt sind, wird von dem $Fire$-Code der Transition überprüft. Nach diesem Test gibt es zwei Varianten der Codeausführung:

Der Test schlug fehl und die Transition muß aus der Menge der potentiell aktivierten Transitionen entfernt werden. Im anderen Fall wird die Transformation der Transition auf dem Datenraum ausgeführt und das Tokenspiel angestoßen. Zum Ende müssen noch alle benachbarten Transitionen bezüglich ihrem ggf. neuen Aktiviertheitszustand administriert werden. Die Struktur des realisierenden Codes ergibt sich nach Tabelle 5.6.

	Codesegment	Beschreibung
(1)	`JumpToSubroutine` $Check_t^A$	Teste, ob alle $s \in^{enb} t$ Schwarz markiert sind
(2)	`JumpOnTrue 3`	True: Springe zur Datenraumtransformation
(3)	`MoveT23toT0` *index pattern*	Deaktiviere t
(4)	`EOT`	Ende der Aktionen von t
(5)	`OperateOnStateSpace` *refNo*	Führe Transaktion auf Datenraum aus
(6)	`UnsetBlackToken` *index pattern*	Entferne Token von ^{cons}t
(7)	`SetBlackToken` *index pattern*	Setze schwarze Token auf t^{del}
(8)	`SetGreyToken` *index pattern*	Setze graue Token auf t^{condi}
(9)	`CaseX` *index pattern* oder	Variabler Administrationscode für
	`MoveXtoY` *index pattern*	alle $t \in MayBeActBy_t \cup DeactBy_t$
(10)	`EOT`	Ende der Aktionen von t

Tabelle 5.6: Codesequenz $Fire_t$-Funktion

3. Erzeugung von Code für die Initialisierung der Betriebsmittel der ANM für das aktuelle Netz

In dieser Phase wird eine Codesequenz erzeugt, in dessen Abarbeitung die Mengen, welche die Zustandsmengen der Transitionen und Stellen repräsentieren, entsprechend der Startmarkierung initialisiert. Dieser Code wird von der Maschine ausgeführt, bevor die Konfliktlösungsstrategie die erste potentiell aktivierte Transition nach dem Prinzip der zufälligen Auswahl selektiert.

5.1.4 Simulation modularer AADL Petri-Netze

Ein AADL Petri-Netz, wie bisher eingeführt, beschreibt das Verhalten einer geschlossenen Hardwarekomponente. Damit das Zusammenspiel vieler Komponenten inklusive des notwendigen Informationsaustausches simuliert werden kann, werden die bisherigen Netze weiter differenziert und um eine *Interfacebeschreibung* angereichert, mit deren Hilfe eine Komposition der Netze semantisch konform zu der abstrakteren AADL-Ebene erfolgen kann. Diese modularen AADL Petri-Netze werden im folgenden *Basisnetze* genannt.

Eine Struktur, mit deren Hilfe eine gewünschte Verknüpfung von Basisnetzen erfolgen kann, wird als Topologie bezeichnet. Sie wird aus AADL Refinementarchitekturen abgeleitet. Eine Simulation eines Designs geht somit immer von einer gegebenen Topologiebeschreibung aus. Von ihr wird ein Topologiebaum abgeleitet der die referenzierten Basisnetze und deren Interfacebeschreibungen beinhaltet, so daß ein einziges logisches AADL Petri-Netz entsteht, welches simuliert werden muß.

Um alle beteiligten Basisnetze zu einem einzigen Netz verkleben (linken) zu können, müssen folgende Operationen durchführbar sein, die auf dem ersten Blick unvereinbar mit einer modularen Compilation der Netze zu sein scheinen:

- Unifikation von Stellen aus verschieden Basisnetzen
- Integration von Kanten zwischen Basisnetzen
- Paralelles Ausführen von Transitionen aus verschieden Basisnetzen, die die synchrone Kommunikation realisieren

In den folgenden Kapiteln werden notwendige Definitionen und Adaptierungen der vorgestellten Compilationsschemata eingeführt. Hieran schließt sich die Beschreibung des AADL-Netz Linkers an, der das Verkleben der Basismodule anhand einer Topologiebeschreibung durchführt.

5.1.4.1 Modulares Simulationsszenario

Um das komplexe Zusammenspiel der Teilkomponenten besser erläutern und motivieren zu können, sei hier der Datenfluß des Systems mit Hilfe einer Graphik (siehe Abbildung 5.3) visualisiert.

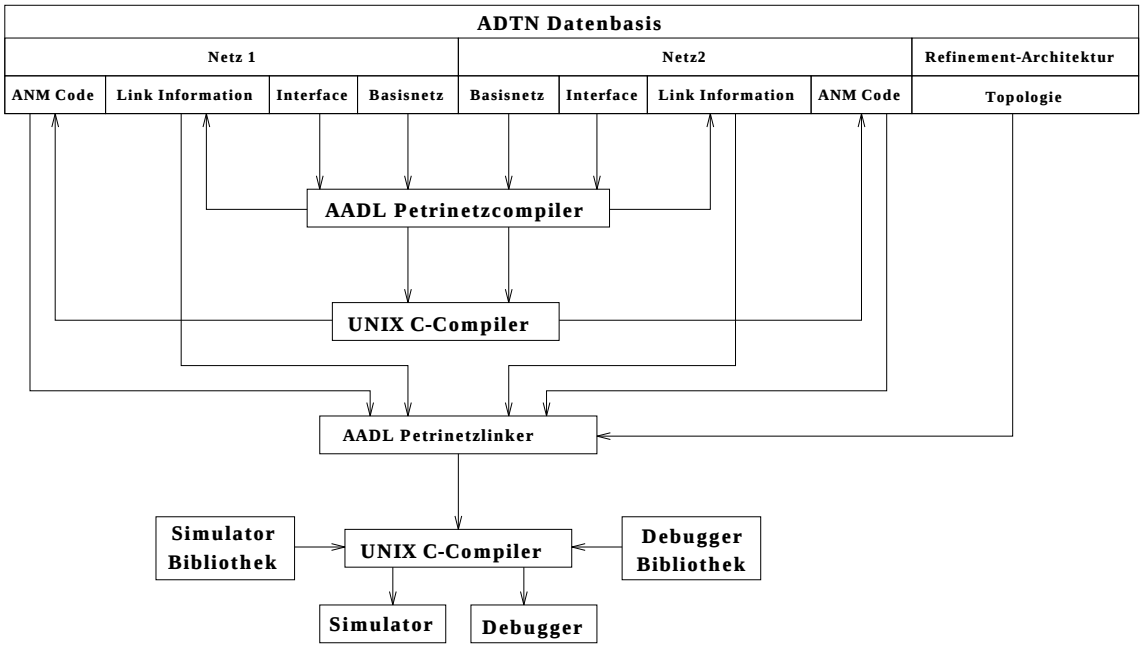


Abbildung 5.3: Simulationsszenario modularer AADL Petri-Netze

Mit Hilfe verschiedener Tools, z.B. dem AADL Strukturediteor, werden AADL Petri-Netze in der Datenbasis abgelegt. Hierbei handelt es sich um Netzobjekte und deren Datenräume, die aus einer Compilation heraus generiert wurden, sowie die zugehörige Interfacebeschreibung. Desweiteren werden auch von Refinementarchitekturen abgeleitet Topologiebeschreibungen in der Datenbasis verwaltet.

Der Netzcompiler kann ein Basisnetz mit seinem Interface einlesen und Code für die ANM erzeugen, der durch einen konventionellen C-Compiler in Objektcode übersetzt wird. Neben den schon bekannten Codesegmenten werden die für die modulare Simulation notwendige *Servicecodemodule* generiert (siehe Kapitel 5.1.4.5).

Der Netzlinker kann nun anhand der Topologiebeschreibung die Teilnetze verkleben. Hierzu werden alle compilierten Codemodule der Basisnetze und ein neu generierter Linkcode zu einem ausführbaren Programm zusammengefügt. Durch Linken einer Simulations- bzw. Debuggerbibliothek wird entweder ein Simulator oder ein Debugger für das AADL Modul generiert, mit dem nun alle Simulations- bzw. Debugläufe

durchgeführt werden.

5.1.4.2 Modulare AADL Petri-Netze

Das Verkleben der AADL Petri-Netze geschieht auf logischer Ebene analog zur Komposition von AADL Modulen [DDG+89]. Hieraus folgt die Notwendigkeit, daß sich das Verkleben der Netzobjekte als eine Unifikation von AADL Variablen des Interfaces, im folgenden als *AADL Objekte* bezeichet, darzustellen hat. Jedem AADL Objekt ist hierbei eine Vielzahl von Netzobjekten zugeordnet [Dö97], die anhand der von einer Refinementarchitektur abgeleiteten Topologiebeschreibung modifiziert werden müssen.

Die synchrone Kommunikation zwischen Teilmodulen eines AADL Designs wird wie bei CSP [Hoa78] durchgeführt und wird den dort postulierten Anforderungen gerecht. Desweiteren wird das Prinzip der Shared Memory Kommunikation unterstützt. Hierbei wird das Konzept der Asynchronen Ports nicht explizit unterstützt, sondern auf den Zugriff auf Shared Memory abgebildet [Kor90].

Da mehrere Inkarnationen eines Basisnetzes in einer Topologiebeschreibung referenziert werden können, resultieren hieraus Konflikte zwischen Namen und Bezeichner der compilierten Teilnetze und der dazugehörigen Datenräume. Diese Konflikte lassen sich durch eine Namenskonvertierung vor dem Verkleben der Teilnetze lösen. Hierbei wird jedem Bezeichner eine eindeutige Inkarnationsnummer zugeordnet:

$$(Inkarnation, Bezeichner)$$

Um eine Verklebung der Netze durchführen zu können, werden noch weitergehende Informationen benötigt, die sich auf AADL Objekte aus der AADL Schnittstellendefinition beziehen. Sie stellt eine Abbildung dar, die jedem AADL Objekt eine Menge von Netzobjekten zuordnet, die bei einem eventuellen Verkleben des zugeordneten AADL Objektes berücksichtigt werden müssen. Es werden folgende drei Mengen von AADL Objekten, welche eindeutig durch ihren Namen und eine Inkarnationsnummer gegeben sind, unterschieden:

1. P_s als Menge der Speicherplatzparameter und asynchronen Ports. Jedem $p_s \in P_s$ werden folgenden Netzobjekte zugeordnet:
   - Eine Menge von zugeordneten Semaphorstellen. Hierbei wird jeder Semaphorstelle als weitere Information noch ihr Level der zu schützenden Critical Section und ihre Inkarnationsnummer zugeordnet.

- Eine Menge der Triggerstellen, die p_s in ihrer Bedingung referenzieren.
- Eine Menge der Transitionen, die eine Transformation auf p_s durchführen.
- Eine Menge der Triggerstellen ohne Bedingung, die von Transitionen, die p_s einen neuen Wert zuweisen, mit einem Token belegt werden müssen.
- Einem Zweitupel (i, j), welches eindeutig einen Speicherplatz für p_s im modular aufgebauten Datenraum zuordnet.

2. P_i als Menge der Synchronen Inports. Jedem $p_i \in P_i$ werden folgende Netzobjekte zugeordnet:
   - Eine Semaphorstelle für die Überwachung des synchronen Kanals.
   - Einer Triggerstelle als Flag, dessen Markierung symbolisiert, ob p_i an einem Kanal angeschlossen ist.
   - Eine Input-Transition, welche die Kommunikation durchführt.
3. P_o als Menge der synchronen Outports. Jedem $p_o \in P_o$ werden folgende Netzobjekte zugeordnet:
   - Eine Semaphorstelle für die Überwachung des Synchronen Kanals.
   - Einer Triggerstelle als Flag, dessen Markierung symbolisiert, ob p_o an einem Kanal angeschlossen ist.
   - Eine Output-Transition, welche die Kommunikation durchführt.
   - Einer Triggerstelle, welche mit dem Pattern annotiert ist, mit dem die Transition matched.

Diese Struktur, welche AADL-Objekte und AADL Petri-Netzobjekte in Relation stellt, wird als *Interface* bezeichnet und kann formal wie folgt angegeben werden:

Definition 5.2 AADL Interface eines Basisnetzes.

Zu einem gegebenen Basisnetz BN ist ein Interface eine Struktur der Form

$$I_{BN} = (AO, ASS)$$

mit

- $AO = P_s \dot\cup P_i \dot\cup P_o$*, die Menge der AADL Objekte (Menge der Speicherplatzparameter, asynchronen Ports, synchronen In- und Outports),*
- $ASS = ass_s \dot\cup ass_i \dot\cup ass_o$ *als Menge von Funktionen, welche Netzobjekte an AADL Objekte binden:*

  $ass_s : P_s \rightarrow (N \times N) \times 2^{(N \times N \times S_{semaphor})} \times 2^{S_{Trigger_{normal}}} \times 2^{T_{normal}} \times 2^{S_{Trigger_{no\_cond}}}$

$ass_i : P_i \rightarrow 2^{(S_{Semaphor} \times S_{Trigger_{no\_cond}} \times T_{input})}$
$ass_o : P_o \rightarrow 2^{(S_{Semaphor} \times S_{trigger_{no\_cond}} \times T_{output} \times S_{trigger_{pm}})}$ ○

Damit die Komposition der Petri-Netze anhand einer Topologiebeschreibung sinnvoll durchgeführt werden kann, muß das Interface gewissen Wohlgeformheitseigenschaften genügen, die in der folgenden Definition dargestellt sind:

Definition 5.3 Wohlgeformtes Interface.

Ein Interface I_{BN} zu einem Basisnetz BN wird als wohlgeformt bezeichnet, gdw.

- *alle in ass_s referenzierten Speicherplätze im Datenraum existieren,*
- *für $((i,j), sem, tri, trans, nvset) \in ass_s(p)$ gilt:*
  - *tri enthält genau die Triggerstellen, die von dem (i,j) zugeordneten Speicherplatz abhängen,*
  - *$trans$ enthält genau die Transitionen, die auf den (i,j) zugeordneten Speicherplatz schreiben,*
  - *sem enthält genau einen Semaphor, der die Synchronisation mit parallel laufenden Architekturen durchführt (Er trägt die Critical Section Nummer 0),*
  - *pro Critical Section existiert für jeden externen Speicherplatz s ein Semaphor, der den Zugriff auf den Speicherplatz innerhalb der Critical Section schützt.*
- *eine Input bzw. eine Output Transition wird maximal an einen synchronen Port gebunden wird,*
- *die Enabling Triggerstelle jeder Input bzw. Output Transition ist nur über eine Enabling Kante mit der Input Transition verbunden ist,*
- *es genau eine Enabling Kante, die von der Pattern Matching Triggerstelle zu der ihr zugeordneten Transition führt, gibt.* ○

Nachdem nun das Interface zu einem AADL Petri-Netz vorgegeben ist können nun modulare AADL Petri-Netze definiert werden:

Definition 5.4 Modulares AADL Petri-Netz.

Ein Modulares AADL Petri-Netz ist eine Struktur, die aus einem Basisnetz und einer Interfacebeschreibung aufgebaut ist, die den o.g. Wohlgeformtheitseigenschaften entsprechen:

$$MN := (BN \times I)$$

5.1.4.3 AADL Refinementarchitekturen

Eine AADL Refinementarchitektur ist eine Struktur, in der die Verbindungsstruktur von AADL Modulen (Subarchitekturen) spezifiziert werden kann. Verbindungen zwischen synchronen und asynchronen Ports von AADL Modulen werden in Strukturen der Topologiebeschreibung transformiert. Desweiteren können innerhalb einer AADL Refinementarchitektur Speicherplätze definiert werden, auf die alle Subkomponenten gemeinsam zugreifen können.

Eine hierarchisch aufgebaute AADL Refinementarchitektur wird bei ihrer Übersetzung in eine Topologiebeschreibung in einen Operatorbaum übersetzt, welcher die Netze der Subarchitekturen referenziert. Die Operatoren selber greifen über AADL Interface Objekte auf die Komponenten der Basisnetze zu. Da die Bezeichner aller Netzobjekte neben ihrem Namen auch die Inkarnationsnummer des Netzes enthalten, lassen sich alle Namenskonflikte auflösen. Der Aufbau des Operatorbaumes wird mit Hilfe von drei Operatoren vorgenommen:

1. rs-Operator:
   Er löst Namenskonflikte auf, welche z.B. durch mehrere Inkarnationen eines Basisnetzes entstehen können.
2. *con*-Operator:
   Er verbindet zwei Netze.
3. $glueNet$-Operator:
   Er verbindet Komponenten eines Netzes zu einem einzigen Netz und erzeugt ein neues Interface.

Um den Aufbau des Operatorbaumes besser nachvollziehen zu können, sei hier ein kleines Beispiel angeben:

$$glueNet(con(rs(1, MC68040), rs(2, RAM1)), topo[\ldots])$$

In den folgenden informellen Definitionen der Netzoperatoren sei $ID = (Cardinal \times String)$ die Menge der eindeutigen Bezeichner für P_s, P_i und P_o aus Definition 5.2. Die detaillierte formale Definition der Operatoren ist in [Kor90] gegeben.

Definition 5.5 rs-Operator.

Der Operator rs setzt die Subarchitekturnummer auf einen neuen Wert. Diese Nummer ist die erste Komponente eines AADL Bezeichners.

Gegeben sei ein Basisnetz $N' = (BN', I')$ und eine Zahl i. Der rs-Operator setzt die Subarchitekturnummer von N' auf i:

$$N = (BN, I) := rs(i, N'),$$

mit

1. *Stellen, Transitionen und Kanten bleiben durch die Anwendung des rs-Operators unberührt.*
2. *Alle im Interface referenzierten Netzobjekte werden in der ersten Komponente des ID auf den Wert i gesetzt.* ◦

Definition 5.6 con-Operator.

Gegeben seien zwei Netze $N' = (BN', I')$ und $N'' = (BN'', I'')$. Der con-Operator ist auf N' und N'' anwendbar, wenn die Interfaceobjekte der beiden Netze unterschiedliche Inkarnationsnummern haben:

Sei con auf N' und N'' anwendbar. Er verbindet die Netze dann wie folgt:

$$N = (BN, I) := con(N', N''),$$

mit

1. *Die Mengen der Stellen und Transitionen von N' und N'' werden disjunkt vereinigt. Die Netzobjekte aus N'' erfahren dabei eine Erhöhung ihrer Netznummern um die Anzahl der Teilnetze, die in N' referenziert werden.*
2. *Alle Kanten werden an die Renummerierung der Netzobjekte in N'' angepaßt.*
3. *Alle von N'' exportierten AADL Objekte werden ebenfalls an die Renummerierung angepaßt.*
4. *Der Datenraum wird vereinigt und an die Renummerierung bezüglich N'' angepaßt.* ◦

Definition 5.7 glueNet-Operator.

Der Operator glueNet wird auf ein Netz N' angewendet und definiert ein neues Netz N, welches entsprechend topo verklebt ist:

$$N = (BN, I) := glueNet(N', topo),$$

wobei $topo \in TopoGlueNet$ ist, mit TopoGlueNet als die kleinste Menge, für die gilt:

1. *Es werden die Semaphoren für den Zugriff auf externe Ports und externe Speicherplätze verklebt. Diese Funktionalität wird von dem glueSem-Operator realisiert*[4].
2. *Es werden Delivering- und Consuming-Kanten zwischen allen Transitionen, die auf gemeinsamen Ports und Speicherplätze schreiben, und Triggerstellen, die diese Objekte referenzieren, gezogen. Das Einfügen redundanter Kanten ist erlaubt (delConsArc-Operator).*
3. *Es werden Delivering Kanten zwischen speziellen Transitionen und Stellen eingefügt, die im Interface von Storage AADL Objekten referenziert werden; bis auf die Kanten und zwischen Stellen und Transitionen, die zum selben externen Speicherplatz gehören(delArc-Operator).*
4. *Es werden die Enablingstellen der an einen Kanal angeschlossenen Input- und Output-Transitionen gelöscht (delTri-Operator).*
5. *Es werden die Input- und Output-Transitionen eines Kommunikationskanals paarweise verklebt und die passenden Triggerstellen erzeugt, indem die Pattern der Input-Transitionen an die Pattern Matching Stellen angeheftet werden (syncLink-Operator).*
6. *Für das neu generierte Netz wird ein neues AADL Interface mittels der Operatoren $expP_s$, $expP_i$ und $expP_o$ erzeugt.* ∘

Damit das Verkleben durch den *glueNet*-Operator korrekt durchgeführt werden kann, muß er folgende Wohlgeformheitseigenschaften genügen:

Definition 5.8 Wohlstrukturierter glueNet-Operator.

Der Operator glueNet heißt wohlstrukturiert, wenn er folgende Bedingungen erfüllt:

1. *Ein Semaphor darf maximal einmal verklebt werden*
2. *Ein asynchroner Port bzw. externer Speicherplatz darf nur in einer der beiden Operatoren delConsArc bzw. delArc referenziert werden.*
3. *Ein synchroner Port darf maximal an einen Kanal angeschlossen bzw. exportiert werden.*
4. *Wird eine Menge von Speicherplätzen exportiert, so müssen die entsprechenden Semaphoren der Critical Sections verklebt werden und den selben Speicherplatz referenzieren.*

[4] Alle Operatoren der Topologiebeschreibung (glueSem, delTri, delConsArc, delArc, syncLink) werden auf $InterfaceObject \in 2^{ID} \neq \emptyset$ angewandt.

5. Alle Objekte, die exportiert werden, haben unterschiedliche Bezeichner. ◦

Im folgenden wird immer davon ausgegangen, daß alle Strukturen und Operatoren ihren Wohlgeformheitseigenschaften entsprechen.

5.1.4.4 Adaption der ANM

Die abstrakte Netzmaschine benötigt nur wenige Erweiterungen, um modulare AADL Petri-Netze simulieren zu können. Hierbei handelt es sich um Erweiterungen der Betriebsmittel und des Maschinenbefehlssatzes.

Erweiterungen der Betriebsmittel

Die Erweiterungen folgen ausschließlich aus der Realisierung der synchronen Kommunikation zwischen AADL Modulen auf Petri-Netzebene. An einen synchronen Port wird jeweils zur Linkzeit des Systems eine Menge von Input- und Output-Transitionen gebunden. Da Input-Transitionen in der Semantik der modularen AADL Petri-Netze nicht in der Lage sind, eine Kommunikation zu initiieren, d.h. eine kommunikationsbereite Output-Transition auszuwählen, muß eine Differenzierung eingeführt werden. Um die Input-Transitionen getrennt von den restlichen Transitionen verwalten zu können, wurde für sie eine getrennte isomorphe Struktur geschaffen, in der sie nicht für die Konfliktlösungsstrategie zugreifbar sind.

Desweiteren benötigt die ANM eine Struktur, in der sich die Startadressen der $Check_t^A$ Funktionen der Input-Transitionen abspeichern lassen. Die $Check_t^A$ Funktion einer Input-Transition wird aus einer anderen Funktion heraus aufgerufen. Sie hat die Aufgabe, aus der Menge der möglicherweise aktivierten Input-Transitionen, welche dem gleichen Kommunikationskanal zugeordnet sind, zufällig eine aktive zu selektieren. Dabei fallen administrative Aufgaben an, die zusätzlich von dieser Funktion ausgeführt werden müssen. Damit die ANM sich die Codestartadresse einer kommunikationsbereiten Input-Transition merken kann, welche von einer temporären Sub-ANM berechnet wurde, wird eine weitere Struktur benötigt.

Die Berechnung einer kommunikationsbereiten Partnertransition basiert auf einer tabellenartigen Struktur, in der jedem Kommunikationskanal seine angeschlossenen Input-Transitionen zugeordnet sind.

Formal lassen sich diese Erweiterungen wieder in AADL spezifizieren, wobei die Konstanten *MAXChannels* und *MAXChannelTransitions* als Designparameter der

Beschreibung	ANM - Befehl	
Update Netzobjekte	`SetInputT0`	`UnsetInputT0`
	`SetInputT1`	`UnsetInputT1`
	`SetInputT23`	`UnsetInputT23`
Administrationscode	`MoveInputT0ToInputT23`	`MoveInputT23ToInputT1`
	`MoveInputT0ToInputT1`	`InputCase2`
	`InputCase5`	`InputCase68`
Kontrollstrukturen	`JumpDynamicToSubroutine`	`tInTo`
	`tInInputT23`	

Tabelle 5.7: Erweiterung des Befehlssatzes der ANM

Architektur anzusehen sind:

```
type
    CommunicationChannels = 1..MAXChannels,
    TransitionPerChannel  = 1..MAXChannelTransitions
storage
    StartAdrPartner = ProgramAddress,
    Channels        = array [CommunicationChannels] of
                        array [TransitionPerChannel] of AmountTransitions,
    StartATCode     = array [AmountTransitions] of ProgramAddress,
    IsInputOK       = array [AmountTransitions] of ProgramAddress,
    InputT0,InputT1 = array [SizeOfTransitions] of pattern,
    InputT23        = array [SizeOfTransitions] of pattern
```

Erweiterungen des Maschinenbefehlsumfanges

Die Erweiterungen ergeben sich aus der Verwaltung der neuen Betriebsmittel der ANM und der Notwendigkeit, eine aktive Input-Transition für eine selektierte Output-Transition zu bestimmen:

Als Ausnahme von dieser kanonischen Struktur ist der `SearchPartner` Befehl zur Bestimmung eines Kommunikationspartners bezüglich eines gegebenen Kanals anzusehen. Er sucht aus der Menge der an einen Kommunikationskanal angeschlossenen Input Transitionen zufällig eine Transition aus, die aktiv ist und deren Pattern für die gerade aktive Output-Transition matched.

Die Realisierung dieser Funktionalität kann auf Grund der sehr komplexen Berechnungsschritte nicht auf ANM Ebene mittels eines einzigen Maschinenbefehls durchgeführt werden. Es wird deshalb temporär eine Sub-ANM inkarniert, welche die notwendigen Berechnungsschritte auf Basis eines „System-Interupts“ berechnet.

5.1.4.5 Modulare Codegenerierung

Durch die Möglichkeit, mehrere modulare AADL Petri-Netze gemeinsam als ein System simulieren zu können, ergibt sich die Notwendigkeit, diese Funktionalität effektiv im Code der Transition zu verankern. Dies wird mittels einer Differenzierung der Netzobjekte in zwei disjunkte Teilklassen erreicht:

1. Objekte, die durch eine Interfacebeschreibung referenziert werden und
2. Objekte, die zum Kern des Basisnetzes gehören, d.h. keine Referenz auf AADL Interface Objekte haben.

Für die im Interface referenzierten Netzobjekte muß die Möglichkeit geschaffen werden, daß auf sie auch nach der Compilation noch modifizierend zugegriffen werden kann, da ihre Netzstruktur durch das Linken verändert werden kann. Folgende Modifikationen sind möglich:

1. nachträgliches Einfügen von Kanten
2. nachträgliches Löschen von Stellen und deren assoziierten Kanten
3. Unifizieren von Stellen
4. Verkleben von Transitionen

Für alle AADL Objekte, die im Interface referenziert werden, ergeben sich jedoch aus der Wohlgeformtheit der Basisnetze nur eingeschränkte, in ihrer Struktur immer wiederkehrende, Update-Anforderungen durch das Verkleben der Basisnetze. Alle o.g. Update-Funktionalitäten sind durch zusätzliche Maschinenbefehlssequenzen, den so genannten Servicefunktionen, realisiert, die durch Subroutine Calls ausgeführt werden können. Für die Compilation der Petri-Netze in Code für die ANM ergeben sich zur modularen Codegenerierung folgende Änderungen:

1. Die Codegenerierung ist von der Interfacebeschreibung abhängig
2. Parametrisierung der Basisnetze und Servicefunktionen:
   - Adressen für reallokierbare Stellen

- Startadresse des Maschinencodes
- Startadressen der Mengenkodierungen von T_0, T_1, T_{23}
- Startadressen der Markierungen der Stellen
- Startadressen einzufügender Codestücke
- Adressen unifizierter Stellen
- Referenzen auf den assoziierten Datenraum

3. Bereitstellung von Servicecodemodulen für den Update von Interface-Objekten
4. Implementierung eines Kommunikationsprotokolls für die synchrone Kommunikation

Eine wichtige Anforderung für die Modularität des erzeugten Maschinencodes ist die Reallokationsfähigkeit und Parametrisierbarkeit des erzeugten Codes. Dies wird mit Hilfe von *Codemodulen* verwirklicht. Alle logisch zusammenhängenden Daten, z.B. compilierte Basisnetze und Servicefunktionen, werden von Funktionen gekapselt, welche minimal parametrisiert sind. Es ist somit keine Information über ihren Kern zur Linkzeit notwendig und die Funktionen können als Black Boxes gleich vom Linker benutzt werden. Als Parameter werden nur die o.g. Parameter benötigt.

Synchrones Kommunikationsprotokoll

Die Realisierung der synchronen Kommunikation über ein Protokoll ist für das Verständnis des Firecodes von Input- und Output-Transitionen notwendig und wird hier nur informell eingeführt (siehe [Kor91]).

An einem Kommunikationskanal sind synchrone Input- und Output-Ports angeschlossen, welche die baumartige Verbindungsstruktur beschreiben. Im Gegensatz zur Semantikdefinition, die die Input- und Output-Transitionen paarweise verklebt, wird die Kommunikation durch ein Protokoll realisiert. Trotz der Input- und Output-Guards ist dieses Protokoll wesentlich einfacher als ein CSP Protokoll [Hoa78], da kein Protokoll für parallel laufende Prozesse realisiert werden muß. Die Auswahlstrategie stellt einen Scheduler dar, in dem das Protokoll integriert ist und der den parallelen Ablauf simuliert. Ein weiterer Grund für dieses Realisierung liegt in der separaten Compilation der Basisnetze, da erst zur Linkzeit eine logische Verknüpfung der Input- und Output-Transitionen durchgeführt werden braucht und somit das vorcompilierte Netz keine Informationen für die spätere topologische Einbettung benötigt.

Das Protokoll wird wie folgt realisiert: Jede Input-Transition, die in der Menge T_{23} liegt, wartet darauf, daß sie von einer Output-Transition, die an dem selben Kanal angeschlossen ist, ausgewählt wird. Wird eine Output-Transition von der Auswahlstrategie ausgewählt, sucht sie unter der Menge der Input-Transitionen, die am selben Kanal angeschlossen sind und in T_{23} liegen, zufällig eine aus, mit deren Pattern das Output-Pattern matcht.

1. Realisierung des Protokolls aus Sicht der Input-Transitionen
   Wechselt eine Input-Transition ihren Aktiviertheitszustand nach T_{23}, werden die Pattern Matching Stellen aller Output-Transitionen dieses Kanals mit einem grauen Token belegt und ihr Aktiviertheitszustand mittels des A4 Admincodes neu bestimmt.
2. Realisierung des Protokolls aus Sicht der Input-Transitionen
   Wird eine Output-Transition von der Auswahlstrategie ausgewählt, so wird mit der $Check_t^A$ ihre Aktiviertheit überprüft. Ist dies der Fall, wird zufällig aus der Menge der aktiven, am gleichen Kanal angeschlossenen, Input-Transitionen eine ausgewählt, deren Pattern matcht und beide Transitionen werden gefeuert. Existiert keine solche Input-Transition, so wird das Token der Pattern Matching Stelle entfernt und die Transition deaktiviert.

Tabelle 5.8 zeigt den Maschinencode, der von einer Output-Transition für eine selektierte Input-Transition angesprungen wird und u.a. das Pattern Matching evaluiert. Dieser Code wird von der Sub-ANM, die im Maschinenbefehl *Search Partner* ak-

Codesegment			Beschreibung
`tInInputT23`	*index*	*pattern*	Ist t potentiell aktiv
`JumpOnTrue`	*2*		Ja : weiter
`EOT`			Nein: Springe zur aufrufenden Funktion zurück
`JumpToSubroutine`	$Check_t^A$		Ist t aktiviert
`JumpOnTrue`	*3*		Ja: weiter
`MoveInputT23ToInputT1`	*index*	*pattern*	Nein: Deaktiviere t
`EOT`			Springe zur aufrufenden Funktion zurück
`EvaluateTrigger`	*netindex*	*idRef*	Führe das Pattern Matching durch
`EOT`			Springe zur aufrufenden Funktion zurück

Tabelle 5.8: Codesequenz: Test von Input-Transition als Kommunikationspartner

tiviert wird, für jede selektierte Input-Transition ausgeführt, bis ggf. eine aktive Input-Transition gefunden wurde.

Servicefunktionen

Die Servicecodefunktionen dienen dazu, Funktionalitäten, welche für das Linken der compilierten Basisnetze benötigt werden, bereitzustellen. Für jedes AADL Objekt, welches in einer Interfacebeschreibung referenziert wird, werden folgenden Funktionen generiert (siehe Definition 5.7), die als Codemodule realisiert werden :

1. Die *delConsArc* Relation hat die Aufgabe, Triggerstellen eines AADL-Objektes zu demarkieren und mit grauen Token zu belegen, und alle abhängigen Transitionen auf deren Aktiviertheit hin mit dem Case68-Admincode zu überprüfen:

Servicecodefunktionsname	*NameDesBasisnetzes_NameDesAADLObjektes*_DC()
Parameter	Codeindex : Startadresse des Maschinencodes
	OffPlace : Offset der Stellenmenge des Basisnetzes
	OffTrans : Offset der Transitionsmenge des Basisnetzes
ANM-Befehle	`UnsetToken` *OffPlace pattern*
	`SetGreyToken` *OffPlace pattern*
	`Case68` *OffTrans pattern*

2. Die *delArc* Relation hat die Aufgabe, Triggerstellen eines AADL-Objektes mit grauen Token zu belegen und alle abhängigen Transitionen auf deren Aktiviertheit hin mit dem A4-Admincode zu überprüfen:

Modulname	*NameDesBasisnetzes_NameDesAADLObjektes*_DA()
Parameter	Codeindex : Startadresse des Maschinencodes
	OffPlace : Offset der Stellenmenge des Basisnetzes
	OffTrans : Offset der Transitionsmenge des Basisnetzes
ANM-Befehle	`SetGreyToken` *OffPlace pattern*
	`A4` *OffTrans pattern*

3. Die *glueSem* Relation identifiziert nur Semaphorstellen, stellt aber selbst keinen Servicecode zur Verfügung. Vielmehr müssen zwei Funktionen für jeden Semaphor zur Verfügung gestellt werden, die

   (a) einen unifizierten Semaphor mit einem Token belegen und benachbarte Transitionen auf Aktiviertheit hin überprüfen oder

   (b) von einem unifizierten Semaphor das Token entfernen und benachbarte Transitionen auf Deaktivierung hin überprüfen.

   Für die ANM bedeutet dies, das nach dem Tokenspiel der A2 bzw. A3 Admincode auszuführen ist, der auch namensgebend für die Servicecodemodule

ist:

Modulname	*NameDesBasisnetzes_SemaphorId*_A2()
Parameter	Codeindex : Startadresse des Maschinencodes
	OffTrans : Offset der Transitionsmenge des Basisnetzes
	OffSema : Offset des unifizierten Semaphors
	SemaPattern : Pattern des unifizierten Semaphors
ANM-Befehle	`UnsetToken` *OffSema SemaPattern*
	`A2` *OffTrans pattern*

Modulname	*NameDesBasisnetzes_SemaphorId*_A3()
Parameter	Codeindex : Startadresse des Maschinencodes
	OffTrans : Offset der Transitionsmenge des Basisnetzes
	OffSema : Offset des unifizierten Semaphors
	SemaPattern : Pattern des unifizierten Semaphors
	AmountTrans : Anzahl der Transitionen der ANM
ANM-Befehle	`SetBlackToken` *OffSema SemaPattern*
	`tInT0` *OffTrans pattern*
	`JumpOnFalse` *4*
	`JumpToSubroutine` $Check_t^{PA}(AmountTrans)$
	`JumpOnFalse` *2*
	`MoveT0toT23` *OffTrans pattern*

4. Die *syncLink* Relation hat die Aufgabe, Pattern Matching Stellen eines synchronen AADL-Outputobjektes mit einem grauen Token zu belegen und alle abhängigen Transitionen auf deren Aktiviertheit hin mit dem A4-Admincode zu überprüfen:

Modulname	*NameDesBasisnetzes_NameDesAADLObjektes*_OpmGA4()
Parameter	Codeindex : Startadresse des Maschinencodes
	OffPlace : Offset der Stellenmenge des Basisnetzes
	OffTrans : Offset der Transitionsmenge des Basisnetzes
ANM-Befehle	`SetGreyToken` *OffPlace pattern*
	`A4` *OffTrans pattern*

Für den Linker werden die Informationen über die Art und Größe der zugreifbaren Servicecodemodule für das compilierte Basisnetz noch in einer komprimierten Form als weiteren Input für den Linkprozeß erzeugt.

Modularer Firecode

Der Firecode der Transition für modulare AADL Petri-Netze besitzt alle Funktionalitäten, die schon im Kapitel 5.1.3 erläutert wurden. Jedoch ergeben sich folgende Erweiterungen der Codesequenzen in Abhängigkeit vom Typ der Transitionen:

1. Normale Transition:
   Nach dem Tokenspiel kann eine Input-Transition aktiviert worden sein, so daß für alle Output-Transitionen, die an dem gleichen Kanal angeschlossen wurden, der Aktiviertheit mit dem *OpmGA4* Code überprüft werden muß. Desweiteren werden, wenn nötig, die *DC,DA*, *A2* und *A3* Codesegmente mit Subroutine Calls angesprungen.
2. Tau Transitionen:
   Bei ihnen entfällt im Gegensatz zu normalen Transitionen die Operation auf dem Datenraum.
3. Output Transitionen:
   Nachdem eine Output-Transition die eigene Aktiviertheit mit dem $Check_t^A$ Code festgestellt hat, wird weiterhin versucht, mittels des $SearchPartner$ Subroutine Calls eine passende aktivierte Input-Transition gesucht. Ist dies nicht der Fall, deaktiviert sich die Transition. Konnte eine Partnertransition gefunden werden, so wird ihr Maschinencode nach der Ausführung des Admincodes der Output-Transition angesprungen und ausgeführt.
4. Input Transition:
   Der Code der Input-Transition unterscheidet sich nur in zwei Punkten von dem einer normalen Transition. Zum einen ist die Aktiviertheit der Transition schon abgetestet und gegeben; somit entfällt der Testaufruf von $Check_t^A$. Zum anderen wird das Ende der Codesequenz nicht mit einem `EOT`-Befehl beendet, sondern durch einen *ReturnFromSubroutine*-Befehl, da er ja auch von einen Subroutine Call aus angesteuert wurde.

5.1.4.6 Linken der modularen AADL Petri-Netze

Der Linker generiert anhand einer Topologiebeschreibung, die von einer AADL Refinementarchitektur abgeleitet wurde, ein lauffähiges Simulationsprogramm als Komposition der referenzierten Basisnetze. Das Linken der Module findet in drei Phasen statt:

Zuerst wird inkrementell anhand des Topologiebaumes eine bottom-up Unifikation

der AADL-Objekten auf Basis der Netzoperatoren durchgeführt. Am Ende dieses Durchganges ist die baumartige Ableitungsstruktur mit ihrem Blockkonzept in eine planare Struktur überführt worden.

Hiernach schließt sich die Unifikation der Netzobjekte an, die sich auf die Komposition der Servicecodefunktionen abstützt.

In der letzten Phase werden alle Referenzen der Sprungadressen aufgelöst, um keine indirekten Sprünge mehr durchführen zu müssen. Auch wird zusätzlicher Link- und Initialisierungscode erzeugt, der dann zusammen mit den Kernbibliotheken zu einem ausführbaren Programm zusammengefügt werden kann.

Die einzelnen Phasen des Linkvorganges sind wie folgt aufgebaut:

Pass 1: Verkleben von AADL-Objekten

Aus Definition 5.7 des *glueNet*-Operators geht hervor, daß alle unterhalb dieses Operators im Topologiebaum vorkommenden Netzobjekte verbunden werden könne. Durch einen postorder-Durchlauf wird der Baum abgearbeitet, wobei für jeden Funktor des glueNet Operators separate Kopien der Mengen von AADL Objekten verwaltet werden, um auf eine ggf. spätere semantische Abänderung der Operatoren nur lokal reagieren zu müssen. Am Ende liegt eine flache Struktur der AADL Objekte als Vorlage zum Linken auf Netz-Ebene vor.

Pass 2: Verkleben von Netzobjekten

In dieser zweiten Phase geschieht eine Differenzierung des Unifikationsverfahrens anhand der verschieden zu vereinigenden Netzobjekte. Drei Klassen von Veränderungsstrukturen lassen sich ausmachen:

1. Veränderung der Kantenstruktur durch die *delConsArc*-Relation:

   Jede Transition, die eine Transformation auf einem AADL Objekt ausführt, das mit dem delConsArc-Operator verklebt wird, muß alle laut Interfacebeschreibung referenzierten Triggerstellen des Gesamtnetzes administrieren. Dies geschieht durch eine Sequenz von Maschinenbefehlen, die durch die Verbindung aller Servicecodefunktionen des unifizierten AADL Objektes erzeugt und mit einem `ReturnFromSubroutine` Befehl abgeschlossen wird. Die relativen Codestartadressen werden durch Addition der Längen der einzelnen Servicecodefunktionen berechnet und zur finalen Adreßberechnung im Pass 3 kommuliert.

Veränderung der Kantenstruktur durch die *delArc*-Relation:

Dieser Operator hat eine ähnliche Semantik wie der *delConsArc*-Operator mit der einzigen Ausnahme, das nur Transitionen in fremden Netzen administriert werden dürfen. Somit werden für jedes Basisnetz eigene Codesequenzen generiert, die die lokale Sequenz der Maschinenbefehle des Servicecodemoduls nicht beinhalten.

2. Unifikation von Stellen durch die *glueSem*-Relation:

   Werden AADL Objekte bezüglich der *glueSem*-Relation verbunden, so müssen alle Semaphorstellen auf allen Critical Section Ebenen unifiziert werden. Dabei wird zwischen Semaphorstellen oberster Ebene (der Außenwelt sichbar) und denen tieferer Ebene unterschieden. Alle Stellen der obersten Schicht werden netzübergreifend zu einer einzigen Stelle unifiziert, wobei tiefergeschachtelte Stellen unterer Critical Section Ebenen nur lokal unifiziert werden müssen. Mit Hilfe einer Matrix deren Spalten Basisnetze und deren Zeilen Critical Sections für ein zu verklebendes AADL Objekt referenzieren, kann die Anzahl der real benötigten Semaphorstellen berechnet werden. Durch diese Unifikation kann aber der notwendige *A2*- bzw. *A3*-Admincode hinfällig werden. Durch verwalten von read/write-Listen kann dann noch der Aufruf dieser Codesequenzen optimiert werden.

3. Synchrone Kommunikation initiiert durch die *syncLink*-Relation:

   Ähnlich der *delConsArc*-Relation wird hier eine Codesequenz pro Kommunikationskanal mittels der Servicecodemodule erzeugt, die alle Output-Transitionen, die an diesem Kanal angeschlossen sind, administriert.

Pass 3: statische Adreßberechnung

Die Berechnungen des Pass 2 haben Codesequenzen berechnet, die den referenzierten AADL-Objekten der Operatoren zugeordnet werden konnten. In dieser letzten Phase des Linkens werden nun die statischen Codeadressen vergeben, da das Wissen um die Codesequenzlängen der Codemodule im System vorhanden ist.

Folgende Anordnung der Codesegmente wurde im Programmspeicher der ANM realisiert:

Die Auflösung aller indirekten Sprungadressen der ANM erfolgt durch einen linearen Analysealgorithmus, der die absoluten Startadressen aller Codesegmente berechnet.

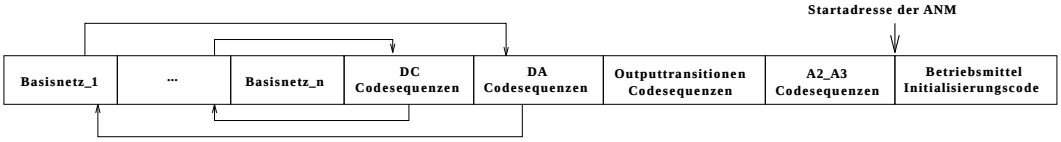


Abbildung 5.4: Codesegmentanordnung der ANM

Die synchronen Kommunikationskanäle des globalen Netzes sind als Matrix in die Laufzeitstruktur der ANM compiliert, damit eine aufwendige Initialisierung des Systems entfallen kann. Durch das Verkleben von Stellen und das Eintragen von Kanten muß noch zur Startzeit der ANM Code ausgeführt werden, der einige Betriebsmittel der ANM initialisiert.

Das engültige Programm wird durch Compilation des Linkcodes, der in der Programmiersprache C vorliegt, und das Linken des Objektcodes der compilierten Basisarchitekturen mit einem Simulationskern durch einen normalen C-Compiler erzeugt und kann „stand-alone“ ausgeführt werden.

5.2 Simulation Symbolischer Zeitdiagramme

Ein generelles Problem bei der formalen Verifikation komplexer Designs liegt in der Erstellung einer formalen Spezifikation, welche die gesamten Systemeigenschaften charakterisiert. Dem Designer muß es bei diesem Prozeß gelingen, seine anfangs informelle Anforderungsspezifikation in einen abstrakten Formalismus zu transformieren, ohne dabei wesentliche Eigenschaften zu verlieren oder durch eine Überspezifikation auszuschließen. Analog zur Erstellung einer korrekten Implementierung entsteht hier das Problem der Erstellung einer korrekten Spezifikation. Dieses resultiert in dem Problem, daß ein vom Model Checker aufgefundener Widerspruch zwischen der Implementierung und der temporallogischen Spezifikation sich nicht zwangsläufig auf einen Fehler innerhalb der Implementierung zurückführen läßt, sondern möglicherweise in einem Fehler in der Spezifikation.

5.2.1 Symbolische Zeitdiagramme

Zur Spezifikation von Anforderungsdefinitionen für AADL-Module werden im Kontext des COMDES-Projektes Symbolische Zeitdiagramme[5] [SD93, DJS93c] eingesetzt, welche als eine graphische Spezifikationssprache angesehen werden können (siehe Abbildung 5.5). Die Notation der STD ist hierbei an die Repräsentation der

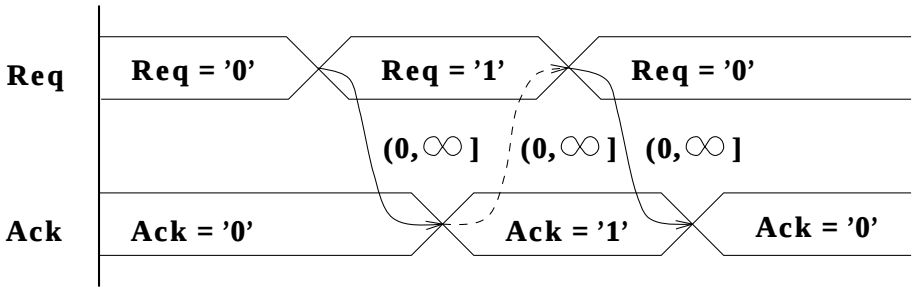


Abbildung 5.5: Request-Acknowledge Handshake Protokoll

klassischen Zeitdiagramme aus dem Bereich des Hardware-Designs angelehnt. Dieser neue Formalismus wurde erstmals durch [SD93] eingeführt und erlaubt eine Definition von *safety*- und *liveness*-Anforderungen sowie Beschreibungen von komplexen Protokolleigenschaften. STD werden im Rahmen dieser Arbeit zur Spezifikation von Wertverläufen über die im Interface eines AADL Moduls referenzierten Ports einge-

[5] engl. Symbolic Timing Diagrams (STD)

setzt. Hieraus folgt, daß die im AADL-Interface beschriebenen Typen und Portvariablen den Datenraum der STD-Simulation festlegen.

Eine Animation bzw. Simulation der Zeitdiagramm-Spezifikation kann in frühen Entwurfsphasen schon einen ersten Eindruck der Systemspezifikation vermitteln und Irrwege und Fehler innerhalb der Zeitdiagrammspezifikation eliminieren. Dies wird besonders dann notwendig, wenn eine Vielzahl von interagierenden Diagrammen zur Beschreibung des Gesamtverhaltens eines AADL-Moduls eingesetzt werden.

Durch eine auf ROBDD basierende symbolische Kodierung des zur Simulation der STD notwendigen Datenraums wird implizit eine Simulation auf Wertemengen realisiert. Jede Variable des Datenraums wird mit einer Menge von Werten ihres zugeordneten Domain assoziiert. Hierdurch wird bei einem Simulationslauf gleich eine Klasse von möglichen Abläufen realisiert.

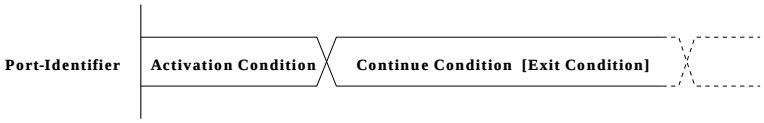


Abbildung 5.6: Symbolic Waveform

Eine Symbolische Zeitdiagrammspezifikation besteht aus einer Sammlung von einzelnen STD, welche mit einer Interfacebeschreibung eines AADL-Moduls[6] assoziiert sind. Ein STD besteht wiederum aus einer Anzahl von *Symbolic Waveforms*, wobei jede durch *Symbolic Events* wieder in *Regionen* aufgeteilt werden, welche mit Prädikaten[7] annotiert sind.

Abbildung 5.6 gibt die graphische Repräsentation einer Symbolic Waveform an. Ähnlich den klassischen Zeitdiagrammen, wird jeder Waveform ein Port-Identifier des Interfaces zugeordnet. Das erste boolesche Prädikat wird als Activation Condition bezeichnet. Ihr können dann beliebig viele *Continue Conditions* folgen, die die Wertbelegung des Port-Identifiers charakterisieren. Eine ggf. vorhandene *Exit Condition* gibt eine Bedingung an, bei deren Erfüllung das Zeitdiagramm legal verlassen bzw. deaktiviert werden kann.

Zwischen einzelnen Events können *Constraints Arcs*, durch Pfeile dargestellt, definiert werden, die eine temporale Ordnung festlegen. Hierbei können sowohl Events

[6] Einer *Entity Declaration* im VHDL-Kontext

[7] Boolesche Ausdrücke über Variablen, die in der Interfacebeschreibung des Moduls referenziert werden.

verschiedener Waveforms als auch Events ein und derselben Waveform in Relation gebracht werden. Abbildung 5.7 gibt eine graphische Darstellung der Constraint

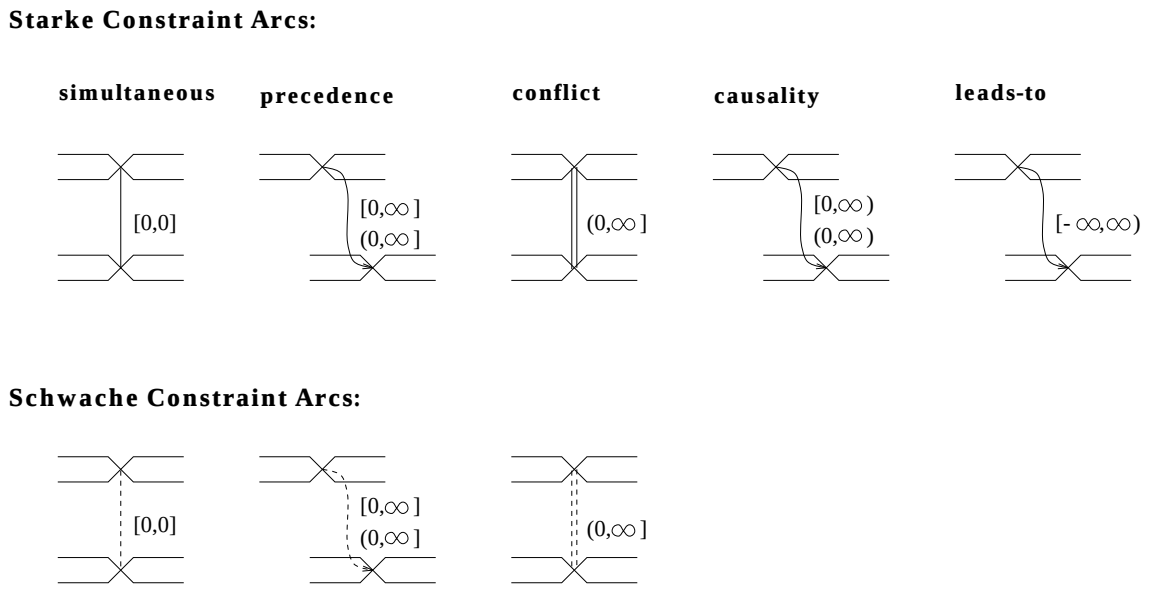


Abbildung 5.7: Constraint Arcs Symbolischer Zeitdiagramme

Arcs [DJS93c] wieder. Folgende fünf Ausprägungen der Arcs werden unterschieden:

Simultanous :

- $[0, 0]$[8] [9] : Beide Events finden parallel zum gleichen Zeitpunkt statt

Conflict :

- $(0, \infty]$: Beide Events dürfen nicht parallel zum gleichen Zeitpunkt stattfinden.

Precedence :

- $(0, \infty]$: Das *Ziel-Event* der Relation muß nach dem *Start-Event* beobachtet werden. Zur Erfüllung des Constraints muß aber das Ziel-Event *nicht* beobachtet werden.
- $[0, \infty]$: Analog zur obigen Beschreibung, jedoch kann das Ziel-Event der Relation auch parallel zum Start-Event beobachtet werden.

[8] Die Intervallschreibweise gibt an, wieviel Zeit verstreichen muß, bis das Event, auf das der Arc zeigt, stattfinden darf.

[9] Sowohl bei Simultanous als auch bei Conflict Arcs handelt es sich um eine Symmetrische Relation, sodaß die Intervallschreibweise beidseitig zu interpretieren ist. In der graphischen Notation werden deshalb beide Arcs ohne Pfeilspitzen dargestellt.

Causality :

- $(0, \infty)$: Das *Ziel-Event* der Relation muß nach dem *Start-Event* beobachtet werden. Zur Erfüllung des Constraints muß aber das Ziel-Event beobachtet werden.
- $[0, \infty)$: Analog zur obigen Beschreibung, jedoch kann das Ziel-Event der Relation auch parallel zum Start-Event beobachtet werden.

leads-to :

- $[-\infty, \infty)$: Bei diesem Constraint Arc muß das Ziel-Event beobachtet werden. Hierbei ist es jedoch irrelevant, ob es vor, gleichzeitig oder nach dem Start-Event beobachtbar ist.

Jedem STD kann ein *initial state* als Menge der möglichen Werte der Portvariablen zugeordnet werden, der sich als Konjunktion der Activation Conditions der einzelnen Waveforms bestimmenen läßt. Stimmen die Werte der Variablen der Interfacebeschreibung eines Moduls mit dem durch den initial state beschriebenen Zustand überein, wird das Zeitdiagramm aktiviert. Im folgenden sind nun alle Constraints dieses STD mit zu berücksichtigen. Der aktuelle Zustand der Abarbeitung eines Zeitdiagramms kann als Schnitt durch alle Waveforms angesehen werden und wird als *Front* bezeichnet. Sie kann entweder einen stabilen bzw. instabilen Zustand des STD charakterisieren in Abhängigkeit, ob noch nicht beobachtete Events durch Causality-Arcs referenziert werden. Ein Voranschreiten der Front kann dann zu folgenden drei Schritten in einem STD führen:

1. *Unwind-Step:*
   Das Fortschreiten der Front kann durch eine bzw. mehrerer Continue-Conditions der Waveforms des STD abgebildet werden.
2. *Exit-Step:*
   Bei diesem Fortschreiten werden eine oder mehrere Exit-Conditions aktiviert. Nach diesem Schritt wird das Zeitdiagramm deaktiviert.
3. *Fail-Step:*
   Weder durch einen Unwind- noch durch einen Exit-Step kann ein Fortschreiten der Front durchgeführt werden. Es kommt zu einer Fehlersituation innerhalb der STD-Spezifikation.

Bei der möglichen Aktivierung von Zeitdiagrammen werden zwei Arten unterschieden: *Initial STD* gelten nur initial und können zu Beginn der Gesamtspezifikation

aktiviert werden wohingegen *Invariant STD* immer dann aktiviert werden, wenn ihr Initial State dies erlaubt.

Constraint Arcs besitzen in ihrer Darstellung auch eine *weak* Form (im Gegensatz zur eigentlichen *strong* Form). Die weak Form drückt mehr eine Prämisse als eine Anforderung aus. D.h. durch das Verletzen einer weak Anforderung wird das Zeitdiagramm nur deaktiviert. Die Verletzung einer strong Anforderung führt dagegen zu einem Fehler. Durch diesen Ansatz ist es dem Designer möglich, eine Fallunterscheidung in seiner Spezifikation auszudrücken.

5.2.2 Übersetzung von Zeitdiagrammen in Petri-Netze

5.2.2.1 Motivation

Um eine schnelle und effektive Simulations- und Animationsumgebung bereitzustellen, muß eine adäquate Struktur zur Semantikrepräsentation der STD gewählt werden. Zwei Ansätze sollen hier kurz präsentiert werden:

1. Die formale Semantik einer STD-Spezifikation ist durch ihre Übersetzung in temporale Logik [SD93] via eines abgeleiteten Kontrollgraphen gegeben. Eine optimale Übersetzung dieser TL in Automaten ist durch [Kor97] gegeben. Nachteilig bei diesem automatenbasierten Ansatz für eine Simulation sind zwei Faktoren: Die Erzeugung bzw. Größe von TL-Ausdrücken aus einer STD-Spezifikation mit wenigen Kanten ist im *worst-case* exponentiell, da die temporallogische Formel „alle möglichen Pfade“ eines STD explizit repräsentieren muß; d.h. Parallelität bzw. Nebenläufigkeit muß bei diesem Ansatz als Serialisierte Darstellung des Kreusproduktes der möglichen Abläufe dargestellt werden. Andererseits ist eine Automatenrepräsentation einer TL-Formel in speziellen Fällen nicht trivial gegeben [uRS94, Kor97] und stellt ein Komplexitätsproblem dar. Simulationsiterationen mit wenig modifizierten STD-Spezifikationen können bei diesem Ansatz zu hohen Compilationszeiten führen.
2. Ein anderer, hier weiter verfolgter, Ansatz basiert auf einer Übersetzung der STD-Spezifikation in Petri-Netze. Sie eignen sich im Gegensatz zu Automaten sehr gut zur Kodierung nebenläufige Systeme, so daß bei einer Übersetzung der STD-Spezifikationen in Petri-Netze Strukturen in annähernd gleicher Größe aufgebaut werden. Zur Simulation der abgeleiteten Petri-Netze wird in dem hier vorgestellten Ansatz ein interpretatives Verfahren angewandt, so daß weitere Compilationszeiten entfallen. Dies ist akzeptabel, da der Designer schon an relativ

kurzen Ablaufsequenzen seiner STD-Spezifikation erste wichtige Einblicke gewinnen kann. Einen qualitativen Nachteil besitzt die auf Petri-Netzen basierende Simulation jedoch auch:
Sollte eine STD-Spezifikation einen Widerspruch enthalten, d.h. die gesamte Spezifikation ist *false*, wird dies im automatenbasierten Ansatz schon während der Übersetzung erkannt und es wird keine Struktur zu einer möglichen Simulation aufgebaut. Im Petri-Netz-basierten Ansatz wird jedoch eine Ablaufsequenz erzeugt, die erst als korrekt angesehen werden kann, falls eine Analyse des Goal-Stacks, d.h. der Menge von Prädikaten, die von der Simulation noch zu erfüllen sind, als konsistent und korrekt angesehen werden kann. Sollte der Goal-Stack nicht zu erfüllende Prädikate enthalten – diese Analyse ist vom Designer eigenverantworlich durchzuführen – kann der Designer anhand der Historie ihrer Erzeugung aber noch immer einen Ansatz zur Modifikation seiner STD Spezifikation ableiten, welches sich beim automatenbasierten Ansatz weitaus schwieriger gestalten würde.

Bei der Behandlung von sich selbst aktivierenden Zeitdiagrammen[10] ergibt sich ein weiterer Vorteil für die Petri-Netzdarstellung. Das dynamische Verhalten der STD-Spezifikation wird hierbei vollständig in den Simulationsalgorithmus verlagert und führt lediglich zur Inkarnation eines weiteren Petri-Netzes. Demgebenüber steht ein komplexer Mechanismus bei einem automatenbasierten Ansatz, da zur Übersetzung dieser speziellen TL-Formeln in Automaten im worst case mit einer exponentiellen Darstellung zu rechnen ist.

In dem hier verfolgten Ansatz wird eine Simulation eines Constraint Systems, wie es durch eine STD-Spezifikation gegeben ist, auf eine Petri-Netz-Simulation abgebildet. Im folgenden wird zunächst auf die Ableitung der Petri-Netzdarstellung von STD eingegangen. Anschließend werden geringfügige Erweiterungen der AADL Petri-Netze eingeführt. Danach erfolgt die Beschreibung der symbolischen Simulation des Constraint Systems welche als Ergebnis Folgen möglicher Wertverläufe der Interface-Variablen einer STD-Spezifikation erzeugen kann.

[10] Hierbei handelt es sich um Invariant-STD, deren Initial State als Front bei ihrer Abwicklung beobachtbar ist.

5.2.2.2 Phasen der STD-Compilation

Die Konstruktion der Petri-Netzrepräsentation für Zeitdiagramme erfolgt in drei Phasen:

1. Generierung eines zyklenfreien Netzes für jede Symbolic Waveform
2. Verkleben der Teilnetze mit Prolog- und Epilog-Netzkomponenten
3. Inkrementelle Interpretation der Constraint Arcs

Die folgenden Paragraphen gehen nun auf die einzelnen Compilationsschritte näher ein:

Generierung der Netze

In dieser ersten Phase wird jeder Waveform ein Netz zugeordnet (siehe Abbildung 5.8). Bei dem Übersetzungsvorgang wird jedem Event eine Transition zugeordnet,

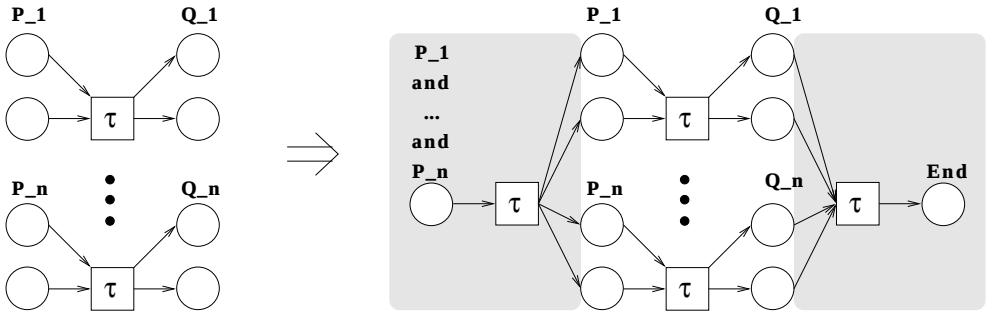


Abbildung 5.8: Prozeßnetzgenerierung

deren Vorbereich und Nachbereich jeweils durch eine Kontrollfluß- und eine Triggerstelle dargestellt wird. Die Triggerstelle im Vorbereich (Nachbereich) ist mit der *current assertion* (*continue assertion)* annotiert. Existiert eine *exit assertion*, so wird eine weitere Transition mit gleichen Vorbereich eingeführt. Der Nachbereich dieser Transition besteht wiederum aus einer Triggerstelle, annotiert mit der *exit assertion* und einer Kontrollflußstelle, die jedoch eine gesonderte Behandlung innerhalb der nächsten, sich anschließenden Compilationsphase erfährt.

Komposition der Prozeßnetze mit Prolog- und Epilog-Netzstrukturen

Sind alle Waveforms in Teilnetze übersetzt worden, so werden sie durch zwei weitere Netzfragmente zu einem einzigen Gesamtnetz verbunden (siehe Abbildung 5.9):

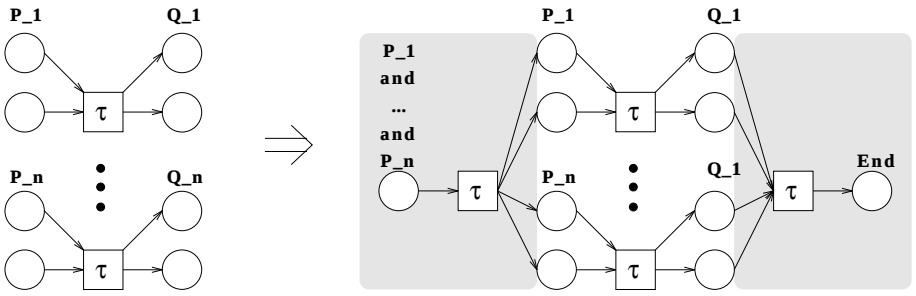


Abbildung 5.9: Verkleben der Teilnetze

- Durch das Prolog-Netzfragment wird der matching state des Zeitdiagramms kodiert. Es besteht aus einer Triggerstelle, die mit der Konjunktion aller Annotationen der ersten Triggerstellen jedes beteiligten Prozeßnetzes beschriften ist, und einer Tau-Transition. Die Transition hat die Aufgabe, die Aktivierung des Zeitdiagramms zu testen und dann ggf. den Kontrollfluß auf die einzelnen Prozeßnetze zu verteilen.
- Das Epilog-Netz entzieht den Prozeßnetzen ihr jeweiliges lokales Kontrollflußtoken und belegt eine ausgezeichnete *End*-Stelle, die die vollständige Abarbeitung des Gesamtnetzes charakterisiert, mit einem Token. Hierdurch kann der Simulator explizit das Ende der Simulation eines Zeitdiagrammnetzes ausmachen und aus der Menge der zu berücksichtigen STD zu entfernen.

Interpretation der Constraint-Arcs

Bei der Übersetzung der Constraint Arcs erfolgt die Kodierung der durch sie vorgegebenen Halbordnungseigenschaften auf den Events in einer Modifikation der möglichen Schaltfolge der Transitionen auf Petri-Netzebene. Dies erfolgt durch bekannte Petri-Netz-Transformationen [Rei85] zur Synchronisation, exklusivem Ausschluß und Aufbau von Halbordnungseigenschaften durch Einführung von *Semaphoren* und Unifikation von Transitionen.

Im folgenden wird die Transformation anhand der unterschiedlichen Arcs auf der Basis des bisher erzeugten Netzskeletts dargestellt. Aus Visualisierungsgründen wird hier nur auf die Interpretation jeweils eines Arcs eingegangen.

Simultanous Arc:

Alle Events, die über einen Simultanous Arc in Beziehung gesetzt sind, werden zu einer einzigen Transition zusammengefaßt, welche als Identifikator die Namen der ursprünglichen Transitionen trägt. Ein paralleles Schalten der beteiligten Transitio-

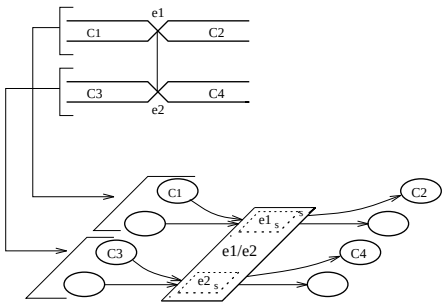


Abbildung 5.10: Simultanous Arc $[0, 0]$

nen wird sichergestellt (siehe Abbildung 5.10).

Conflict Arc:

Alle Transitionen, die über Conflict Arcs in Beziehung gesetzt sind, werden mit einer neu erzeugten Semaphorstelle verbunden, die den wechselseitigen Ausschluß der Events garantiert (siehe Abbildung 5.11).

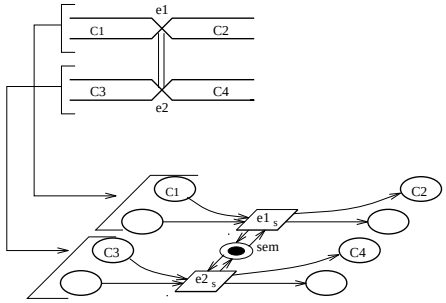


Abbildung 5.11: Conflict Arc $(0, \infty]$

Precedence Arc:

$(0, \infty)$**:** Durch die Einführung einer weiteren Kontrollflußstelle im Nachbereich der Transition e_1 und im Vorbereich der Transition e_2 wird die Netzstruktur dahingehend modifiziert, daß die durch den Constraint Arc induzierte Halbordnungsrelation eingehalten wird (siehe Abbildung 5.12 a))

$[0, \infty)$**:** Die Netztransformation ist analog dem obigen Vorgang, jedoch muß bei diesem Constraint auch ein mögliches paralleles Stattfinden der Events e_1 und e_2 beobachtbar sein, welches sich auf Netzebene durch die Einführung einer zusätzliche Transition e_1/e_2 niederschlägt (siehe Abbildung 5.12 b)).

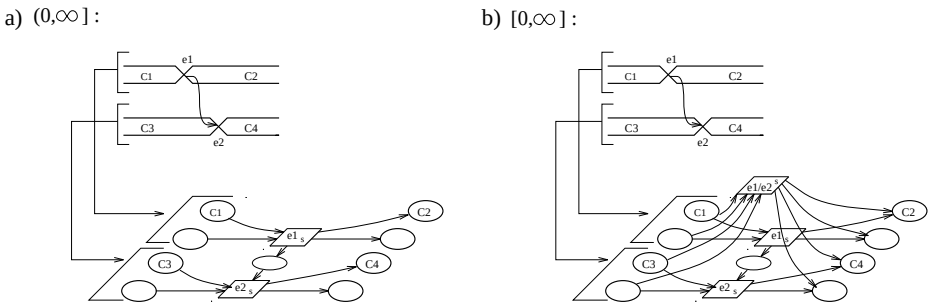


Abbildung 5.12: Precedence Arc $(0, \infty]$ und $[0, \infty]$

Causality Arc:

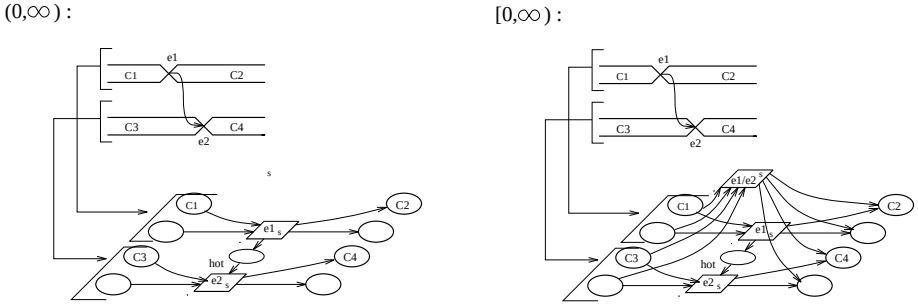


Abbildung 5.13: Causality Arc $(0, \infty)$ und $[0, \infty)$

Im Gegensatz zum Precedence Arc, durch den nur eine Halbordnungseigenschaft manifestiert wird, fordert der Causality Arc auch die Beobachtung des durch e_1 angestoßenen Events e_2. Die Simulation dieses Verhaltens auf Petri-Netzebene wird durch Einführen einer besonderen Kontrollflußstelle, einem s.g. *hot place*, realisiert. Der Simulator kann an der Markierung eines hot places später identifizieren, ob ein STD in einem stabilen Zustand ist oder ob noch weitere Events beobachtet werden müssen. Bei einem parallelen Schalten von e_1 und e_2 wird das hot Place nicht referenziert, da e_2 schon beobachtet wurde (siehe Abbildung 5.13).

Leads-To Arc:

Der Aufbau des Petri-Netzes für die Simulation des Leads-to Arcs stellt eine Erweiterung des Causality Arcs dar und ist als Fallunterscheidung anzusehen. Neben den festgelegten Eigenschaften muß bei diesem Constraint noch eine weitere Eigenschaft kodiert werden: Das Event e_2 darf vor dem Event e_1 beobachtet werden und hierdurch ist das Constraint schon vollständig erfüllt. Auf Petri-Netzebene wird der aufzubaueneden Netzstruktur für den $[0, \infty)$-Causality Arc noch eine sequentielle

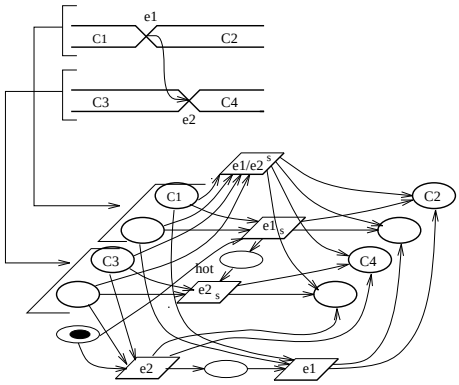


Abbildung 5.14: Leads-To Arc $[-\infty, \infty)$

Abfolge der Transitionen e_2 und e_1 mit einer Kontrollflußstelle hinzugefügt. Um ein ungewolltes Markieren des hot Places durch Transition e_1 zu verhindern, wird noch eine weitere Kontrollflußstelle eingeführt, die die erfolgte Fallunterscheidung auf Petri-Netzebene kodiert (siehe Abbildung 5.14).

Weak Constraint Arcs:

Die Übersetzung der Weak Constraint Arcs erfolgt analog zu den gerade beschriebenen Strong Constraint Arcs. Für spätere Simulationsläufe muß dem Simulationsalgorithmus jedoch die Information zugänglich sein, welche Netzkonstrukte auf der Basis eines Weak Constraints erfolgt, um ggf. gegen sie verstoßen zu können mit gleichzeitiger Deaktivierung des zugehörigen STD. Hierzu werden die neu erzeugten Netzobjekte mit einem weak-Attribute versehen. Für die Simultanous und Conflict Arcs müssen hierzu neben den Attributen auch Äquivalenzklassen für Strong und Weak Arcs verwaltet werden, um eine exit bzw. failure Bedingung des STD korrekt ableiten zu können.

Exit Conditions:

Eine Besonderheit bei dem Fortschreiten bzw. Abarbeiten eines STD liegt in der Behandlung von Exit-Conditions. Sollte die Continue Condition C_2 nicht erfüllt werden, sondern die Exit Condition C_3, dann wird das Zeitdiagramm analog zur Verletzung eines Weak Constraint Arcs deaktiviert. Dies wird auf Petri-Netzebene durch das Einführen einer weiteren Transition, die ein Token auf die End-Stelle des Epilog-Netzes legt, dargestellt (siehe Abbildung 5.15). Hierdurch kann der Simulator das Petri-Netz aus der Menge der aktiven Zeitdiagrammnetze entfernen.

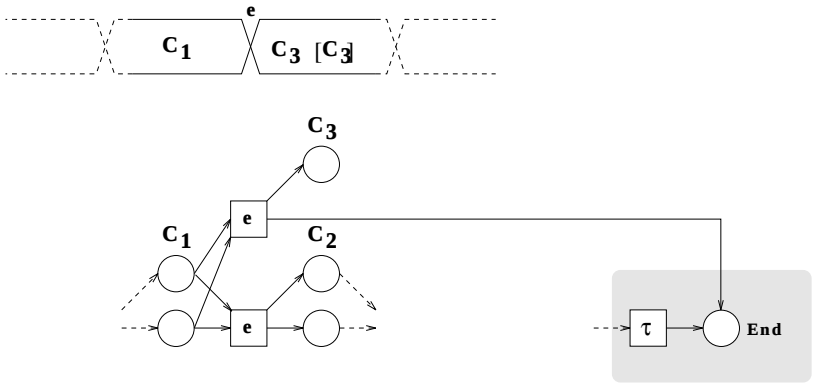


Abbildung 5.15: Kodierung von Exit-Conditions

5.2.3 Symbolische Simulation

Die Simulation einer Zeitdiagrammspezifikation basiert auf der symbolischen Simulation der von ihr abgeleiteten AADL Petri-Netze. Der Simulationsalgorithmus weicht von dem im Kapitel 5 vorgestellten Verfahren ab, da die Menge der an der Simulation beteiligten Inkarnationen der AADL Petri-Netze sich dynamisch ändert und auch die Schaltregel zur Petri-Netz gesteuerten Animation der Zeitdiagramme abweichend ist. Transitionen führen bei diesem Ansatz keine Datentransformationen zur Berechnung der Wertverläufe der beteiligten Variablen aus, vielmehr berechnet der Simulationsalgorithmus sie implizit über die Gültigkeit der booleschen Bedingungen der Triggerstellen der aktivierten Transitionen.

Durch das vorgestellte Verfahren zur Simulation der *Zeitdiagramm-Netze* wird eine Sequenz von Werten für die Port-Variablen des Interfaces inkrementell vom Simulator berechnet. Der Ablauf kann wie folgt beschreiben werden (siehe Abbildung 5.16):

Kernstruktur des Systems ist der Simulationszustandsraum. Er besteht aus einer Menge von Inkarnationen aktivierter Zeitdiagramm-Netze sowie einem ROBDD basierten Datenraum. Die Menge der Zeitdiagramm-Netze steuert hierbei als abstraktes Kontrollprogramm die möglichen Wertverläufe der Variablen. Durch die symbolische Kodierung des Datenraums ist es möglich, jeder Variablen eine Menge von Werten zuzuordnen. Hierdurch wird bei jedem Simulationslauf gleich eine Klasse von möglichen Abläufen generiert, die alle auf dem selben Fortschreiten der Front basieren.

Ausgehend von der Menge der „*initial mode*“-Zeitdiagramme wird eine initiale Konfiguration des Simulationsraums erzeugt. Für jede Port-Variable wird die Menge sei-

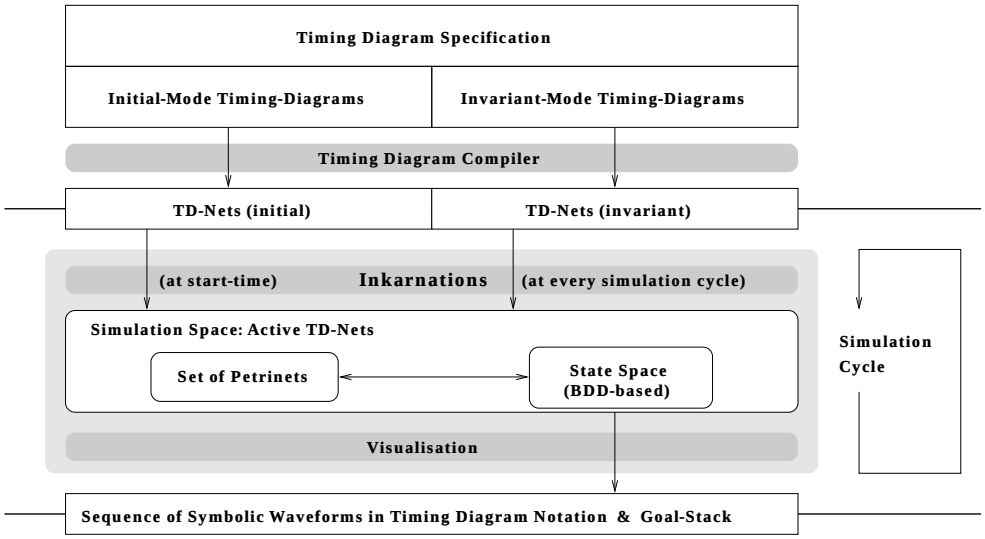


Abbildung 5.16: Aufbau des Zeitdiagrammsimulationsmodells

ner charakterisierenden Startbedingungen aus den *matching states* jedes o.g. Zeitdiagramms selektiert. Sollten sich Prädikate bzgl. eines Ports widersprechen, d.h. deren Konjunktion evaluiert sich zu *false*, so kann schon innerhalb der Initialisierungsphase festgestellt werden, daß die Zeitdiagrammspezifikation als Gesamtsystem nicht erfüllbar ist und somit vom Designer eine falsche Spezifikation erfolgte. Wird zu einem Port kein einschränkendes Prädikat durch die Zeitdiagrammspezifikation angegeben, so startet die Simulation mit allen möglichen Wertbelegungen für diese Port-Variable. Nach einer erfolgreichen Initialisierungsphase besteht der Simulationsraum somit aus den Inkarnationen der „initial mode“ Zeitdiagramm-Netze und einem ROBDD, der die Wertemengen aller Variablen charakterisiert.

An diese Phase schließt sich der eigentliche Simulationszyklus an: Alle „*invariant mode*“ Zeitdiagramm-Netze werden bzgl. ihres matching state auf „Verträglichkeit“ mit in dem aktuell existierenden Zustand des Datenraums hin überprüft, d.h. es wird geprüft, ob die Konjunktion aus der Bedingung des matching state und der Zustandskodierung des Simulationsraumes ungleich *false* ist. In diesem Fall wird eine Inkarnation des Netzes in den Simulationsraum aufgenommen und der Zustand des Datenraums durch die Aufnahme der matching state Bedingung weiter eingeschränkt. In dieser Phase findet Nichtdeterminismus seinen Eingang in die Simulation durch ggf. nicht stark genug eingeschränkte Wertebereiche der Port-Variablen. Existieren z.B. mehrere Zeitdiagramm-Netze, die jedes für sich mit der Konfiguration des Datenraums verträglich sind, sich jedoch gegenseitig ausschließen, muß eine zufällige Selektion vorgenommen werden. Damit alle möglichen Abfolgen ge-

neriert werden können, muß der gesamte Simulationsablauf ggf. öfters durchlaufen werden.[11]

Ist diese Phase beendet, kann der Simulator einen Simulationsschritt durchführen, indem er eine Teilmenge von aktiven Transitionen berechnet, deren Tokenspiel durchführt und den Zustand des Datenraums neu anhand der markierten Triggerstellen durch einen ROBDD charakterisiert. Desweiteren werden die Events, welche auf Petri-Netzebene durch *hot* Stellen markiert sind, in einen *Goal Stack* integriert und dort für spätere Simulationszyklen verwaltet. Zur Beobachtung des Simulationsverlaufes hat der Designer Zugriff auf den Simulationsraum, d.h. er kann auf die gerade gültigen Wertemengen jeder Port-Variablen zugreifen, er hat Zugriff auf die Menge der aktuell existierenden Inkarnationen der Petri-Netze und auf die Menge der noch zu erfüllenden Prädikate des Goal-Stack. Ist eine Inkarnation eines Zeitdiagramm-Netzes vollständig und korrekt abgearbeitet, d.h. die End-Stelle ist mit einem Token belegt worden, so wird es aus dem Simulationsraum entfernt und kann so nicht weiter Einfluß auf die Simulation nehmen.

Ein neuer Simulationszyklus kann nun initiiert werden, indem die Menge der „*invariant mode*“-Zeitdiagramm-Netze nach potentiellen Kandidaten hin abgesucht und ggf. in den Simulationsraum integriert werden können.

5.2.3.1 Berechnung eines Simulationsschrittes

Um ein Fortschreiten der Front innerhalb der Zeitdiagrammspezifikation zu berechnen, werden die Inkarnationen der Zeitdiagramm-Netze analysiert. Ziel ist die Berechnung einer Menge von konfliktfrei schaltbarer Transitionen der jeweiligen Zeitdiagramm-Netze und eine neue Charakterisierung des Zustands des Datenraums.

Für jede Port-Variable werden daher zwei Mengen von Transitionen konstruiert:

1. Aktivierte Transitionen, welche die Markierung von Triggerstellen verändern, die mit einem Prädikat über die Port-Variable annotiert sind.
2. Nicht aktivierte Transitionen mit o.g. Eigenschaft und einer markierten Kontrollflußstelle im Vorbereich. Diese Transitionen könnten somit potentiell die Markierung von Triggerstellen verändern, die mit einem Prädikat über die Port-Variable annotiert sind, jedoch scheint ihnen eine weitere Prämisse zur eigentlichen Aktivierung zu fehlen.

[11] Dies ist ein generelles Problem bei der Simulation nichtdeterministischer Systeme und stellt keinen Nachteil dar, der nur dieser Realisierung anhaftet.

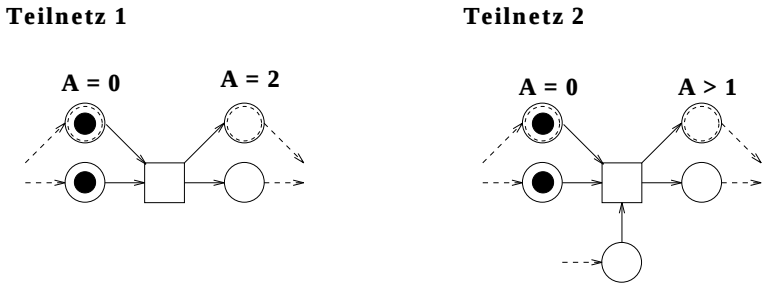


Abbildung 5.17: Abhängigkeiten zwischen Zeitdiagramm-Netze

Der Simulationsalgorithmus kann nun anhand beider Mengen für jede Port-Variable eine Menge von Transitionen konstruieren, die geschaltet werden können und hierdurch eine neue Markierung der Triggerstellen und so eine neue Charakterisierung des Zustandes des Datenraums festlegen:

Die nicht aktivierten Transitionen implizieren über die Prädikate ihrer Triggerstellen im Nachbereich die Wertebereiche einer Portvariablen, die nicht im nächsten Schritt angenommen werden dürfen und induzieren deshalb ein „Sperrverhalten“ auf die Menge der aktivierten Transitionen. Der Simulator muß nun eine *konsistente* Teilmenge der aktivierten Transitionen durch Ausschlu berechnen, die nicht im Widerspruch zum Sperrverhalten steht, d.h. die Prädikate der durch das Schalten markierten Triggerstellen widersprechen sich nicht. Sollten mehrere konsistente Teilmengen konstruiert werden können, wird nichtdeterministisch vom Simulator eine Teilmenge ausgewählt und geschaltet.

Kann durch das implizite Sperrverhalten der nicht aktivierten Transitionen oder durch das Schalten von aktiven Transitionen keine konsistente Teilmenge konstruiert werden, versucht der Simulator durch Interpretation der weak-Annotationen das System abzuschwächen. Hierdurch können (aktivierte und sperrende) Transitionen aus der jeweiligen Menge entfernt werden. Sollte es hierdurch möglich sein, eine konsistente Teilmenge von schaltbaren Transitionen zu generieren, werden die durch die Verletzung deaktivierten Inkarnationen der Zeitdiagramm-Netze aus dem Simulationsraum entfernt. Kann trotz dieser Abschwächung keine konsistente Menge von schaltbaren Transition generiert werden, bricht der Simulator die Simulation mit einem Prädikat-Fehler ab.

Abbildung 5.17 zeigt das Sperrverhalten des zweiten Teilnetzes auf das erste Teilnetz. Die Port-Variable A hat im aktuellen Zustand des Datenraums den Wert 0. Durch das Schalten der aktiven Transition aus Teilnetz 1 würde $A = 2$ erzwungen werden, was jedoch im Widerspruch zu Teilnetz 2 steht, da dann die Bedingung $(A = 0) \wedge$

$(A = 2) = false$ gelten müßte.

5.2.3.2 Ergebnisse der Simulation

Terminiert der Simulationsalgorithmus, so können zwei Fälle unterschieden werden:

1. Sind alle Inkarnationen der Zeitdiagramm-Netze erfolgreich abgearbeitet, so wurde die Simulation erfolgreich beendet.
2. Existieren noch Inkarnationen der Zeitdiagramm-Netze im Simulationsraum, so wurde die Simulation durch einen Prädikat-Fehler abgebrochen und die Simulation ist somit fehlgeschlagen. Dem Simulationsalgorithmus ist es nicht gelungen, trotzt Abschwächung durch ggf. vorhandener Weak-Annotation die Simulation der Front weiter zu treiben. Die Simulation kann nun von Benutzer erneut angestoßen werden, um sich ggf. einen anderen Verlauf der Front berechnen zu lassen.

Im zweiten Fall können auch zwei Situationen vom Benutzer unterschieden werden, die vom Inhalt des Goal-Stack abhängen:

1. Sollten keine Prädikatem auf dem Goal-Stack vorhanden sein, so ist die Simulation in einem konsistenten Zustand und die Simulation ist erfolgreich.
2. Befinden sich auf dem Goal-Stack nur erfüllbare Prädikate, die wiederkehrend abgearbeitet werden, so kann die Simulation auch als erfolgreich angesehen werden. D.h. die Zeitdiagrammspezifikation ist korrekt und der Benutzer kann anhand des erzeugten Traces seine Spezifikation nachvollziehen.
3. Sollten auf dem Goal-Stack Prädikate vorhanden sein, die nicht zu erfüllen sind, so ist die Simulation fehlgeschlagen und alle bisherigen Berechnungen zum Fortschreiten der Front bzw. die erzeugten Sequenzen von Wertverläufen der Port-Variablen sind falsch. Die Zeitdiagrammspezifikation ist somit als Gesamtsystem als falsch anzusehen.

5.3 Abgrenzung zu anderen Netzsimulatoren

In den letzten Jahren sind eine Vielzahl von Petri-Netzsimulatoren implementiert und via Internet publiziert worden. Eine Abgrenzung zu den derzeit über 100 Simulatoren weltweit kann im Einzelnen im Rahmen dieser Arbeit aus komplexitäts-

gründen nicht vorgenommen werden. Desweiteren ist eine vergleichende Diskussion jeweils abhängig von der zugrunde liegenden Petri-Netzklasse des Simulators. Einen, wenn auch nicht mehr sehr aktuellen, Überblick über Netzsimulatoren kann [FJ91] vermitteln. In diesem Abschnitt soll deshalb nur ein Vergleich anhand des an der Oldenburger Universität bzw. OFFIS entwickelten DNS-Systems durchgeführt werden, da die Netzklasse „nahe“ an der Klasse der AADL Petri-Netze liegt und auch ein compilatives Simulationsverfahren verfolgt wird.

Das Projekt DNS (*D*istributed *N*et *S*imulation) hat das Ziel, reale Systeme mit Hilfe einer high-level Sprache zu beschreiben und die Modelle effizient zu simulieren [SSW94]. Die verwendete Sprache basiert auf einer Klasse beschrifteter, hierarchisch strukturierter, höherer Petri-Netze, die als THORN (*T*imed *H*ierarchical *O*bject-Related *N*ets) bezeichnet wird. Einige Konzepte der THORN sollen hier dargestellt werden:

- Token sind C++ Objekte und werden als C++ Klassen definiert,
- Stellen verwalten die auf ihnen liegenden Token strukturiert, d.h. der Zugriff auf Token erfolgt anhand der Struktur der Stelle als z.B. multiset, stack, queue oder priority queue,
- Stellen können immer nur Token einer Klasse verwalten, die Kapazität der Stellen kann auch unbegrenzt sein,
- Transitionen besitzen explizit eine Aktivierungsbedingung und einen Aktionsblock,
- Transition können zeitbeschriftet sein, d.h. es kann eine *firing time* und eine *delay time* spezifiziert werden [SSW95],
- Als Kantentypen werden *enabling*, *inhibitor* und *consuming* Kanten unterstützt,
- Als Hierarchisierungskonzept wird dynamisches *transition refinement* angeboten. Hierbei wird eine Transition durch ein Petri-Netz ersetzt, wobei eine Verschmelzung der Randstellen der verfeinerten Transition mit den Randstellen des neu eingefügten Netzes erfolgt.

Die Netzklassen weisen ähnliche Grundstrukturen auf. Als wichtiges Unterscheidungsmerkmal läßt sich die Komposition von Petri-Netzen ausmachen. Im DNS-System wird dies hierarchisch durch Verfeinerung von Transitionen durchgeführt wohingegen im COMDES-System Petri-Netze anhand einer abstrakten Schnittstelle komponiert werden, wobei dann Transitionen und Stellen unifiziert werden müssen und nachträglich Kanten in die Netzstruktur eingeführt werden müssen.

Zur Simulation der Netzklasse im DNS-System ist in sequentieller und ein verteilter Netzsimulator entwickelt worden. Um einen Vergleich der Performance zu erhalten, soll hier nur der sequentielle Simulator betrachtet werden.

Zur Simulation eines THORN wird die Aktivierungsbedingung und der Aktionsblock jeder Transition in C++ Funktionen transformiert und mit einem Simulationskern zusammen in ein ausführbaren Programm übersetzt. Dieser Ansatz ist für Netze mit über 100.000 Transition nicht mehr anwendbar, da ein normaler Linker mit einer Anzahl von mehr als 16.000 verschieden Funktionen nicht mehr einen ausführbaren Programmcode erzeugen kann. Auch die Linkzeit steigt rasant mit der Anzahl der zu linken Funktionen an. Das „Linken“ im COMDES-System wurde deshalb als Array-Initialisierung mit ca. 50 verschieden Funktionspointern kodiert, so daß selbst für ein Netz mit ca 100.000 Transitionen dieser Vorgang unter 10 Sekunden beendet ist.

Durch die Übersetzung in C++ ohne Bereitstellung eines expliziten Debuggers, der eine Rekonvertierung der abgeleiteten Funktionen auf Petri-Netzebene bereit stellt, kann eine Analyse des Petri-Netzes nur via eines C++ Source-Code Debuggers erfolgen. Der COMDES Debugger hingegen ist schon alleine durch eine Visualisierung der Betriebsmittel der ANM gegeben, da hier alle Netzstrukturen, Markierungen und Aktiviertheitszustände der Transitionen explizit auf Netzlevel sichtbar sind. Eine Rückrechnung auf den erzeugenden AADL Source-Code via Referenzen ist einfach durchzuführen.

In [SSW94] wird eine Transitionsschaltfolge von ca. 1.500 Transitionen pro Sekunden angegeben. Die Schaltzeiten des COMDES-System werden z.b. bei der Multiprozessorfallstudie nur durch die Trennung des Zustandsraumes vom Petri-Netz durch eine ggf. notwendige Netzwerkkommunikationen auf ca. 10.000 Transitionen pro Sekunden begrenzt. in einem einzigen monolithischem Programm steigt der Durchsatz pro Sekunde auf über 25.000 Transitionen.

Teil IV

Symbolische Verifikationstechniken für den Hardware Entwurf auf Systemebene

Kapitel 6

Symbolische Verifikation

Während im vorigen Teil der Arbeit eine Validierung eines AADL-Designs auf der Basis von Simulationstechniken eingehend beschrieben wurde, wird in diesem Kapitel auf die formale Verifikation der erstellten AADL Entwürfe eingegangen. Wie schon in der Einleitung motiviert wurde, kann eine Simulation nur eine partielle Aussage über die Korrektheit eines Systems geben, wenn nicht alle möglichen Eingabewerte für das System vollständig simuliert wurden.

Die formale Verifikation eines Systems, wie z.B. das in dieser Arbeit vorgestellte *Finite State Model Checking*, erlaubt hingegen die Aussage, daß ein System keinen Fehler in Bezug auf die gegebene Anforderungsspezifikation besitzt - und zwar in Bezug zu einem vollständigen mathematischen Kalkül.

Da eine explizite Repräsentation des Erreichbarkeitsgraphen, welcher für *finite state* Verifikationsmethoden zwingend notwendig ist, für realistische, industrielle Anwendungen mit mehr als 10^6 Zuständen nicht mehr direkt darstellbar ist, wird im folgenden Kapitel 6.1 eine symbolische, auf ROBDDs basierende Kodierung der Struktur motiviert und vorgenommen. Hierauf aufbauend wird im Kapitel 6.2 eine symbolische Kodierung der AADL Petri-Netze inklusive des AADL-Datenraums vorgenommen, wobei die Kontrollstruktur des Petri-Netzes zur Kodierungsoptimierung ausgenutzt wird. Um die Effizienz dieses Verfahrens zu verdeutlichen, wird im Kapitel 6.3 die Kodierung von aus VHDL-Beschreibungen erzeugten AADL Petri-Netzen vorgestellt und ein Vorschlag zur Reduktion der symbolischen Struktur auf die für die Umgebung beobachtbaren Schritte motiviert. Als Besonderheit ergibt sich hierbei für deterministische Systeme eine funktionale Kodierung des Modells. Eine weitere Optimierung kann durch eine Abstraktion (Slicing) des Modells erfolgen, welche im

Kapitel 6.4 vorgestellt wird. Bei dem gewählten Ansatz wird in Abhängigkeit der zu verifizierenden temporallogischen Formel eine Projektion des Modells gebildet, welche noch den Nachweis aller Eigenschaften des Originalmodells zur Verifikation der Formel erlaubt. Im Kapitel 6.5 erfolgt kurz die Vorstellung eines funktionalen symbolischen Modell Checking Verfahrens, welches die funktionale Beschreibung des Modells verarbeiten kann.

6.1 Modellierung von B/E Systemen

Sei $N = (S, T, F, m_0)$ ein B/E-System (siehe Definition 4.4). Eine Markierung von N kann durch eine Menge von Stellen m dargestellt werden, wobei $s_i \in m$ die Eigenschaft beschreibt, daß auf der Stelle s_i ein Token liegt. Jede Menge von Markierungen kann durch eine Menge M von Teilmengen von S dargestellt werden. M_S sei die Menge aller möglichen Markierungen eines B/E-Systems mit $|S|$ Stellen ($|M| = 2^{|S|}$). Das System

$$(2^{M_S}, \cup, \cap, \emptyset, M_s)$$

ist eine boolesche Algebra von Mengen von Markierungen (siehe Kapitel 4.2). Dieses System ist isomorph zu der booleschen Algebra der n-stelligen logischen Funktionen mit $n = |S|$ ist.

In diesem Kapitel wird im folgenden nicht weiter zwischen der Stelle $s_i \in S$ und ihrer entsprechenden booleschen Variablen unterschieden. Es besteht deshalb eine direkte Verbindung zwischen den Markierungen von M_S und Elementen von B^n. Eine Markierung $m \in M_S$ kann mittels einer Kodierungsfunktion $\kappa : M_S \rightarrow B^n$ in ein entsprechendes Element $(s_1, \ldots, s_n) \in B^n$ abgebildet werden. Hierbei gilt

$$s_i = \begin{cases} true & gdw.\ s_i \in m \\ false & gdw.\ s_i \notin m \end{cases}$$

6.1.1 Charakteristische Funktionen und binäre Relationen

Die charakteristische Funktion χ_E einer Menge von Elementen $E \subseteq B^n$ ist definiert als logische Funktion, die für die Elemente von B^n zu $true$ evaluiert, die sich in E befinden:

$$\forall e \in B^n : e \in E \Leftrightarrow \chi_E(e) = true$$

Wird die Anwendung der Kodierungsfunktion κ auf Mengen von Markierungen $M \in 2^{M_S}$ hin erweitert, so besitzt jede von ihnen eine charakteristische Funktion $\chi_M^\kappa : B^n \to B$, die für diejenigen Elemente zu $true$ evaluiert, die zu M gehören. Im folgenden wird bei einer fest gegebenen Kodierungsfunktion κ die charakteristische Funktion von M festgelegt und hat folgende Eigenschaft.

$$\chi_M^\kappa(s_1, \ldots, s_n) = 1 \; gdw. \; \exists m \in M : \kappa(m) = (s_1, \ldots, s_n)$$

Aus Vereinfachungsgründen wird nicht weiter zwischen M und der charakteristischen Funktion χ_M unterschieden. Alle Mengenoperationen können direkt auf cha-

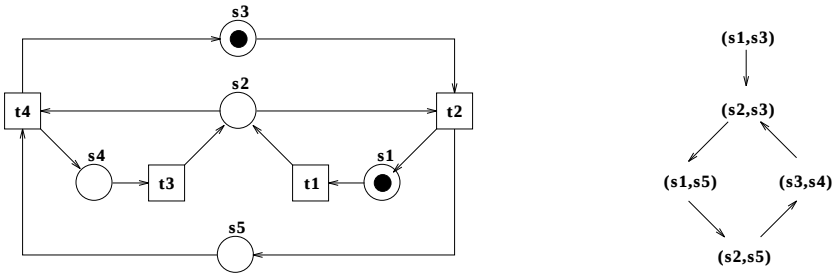


Abbildung 6.1: BE/System und zugehöriger Fallgraph

rakteristische Funktionen angewendet werden. Z.B. können auf die Markierungen $M_1, M_2 \in M_S$ eines B/E-Systems folgende Operationen angewendet werden:

$$\begin{aligned} \chi_{M_1 \cup M_2} &= \chi_{M_1} + \chi_{M_2} \\ \chi_{M_1 \cap M_2} &= \chi_{M_1} * \chi_{M_2} \\ \chi_{\bar{M}_1} &= \bar{\chi}_{M_1} * \chi_{M_S} \end{aligned}$$

Werden charakteristische Funktionen mit Hilfe von ROBDDs implementiert, so sind deren Repräsentationen im Allgemeinen sehr kompakt und effizient dargestellt. Charakteristische Funktionen können auch zur Repräsentation von binären Relationen verwendet werden, in dem die Teilmengen des kartesischen Produktes der Mengen dargestellt werden. Um die binäre Relation $R \subseteq M_1 \times M_2$ darstellen zu können, müssen M_1 und M_2 durch verschiedene Mengen von Variablen kodiert werden. Bei einer gegebenen Relation R zwischen den Mengen M_1 und M_2 kann die Menge der Elemente von M_2, die in Relation zu Elementen aus M_1 stehen, wie folgt charakterisiert werden:

$$V_R(M_1) = \{m_2 \in M_2 \mid \exists m_1 \in M_1 : (m_1, m_2) \in R\}.$$

Die charakteristische Funktion von V kann dann mittels der charakteristischen Funktion von R angegeben werden:

$$\chi_V(x_1, \ldots, x_n) = \exists_{y_1, \ldots, y_n} \chi_R(x_1, \ldots, x_n, y_1, \ldots, y_n).$$

6.1.2 Schalten von Transitionen

Im Kontext von symbolischem Model Checking kann die Transitionsfunktion $\vec{\delta}$ eines B/E-Systems als

$$\vec{\delta} : 2^{M_S} \times T \rightarrow 2^{M_S}$$

definiert werden. Sie transformiert für jede Transition des Netzes eine Menge von Markierungen M_1 in eine neue Menge von Markierungen M_2:

$$\vec{\delta}(M_1, t) = M_2 = \{m_2 \in M_S | \exists m_1 \in M_1, m_1[t > m_2\}$$

Die Definition von $\vec{\delta}$ kann zur Transitionsrelation

$$\vec{\Delta} : 2^{M_S} \rightarrow 2^{M_S},$$

wobei

$$\vec{\Delta}(M) = \bigcup_{t \in T} \vec{\delta}(M, t)$$

des Netzes generalisiert werden, indem in dieser Relation alle Transitionen des B/E-Systems eingebettet werden. $\vec{\Delta}$ transformiert eine Menge von Markierungen in eine neue Menge von Markierungen, die durch das Schalten einer Transition in einem Schritt erreicht werden können.

6.1.3 Symbolische Transitionssysteme (STS)

Der symbolische Transitionsgraph eines B/E-Systems, der die Basis für das symbolische Model Checking darstellt, kann implizit durch die symbolische Transitionsrelation und die charakteristische Funktion der Startmarkierung des Netzes repräsentiert werden.

Um die symbolische Transitionsrelation eines B/E-Systems zu generieren, muß eine boolesche Kodierung der Stellen des Netzes vorgenommen werden. Hierbei kann bei der gegebenen Klasse von Netzen jede Stelle durch eine einzige boolesche Variable kodiert werden, d.h. die Menge der zur Kodierung notwendigen Variablen V ist gleichmächtig S, d.h. $|V| = |S|$. Somit hat jede Stelle $s_i \in S$ eine ihr zugeordnete boolesche Variable $v_i \in V$. Da in der Transitionsrelation Aussagen über den Folgewert einer Stelle gemacht werden sollen, wird eine Verdoppelung des booleschen Domains notwendig. Die Menge der hierfür notwendigen binären Variablen wird mit V' gekennzeichnet und besitzt für jedes $v_i \in V$ ein Äquivalent $v_i' \in V'$, welches den Folgewert der Stelle nach dem Schalten einer Transition widerspiegelt.

Für das Beispiel in Abbildung 6.1 sind die Mengen der Variablen wie folgt gegeben

$$V = \{v_1, v_2, v_3, v_4, v_5\} \quad V' = \{v'_1, v'_2, v'_3, v'_4, v'_5\}$$

Hierauf aufbauend kann nun die Transitionsrelation bei einem fest vorgegebenen Netz N für jede Transition $t \in T$ bestimmt werden. Hierzu werden die folgenden charakteristischen Funktionen benötigt:

$$\begin{aligned} E_t &= \bigwedge_{s_i \in {}^\cdot t} v_i \\ Pred_t &= \bigwedge_{s_i \in {}^\cdot t} \bar{v}'_i \\ Succ_t &= \bigwedge_{s_i \in t^\cdot} v'_i \\ U_t &= \bigwedge_{s_i \notin ({}^\cdot t \cup t^\cdot)} v_i \Leftrightarrow v'_i \end{aligned}$$

Mit diesen charakteristischen Funktionen kann nun die Transitionsrelation R des Netzes auf den booleschen Mengen V und V' gebildet und als ROBDD repräsentiert werden:

$$R = \bigvee_{t \in T} E_t \wedge Pred_t \wedge Succ_t \wedge U_t$$

E_t selektiert aktive Transitionen im aktuellen Zustand. $Pred_t$ und $Succ_t$ charakterisieren das Tokenspiel der Transition t, und U_t die Eigenschaft, daß auf den von t unabhängigen Stellen des Netzes keine Veränderung eintritt.

Die Menge der Startmarkierungen S_0 kann auch als charakteristische Funktion über V mittels eines ROBDDs dargestellt werden, so daß das gesamte B/E-System symbolisch kodiert wird.

6.2 Symbolische Transitionssysteme für AADL Petri-Netze

AADL Petri-Netze besitzen im Gegensatz zu B/E-Systemen einen zugeordneten getypten Datenraum, dessen Zustand neben der jeweiligen Markierung des Netzes mit in den globalen Zustand des AADL Petri-Netzes eingeht (siehe Definition 4.5). Durch die Trennung von Kontroll- und Datenaspekten in AADL Petri-Netzen können für die verschiedenen Strukturen jeweils optimierte Compilationsschemata zur symbolischen Repräsentation angewendet werden, die eine sehr kompakte symbolische Darstellung ermöglichen.

6.2.1 Kodierung des AADL Datenraums

Da ein endliches Transitionssystem erstellt werden soll, dürfen in der Spezifikation des Datenraums nur endliche Datentypen und nichtrekursive Funktionen referenziert werden. D.h. als zulässige Typen werden nur Aufzählungsdatentypen, Subranges, Arrays mit endlichen Indexmengen und Records als Datenstrukturen zugelassen. Nur für einen so eingeschränkten Datenraum kann dann eine vollständig durch ROBDDs repräsentierte Struktur zur Verfügung gestellt werden [PH94, HP94, Her92].

Um die Übersetzung von AADL Petri-Netzen in eine ROBDD-Darstellung zu beschreiben, wird zuerst auf die symbolische Kodierung des Datenraums eingegangen, bevor die Kodierung der Kontrollaspekte beschrieben werden kann.

Datenraum

Die Grundelemente des Datenraums lassen sich auf getypte *input* und *lokale* Variablen reduzieren, da lokale und *output* Variablen keinen Unterschied in der Behandlung innerhalb des Transitionssystems erfahren.[1] Input Variablen der Menge I_{AADL} können von der Umgebung beschrieben werden. Daraus folgt, daß sie ihren Wert zu jeder Zeit ändern können.

Lokale Variablen werden in zwei Mengen V_{AADL} und V'_{AADL} unterteilt, wobei V'_{AADL} das Bild von V_{AADL} ist, in dem alle Variablen aus V_{AADL} ein gestrichenes „Aquivalent“ besitzen. Der Wert jede gestrichene Variable $v' \in V'_{AADL}$ repräsentiert bei diesem Ansatz den neuen Wert der Variablen $v \in V_{AADL}$ im nächsten Zustand. Input Variablen der Menge I_{AADL} werden jederzeit von der Umgebung beschrieben. Aus diesem Grund entfällt für sie eine Darstellung als gestrichene Variablen. Ein für die ROBDD-Generierung geeigneter Datenraum läßt sich nun wie folgt angeben:

Definition 6.1 Datenraum.

Ein Datenraum ist eine Struktur $\Upsilon = (DT_{AADL}, I_{AADL}, V_{AADL}, V'_{AADL}, init)$ *mit*

- DT_{AADL} *ist eine endliche Menge von endlich repräsentierbaren Datentypen,*
- I_{AADL} *ist eine endliche Menge von Input Variablen,*
- V_{AADL} *ist eine endliche Menge von lokalen Variablen,*

[1] Um im folgenden Text eine Unterscheidung zwischen den Variablen des AADL Datenraums des Petri-Netzes im Gegensatz zu den binären Variablen der symbolischen, BDD basierten Darstellung zu erhalten, werden Mengen jeweils mit $_{AADL}$ bzw. $_{BDD}$ indizert.

- *V'_{AADL} ist eine endliche Menge von gestrichenen, lokalen Variablen*
- *init ist eine Funktion, die jeder lokalen Variablen einen initialen Wert bzgl. ihres Datentyps zuordnet.* ∘

Um eine symbolische Kodierung des AADL Datenraums zu erlangen, muß eine binären Repräsentation jeder AADL Variable erzeugt werden. In Abhängigkeit von der auf verschiedenen Heuristiken basierenden Variablenordnung δ wird der binäre Domain (BD) durch zwei endliche Mengen von BDD-Variablen V_{BDD} und V'_{BDD} ($BD = V_{BDD} \cup V'_{BDD}$) analog zu V_{AADL} und V'_{AADL} dargestellt. Dabei definiert β eine Abbildung, die jeder AADL Variablen ihre zur Kodierung notwendigen BDD Variablen zuordnet. Desweiteren wird zwischen einer BDD-Variablen $v_i \in V_{BDD}$ und dem Index $i \in V_{index}$ der Variablen unterschieden; V_{index} (V'_{index}) bezeichne in diesem Zusammenhang die Menge der Indizes aller Variablen $v_i \in V$ ($v'_i \in V'$). Sei B eine Menge von BDD-Variablen. Dann beschreibt $ROBDD(B)$ die Menge aller ROBDDs, die nur Variablen aus B referenzieren.

Die Größe eines ROBDDs hängt von der gewählten BDD Variablenordnung δ ab. Um Model Checking für große, industrielle Designs überhaupt zu ermöglichen, ist das Auffinden einer nahezu optimalen Variablenordnung δ eine Grundvoraussetzung. [2] Das Problem hierbei ist nicht nur eine kleine Repräsentation des Transitionssystems sondern auch die Größe der temporären ROBDDs, welche während des Model Checkings als Charakterisierung von temporären Zustandsmengen, z.B. der bei Fixpunktberechnung, aufgebaut werden. Dynamische Reorganisationsverfahren der Variablenordnung zur Laufzeit des Model Checkers können in einigen Fällen helfen. Ihr Vorteil geht doch meistens durch einen enormen Zeitaufwand für ihre Berechnung wieder verloren [FYBSV93, Rud93]. Das Auffinden einer initialen, globalen Variablenordnung δ wird hierdurch unbedingt notwendig. Sie kann u.a. durch eine Berechnung von Abhängigkeitsrelationen und von Datenflußanalysen festgelegt werden [Pö95]. Desweiteren kann eine Reachability-Berechnung als Analyse durchgeführt werden. Da auch sie auf einer Fizpunktberechnung basiert, kann eine Berechnung der Variablenordnung auf dieser Basis ggf. auch gute Ergebnisse liefern.

Notation

Um die Generierung von symbolischen Transitionssystemen zu beschreiben, werden

[2] Zwar ermöglichen dynamische Reordering Techniken eine Optimierung der Darstellung, jedoch ist ihr Laufzeitverhalten ggf sehr groß. Des weiteren sind die intermediären Datenstrukturen des Reorderings oft sehr komplex und erzwingen weitere hohe Laufzeiten des BDD-Toolsets.

noch folgende Hilfsfunktionen benötigt:

Basierend auf dem AADL Datenraum Υ und dem binärem Domain BD werden zwei Funktionen zur ROBDD-basierten Handhabung von booleschen Ausdrücken ($expr2bdd$) und Datentransformationen ($trans2bdd$) eingesetzt:

$$expr2bdd_{\Upsilon,\beta} : Bexpr \rightarrow ROBDD(BD)$$

berechnet einen einzigen ROBDD für einen gegebenen booleschen Ausdruck, der genau dann zu 1 evaluiert wird, wenn der Ausdruck $boolExpr$ im aktuellen Zustand von BD zu $true$ evaluiert.

$$trans2bdd_{\Upsilon,\beta} : Cont \rightarrow 2^{(V'_{index} \times ROBDD(BD))}$$

berechnet zu einer Datentransformation eine Menge von Paaren (i, bdd_i), welche symbolisch die bitweise Datentransformation charakterisieren; hierbei kennzeichnet i den Index der BDD Variablen $v'_i \in V'$ und $bdd_i \in ROBDD(BD)$ den ROBDD, welcher den neuen Wert nach der Transformation beschreibt. Die Funktion

$$valid_{\Upsilon,\beta} : Cont \rightarrow ROBDD(BD)$$

ordnet einer Datentransformation einen ROBDD zu, welcher charakterisiert, ob sie unter der aktuellen Belegung der binären Variablen gültig ist, d.h. daß keine Laufzeitfehler durch Indexverletzung erzeugt werden. Dies wird u.a. notwendig durch die binäre Kodierung von Datentypen[3][Her92].

Als abkürzende Schreibweise werden folgende Funktionen definiert:

$$writeSet_{\Upsilon,\beta} : 2^{(V'_{index} \times ROBDD(BD))} \rightarrow 2^{V_{index}},$$

eine Funktion, welche die Indexenge der durch eine Transformation veränderten binären Variablen berechnet,

$$readSet_{\Upsilon,\beta} : 2^{(V'_{index} \times ROBDD(BD))} \rightarrow 2^{BD},$$

eine Funktion, welche zu einer Datentransformationen die Menge von BDD-Variablen berechnet, von denen sie abhängt, und die Funktion

$$sb : V'_{index} \times 2^{(V'_{index} \times ROBDD(BD))} \rightarrow ROBDD(BD),$$

[3]Eine binäre Kodierung einer Variablen x mit Subranges $0 \ldots 8$ benötigt 4 binäre Variablen zur Kodierung. Dies würde aber dann eine Variable vom Subrange von $0 \ldots 15$ implizit definieren. Als Gültigkeitsbedingung könnte dann die Bedingung $x \leq 8$ angegeben werden.

die eine Projektion auf einen einzelnen Transformations-BDD bezüglich eines Index durchführt. Sei e eine AADL Datentransformation, dann beschreibt $writeSet(trans2bdd_{\Upsilon,\beta}(e))$ die Menge von BDD Variablen Indizes, die die linke Seite der Transformation repräsentieren. In

$$bdd_i = sb(i, trans2expr_{\Upsilon,\beta}(e)),$$

wobei $i \in writeSet(trans2bdd_{\Upsilon,\beta}(e))$ gilt, charakterisiert bdd_i den Wert der binären Variablen $v'_i \in V'$. Die Funktion

$$bdd : BD \rightarrow ROBDD(BD)$$

bildet eine Variable auf einen ROBDD ab.

6.2.2 Kodierung der AADL-Kontrollstruktur

Eine Kodierung der Kontrollstruktur des Netzes kann auf zwei Arten vorgenommen werden:

1. Die Markierung der Kontrollflußstellen des Netzes sowie die Transitionsrelation werden analog der zu B/E-Systemen kodiert. Sie benötigt dann zur Kodierung $2 * |S_{Control}|$ binäre Variablen.
2. Alle möglichen Markierungen der Kontrollflußstellen des Netzes werden explizit in einem Fallgraph repräsentiert, welcher erst in einem zweiten Schritt symbolisch kodiert wird. Zur Kodierung werden dann nur $2 * log_2(|V|)$ Variablen benötigt, wobei $|V|$ im *worst case* $2^{|S_{Control}|}$ Zustände zur Kodierung benötigt werden und welche dann die gleiche Komplexität wie Ansatz 1 besitzt.

Die erste Kodierungsmöglichkeit stellt die logische Konsequenz aus der Kodierung der B/E-Systeme dar und ist für parallele, unabhängige Systeme adäquat, obwohl sie den Nachteil der hohen Anzahl von notwendigen booleschen Variablen zur Kodierung impliziert. Soll jedoch ein System mit relativ wenigen nebenläufigen Prozessen, welche aus längeren sequentiellen Berechnungsschritten aufgebaut sind, kodiert werden, so ist der zweite Ansatz bezüglich der notwendigen binären Variablen vorzuziehen.

Die aus AADL Spezifikationen, welche von einem Menschen (d.h. nicht maschinell) erzeugt wurden, abgeleiteten Petri-Netze erfüllen meistens diese gutartige Eigenschaft und der Aufbau der fallgraphähnlichen Struktur, basierend auf der Markierung der Kontrollflußstellen, ist keine speicherplatzkritische Phase der Übersetzung.

Das AADL Petri-Netz agiert letztlich nur wie ein abstrakter Programmzähler, der, in Abhängigkeit von Bedingungen, Datentransformationen selektiert und ausführt. Die Kontrollstruktur kann deshalb in einen expliziten *Kontrollautomaten (KA)* übersetzt werden. Dies führt zu einer Abstraktion von der Markierung der Stellen im Netz und kann deshalb zu einer kompakteren Darstellung führen. Die Transitionen des KA sind mit einer Aktivierungsbedingung und einer Menge von Datentransformationen in symbolischer Form annotiert, die sich von der Beschriftung des AADL Petri-Netzes ableitet. Die ROBDD Realisierung der Aktivierungsbedingung ψ einer Transition $t \in T$ des AADL Petri-Netzes ist durch

$$\psi(t) = \bigwedge_{s \in {}^{\cdot e}t \cap S_{Trigger}} expr2bdd_{\Upsilon,\beta}(\pi(s))$$

zu erlangen. Die Datentransformationen ϕ einer Transition $t \in T$ und die Berechnung ihrer Gültigkeitsbedingung ζ wird durch

$$\phi(t) = trans2bdd_{\Upsilon,\beta}(\Gamma(t))$$

und

$$\zeta(t) = \bigwedge_{trans \in \Gamma(t)} valid(\Gamma(t))$$

aufgebaut. Der Kontrollautomat für ein AADL Petri-Netz kann nun wie folgt beschrieben werden:

Definition 6.2 Kontrollautomat.

Der Kontrollautomat KA eines AADL Petri-Netzes N ist eine Struktur

$KA(N) = (S_{KA}, \psi, \phi, \zeta, T_{KA}, s_{init}, \Upsilon, \beta, BD)$ *mit*

- S_{KA} *ist eine endliche Menge von Zuständen* ($|S_{KA}| \ll 2^{S_{Con}}$),
- $\psi \subseteq ROBDD(BD)$ *ist eine Menge von Aktivierungsbedingungen,*
- $\phi \subseteq 2^{(V'_{index} \times ROBDD(BD))}$ *ist eine Menge von Datentransformationen,*
- $\zeta \subseteq ROBDD(BD)$ *ist eine Menge von Gültigkeitsbedingungen,*
- $T_{KA} \subseteq S_{KA} \times S_{KA} \times \psi \times \phi \times \zeta$ *ist eine endliche Menge von Transitionen,*
- $s_{init} \in S_{KA}$ *ist ein Startzustand des Kontrollautomaten,*
- Υ *ist ein Datenraum (siehe Definition 6.1),*
- β *ist eine binäre Kodierung der Typen und Variablen von* Υ
- $BD = (V \dot{\cup} V' \dot{\cup} I)$, *der boolesche Domain als Kodierung der Variablen von* Υ.
- δ *ist eine totale Ordnungsrelation auf* BD ∘

Ein Kontrollautomat, der aus einem von einer AADL Spezifikation abgeleiteten AADL Petri-Netz entstanden ist, hat folgende Lebendigkeitseigenschaft (input-enabledness): Alle von einem Zustand ausgehenden Kanten sind mit wohlgeformten Aktivierungsbedingungen beschriftet, so daß immer mindestens eine von ihnen in einem gültigen Zustand des Datenraums zu *true* evaluierbar ist, wodurch immer ein Fortschreiten in einen neuen Zustand gegeben ist. Ein Kontrollautomat wird als *deterministisch* bezeichnet, wenn immer nur genau eine dieser Bedingung zu *true* evaluiert werden kann.

6.2.3 Relationale Symbolische Transitionssysteme

Auf der Struktur des Kontrollautomaten aufbauend kann ein relationales symbolisches Transitionssystem als Zielstruktur für die Eingabe in einen symbolischen Model Checker einfach abgeleitet werden:

Definition 6.3 Relationales Symbolisches Transitionssystem (RSTS).
Ein relationales symbolisches Transitionssystem ist eine Struktur

$RSTS = (V_{BDD_{input}}, V_{BDD_{state}}, V'_{BDD_{state}}, \delta, S_{start}, NextStep)$, *mit*

- $V_{BDD_{input}}$ *ist eine endliche Menge von booleschen Input-Variablen,*
- $V_{BDD_{state}}$ *ist eine endliche Menge von booleschen Zustands-Variablen,*
- $V'_{BDD_{state}}$ *ist eine endliche Menge von gestrichenen, booleschen Zustands-Variablen* $v' \in V'_{BDD_{state}}$*, die den neuen Wert der Zustands-Variablen* $v \in V_{BDD_{state}}$ *beschreiben,*
- δ *ist eine totale Ordnungsrelation auf* $V = V_{BDD_{input}} \dot{\cup} V_{BDD_{state}} \dot{\cup} V'_{BDD_{state}}$,
- $S_{start} \in ROBDD(V_{BDD_{state}})$ *ist ein ROBDD, der einen initialen Zustand (ggf. mehrere initiale Zustände) von* $V_{BDD_{state}}$ *charakterisiert,*
- $NextStep \subseteq \{bdd \mid ROBDD(V_{BDD_{input}} \cup V_{BDD_{state}} \cup V'_{BDD_{state}})\}$ *ist eine Menge von relationalen NextStep-ROBDDs, die implizit als Disjunktion betrachtet wird.*

$\circ$

Im folgenden Abschnitt des Kapitels wird die Konstruktion der Teilkomponenten eines RSTS für einen gegebenen Kontrollautomaten beschrieben.

Die Mengen V_{BDD} und V'_{BDD} des booleschen Domains BD werden um die binären Variablen erweitert, die zur Kodierung der Zustände des KA notwendig sind. Um sie zu kodieren, werden $j = \lceil log_2(|S_{KA}|) \rceil$ boolesche Variablen benötigt. Diese neuen

Variablen zusammen mit V_{BDD} beschreiben die Menge $V_{BDD_{state}}$. Ihre gestrichene Version vereinigt mit V'_{BDD} ergibt die Menge $V'_{BDD_{state}}$. Die Menge $V_{BDD_{input}}$ bleibt durch die Kodierung der Kontrollzustände unverändert.

Die existierende globale Variablenordnung δ, die aus der Kodierung des AADL Datenraums abgeleitet wurde, wird um die Variablen erweitert, die zur Kodierung des KA notwendig sind. Da die Kontrollstruktur des KA wie ein abstrakter Programmzähler agiert, welcher die Datentransformationen in Abhängigkeit von gegebenen Bedingungen selektiert, haben sie die größte Priorität in der globalen Variablenordnung. Ihre gestrichenen und ungestrichenen Variablen werden dabei verschachtelt angeordnet. Alle anderen Anordnungsverfahren führen zu weit größeren Repräsentationen der ROBDDs des Transitionssystems [PH94].

Die symbolische Repräsentation des Startzustandes S_{start} ist eine Konjunktion der symbolischen Kodierung der Initialisierungsausdrücke des AADL Datenraums und der Kodierung des Startzustandes des KA:

$$S_{start} = sbdd(s_{init}) \bigwedge_{v \in V_{AADL}} expr2bdd_{\Upsilon,\beta}(v = init(v))$$

Hierbei beschreibt $sbdd : S_{KA} \rightarrow ROBDD(V_{BDD_{state}})$ eine Funktion, die jedem Zustand des KA einen ROBDD zuordnet, der ihn eindeutig charakterisiert.

Die wichtigste Komponente des RSTS ist die Menge der *NextStep*-Relationen zwischen den Zuständen der binären Variablen von V'_{state} in Abhängigkeit von V_{input} und V_{state}. Hierbei wird jede Transition $(s_x, s_y, \psi_i, \phi_i, \zeta_i) \in T_{KA}$ durch einen separaten NextStep-ROBDD $NextStep_t$ repräsentiert, d.h. $|T_{KA}| = |NextStep|$. Diese Partitionierung der Relation hat sich als beste Grundlage für ein relationales Model Checking herausgestellt [PH94]. Die vollständige Transitionsrelation des RSTS ergibt sich dann als

$$NextStep = \bigvee_{t \in T_{KA}} NetxStep_t.$$

Für die Darstellung der NextStep-Relationen werden noch zwei Hilfsfunktionen benötigt, die einen Zustand $s \in S_{KA}$ durch einen ROBDD eindeutig charakterisieren. Die Funktion

$$actualState_{\Upsilon,\beta} : S_{KA} \rightarrow ROBDD(V_{state})$$

charakterisiert einen Zustand mit ungestrichenen Variablen, wobei die Funktion

$$nextState_{\Upsilon,\beta} : S_{KA} \rightarrow ROBDD(V'_{state})$$

dies mit gestrichenen Variablen durchführt. Die NextStep Relation kann durch

$$NextStep = \bigvee_{(s_x,s_y,\psi,\phi,\zeta)\in T_{KA}} actualState(s_x) \wedge nextState(s_y) \wedge \psi \wedge \zeta \wedge dataTrans(\phi),$$

mit

$$dataTrans(\phi) = (\bigwedge_{i\in writeSet(\phi)} sb(i,\phi) \Leftrightarrow bdd(v_i')) \wedge (\bigwedge_{j\in V_{index}-writeSet(\phi)} bdd(v_j) \Leftrightarrow bdd(v_j'))$$

angegeben werden.

6.2.4 Funktionale Symbolische Transitionssysteme

Die Transformation eines KA in ein funktionales symbolisches Transitionssystem basiert auf der Voraussetzung, daß der KA deterministisch ist, d.h. daß die Bedingungen der Transitionen, die von einem Zustand ausgehen, exklusiv sind. Diese Eigenschaft kann durch einen Algorithmus mit linearer Laufzeit für einen gegebenen KA berechnet werden. Im folgenden Abschnitt des Kapitels wird die BDD basierte Kodierung der Kontrollstruktur des KA und die Integration der Datentransformationen in ein funktionales Transitionssystem beschrieben. Die Zielstruktur ist ein:

Definition 6.4 Funktionales Symbolisches Transitionssystem (FSTS).

Ein Funktionales Symbolisches Transitionssystem ist eine Struktur

$RSTS = (V_{BDD_{input}}, V_{BDD_{state}}, V'_{BDD_{state}}, \delta, S_{start}, NextStep)$, *mit*

- $V_{BDD_{input}}$ *ist eine endliche Menge von booleschen Input-Variablen,*
- $V_{BDD_{state}}$ *ist eine endliche Menge von booleschen Zustands-Variablen,*
- $V'_{BDD_{state}}$ *ist eine endliche Menge von gestrichenen, booleschen Zustands-Variablen* $v' \in V'_{BDD_{state}}$*, die den neuen Wert der Zustands-Variablen* $v \in V_{BDD_{state}}$ *beschreibt,*
- δ *ist eine totale Ordnungsrelation auf* $V = V_{BDD_{input}} \dot\cup V_{BDD_{state}} \dot\cup V'_{BDD_{state}}$*,*
- $S_{initial} \in ROBDD(V_{BDD_{state}})$ *ist ein ROBDD der den initialen Zustand von* $V_{BDD_{state}}$ *charakterisiert,*
- $NextStep : V'_{BDD_{state}} \rightarrow ROBDD(V_{input} \dot\cup V_{state})$ *assoziiert einen NextStep-BDD zu jeder Variablen* $v' \in V'_{BDD_{state}}$ ∘

Die Generierung eines FSTS für einen KA ist analog zu der Generierung eines RSTS, bis auf die Erzeugung der $NextStep$-Funktionen, die den Wertübergang der einzel-

nen binären Variablen beschreiben. Für jede Variable $v_i' \in V_{BDD'_{state}}$ muß eine Funktion f_i generiert werden, welche den Wert der Variablen im nächsten Zustand in Abhängigkeit vom aktuellen Zustand und den Input Variablen berechnet:

$$v_i' = f_i(V_{input} \cup V_{state})$$

Hierbei wird implizit davon ausgegangen, daß die i-te $NextStep$-Funktion mittels f_i als

$$NextStep_i = (v_i' \Leftrightarrow f_i)$$

definiert werden kann. Die vollständige $NextStep$-Funktion kann dann als Komposition der Teilfunktionen angesehen werden. Da die Umkodierung der einzelnen Teilfunktionen unktionen zu der globalen Funktion $NextStep$ im Falle des funktionalen Symbolischen Model Checkings redundant ist und nur eine Umkodierung darstellt, wird im folgenden bereits die Funktion f_i als $NextStep_i$ bezeichnet.

Um funktionales Model Checking zu unterstützen, muß jede einzelne Funktion total sein.[4] Die symbolische Repräsentation hat diese Eigenschaft sicherzustellen, wobei drei Probleme besonders beachtet werden müssen:

1. Durch die logarithmische Kodierung von S_{KA} können mehr Zustände beschrieben werden als S_{KA} Elemente hat, d.h. $2^j > |S_{KA}|$.
2. Die NextStep-Funktion muß auf Zustände anwendbar sein, die keine ausgehenden Transitionen besitzen.
3. Es muß ein expliziter Fehlerzustand verwaltet werden, um Laufzeitfehler bei Datentransformationen erkennen zu können.

Beispiel 6.1 Erweiterung der Transitionsrelation *Abbildung 6.2 zeigt auf der linken Seite die Struktur eines KA. Die binäre Kodierung der drei Zustände des KA mit zwei Variablen führt zu einer Kodierung von vier Zuständen in FSTS. Um das totale Transitionsystem mit dem neuen Zustand bilden zu können, müssen zwei weitere Transitionen eingefügt werden. Es sind Schleifen für die Zustände 3 und 4, die auf sich selber referenzieren, und mit den Bedingung true und der Identität als Datentransformation annotiert werden. Der neue Zustand „4“ stellt für diesen Zustand jedoch keinen Overhead dar, da er als expliziter Fehlerzustand angesehen werden kann, in den dann implizit während eines Model Checking Laufes bei einer Fehlersituation verzweigt werden kann. Dies führt dann zur sofortigen Terminierung*

[4] Das zu transformierende AADL Petri-Netz muß deshalb deterministisch bzgl. seines Kontrollflusses sein.

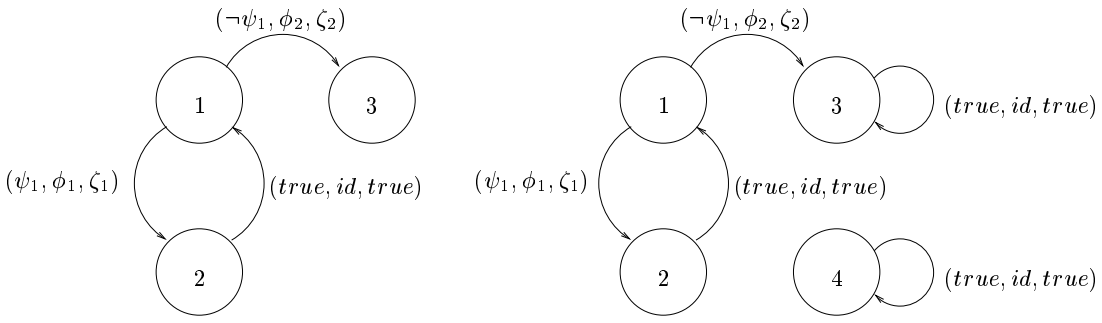


Abbildung 6.2: Erweiterung der Transitionsrelation

einer Fixpunktberechnung, da kein weitere Kontrollzustand von diesem Zustand aus angenommen werden kann und auch keine weitere Datentransformation vorgenommen wird. ○

Drei aufeinanderfolgende Compilationsschritte müssen durchgeführt werden, um die Menge von NextStep-Funktionen für einen KA zu erzeugen:

1. Jede Transition des KA wird in eine symbolische Struktur übersetzt und verändert die Menge der Funktionen $\tilde{f}$. Für jede Zustands-Variable wird die zugehörige charakteristische NextStep Funktion aufgebaut, die angibt, welchen Wert sie in Abhängigkeit vom aktuellen Zustand im nächsten Zustand des Systems annehmen wird.
   Die Konstruktion von $\tilde{f}$ ist jedoch unterschiedlich für Variablen, die den Kontroll- bzw. den Datenfluß des KA kodieren. Durch die statische Kontrollstruktur des KA kann schon zur Compilezeit des Systems mehr Information in $\tilde{f}$ integriert werden, als es für die allgemeineren Datentransformationen möglich ist.

   Kontrollflußkodierung
   Für die i-te binäre Variable v_i', die zur Kodierung eines Zustandes des KA benötigt wird, wird $\tilde{f}_{v_i'}$ inkrementell durch die Interpretation jeder Transition des KA aufgebaut. Die Kodierung ist hierbei abhängig vom Zielzustand s_y der gerade untersuchten Transition $(s_x, s_y, \psi_i, \phi_i, \zeta_i) \in T_{KA}$ und ihrer zugeordneten Bedingung ψ. Es werden nur die Funktionen $\tilde{f}$ verändert, für die die Zustandsvariable v_i mit *true* bei der binären Kodierung des Zielzustandes belegt wird.[5] Diese Eigenschaft

[5] Dies verhält sich analog dem Aufbau von booleschen Formeln aus einer booleschen Wahrheits-

kann durch die Funktion

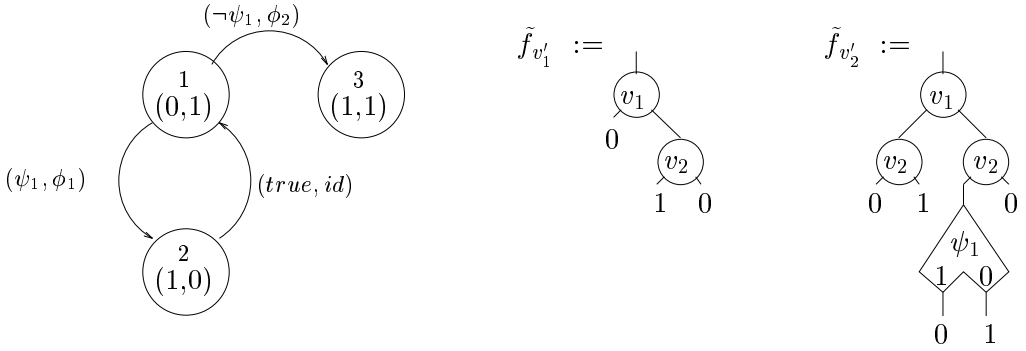
$$binary : S_{KA} \times \{1, \ldots, \lceil log_2(|S_{KA}|)\rceil\} \rightarrow \{0, 1\}$$

repräsentiert werden. In diesem Fall wird $\tilde{f}$ um die Konjunktion des ROBDD $sbdd(s_x)$, welcher den Startzustand kodiert, dem ROBDD ψ, der die Bedingung der Transition charakterisiert, und der Gültigkeitsbedingung ζ erweitert.

$$\tilde{f}_{v_i'} := \bigvee_{\substack{(s_x, s_y, \psi, \phi, \zeta) \in T_{KA} \wedge \\ binary(s_y, i) = 1}} sbdd(s_x) \wedge \psi \wedge \zeta$$

Wird die Bedingung ζ nicht erfüllt, so wird der Wert der NextStep Funktion beim Model Checking zu $false$ evaluiert. Der Fehlerzustand hat somit die Kodierung 0 zu tragen und wird als einzelner, isolierter Zustand im Kontrollautomaten verwaltet.

Beispiel 6.2 Symbolische Kodierung der Transitionsrelation



Transitionsrelation eines KA

Symbolische Repräsentation von Zwischenfunktionen

Abbildung 6.3: Symbolische Kodierung der Transitionsrelation

Die binäre Kodierung der drei Zustände des KA in Abbildung 6.3 erfolgt durch zwei binäre Variablen (v_1, v_2). Die Kodierung der Transitionen des KA (hier der besseren Übersicht wegen ohne Gültigkeitsbedingung ζ) führt zu den beiden booleschen Funktionen $\tilde{f}_{v_1'}, \tilde{f}_{v_2'}$. Befindet sich das System z.B. im Zustand 1*, so ist der Wert der Variablen v_1 im nächsten Zustand unabhängig von dem Ergebnis der Bedingung ψ_1 und wird immer zu* 1 *evaluiert. Der Folgezustand der Variablen v_2 dagegen ist die Negation des Resultats der Bedingung ψ_1, wie er durch $\tilde{f}_{v_2}$ charakterisiert wird.* ∘

tabelle, bei der auch nur „Zeilen“ berücksichtigt werden, deren Funktionswert 1 ist.

Datentransformationskodierung
Für Zustands-Variablen, die den Datenraum kodieren, können die $\tilde{f}$ Funktionen nicht in gleicher Weise aufgebaut werden. Der Wert dieser Variablen im nächsten Zustand hängt von dem Kontrollzustand, der Aktivierungsbedingung und der Datentransformation ab. Aus diesem Grund muß eine Konjunktion der drei ROBDDs aufgebaut werden. Für eine binäre Variable $v'_j \in V'_{BDD_{state}}$ ist die Funktion $\tilde{f}_{v'_j}$ wie folgt gegeben

$$\tilde{f}_{v'_j} := \bigvee_{(s_x,s_y,\psi,\phi,\zeta)\in T_{KA}} sbdd(s_x) \wedge \psi \wedge sb(j,\phi)$$

2. In diesem Schritt der Compilation wird eine charakteristische Funktion $\hat{f}$ aufgebaut, die alle Zustände beschreibt, die mindestens eine ausgehende Kante besitzen:

$$\hat{f} := \bigvee_{(s_x,s_y,\psi,\phi,\zeta)\in T_{KA}} sbdd(s_x)$$

Beispiel 6.3 Symbolische Kodierung von $\hat{f}$

$\hat{f}$:=

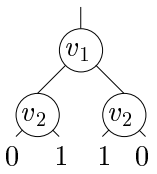


Abbildung 6.4: Symbolische Kodierung von $\hat{f}$

Für den KA aus Abbildung 6.3 ist der ROBDD, der alle Zustände mit ausgehenden Kanten charakterisiert in Abbildung 6.4 angegeben. In diesem Beispiel ist es die Menge der Zustände 1 und 2. ◦

3. Im letzten Schritt werden die Funktionen $\tilde{f}$ und $\hat{f}$ zur eigentlichen NextStep Funktion zusammengefaßt. Dies geschieht wiederum für jede Variable $v' \in V'_{BDD_{state}}$, wobei die Funktion $NextStep_{v'}$ erzeugt wird. Sie beschreibt den neuen Wert von v' mit folgender Pragmatik:
Entweder kann das Transitionssystem in einen neuen Zustand übergehen ($\hat{f} = true$) und $\tilde{f}_{v'}$ beschreibt den neuen Wert von v' oder das Transitionssystem verbleibt im aktuellen Zustand ($\neg\hat{f} = true$) und der neue Wert der Variablen v' ergibt sich durch den aktuellen Wert von v. Da

$$\tilde{f} \Rightarrow \hat{f}$$

kann die Funktion $NextStep_{v'}$ wie folgt beschrieben werden:

$$NextStep_{v'} := \tilde{f}_{v'} \vee (\neg \hat{f} \wedge bdd(v))$$

Beispiel 6.4 Symbolische Kodierung der $NextStep$ Funktionen

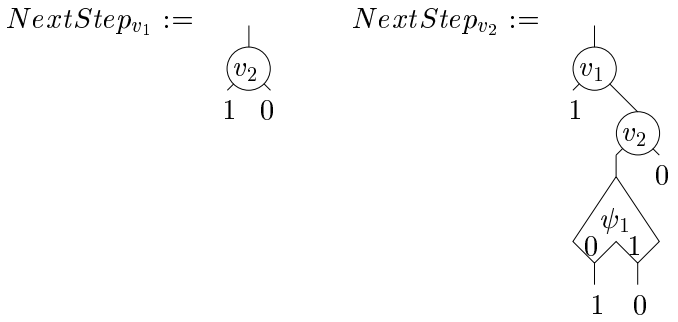


Abbildung 6.5: Symbolische Kodierung der $NextStep$ Funktionen für (v_1, v_2)

Abbildung 6.5 zeigt die NextStep Funktionen für die Zustands-Variablen, die die Kontrolle des KA aus Abbildung 6.3 kodieren. ◦

Mit der gerade präsentierten Methode zur Ableitung eines symbolischen Transitionssystems für einen Kontrollautomaten ist nun das symbolische Transitionssystem einer AADL Spezifikation für Symbolisches Model Checking vorbereitet.

6.3 Symbolische Transitionssysteme für VHDL/S

Um die Relevanz dieses Ansatzes auch für ein industrielles Umfeld nachzuweisen, sind die bisher vorgestellten Ansätze im Rahmen des ESPRIT Projektes FORMAT für die Hardwarebeschreibungssprache VHDL bzw. VHDL/S StateCharts realisiert worden.

Die Besonderheit dieser beiden Sprachen ist das Konzept der Sichtbarkeit von Modifikationen auf Variablen ihrer Interfacebeschreibungen. Nur zu ganz bestimmten Zeitpunkten, den *wait*-Konfigurationen, kann die Umgebung durch Sicht auf das Interface Wertveränderungen beobachten. Hieraus folgt für die Modellgenerierung, daß alle Zustände zwischen diesen von der Umgebung beobachtbaren Zuständen abstrahiert (zusammengefaltet bzw. kolabiert) werden können. Dies führt zu Modellen mit weniger Zuständen, was eine signifikante Beschleunigung für das Model Checking zur Folgen hat.

Bevor die zur Realisierung notwendigen Transformationen vorgestellt werden, soll auf die Struktur der AADL Petri-Netze eingegangen werden, die bei der Übersetzung von VHDL bzw. VHDL/S StateCharts aufgebaut werden. Da die Modellgenerierung auf dieser Zwischenrepräsentation aufgebaut ist, muß eine Kodierung von ihr abhängig sein, d.h. realisierte Kontrollkonzepte auf der Petri-Netzebene werden sich auf Modellebene widerspiegeln.

6.3.1 VHDL

VHDL ist eine standardisierte Hardwarebeschreibungssprache [IEE87], welche von allen führenden CAD Tool-Herstellen unterstützt wird. Eine detaillierte Einführung in den Sprachdesign und die Anwendung von VHDL für konkrete, komplexe Designs wird in [Ash90b, LSU89, DJS93c, IEE87] durchgeführt. Im folgenden Abschnitt soll nur kurz auf Basiskonstrukte der Sprache eingegangen werden, welche als relevant für die Konstruktion von endlichen Transitionssystemen angesehen werden.

VHDL stellt sich als eine komplexe Sprache mit einer Vielzahl von mächtigen Modularisierungsmöglichkeiten dar. Das an ADA angelehnte *package*-Konzept erlaubt eine modulare Spezifikation eines Designs und dessen Datentypen. Desweiteren wird auch das Konzept der Konfiguration unterstützt, welches einen inkrementellen Entwurf eines Gesamtdesigns erlaubt. Eine VHDL *Design Entity* besteht immer aus zwei Komponenten:

1. *Entity Deklaration:*
   Sie definiert die Komponenten des Interfaces und die initialen Werte von Variablen. Es kann formal als Tupel

   $$(I, O, V_{gen}, init)$$

   angegeben werde, wobei I und O disjunkte Menge von Input- und Outputports sind, V_{gen} eine Menge von *generics* ist und *init* eine Funktion, die jedem Outputport einen Wert zuweist.
2. *Architecture Body:*
   Er definiert das Verhalten der Komponente, wie es im Interface sichbar wird. Der Architecture Body kann entweder aus einer Struktur- oder einer Verhaltensbeschreibung bestehen. Eine Verhaltenbeschreibung kann wiederum durch mehrere parallele Prozesse gegeben sein, die aus sequentiellem Programmcode aufgebaut sind, der hier nicht detaillierter beschrieben werden soll (siehe [IEE87]).

Abbildung 6.6 zeigt den Aufbau der *Master-Slave*-Komponenten des *Request-Acknowledge* Protokolls. Tabelle 6.1 gibt eine mögliche VHDL-Implementierung der beiden Komponenten an. Im Gegensatz zu AADL erlaubt VHDL u.a. eine Definition

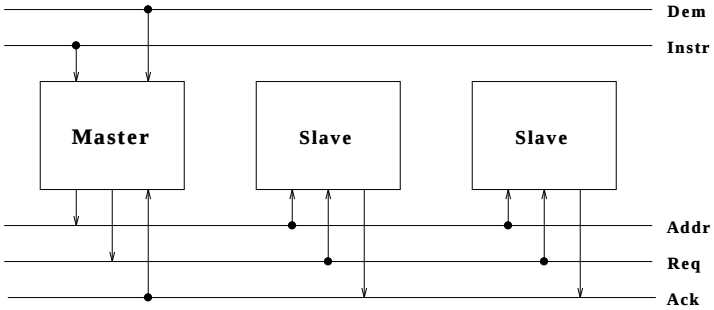


Abbildung 6.6: Master Slave Topologie

```
entity Master is
   port (Instr : in bit;
         Dem   : in bit;
         Addr  : out bit;
         Req   : out bit;
         Ack   : in bit);
end Master;

architecture behaviour of Master is
begin
   process
   begin
      wait on Dem,Ack
      until Dem = '1' and Ack = '0';
      Addr <= Instr
      Req <= '1';
      wait on Ack until Ack = '1'
      Req <= '0';
      wait on Dem until Dem = '0';
   end process;
end behaviour
```

```
entity Slave is
   generic (id : bit := '0');
   port (Addr : in bit;
         Req  : in bit;
         Ack  : out bit);
end Slave;

architecture behaviour of Slave is
begin
   process
   begin
      wait on Req until Req = '1';
      if (Addr = id) then
         Ack <= '1';
         wait on Req = '0'
         Ack <= '0';
      end if;
   end process;
end behaviour
```

Tabelle 6.1: VHDL Beschreibung des Master- und Slave-Moduls des Req-Ack Protokolls

von Zeitannotationen zu Datentransformationen.

6.3.1.1 Übersetzung von VHDL in AADL Petri-Netze

Die Beschreibung des VHDL Standards in [IEE87] gibt nur eine informelle Beschreibung eines Ausführungsmodells für VHDL an. Seine formale Semantik, welche alle Aspekte eines VHDL Designs vollständig abdeckt, ist noch nicht als Industriestandard festgelegt worden.

Eine formale Semantik im Kontext von interaktiven Theorembeweisern, z.B. HOL, wurde von [vT90, BPS91, BEP93] definiert, welche jedoch nicht adäquat für vollautomatische Beweisverfahren sind. Der Ansatz von [OC93] stützt seine Semantik dagegen auf Coloured Petri-Netze ab. Hierbei werden jedoch Datenaspekte von VHDL mit unendlichen Datentypen modelliert, so daß automatische Verfahren, basierend auf endlichen Automaten, nicht angewendet werden können. Die dieser Arbeit zu Grunde liegende Semantik von VHDL, d.h. die Generierung von Symbolischen Transitionssystemen, basiert auf der Arbeit von [Döh94], die sich an frühere Arbeiten von [DJS93b, DJS93a, DJS93c] anschließt. Im folgenden Abschnitt wird kurz auf die generelle Struktur eines AADL Petri-Netzes eingegangen, wie es bei der Übersetzung von VHDL inkrementell erzeugt wird.

Generierung des Datenraums

Eine von der VHDL-Implementierung abgeleitete Symboltabelle, eine Struktur, in der alle referenzierten Typen und Variablen aufgeführt sind, repräsentiert den Datenraum des AADL Petri-Netzes. In ihr werden Typen, lokale Variablen und Funktionen des Systems unter einem eindeutigen Bezeichner abgelegt und einem abstrakten Datentyp gleich zugreifbar gemacht. Für Ports werden komplexere Datenstrukturen, sog. *projected waveforms*, abgeleitet, die das zeitbehaftete Schreiben auf Ports widerspiegeln (siehe [Döh94]).

Generierung des Petri-Netzgerüstes

Die Struktur eines von einer VHDL-Implementierung abgeleiteten AADL Petri-Netzes, wie es in Abbildung 6.7 zu sehen ist, wird inkrementell aufgebaut. Ausgehend von der Übersetzung einzelner VHDL-Statements wird das sequentielle Teilnetz eines Prozesses durch Aneinanderkettung aufgebaut. Nachdem alle Prozesse in Teilnetze übersetzt wurden, findet eine sequentielle Aneinanderreihung statt. Hierbei wird zwischen der *globalen* und *lokalen* Kontrolle unterschieden. Jeder VHDL-Prozeß, angesteuert von der globalen Kontrolle, führt lokal seine Berechnungen durch, bis er auf

ein *wait*-Statement trifft und die globale Kontrolle an den folgenden VHDL-Prozeß weiterleitet. Nachdem alle Prozesse durchlaufen sind, befindet sich das System in einer *wait-Konfiguration*. Hiernach wird in der weiteren Netzstruktur ein Update auf die Projected Waveforms der Signale durchgeführt und die Werte der neuen Input-Variablen gelesen. Jeder Prozeß führt dann seine Berechnungsschritte relativ zu seiner lokalen Kontrolle weiter fort.

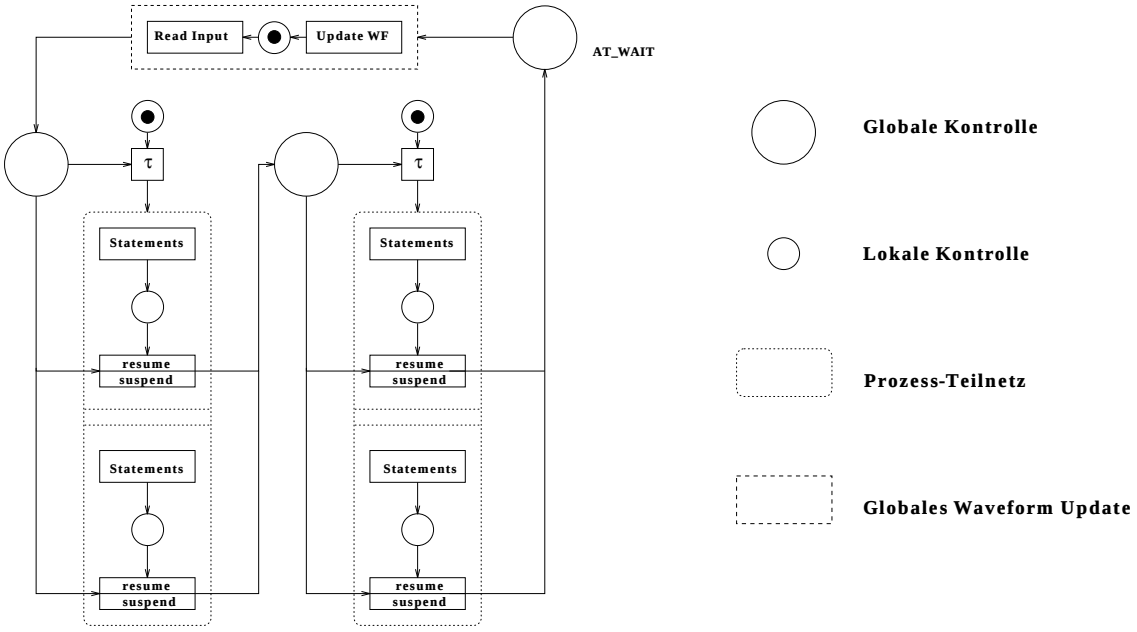


Abbildung 6.7: Struktureller Aufbau eines Petri-Netzes aus einer VHDL-Beschreibung mit zwei Prozessen

6.3.2 VHDL/S StateChart

VHDL/S StateChart ist eine graphische, automatenbasierte Spezifikationssprache für reaktive Systeme. Sie wurde im Kontext des ESPRIT Projektes Nr. 6128 FORMAT entwickelt und basiert auf dem StateChart-Sprachentwurf von D. Harel Mitte der 80er Jahre [Har87]. VHDL/S StateCharts verbindet die Sprachen VHDL (siehe Kapitel 6.3.1) und StateChart zu einer homogenen Spezifikationssprache.

Das Basiskonzept besteht dabei aus der sequentiellen und parallelen Komposition von endlichen Automaten. Die Kommunikation zwischen einzelnen Automaten wird mittels sogenannter *atomic events* realisiert. Transitionen verbinden Zustände und kennzeichnen so die Kontrollstruktur einer VHDL/S StateCharts. Sie sind mit

booleschen Ausdrücken über Events oder Bedingungen über dem Datenraum und mit einer Liste von Events, die bei der Ausführung der Transition generiert werden, annotiert. Transitionen verbinden hierbei nicht nur Zustände gleicher Hierarchiestufen, sondern sie können zwischen beliebigen Zuständen eingeführt werden [Har87, HPSS87, PS91]. Die graphische Repräsentation der Objekte ist in Abbildung 6.8 gegeben. Die Integration der beiden Sprachen wurde wie folgt durchgeführt: Die

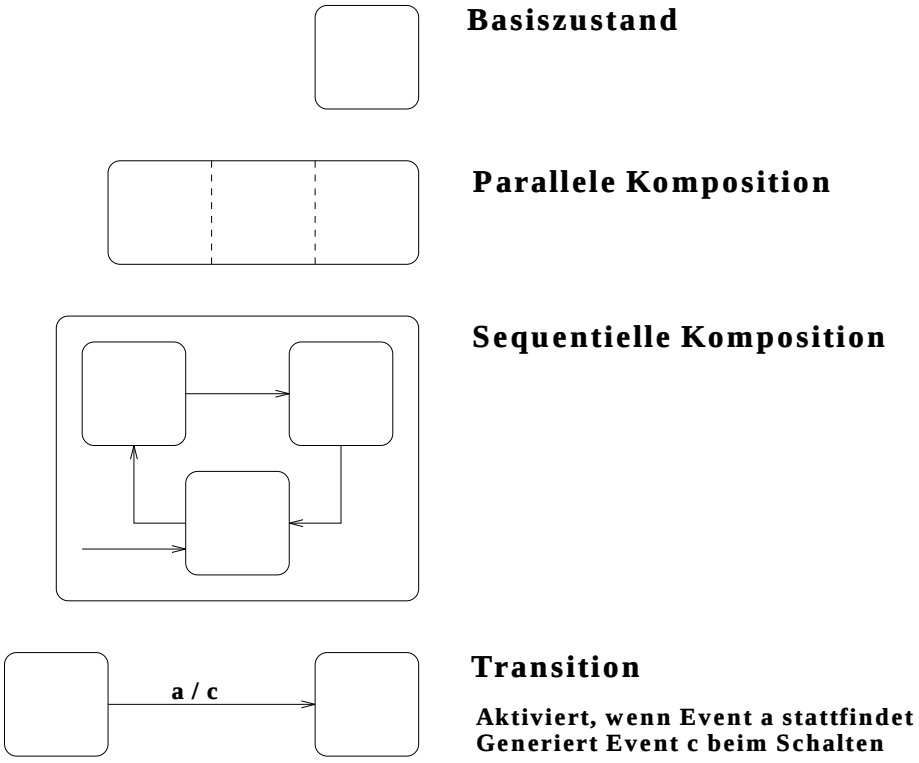


Abbildung 6.8: Graphische Objekte von VHDL/S StateChart

Kernsprache ist StateChart, d.h. die Dekomposition eines Designs wird durch die Sequenz- und Parallel-Operatoren von StateChart realisiert. Sowohl die VHDL Datentypwelt als auch die VHDL Kontrollstrukturen dienen zur Bereicherung der reinen StateChart Kontrollstruktur. In VHDL/S StateChart können die Transitionen mit sequentiellem VHDL Code ohne wait-Statements annotiert sein. Dies schließt die arithmetischen Standardoperationen auf Datentypen und die Wertzuweisungen an Signale, dem Kommunikationsmechanismus von VHDL mit ein. Somit können VHDL/S StateChart Designs problemlos in reine VHDL Umgebungen eingebettet und verifiziert werden, so daß eine industrielle Akzeptanz erreicht wird.

Die Semantik einer VHDL/S StateChart Spezifikation ist eine Menge von Sequenzen von beobachtbaren Portbelegungen des Interfaces der gegebenen Spezifikation. Die

Berechnung der Sequenzen durch ein AADL Petri-Netz wird in [HK94] aufgezeigt. Anhand eines Beispiels (siehe Abbildung 6.9) sollen nocheinmal kurz die Syntax und die Semantik einer VHDL/S StateChart Implementierung beschrieben werden.

Beispiel 6.5 VHDL/S StateChart

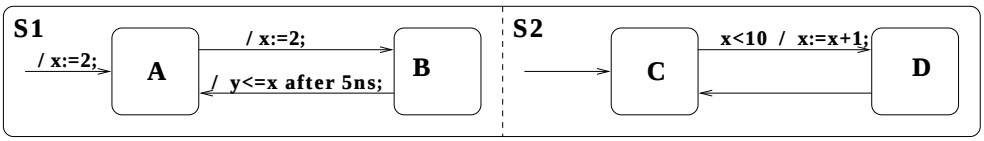


Abbildung 6.9: Eine VHDL/S StateChart Spezifikation

Die Beispielspezifikation besteht aus der parallelen Komposition zweier sequentieller VHDL/S StateCharts $S1$ und $S2$. Das System beginnt mit der Ausführung in den Zuständen A und C. Die Variable x hat zu Beginn der Ausführung den Wert 2. Da sowohl die Transition vom Zustand A nach B als auch die Transition von Zustand C nach D aktiviert sind, können beide Transitionen schalten. Die Transition von $S1$ weist der Variablen x den Wert 2 zu, wohingegen die Transition aus $S2$ den Wert von x inkrementiert. Der tatsächliche Wert von x wird nichtdeterministisch ausgewählt. Nach dem ersten Schritt des System sind die Zustände B und D aktiviert und die Variable x hat entweder den Wert 2 oder 3, je nachdem in welcher Reihenfolge die Transitionen ausgeführt wurden. Danach sind wieder die Zustände A und C aktiviert. Die Transition aus $S2$ hat keine Aktion ausgeführt, wogegen die Transition von $S1$ den Wert von x (2 oder 3) dem Signal y mit dem Zeitstempel $5ns$ zugewiesen hat, d.h. y wird nach $5ns$ den aktuellen Wert von x annehmen. Der Evaluationsprozeß wird nun fortschreiten und die Variable x kann dabei jeden Wert zwischen 2 und 10 annehmen, wobei y den jeweiligen Wert nach $5ns$ propagieren wird.

6.3.2.1 Übersetzung von VHDL/S StateCharts in AADL Petri-Netze

Der prinzipielle Aufbau eines von einer VHDL/S StateChart Spezifikation abgeleiteten AADL Petri-Netzes wird beispielhaft in Abbildung 6.10 beschrieben. Globale boolesche Variablen kodieren den aktuellen Zustand des Systems, d.h. zu welchen Zeitpunkten welche Zustände aktiviert sind. Diese Menge von Variablen wird zu Beginn kopiert, um eine Folgezustandsberechnung durchzuführen [HK94]. Im anschließenden Teilnetz *Berechnung der Folgekonfiguration* werden die VHDL/S StateChart Transitionen simuliert. Neben Modifikationen des Datenraums werden

auch die Variablen zur Bestimmung des Folgezustandes berechnet. Durch das Schal-

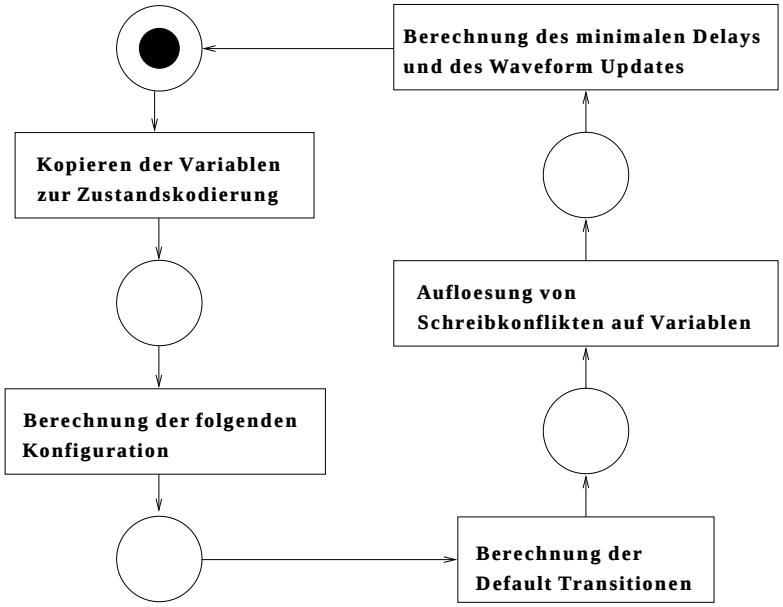


Abbildung 6.10: Globale Petri-Netzstruktur einer VHDL/S StateChart Spezifikation

ten der StateChart Transitionen können hierarchische Zustände betreten werden, deren default-Transitionen ausgeführt werden müssen, bevor eine neue Konfiguration angenommen werden kann. In VHDL/S StateChart können Variablen in parallelen Komponenten beschrieben werden. Diese Eigenschaft führt Nichtdeterminismus ein, da parallele Transitionen einer Variablen verschiedene Werte zuweisen können. Dieser Nichtdeterminismus wird im Petri-Netz aufgelöst, indem für jede Variable eine Kopie pro paralleler Komponente erzeugt wird, die ausschließlich lokal benutzt wird. Nach einem Simulationszyklus werden die lokalen Variablen *synchronisiert*, d.h. aus der Menge der für eine Variable möglichen Werte wird einer nichtdeterministisch ausgewählt. Nach dieser Auswahl müssen alle Variablen mit dem neuen Wert aktualisiert werden, dies geschieht in der Komponente *Auflösung von Schreibkonflikten auf Variablen*. Im letzten Abschnitt des Netzes wird die für diesen Simulationszyklus benötigte Zeit berechnet und ein Update aller Waveforms ausgeführt. Hiernach ist ein Simulationszyklus vollständig abgearbeitet und ein neuer Zyklus kann beginnen.

6.3.2.2 Schalten von VHDL/S StateChart Transitionen

Das AADL Petri-Netz für die Berechnung der Folgekonfiguration wird induktiv über die StateChart Struktur aufgebaut. Hierbei werden StateChart-Transitionen auf Petri-Netz-Transitionen abgebildet.

Eine parallele StateChart-Komponente wird in eine Anzahl von sequentiellen Teilnetzen abgebildet. Hierdurch werden alle parallelen Komponenten sequentiell nacheinander abgearbeitet. Hierarchische StateChart Strukturen werden in Petri-Netze übersetzt, in denen Transitionen im Konflikt stehen. Hierdurch wird sichergestellt, daß das Verhalten des Petri-Netzes zur VHDL/S StateChart genau korrespondiert. In der Abbildung 6.11 wird diese Struktur nocheinmal veranschaulicht. Der parallele

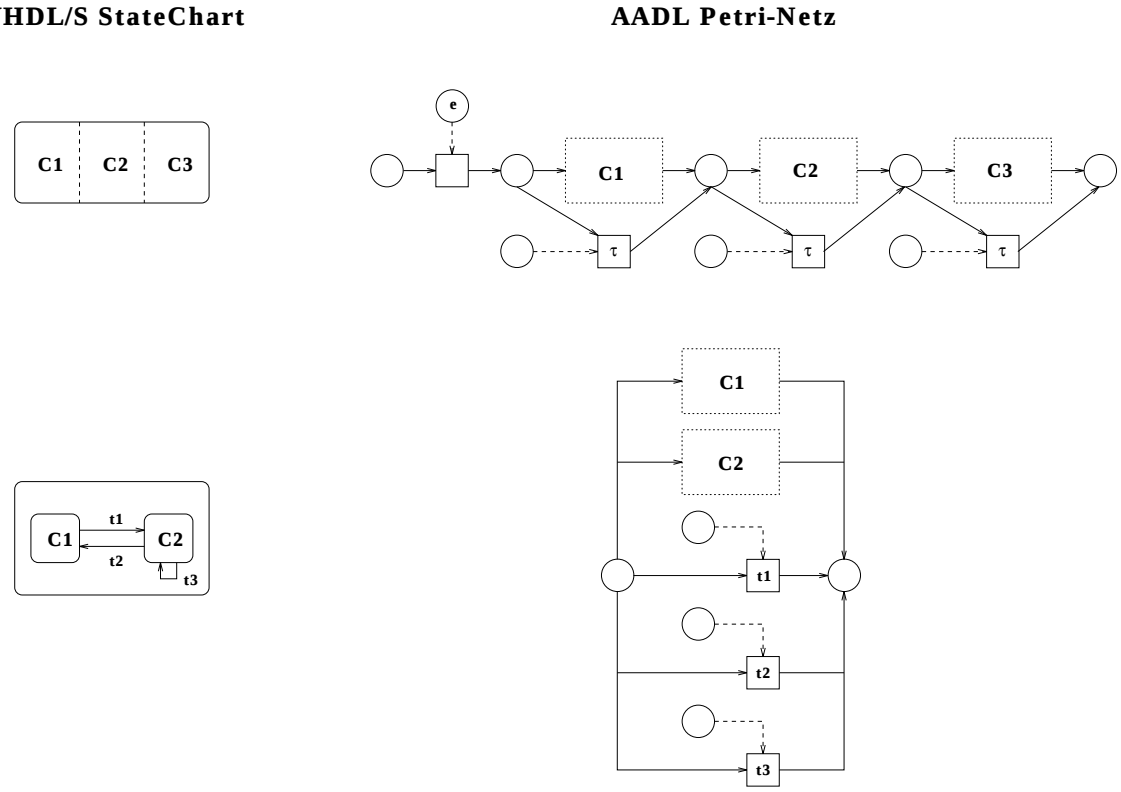


Abbildung 6.11: Compilationsschema für VHDL/S StateChart

StateChart-Zustand besteht aus den drei Teilkomponenten *C1*, *C2* und *C3* und wird in ein Petri-Netz übersetzt, in dem die Teilnetze der Komponenten sequentiell verbunden sind. Die erste Transition testet, ob überhaupt eine StateChart-Transition aktiviert ist (Bedingung e). Für jede Teilkomponenten existiert eine τ-Transition, die aktiviert ist, wenn keine Transition ihrer Teilkomponente aktiviert ist.

Der sequentielle StateChart Zustand besteht aus den Zuständen *C1* und *C2* und den Transitionen *t1*, *t2* und *t3*. Dieses System wird in ein Petri-Netz übersetzt, in dem alle fünf Objekte im Konflikt stehen, d.h. wenn die StateChart-Transition *t1* schaltet, können weder *t2* oder *t3* noch anderweitige Transitionen von *C1* oder *C2* schalten. Die Petri-Netztransitionen der entsprechenden StateChart-Transitionen

sind mit Triggerstellen verbunden, welche diese Zustandsbedingungen eindeutig kodieren.

Das AADL Petri-Netz, welches für eine VHDL/S StateChart-Struktur erzeugt wurde, wird um eine weitere Transition erweitert, die parallel zum generierten Netz plaziert wird. Diese Transition ist nur genau dann aktiviert, wenn keine Transition des VHDL/S StateChart *Kern*-Netzes aktiviert ist. Diese Struktur garantiert die Lebendigkeit des Netzes.

6.3.3 Reduktionstechniken

Wie schon eingangs erläutert, sind bei VHDL bzw. VHDL/S StateCharts Zustände, die keinen wait-Konfigurationen entsprechen, nicht für die Umgebung beobachtbar. Sie können deshalb mittels Substitution aus dem Modell herausgerechnet werden. Somit besteht die Möglichkeit, alle Transformationen zwischen zwei wait-Konfigurationen - im deterministischen Fall - durch eine äquivalente Transformation zu ersetzen oder - im nichtdeterministischen Fall - durch eine Anzahl von Transformationen zu ersetzen, die den möglichen Wertverläufen zwischen den Konfigurationen entsprechen.

Die Transformation des Kontrollautomaten ist durch eine Substitution der Bedingungs- und Datentransformationen der ROBDDs der Transitionen gegeben. Die VHDL Kontrollstrukturen sind stark eingeschränkt und führen zu einer Graphstruktur, welche von drei Transformationsregeln inkrementell reduziert werden kann [PH95].

Die im folgenden vorgestellten Transformation des Kontrollautomaten sind durchführbar, da alle Datentransformationen deterministisch sind und der abgeleitete Kontrollautomat aus VHDL bzw. VHDL/S StateCharts Implementierungen wohlgeformt ist, so daß die Anwendung der Transformationsregeln assoziativ und die vollständige Transformation des Kontrollautomaten durch die Erfüllung der *Chuch-Rosser-Eigenschaft* der einzelnen Transformationen erfüllt ist (siehe u.a. Ersetzungssysteme in [Old91b, ASU87, FL88]).

Nachdem alle Zustände, die nicht als eine wait-Konfiguration auszeichnet sind, eliminiert wurden, führt das kompaktere Transitionssystem zu einer sehr viel schnelleren Fixpunktberechnung beim symbolischen Model Checking (siehe Kapitel 6.3.3.4 Experimentelle Ergebnisse). Diese Beobachtung ist generell nicht zu erwarten, da zwar die Anzahl der Iterationsschritte abnimmt, die Größe der ROBDD-Repräsentation

der NextStep-Funktionen(-Relationen) stark zunehmen kann.

6.3.3.1 Reduktionsregeln

Die Reduktionsregeln stützen sich auf die Grundlagen der Substitutionstechniken für Datentransformationen ab, die in [Old91b, Ste88] detailliert eingeführt wurden.

Elimination paralleler Kanten

Zwei Transitionen, die einen gemeinsamen Startzustand und einen gemeinsamen Zielzustand besitzen, können durch eine neue Transition ersetzt werden, wenn die beiden Bedingungen der Transitionen sich ausschließen, d.h. die Konjunktion $false$ ergibt. Desweiteren darf s_k nicht zu einer Schleife gehören (siehe Elimination von Schleifen).

Die Bedingung der neu in in den Kontrollautomaten zu integrierten Transition wird mit der Disjunktion der Bedingungen der beiden alten Transition annotiert. Die Datentransformationen der beiden alten Transitionen werden dabei jeweils mit ihren zugehörigen Bedingungen über eine Konjunktion verbunden, um einen selektiven Zugriff in Abhängigkeit des aktuellen Zustands des Datenraums zu gewährleisten.

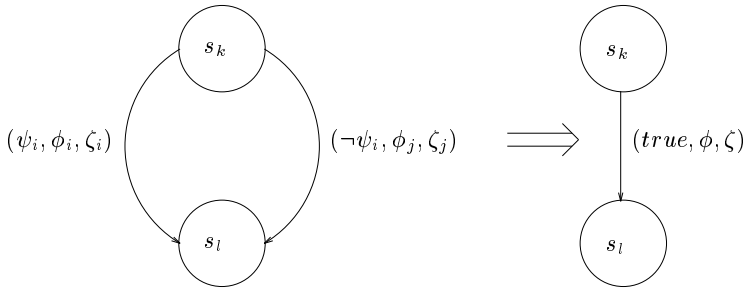


Abbildung 6.12: Elimination paralleler Transitionen

Die Elimination zweier paralleler Kanten eines gegeben Kontrollautomaten KA verändert die nur die Menge T_{KA} seiner Transitionen.

Sei $u = writeSet(\phi_p) - writeSet(\phi_q)$, $v = writeSet(\phi_p) \cap writeSet(\phi_q)$ und $w =$

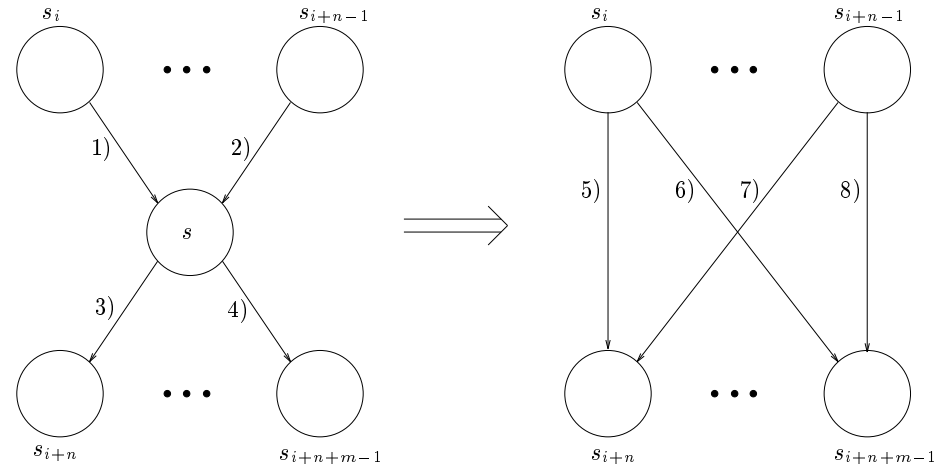
$writeSet(\phi_q) - writeSet(\phi_p)$, dann ergibt sich die Transformation als:

$$
\begin{aligned}
T_{KA} := T_{KA} \quad - \quad & \{(s_k, s_l, \psi_p, \phi_p, \zeta_p), (s_k, s_l, \psi_q, \phi_q, \zeta_q)\} \\
\cup \quad & \{(s_k, s_l, \psi_p \vee \psi_q, \\
& \bigcup_{i \in u} (i, ((\psi_p \wedge sb(i, \phi_p)) \vee (\psi_q \wedge v_i \Leftrightarrow v_i'))) \cup \\
& \bigcup_{i \in v} (i, ((\psi_p \wedge sb(i, \phi_p)) \vee (\psi_q \wedge sb(i, \phi_q)))) \cup \\
& \bigcup_{i \in w} (i, ((\psi_q \wedge sb(i, \phi_q)) \vee (\psi_p \wedge v_i \Leftrightarrow v_i'))), \\
& \psi_p \wedge \zeta_p \vee \psi_q \wedge \zeta_q\}
\end{aligned}
$$

Dieses Verfahren kann iterativ auf alle parallelen Transitionen angewendet werden, so lange sich die Bedingungen der zu verschmelzenden Transitionen ausschließen.

Elimination von Zuständen

Ein Zustand s kann aus dem Kontrollautomaten entfernt werden, wenn es sich bei ihm nicht um eine wait-Konfiguration handelt und er nicht Teil eines lokalen Zyklusses ist. Alle Transitionen, die auf den Zustand s zeigen oder von ihm ausgehen, werden durch neue Transitionen mittels Kreuzproduktbildung der verschiedenen sequentiellen Konfigurationen erzeugt. Das Falten zweier sequentieller



1: $(\psi_i, \phi_i, \zeta_i)$
2: $(\psi_{i+n-1}, \phi_{i+n-1}, \zeta_{i+n-1})$
3: $(\psi_{i+n}, \phi_{i+n}, \zeta_{i+n})$
4: $(\psi_{i+n+m-1}, \phi_{i+n+m-1}, \zeta_{i+n+m-1})$

5: $(\psi_i', \phi_i', \zeta_i')$
6: $(\psi_{i+1}', \phi_{i+1}', \zeta_{i+1}')$
7: $(\psi_{i+n+m-1}', \phi_{i+n+m-1}', \zeta_{i+n+m-1}')$
8: $(\psi_{i+n+m-2}', \phi_{i+n+m-2}', \zeta_{i+n+m-2}')$

Abbildung 6.13: Elimination eines Kontrollautomatenzustandes

Transitionen wird durchgeführt, indem in den Transformationen und Bedingungen der zweiten Transition das Vorkommen aller veränderten ROBDD Variablen der ersten Transformation durch die entsprechenden Transformations ROBDDs substituiert werden. Dies geschieht unter der Berücksichtigung der Bedingung der ersten Transition. Sei $s \in S_{KA}$ ein zu entfernender Zustand des Kontrollautomaten KA, $Pre_s = \{(s_i, s, \psi_i, \phi_i, \zeta_i) \in T_{KA}\}$ die Menge der Kanten, die s als Zielzustand und $Post_s = \{(s, s_j, \psi_j, \phi_j, \zeta_j) \in T_{KA}\}$ die Menge der Kanten, die s als Ausgangszustand haben, dann kann die Transformation Menge der Transitionen und Zustände des Kontrollautomaten wie folgt angegeb werden:

$$
\begin{aligned}
T_{KA} := \quad & (T_{KA} \setminus (Pre_s \cup Post_s)) \cup \\
& \{\ (s_k, s_m, \psi_q[v_i \backslash \psi_p \wedge sb(i, \phi_p)]_{i \in writeSet(\phi_p)}, \\
& \qquad \phi_q[v_i \backslash \psi_p \wedge sb(i, \phi_p)]_{i \in writeSet(\phi_p)}, \\
& \qquad \zeta_p \wedge \zeta_q[v_i \backslash \psi_p \wedge sb(i, \phi_p)]_{i \in writeSet(\phi_p)}) \\
& \qquad\quad \mid (s_k, s, \psi_p, \phi_p, \zeta_p), (s, s_m, \psi_q, \phi_q, \zeta_q) \in T_{KA}\ \}
\end{aligned}
$$

$$
S_{KA} := \quad S_{KA} \setminus \{s\}
$$

Elimination von Schleifen

Das Auflösen von Schleifen, d.h. Transitionen mit gleichen Start- und Zielzuständen, des Kontrollautomaten (siehe Abbildung 6.14 (a)) stützt sich auf die Berechnung von kleinsten Fixpunkten ab. Hierbei wird in einer ersten Phase eine endliche Entfaltung der Schleife (b) berechnet, welches als eine Transformation der Schleife in eine *Fallunterscheidung*, die die Anzahl der möglichen Iterationsfolgen widerspiegeln, angesehen werden kann. Diese Schleifen werden anschließend wieder durch eine Parallelkomposition in eine einzige Kante überführt (c), die dann per Substitution in nachfolgende Transitionen integriert werden kann (d).

Die Bedingungen und Transformationen der neuen Transitionen ergeben sich durch:

$$
\begin{aligned}
\psi_j^1 &= \psi_j \\
\psi_j^2 &= \psi_j \wedge \psi_j^1[v_i \backslash \psi_j^1 \wedge sb(i, \phi_j)]_{i \in writeSet(\phi_j)} \\
&\vdots \qquad \vdots \\
\psi_j^n &= \psi_j \wedge \psi_j^{n-1}[v_i \backslash \psi_j^{n-1} \wedge sb(i, \phi_j)]_{i \in writeSet(\phi_j)}
\end{aligned}
$$

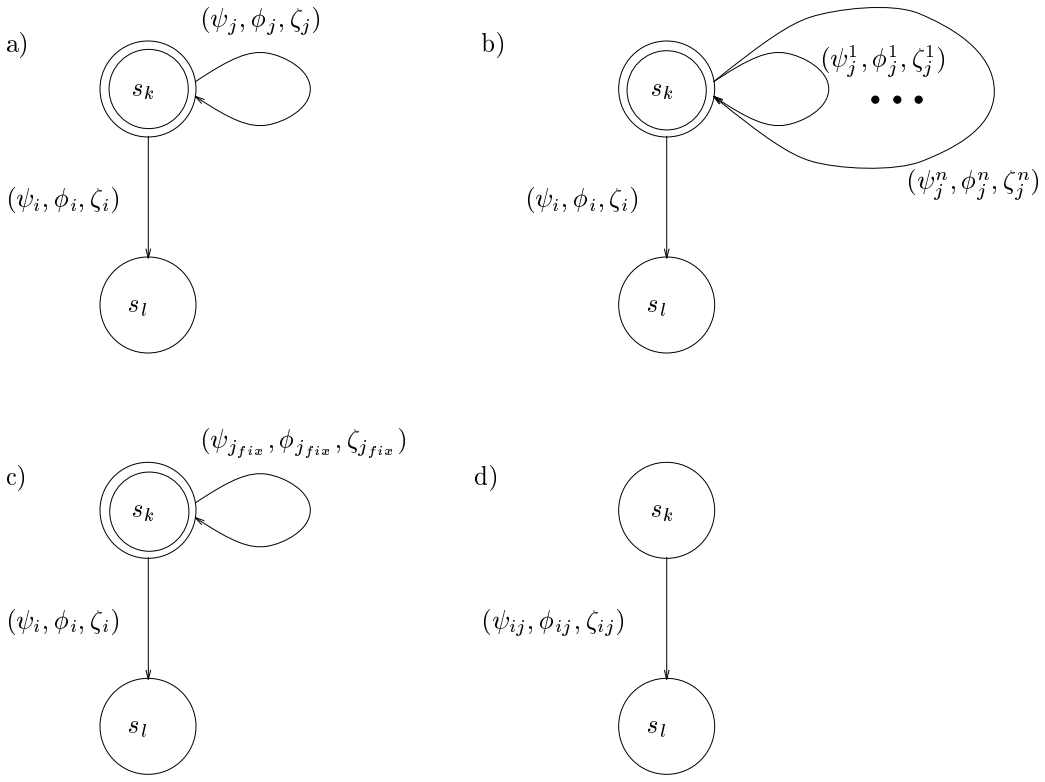


Abbildung 6.14: Elimination von Schleifen

$$\begin{aligned}
\phi_j^1 &= \phi_j \\
\phi_j^2 &= \phi_j^1[v_i \backslash \psi_j^1 \wedge sb(i, \phi_j)]_{i \in writeSet(\phi_j)} \\
&\vdots \quad\quad \vdots \\
\phi_j^n &= \psi_j^{n-1}[v_i \backslash \psi_j^{n-1} \wedge sb(i, \phi_j)]_{i \in writeSet(\phi_j)}
\end{aligned}$$

Das Iterationsverfahren terminiert aus den zwei folgenden Gründen:

1. Die Schleifenbedingung wird bei einer Iteration zu *false* evaluiert, d.h.

   $$\psi_j \wedge \psi_j^n[v_i \backslash \psi_j^n \wedge sb(i, \phi_j)]_{i \in writeSet(\phi_j)} = false$$

   oder
2. die Schleifenbedingung wird nie verletzt und die Berechnung divergiert. Da alle Berechnungen jedoch auf endlichen Wertebereichen agieren, terminiert auch in diesem Fall das Berechnungsverfahren, wenn folgende Eigenschaften gelten:

   $$\psi_j^n = \psi_j^{n-1} \wedge \phi_j^n = \phi_j^{n-1}$$

und es kann eine Divergenzbedingung ω angegeben werden

$$\omega = \bigvee_{k=1}^{n} \neg\psi_j^k$$

Die neu erzeugten Transitionen können nun mittels des Verfahrens zur Elimination paralleler Kanten zu einer einzigen Transition $(s_k, s_k, \psi_{j_{fix}}, \phi_{j_{fix}}, \zeta_{j_{fix}})$ zusammengefaßt werden.

6.3.3.2 Integration von Standard-Assumptions

Die Simulationssemantik von VHDL [IEE87] legt fest, daß zu Beginn einer Simulation jeder Prozeß seine Berechnungen bis seinem ersten wait-Anweisung ausführt. Die Semantik von VHDL ordnet den Input-Variablen des Moduls für diese Berechnung feste, konstante Werte zu, die sich aus den expliziten und impliziten Initialisierungen der Input-Variablen ergeben. In einer ersten Realisierung wurden diese Eigenschaften in temporaler Logik als Assumptions für das Model Checking kodiert. Diese Formeln werden bei diesem Verfahren in eine automatenähnliche Struktur, dem s.g. Tableau, übersetzt und mit dem symbolischen Transitionssystem geschnitten, so daß nur noch Abläufe im Transitionssystem möglich sind, die vom Tableau her erlaubt sind.

Da von dem Startzustand des Kontrollautomaten genau eine Transition mit einer gültigen Bedingung zu einem Zustand führt, der eine wait-Konfiguration charakterisiert, kann diese erste deterministische Berechnung bereits in das Modell mit integriert werden und erspart somit die Integration via Tableau.

6.3.3.3 Heuristiken zur Anwendung der Reduktionen

Neben dem Auffinden einer optimalen Variablenordnung zur Generierung kleiner ROBDDs, die das STS beschreiben, ist die effiziente Erzeugung des STS für einen Kontrollautomaten von vielen Faktoren abhängig, die u.a. im Rahmen von [Pö95, Bro95] näher untersucht wurden. Zwei zeit- und speicherplatzkritische Phasen standen zur Optimierung an:

1. Reduktion des Kontrollautomaten
2. Erzeugung des RSTS für einen Kontrollautomaten

Ausgehend von den o.g. Reduktionsregeln führt ihre straight forward Anwendung zu sehr schlechten Laufzeiten innerhalb der Reduktionsphase, und es entstehen intermediäre ROBDDs, die in ihrer Größe und Struktur eine weitere Anwendung der Reduktionsregeln für reale Beispiele unmöglich machen. Um dieses Problem zu lösen, können zwei Ansätze verfolgt werden:

1. Optimierung der Anwendung von Reduktionsregeln (Top-Level) und
2. Optimierung der Reihenfolge zur Substitution von binären Variablen (Bottom-Level)

Top-Level

Ein Messen des Zeitaufwands für die verschiedenen Regeln ergibt, daß die Substitution von binären Variablen durch die funktionalen Transformations-ROBDDs das zeitaufwendigste Verfahren ist. Hieraus ergibt sich dann die folgende, nach zunehmenden Zeitdauer sortierte Reihenfolge der Regeln:

1. Elimination von Zuständen
2. Komposition zweier paralleler Transitionen
3. Berechnung von Fixpunkten
4. Komposition zweier sequentieller Transitionen

Eine Anwendung der Reduktionsregeln mittels dieser Prioritätenliste kann nun strukturiert anhand folgender Heuristiken erfolgen, die ihrer Effektivität nach aufgelistet sind:

1. Heuristiken, die ohne Kenntnis der Struktur des Kontrollautomaten agieren
   - *Ausgehend vom Startzustand in Kantenrichtung des Kontrollautomaten*
     Diese Strategie hat die schlechtesten Ergebnisse, da sie dem straight-forward Ansatz am nächsten kommt.
   - *Gleichverteilte Reduktion*
     Durch die Einführung von Reduktions-Leveln wird eine gleichverteilte Reduktion des Kontrollautomaten erreicht, bei dem das Anwachsen der Größe der ROBDDs früh zu relativ großen Strukturen führt.
   - *Zufällige Auswahl von Transitionen*
     Durch eine zufällige Auswahl der Transitionen kann obiger Nachteil der Gleichverteilung zeitweise aufgefangen werden.

- *Ausgehend vom Startzustand entgegen dem Kontrollfluß*
  Dieser Ansatz brachte für aus VHDL abgeleitete Kontrollautomaten gute Ergebnisse, da zufällig komplexe Berechnungen zu Anfang der Kontrolle ausgeführt werden.
- *Selektion von Transitionen anhand von Prioritäten*
  Bei diesem Ansatz wurden Transitionen Prioritäten zugeordnet, von denen eindeutig angenommen werden konnte, daß sie mit komplexen, schlecht zu substituierenden Transformationen und Bedingungen annotiert sind.

2. Heuristiken, die auf der Analyse der Annotationen des Kontrollautomaten beruhen

   - *Unabhängigkeit der Datentransformation zweier Transitionen*
     Bei diesem Ansatz werden Transitionen ausgewählt, die disjunkte Datenbereiche modifizieren. Dieses Verfahren ist zu Anfang der Reduktion am schnellsten, verlangsamt sich jedoch nach jeder Anwendung, da die durchschnittliche Größe der transformierenden Datenbereiche sehr stark anwächst.
   - *Anzahl der funktionalen Transformations-ROBDDs einer Transitionen*
     Hierbei wird den Transitionen des Kontrollautomaten eine Priorität zugeordnet, die sich direkt aus der Anzahl ihrer Transformations-ROBDDs ergibt. Dieser quantitativer Ansatz führt zu besseren Ergebnisses als der auf Datenbereichen basierende Ansatz.
   - *Größe der funktionalen Transformations-ROBDDs einer Transitionen*
     Bei diesem Ansatz wird eine Kenngröße aus der Anzahl der ROBDD-Knoten der einzelnen funktionalen Transformations-ROBDDs abgeleitet. Dieses Verfahren führt zu guten Ergebnissen bezüglich der Größe der intermediären ROBDDs, jedoch ist der Zeitaufwand für die Berechnung der Größe der ROBDDs ein nicht akzeptabler Overhead.

3. Heuristik, die Information über die Struktur des Kontrollautomaten ausnutzt
   Dieser Ansatz stellt eine Optimierung des Ansatzes dar, bei dem einzelnen Transitionen Prioritäten zugewiesen werden. Bei diesem Ansatz werden isomorphe Teilgraphen nur ein einziges mal reduziert und dann entsprechend substituiert.
   Für Kontrollautomaten, die aus VHDL abgeleitet werden, ist dieses Verfahren effizient, da sie als Kreuzprodukte der Teilgraphen, die einzelne Prozeßfragmente repräsentieren, aufgebaut werden und diese Information einfach auf

Kontrollautomatenebene repräsentiert werden kann.
Für aus VHDL/S StateChart abgeleitet Kontrollautomaten bringt dieses Verfahren keinerlei Vorteil, da hier keine wait-Konfigurationen unterschieden werden können und somit auch keine isomorphe Teilgraphen existieren.
Der Zeitvorteil dieses Verfahrens gegenüber dem rein prioritätsorientierten Ansatz beträgt aber nur ca. 10-15 %. Diese überraschend geringe Optimierung liegt an der Tatsache, daß über 70 % der Reduktionszeit für die Substitutionen der Transformationen des globalen Netzes in die reduzierten wait-Konfigurations-Teiltransformationen benötigt wird. Da diese Transformationen aber nicht umgangen werden können, ist das Ergebnis trotzdem als gut zu bewerten.

Fazit:

Eine gute Heuristik für die Reduktion eines Kontrollautomaten muß so lange wie möglich mit kleinen ROBDDs agieren. Hierbei ist es auch von Vorteil, kleiner ROBDDs in sehr große ROBDDs zu integrieren, als dieses Verfahren mit mittelgroßen ROBDDs durchzuführen.

Bottom-Level

Kommt eine Reduktionsregel zum Einsatz, müssen gegebenenfalls Substitutionen von binären Variablen mit funktionalen Transformations-ROBDDs durchgeführt werden. Hierbei stehen zwei Vorgehensweisen zur Verfügung, die sich an der Priorität der einzelnen ROBDD Variablen orientieren:

1. Substitution der Variablen in einer Reihenfolge mit fallender Priorität
2. Substitution der Variablen in einer Reihenfolge mit steigender Priorität

Es hat sich gezeigt, daß die zweite Variante der Substitutionsreihenfolge eindeutig schneller ist als die erste Variante. Dies läßt sich wiederum durch das o.g. Fazit beschreiben. Variablen mit geringer Priorität verändern (vergrößern) einen bestehenden ROBDD weniger als eine Variable mit hoher Priorität.

Erzeugung des RSTS

Bei der Erzeugung des relationalen symbolischen Transitionssystems wird jede Transition des Kontrollautomaten vollständig in eine eigene Relation übersetzt. Folgende Heuristik für die Konstruktion des Ergebnis-ROBDDs hat sich als effizient gezeigt:

1. Konstruktion der Äquivalenzrelation zwischen den ROBDD Variablen, die sich nicht durch die Transformation der aktuellen Transition verändern.
2. Konstruktion der Äquivalenzrelation zwischen den ROBDD Variablen und den funktionalen Transformations-ROBDDs, die die Datenmodifikation beschreiben. Dieses Verfahren wird wiederum für die Variablen mit steigender Priorität durchgeführt.
3. Kombination der beiden Teilrelationen zur NextStep-Relation der aktuellen Transition.

6.3.3.4 Experimentelle Ergebnisse

Um eine quantitative Bewertung der Realisierung durchführen zu können, wurde eine Menge von sieben VHDL/S Implementierungen als Referenzmenge von industriellen Beispielen ausgewählt, welche auch im Rahmen des FORMAT Projektes evaluiert wurden. Tabelle 6.2 führt diese Fallstudien auf und kennzeichnet dazu die Anzahl der wait-Konfigurationen, die vom Modul angenommen werden können.

VHDL Modul	Kürzel	wait-Konfigurationen	Services
DepositBelt	DBCONT	4	18
Traffic Light Controller	TLC	1	12
Master/Slave	MS	6	6
Communication Controller	CC	1	18
Table Controller	VHDLTC	40	8
VHDL/S StateChart Modul			
Mutex	MU	1	3
Table Controller	SSTC	1	8

Tabelle 6.2: VHDL/S Module des Benchmarksets

Als Kenngröße der Güte der angewandten Reduktionen wird einerseits die Zeit für die Erzeugung des symbolischen Transitionssystems als auch die notwendige Zeit zur Verifikation eines Services für ein gegebenes Modul angesehen. Um möglichst allgemeine Aussagen treffen zu können, wird jeweils eine Menge von Referenzservices behandelt und deren Zeit zur Verifikation bzw. Widerlegung gemittelt, um Extremfälle, sowohl positiv als auch negativ, ausschließen zu können.

Tabelle 6.3 stellt die benötigte Zeit der einzelnen Phasen zur Generierung des STS deren Gesamtzeit gegenüber. Hierbei werden können Phasen unterschieden werden:

- Generierung des AADL Petri-Netzes aus der VHDL Beschreibung (PN),
- Generierung des booleschen Domains und deren initialen Variablenordnung (DT),
- Generierung des Kontrollautomaten mit symbolischer Annotation der Bedingungen und Datentransformationen(KA),
- Anwendung der Reduktionsregeln auf dem Kontrollautomaten (Reduktion) und
- Transformation des Kontrollautomaten in ein symbolisches Transitionssystem

VHDL Modul	PN	DT	KA	STS	$\sum$ *Micro*	Reduktion	STS	$\sum$ *Makro*
DBCONT	6.2	3.9	10.7	0.1	20.9	40.6	0.1	61.5
TLC	9.9	3.3	13.1	0.1	26.4	56.0	0.4	82.7
MS	5.9	3.3	10.6	0.2	28.0	410.3	4.7	434.8
CC	35.8	8.0	38.4	0.5	82.7	117.4	0.8	200.4
VHDLTC	9.9	7.1	30.9	4.2	52.1	2521.1	16.7	2585.7
MU	8.1	5.7	8.6	1.1	23.5	10.5	2.9	35.8
SSTC	9.2	8.2	13.2	0.2	30.8	319.9	0.1	350.6

Tabelle 6.3: Zeitaufwand zur Generierung des STS für VHDL in Sekunden

Für die Erzeugung des nicht reduzierten STS (Micro Modell) kann ein lineares Laufzeitverhalten der einzelnen Phasen beobachtet werden, d.h. die jeweils benötigte Zeit ist proportional zur Größe des Moduls. Das Verhalten der Laufzeit für die Reduktion des STS ist dagegen nicht eindeutig aus der Größe vorausberechenbar (siehe Module MS und SSTC), ergibt sich aber aus der Anzahl der wait-Konfigurationen des Systems und überschreitet gewöhnlich die Gesamtzeit zur Berechnung des nicht optimierten Modells ($\sum$ *Micro*) bis zum 50-fachen beim Modul VHDLTC.

Ein entscheidender Faktor der Reduktion ist die Größe der ROBDDs für die Next-Step Berechnung. Tabelle 6.4 gibt einen Überblich über die benötigten BDD Knoten zur Repräsentation, wobei Abbildung 6.15 die minimalen und maximalen ROBDDs der Micro- und Macro-STS gegenüberstellt. Es ist zu erkennen, daß die Reduktion eine extreme Veränderung der Größe der Darstellung der NextStep-ROBDDs nach sich zieht. Durch die logarithmische Skalierung wird deutlich, daß es sich um ein exponentielles Wachstum handelt, welches im worst case die Berechnung nicht terminieren läßt. Strukturanalysen des Kontrollautomaten ergebenen, daß die komplexesten ROBDDs im *globalen Kontrollteil*, z.B. die Berechnung des lokalen, minimalen Zeitschrittes des Moduls, angesiedelt sind. Sinnvoll wäre ein Reordering nach einer Substitution des globalen Netzes in ein Prozeßnetz, welches einer wait-Konfiguration

VHDL/S	NextStep-ROBDDs		Min-ROBDD		Max-ROBDD		∅-ROBDD	
Modul	Mikro	Makro	Mikro	Makro	Mikro	Makro	Mikro	Makro
DBCONT	40	37	16	7	72	83	30	28
TLC	68	63	10	10	94	641	31	251
MS	59	55	19	8	115	5578	42	299
CC	95	88	12	11	216	2106	35	218
VHDLTC	83	79	79	12	547	960	139	100
MU	36	31	1	3	56	86	22	34
SSTC	91	85	3	3	335	832	37	58

Tabelle 6.4: Anzahl von Knoten der NextStep-ROBDDs

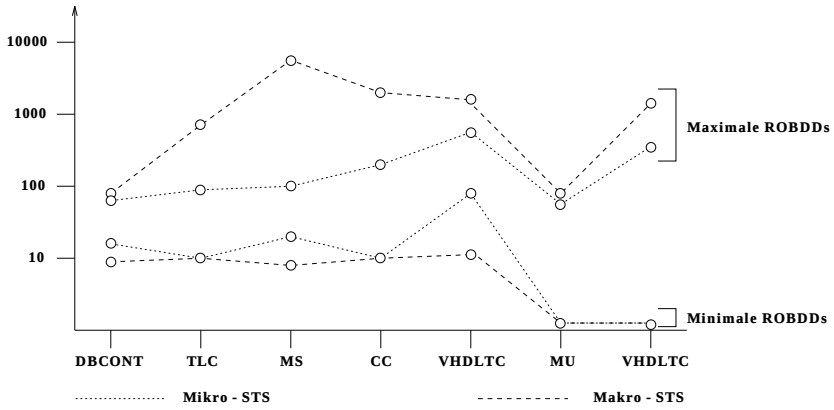


Abbildung 6.15: Größe der ROBDDs für Mikro- und Makro-STS

zugeordnet ist. Diese Schlußfolgerung ergibt sich aus der Berechnung einer Variablenordnung für das Model Checking basierend auf einer Reachability-Berechnung des STS.

Für ein Reordering während der Generierung des STS wird jedoch ein BDD-Package benötigt, welches ein Reordering auf einer Teilmenge von ROBDDs erlaubt. Ein globales Reordering führt zu keiner Verbesserung, da sich im BDD-Cache noch Inkarnationen von ROBDDs befinden, die die gewünschte Optimierung verhindern. Der Overhead der Reduktion des STS wird durch das schnellere Model Checking ausgeglichen. Tabelle 6.5 gibt die gemittelte benötigte Zeit pro Phase des Model Checkings für den Benchmarkset der Services an, wobei folgende Phasen unterschieden werden:

- Tableau-Konstruktion (T)
- Sicherheits-, Lebendigkeits- und Fairness-Eigenschaften (FN)
- Reachability (RB)
- Symbolic Model Checking (MC)

VHDL	∅-T		∅-FN		∅-RB		∅-MC		$\sum$ *Services*	
Modul	Mikro	Makro	Mikro	Makro	Mikro	Makro	Mikro	Makro	Mikro	Makro
DBCONT	21	0	128	4	18	6	61	14	4080	900
TLC	8	0	63	2	29	32	15	3	960	180
MS	10	0	17	1	23	23	58	4	840	240
CC	11	0	1	1	732	875	434	203	10620	6720
VHDLTC	472	0	2367	1034	126	73	12	8	9720	2640
MU	3	0	9	3	125	3	207	7	1500	60
SSTC	53	0	850	274	–	–	47	17	10560	3120

Tabelle 6.5: Zeitaufwand in Sekunden pro Phase des Model Checking

Es konnten jedoch keine Laufzeitergebnisse für die Reachability Berechnung des Moduls SSTC gewonnen werden, da die Berechnung sowohl für das normale als auch reduzierte STS nach ca. 30 Stunden nicht terminierte und abgebrochen wurde. Die

VHDL/S Modul	Mikro	Makro
DBCONT	161	15
TLC	28	6
MS	118	9
CC	51	17
VHDLTC	143	28
MU	6203	19
SSTC	79	8

Tabelle 6.6: Anzahl der Iterationsschritte beim MC für RB-Berechnung

drastische Reduktion der Laufzeit des Model Checkers trotz der komplexeren Next-Step Relation beruht auf der sehr viel kleineren Zahl von Iterationsschritten beim Symbolic Model Checking, wie es in Tabelle 6.6 aufgezeigt wird. Hierdurch wird eine schnellere Berechnung der Fixpunkte ermöglicht, welches nicht von Anfang an zu erwarten war. Der absolute Zeitgewinn durch die Reduktionen wird für den gewählten Benchmarkset in Tabelle 6.7 aufgeführt. In ihr wird die Modelgenerierungszeit (MG) und ein Lauf des Model Checkers (MC) gegenübergestellt. Das beste Ergebnis wurde für das VHDL/S StateChart Modul Mutex erzielt, wobei die Model Check Zeit von 25 Minuten auf 1 Minute reduziert werden konnte.

6.4 Slicing

Trotz der bisher vorgestellten Methoden zur Erstellung und Reduktion von symbolischen Transitionssystemen für reaktive Systeme mit endlichen Datentypen kann

VHDL	Mikro–Maschine		Makro–Maschine		Mikro/Makro
Modul	MG	MC	MG	MC	Zeitgewinn
DBCONT	21	4080	62	900	3139
TLC	26	960	83	180	723
MS	28	840	435	240	193
CC	82	10620	200	6720	3782
VHDLTC	52	9720	2586	2640	4546
MU	27	1500	37	60	1430
SSTC	41	10560	358	3120	7123

Tabelle 6.7: Absoluter Zeitgewinn der Reduktionen in Sekunden

das zu verifizierende Modell zu groß für eine Anwendung von symbolischem finite state Model Checking sein. Ein Ausweg aus dieser Situation wurde mit [DDG+94] angeboten. In dieser Arbeit wird ein adaptives Verfahren vorgestellt, welches anhand einer gegeben temporallogischen Formel ein abstraktes minimales Transitionssystem ableitet. Eine effiziente Realisierung konnte bis jetzt jedoch nicht angegeben werden, da die iterative Entwicklung des abstrahierten Modells mittels eines Splitting-Algorithmus, welcher vollständige konkrete Modelle benötigt, berechnet wird.

Der hier vorgestellte Ansatz verfolgt eine Abstraktion der Datentransformationen und des Kontrollflusses auf der Ebene des Kontrollautomaten [Par96]. Das realisierte Verfahren basiert auf folgenden Beobachtungen:

- In Hardware-Designs herrscht meistens eine sehr lockere Verbindung zwischen Kontroll- und Datenpfaden.
- Mehrere parallele Prozesses führen einen Großteil ihrer Berechnungen auf nicht geteilten, d.h. auf lokalen, Datenbereichen aus.
- Zu verifizierende Eigenschaften referenzieren bzw. benötigen zu ihrem Nachweis meistens nur eine Teilmenge der Variablen des Designs.
- Der Designer möchte eine Verifikation für bestimmte, kritische Input-Werte des Designs schnellstmöglich erhalten.

Drei weitere Anforderungen wurden an eine optimale Modellgenerierung gestellt:

1. Es sollte eine Methodik zur vollautomatischen Modellgenerierung für VHDL-Module bereit gestellt werden, die durch ihre Größe bisher nicht behandelbar waren.[6]

[6]Dynamic Reordering Funktionen waren zu diesem Zeitpunkt noch nicht zur Modellgenerie-

2. Eine schnellere Berechnung eines symbolischen Modells für eine gegebene Klasse von Spezifikationen.
3. Eine Beschleunigung der Fixpunktberechnung während des Model Checkings.

Das im folgenden vorgestellte Verfahren des Slicings stellt eine sichere Reduktion eines Modells M in Bezug zu einer temporallogischen Formel φ dar, da sowohl

$$\mathcal{M}_{sliced_\varphi} \models \varphi \Rightarrow \mathcal{M} \models \varphi$$

als auch

$$\mathcal{M}_{sliced_\varphi} \not\models \varphi \Rightarrow \mathcal{M} \not\models \varphi$$

gilt. Desweiteren wird das reduzierte Modell ohne vorherige Konstruktion des vollständigen Modells erzeugt.

6.4.1 Berechnung des abstrahierten Modells

Die Grundidee der Abstraktion basiert auf der Annahme, daß sich ein Teilmodell des Designs auf Grund einer Datenflußanalyse als Projektion ableiten läßt. Ausgehend von der Menge der Variablen, die in der von der STD Spezifikation abgeleiteten CTL-Formel referenziert werden, wird eine transitive Hülle von Variablen auf der Basis der Schreib/Lese-Zugriffe eines Datenflußgraphes aufgebaut. Diese Analyse wird auf der Struktur des Kontrollautomaten durchgeführt.

Das vorgestellte Verfahren kann u.a. auf VHDL Implementierungen angewandt, die folgende Eigenschaften besitzen:

- *nur δ-Delay Signalzuweisungen*
  Werden Wertzuweisungen an Signale mit einer Verzögerungszeit eliminiert, so kann sich das temporale Verhalten des gesamten Designs verändern und somit zu falschen Aussagen des Model Checkers führen.
- *Kontrollstrukturen dürfen/sollten nicht von Variablen abhängen, die zum Datenanteil des Designs gehören*
  Um weiterhin ein deterministisches Verfahren für den abstrahierten Kontrollautomaten anbieten zu können, müssen die Bedingungen der Transitionen erhalten bleiben. Sollten hier Variablen des Datenanteils referenziert werden, müßten sie bei diesem Ansatz mit zur Berechnung der transitiven Hülle herangezogen werden, was zu einer deutlich geringeren Abstraktion führen könnte.

---

rungszeit zugänglich!

Das Verfahren basiert, wie schon eingangs erläutert, auf der Berechnung einer Teilmenge der vom Design referenzierten binären Variablen und deren funktionalen Datentransformationen anhand einer gegeben CTL Formel zur Ableitung eines abstrahierten Kontrollautomaten KA_{sliced} von einem nicht abstrahierten Kontrollautomaten $KA(N) = (S_{KA}, \psi, \phi, \zeta, T_{KA}, s_{init}, \Upsilon, \beta, BD)$. Dies wird in zwei Phasen durchgeführt:

Berechnung der minimalen binären Variablenmenge $V_{BDD_{min}}$

Ausgehend von einer initialen Menge Ψ_0 von binären Variablen $v \in BD$, die aus den in der temporallogischen Formel referenzierten Variablen und den zur Kontrollflußkodierung notwendigen Variablen des KA abgeleitet werden, wird inkrementell diese Menge Ψ_i bzgl. „Lese"-Abhängigkeiten expandiert, bis sie sich nicht weiter vergrößern läßt. Wenn $\Psi_n = \Psi_{n-1}$ gilt, dann ist die minimale Menge von BDD-Variablen mit $V_{BDD_{min}} = \Psi_n$ gefunden worden. Aus ihr kann dann die Menge $V_{index_{min}}$ der Indices der zur Kodierung notwendigen binären Variablen abgeleitet werden. Im worst-case kann $V_{BDD_{min}}$ vollständig die Menge BD umfassen und die Abstraktion war nicht erfolgreich, d.h. der Anzahl der binären Variablen konnte nicht reduziert werden.

Reduktion des Kontrollautomaten bezüglich $V_{BDD_{min}}$

KA_{sliced} ergibt sich direkt aus KA, indem die Menge der Datentransformationen ϕ zu der Menge ϕ_{sliced} verkleinert wird. In ihr werden nur noch die Transformations-ROBDDs des KAs aufgenommen, die Variablen aus $V_{BDD_{min}}$ modifizieren. Somit kann die Transitionsrelation des Kontrollautomaten zu $T_{KA_{sliced}} \subseteq S_{KA} \times S_{KA} \times \psi \times \phi_{sliced} \times \zeta$ mit

$$\phi_{sliced} = \{(i, robdd) \in \phi \mid i \in V_{index_{min}}\}$$

verkleinert werden.

Integration von Invarianten

Eine Einschränkung der Werte von Input-Variablen kann durch die Definition von Invarianten durchgeführt werden. Hierbei ist es dem Designer möglich, boolesche Ausdrücke zu definieren, die mit den Bedingungen ψ des Kontrollautomaten konjugiert werden. Durch eine Analyse des Kontrollautomaten, welche Zustände noch erreichbar sind, kann eine weitere Reduktion des Graphen erfolgen.

Durch diese Transformation können jedoch auch wait-Konfigurationen entfernt werden, so daß sich das Synchroniationsverhalten des Modells maßgeblich verändert. Der Designer hat eigenverantwortlich darauf zu achten, daß seine zu verifizierende Eigenschaft durch die Integration von Invarianten nicht direkt beeinflußt wird.

6.4.2 VHDL-Level Slicing

Neben der Reduktion des Kontrollautomaten durch Anwendung von Slicing kann auch versucht werden, den als Eingabe benutzten VHDL-Code zu modifizieren, um ihn in weiteren Verfahren für „fremde“ CAD Umgebungen nutzbar zu machen. Es lassen sich hier wiederum zwei Klassen von Modifikationen ausmachen:

1. Transformation der Datenstrukturen:
   - Unreferenzierte Variablen werden gelöscht.
   - Unreferenzierte Recordkomponten werden gelöscht.
   - Arrays werden in kleinere Teil-Arrays unterteilt, die nur noch die benötigten Komponenten erhalten; unreferenzierte Komponenten werden gelöscht.
   - Die Datentypen von unreferenzierten Ports werden zu Bit transformiert, jedoch nicht gelöscht, um das Synchronisationsverhalten des Moduls nicht zu modifizieren.
2. Modifikation der Kontrollstruktur:
   - Nicht mehr notwendige Variable Assignment Statements werden gelöscht.
   - IF Statements werden durch ihren THEN bzw. ELSE Teil erzetzt, wenn dies durch Invarianten erlaubt wird.
   - Nicht mehr erreichbare CASE-Zweige werden durch ein NULL-Statement ersetzt.

6.4.3 Modifikation der Softwarearchitektur

Die im Kapitel 3.1 vorgestellte Systemarchitektur für die Modellgenerierung (siehe Abbildung 3.1) stellt sich für die Erzeugung von symbolischen Modellen unter Ausnutzung des Slicing-Verfahrens weitaus komplexer dar. Abbildung 6.16 gibt hierfür das zu Grunde liegende Compilationsschema an. Die grau unterlegten Komponenten stellen neue bzw. modifizierte Tools dar:

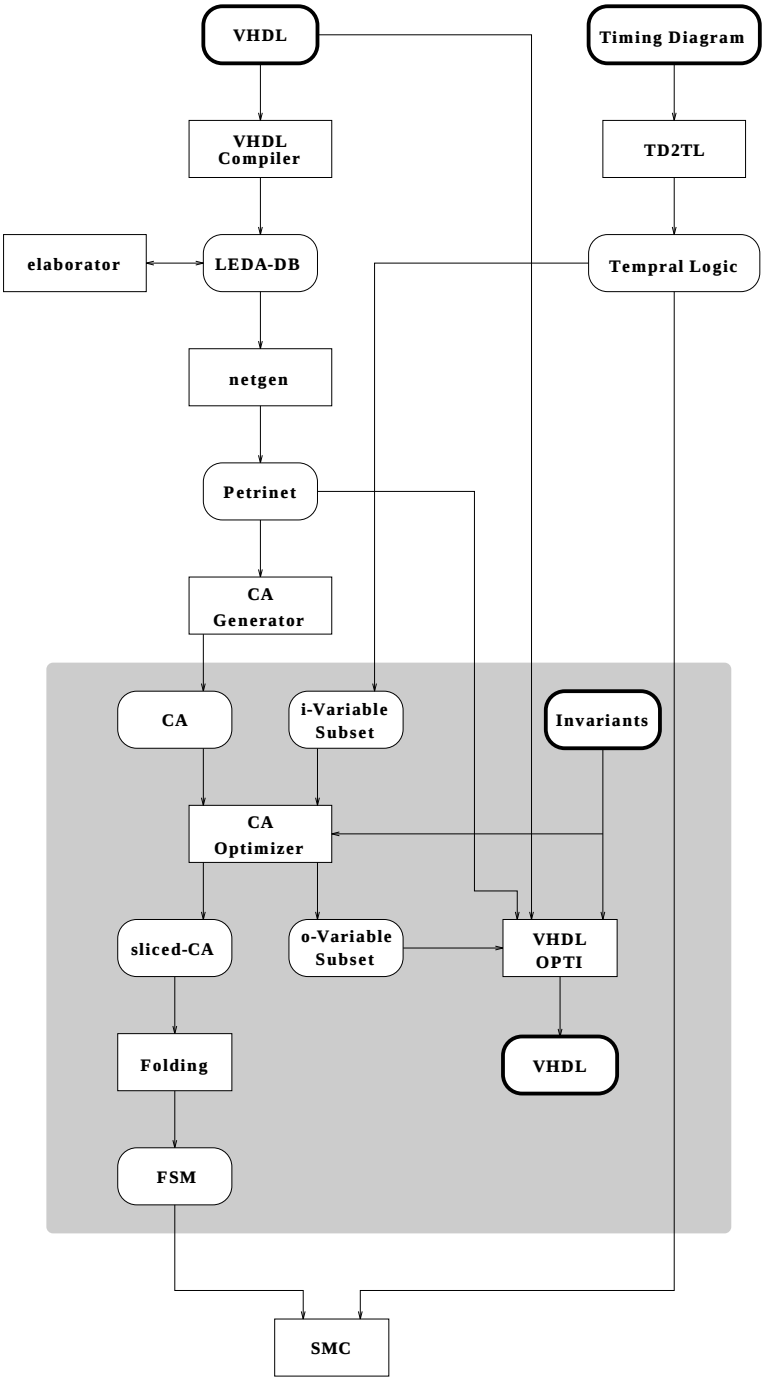


Abbildung 6.16: Compilationsschema mit Slicing

Aus der von einem STD abgeleiteten temporallogischen Formel wird die Menge von BDD Variablen berechnet, anhand derer der Kontrollautomat optimiert werden soll. Desweiteren können Invarianten über die Belegung von Input-Variablen definiert werden, welche den Kontrollfluß innerhalb der Architektur einschränken

können. Als Ausgabe wird ein reduzierter Kontrollautomat erzeugt und eine Menge zur Kodierung notwendiger BDD-Variablen berechnet. Der Kontrollautomat findet nun Eingang in das weitere, unveränderte Verifikationsverfahren. Um ein reduziertes VHDL-Modul zu erhalten, kann nun durch Analyse der Invarianten, der BDD-Variablen, des AADL Petri-Netzes und des VHDL-Source-Codes eine neue, reduzierte VHDL Architektur inkl. Entity Deklaration erzeugt werden.

6.4.4 Quantitative Ergebnisse

Als eine Fallstudie für *safety* und *lifeness Eigenschaften* wurde eine VHDL-Implementierung eines Mutual Exclusion Controllers (MU) herangezogen [Par96]. Die Sicherheitseigenschaft besagte, daß der Controller es nie erlauben wird, daß die Komponenten sich gleichzeitig in ihren critical sections befinden. Die Liveness-Eigenschaft wurde dadurch vorgegeben, daß jede Anforderung einer Komponente auch irgendwann erfüllt würde. Tabelle 6.8 zeigt die Verringerung des Zeitaufwan-

Eigenschaft	Zeit in Sekunden	
	ohne Slicing	mit Slicing
safety	18	3
liveness	565	11

Tabelle 6.8: Benötigte Zeit für die Verifikation

des für die Verifikation der Beispielarchitektur. Für eine modifizierte Version des Traffic Light Controllers - er wurde um eine weitere Straße erweitert - wurden eine Reachability-Analyse durchgeführt (siehe Tabelle 6.9). Hierbei wurde einmal das gesamte Modell und einmal nur beschränkt auf eine Straße durchgeführt.

Eigenschaft	vollständiges Modell	eine Straße
Zeit für Modellberechnung	914 sec.	560 sec.
Größe des Modells	5.8 MB	3.9 MB
Zeit für die Berechnung der Reachabilty	1418 sec.	145 sec.

Tabelle 6.9: Benötigte Zeit für Reachabilty-Analyse

Die Zeit für die Berechnung des vollständigen Modells weicht nur gereingfügig von der Erstellung des Teilmodells ab. Ein ähnliches Verhalten kann auch für die Größe des Modells festgestellt werden. Der eigentliche Vorteil wird jedoch bei der Be-

rechnung der Reachabilty deutlich. Hier wird nur etwa 10 Prozent der Zeit für die Berechnung benötigt.

Wird diese Analyse auf die Anzahl der zur Kodierung notwendigen binären Variablen angewendet, kann für den Benchmarkset dieser Arbeit in Verbindung mit den zugeordneten Zeitdiagrammspezifikationen eine durchschnittliche Reduktion von etwa 25 Prozent ausgemacht werden.

Zusammenfassend lassen sich folgenden Aussagen über die Leistungsfähigkeit des vorgestellten Slicing-Konzeptes machen:

- Die Effizienz steigt mit der Große des Modells und der „Lockerheit“, mit der es vermascht ist. Hierdurch steigt die Wahrscheinlichkeit, daß es voneinander unabhängige Systembereiche gibt, die einzeln betrachtet werden können; z.B. eine von Datentransformationen unabhängige Kontrollflußkodierung.
- Der Zeitaufwand für Slincing ist zu vernachlässigen - die Anwendung führt im worst case nur zur Berechnung des vollständigen Modells.
- STDs sollten so klein wie möglich in der Anzahl ihrer referenzierten Ports spezifiziert werden, um die Startmenge von benötigten Variablen so klein wie möglich zu halten.
- Die Erzeugung von optimiertem VHDL Code führt nicht zu gleichen Reduktionen wie auf dem binären Modell des Kontrollautomaten. Dies wird u.a. dadurch verursacht, das einer VHDL-Variablen eine Vielzahl von binären Variablen zugeordnet ist; z.B. projected waveforms. Sollte auch nur eine dieser Komponenten referenziert werden, muß die „gesamte“ VHDL-Variable im VHDL-Code erhalten bleiben.

6.5 Symbolisches Model Checking

6.5.1 CTL

Um für ein gegebenes endliches System festzustellen, ob es einer temporallogischen Spezifikation genügt, kann Model Checking als automatische Verifikationsmethode eingesetzt werden. Als Spezifikationssprache wird CTL [EC81] eingesetzt, welche zum einen ausdrucksstark genug ist, um eine Vielzahl interessanter temporaler Eigenschaften zu spezifizieren, und zum anderen auch sehr effizient verifiziert werden kann, da das Laufzeit- und Speicherplatz-Verhalten in einer expliziten Graphre-

präsentation des Transitionssystem linear zur Länge der temporallogischen Formel und der Größe des Transitionssystems des zu verifizierenden Systems ist.

Die Logik CTL wird über eine Menge von atomaren Propositionen $a \in \mathcal{A}$ definiert. Die Syntax von CTL ist gegeben als:

$$\begin{aligned} \rho &::= a \mid (\rho) \mid \rho_1 \wedge \rho_2 \mid \rho_1 \vee \rho_2 \mid \neg\rho \mid \forall\varrho \mid \exists\varrho \\ \varrho &::= O\rho \mid \rho_1 \text{ until } \rho_2 \mid \rho_1 \text{ unless } \rho_2 \end{aligned}$$

Die Semantik einer CTL Formel wird in Abhängigkeit von einer Kripke Struktur definiert. Eine Kripke Struktur ist ein totales Transitionssystem $(S, R \subseteq S \times S, \mathcal{I})$. $\mathcal{I}$ ist die Interpretation der atomaren Propositionen a, die sie auf Mengen von Zuständen abbildet. Die Semantik einer CTL Formel ρ ist die Teilmenge von S, in denen ρ gültig ist. Die Semantik einer CTL Pfad Formel ist eine Menge von Pfaden. Ein Pfad π ist eine unendliche Sequenz von Zuständen mit folgender Eigenschaft:

$$\pi = s_1 \ldots s_n \ldots, \ mit \ \forall i \in I\!N : (s_i, s_{i+1}) \in R$$

$\pi[i]$ bezeichnet den Zustand $s_i \in S$, wobei π^i den Suffix von π mit Startzustand s_i beschreibt. Die Menge aller Pfade von R wird als Π bezeichnet. Somit kann die Semantik von CTL formal wie folgt angegeben werden:

$$\begin{aligned} [\![.]\!] : \mathcal{L}_\rho &\rightarrow 2^S \\ \text{mit} & \\ [\![a]\!] &= \mathcal{I}(a) \\ [\![\rho_1 \wedge \rho_2]\!] &= [\![\rho_1]\!] \cap [\![\rho_2]\!] \\ [\![\rho_1 \vee \rho_2]\!] &= [\![\rho_1]\!] \cup [\![\rho_2]\!] \\ [\![\neg\rho]\!] &= S \setminus [\![\rho]\!] \\ [\![\forall\varrho]\!] &= \{s \in S \mid \forall \pi \in \Pi : \pi[0] = s \Rightarrow \pi \in [\![\varrho]\!]\rho\} \\ [\![\exists\varrho]\!] &= \{s \in S \mid \exists \pi \in \Pi : \pi[0] = s \wedge \pi \in [\![\varrho]\!]\rho\} \\ \text{und} & \\ [\![.]\!]\rho : \mathcal{L}_\varrho &\rightarrow 2^\Pi \\ \text{mit} & \\ [\![O\varrho]\!]\rho &= \{\pi \in \Pi \mid \pi^1 \in [\![\varrho]\!]\rho\} \\ [\![\varrho_1 \text{ until } \varrho_2]\!]\rho &= \{\pi \in \Pi \mid \exists i \in I\!N : \pi^i \in [\![\varrho_2]\!]\rho \wedge \forall j \in I\!N : j < i \Rightarrow \pi^j \in [\![\varrho_1]\!]\rho\} \\ [\![\varrho_1 \text{ unless } \varrho_2]\!]\rho &= \{\pi \in \Pi \mid (\exists i \in I\!N : \pi^i \in [\![\varrho_2]\!]\rho \wedge \forall j \in I\!N : j < i \Rightarrow \pi^j \in [\![\varrho_2]\!]\rho) \vee \\ & \qquad \forall i \in I\!N : \pi^i \in [\![\varrho_1]\!]\rho\} \end{aligned}$$

Eine Kripke Struktur $(S, R, \mathcal{I})$ erfüllt dann eine CTL Formel φ, wenn alle Zustände von S φ erfüllen:

$$S \subseteq [\![\varphi]\!]$$

6.5.2 Symbolisches CTL Model Checking

Das ursprüngliche von Clarke et al. entwicklete Model Checking Verfahren [EC86] basiert auf einer expliziten Darstellung des Transitionssystem. Der vorgestellte Algorithmus, bei dem der Zustandsraum enumerativ durch eine Tiefensuche traversiert wurde, konnte durch die Darstellung von Zustandsmengen mit Hilfe von ROBDDs und dem damit möglichen Übergang zur Traversierung durch Breitensuche entscheidend verbessert werden [BCL91].

Symbolisches Model Checking berechnet die Semantik einer temporallogischen Formel ϕ (gegeben z.B. in CTL) relativ zu einer Kripke Struktur $(S, R, \mathcal{I})$ [Jos90] oder einem Transitionssystem unter Ausnutzung der booleschen Charakterisierung für Mengen. Es gibt vier einzelne Phasen der symbolischen Verifikation, die durchlaufen werden müssen:

1. Auffinden einer eindeutigen booleschen Kodierung β für die Zustände S des Transitionssystems.
   Für $s_1, s_2 \in S$ mit $s_1 \neq S_2$: $\beta(s_1) \wedge \beta(s_2) \equiv 0$. Die boolesche charakteristische Funktion einer Teilmenge von S ($S' \subseteq S$) ist die Disjunktion der Kodierungen der Elemente von S':
   $$\Sigma_\beta(S') = \bigvee_{s \in S'} \beta(s)$$
2. Ein Compiler kann dann hierauf aufbauend die charakteristischen Mengen der Kripke Struktur berechnen, d.h. $(\Sigma_\beta(S), \Sigma_\beta(R), \mathcal{I})$. Die booleschen Funktionen werden dann mittels ROBDDs repräsentiert.
3. Hiernach muß $\Sigma_\beta([\![\phi]\!])$ durch boolesche Operatoren berechnet werden.
4. Es muß überprüft werden, ob $\Sigma_\beta(S) \Rightarrow \Sigma_\beta([\![\phi]\!])$, d.h. $\neg\Sigma_\beta(S) \vee \Sigma_\beta([\![\phi]\!])$ eine Tautologie darstellt. Ist dies der Fall, dann erfüllt das System die temporal logische Spezifikation.

Die Erzeugung der charakteristischen Funktionen zur Kodierung der Transitionsrelationen wurde bereits eingehend diskutiert und wird hier nicht weiter verfolgt. Es soll hier nur kurz auf die 3. Phase des symbolischen Model Check Verfahrens eingegangen werden

Semantik des Basisoperatoren

Der symbolische Model Check Algorithmus [BCMD90, BCM+90, BCL91] arbeitet induktiv über der Struktur der CTL Formel. Aus Gründen der Dualität des Existenz-

und Allquantors sowie der until und unless-Operatoren, kann jede CTL Formel in eine äquivalente Formel f transformiert werden, die neben den booleschen Operatoren nur noch aus folgenden Operatoren besteht:

$$\exists O\ f,\ \exists\forall f\ und\ \exists(f\ until\ g)$$

Die Realisierung des ersten Operators $\exists O\ f$ ist straight forward:

$$\exists O\ f = \exists v'_0, \ldots, v'_{n-1}[f(v'_0, \ldots, v'_{n-1}) \wedge R(v_0, \ldots, v_{n-1}, v'_0, \ldots, v'_{n-1})]$$

Die Algorithmen für die Berechnung der beiden anderen Funktionen basieren auf einer Fixpunktberechnung unter Ausnutzung der Realisierung von $\exists O$.

$$\exists\forall f = f \wedge \exists O\ f(\exists\forall f)$$

Für jeden Zustand s gibt es einen Weg, der in s beginnt und an dem f gilt, gdw. f gilt im Zustand s und s hat einen Nachfolgezustand s', an dem auch f gilt. Die Fixpunktcharakterisierung für den Operator

$$\exists[f\ until\ g] = g \vee (f \wedge \exists O(\exists(f\ until\ g)))$$

ist etwas komplexer. Wird an einem Zustand s gestartet, dann wird es einen Pfad geben so daß $\exists(f\ until\ g)$ gilt, gdw. g im aktuellen Zustand s gültig ist oder f gültig ist und s einen Nachfolgerzustand s' besitzt, so daß es einen Pfad mit Startpunkt s' gibt, an dem $f\ until\ g$ gültig ist.

6.5.3 Funktionales symbolisches CTL Model Checking

Funktionales symbolisches CTL Model Checking basiert auf einer modifizierten Berechnung der Vorgänger bzw. Nachfolgerrelation R. Für ein deterministisches Transitionssystem, welches durch n binäre Zustandsvariablen ($V = \{v_0, \ldots, v_{n-1}\}, V' = \{v'_0, \ldots, v'_{n-1}\}$) kodiert werden kann, sei für jede Variable $v' \in V'$ eine NextStep-Funktion f_i gegeben, so daß

$$v'_i = f_i(V).$$

Diese Gleichung kann dazu benutzt werden, die einzelnen Relationen des Transitionssystem zu erzeugen:

$$R_i(V, V') = (v'_i \Leftrightarrow f_i(V)).$$

Die Gesamtrelation R des Systems kann dann als

$$R(V, V') = R_0(V, V') \wedge \cdots \wedge R_{n-1}(V, V').$$

aufgefaßt werden. Mit dieser Relation kann dann das klassische symbolische Model Checking durchgeführt werden. Um jedoch die sehr zeitaufwendige Existenzquantifizierung des Verfahrens, hervorgerufen durch die Konstruktion der R_i, nicht durchführen zu müssen, kann für deterministische Systeme eine reine Substitution zur Anwendung kommen. Für die Realisierung des $\exists O$-Operators bedeutet dies

$$\exists O\ g = g(f_0(V), \ldots, f_{n-1}(V)[v'_0, \ldots, v'_{n-1} \setminus v_0, \ldots, v_{n-1}]).$$

Teil V

Zusammenfassung und Ausblick

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wird eine Realisierung eines CAD-Systems für den Hardware Entwurf auf Systemebene präsentiert. Basierend auf der Hardwarebeschreibungssprache AADL ist ein Framework entwickelt worden, welches alle Phasen eines komplexen, hierarchischen Designs unterstützt. Um das COMDES-Toolset zu realisieren, mußten neben theoretischen auch eine Vielzahl von softwaretechnologischen Problemstellungen, die eine ingenieurmäßige Vorgehensweise verlangen, gelöst werden. Die Verbindung dieser beiden Aspekte stellt die eigentliche Herausforderung dieser Arbeit dar.

Ausgehend von der COMDES-Entwurfsmethodik, wie sie in Abbildung 1.2 graphisch aufbereitet ist, sind eine Vielzahl von Tools realisiert worden, die die Spezifikation, Simulation und Verifikation von AADL-Designs ermöglichen. Der Beitrag dieser Arbeit für das Framework läßt sich in drei Bereiche untergliedern:

1. Erstellung eines offenen Toolsets mit homogenen Userinterface und Datenhaltung.
2. Entwicklung von Konzepten für eine effiziente Simulation von AADL Designs inklusive einer komplexen Debug-Umgebung sowie einer Animation der zur Spezifikation zum Einsatz kommenden symbolischen Zeitdiagramme.
3. Entwicklung von Konzepten für die symbolische, ROBDD basierte Repräsentation von Modellen für AADL Module, um sie symbolischen Verifikationsverfahren zugänglich zu machen.

Anhand dieser ausgezeichneten Bereiche sollen die Ergebnisse dieser Arbeit zusammengefaßt werden:

Das COMDES-System ist als Client/Server-Architektur realisiert worden, um es flexibel bzgl. der Datenhaltung und der Integration von weiteren Tools zu gestalten. Durch diese Systemarchitektur wird eine optimale Ausnutzung der Betriebsmittel eines Computer-Netzwerkes ermöglicht, da sowohl leistungsstarke Rechner als Compute-Server problemlos integriert werden können als auch die X11-basierte Visualisierung problemlos auf mehrere Computer verteilt vorgenommen werden kann. Durch einen zentralen Masterprozeß, der lokal an einen Computer mit einer Designdatenbank gebunden ist, werden alle administrativen Informationen netzwerkweit zentral verbucht. So ist es möglich, für jeden Benutzer einen eigenen, netzwerkweiten Design-Kontext aufzubauen. Außerdem kann mit dieser Realisierung die Konsistenz der Designdatenbasis im Mehrbenutzerbetrieb sichergestellt werden.

Die Simulation des Verhaltens von AADL Modulen basiert auf der Simulation der von ihnen als Zwischenrepräsentation abgeleiteten AADL Petri-Netze. Industrielle Fallstudien, wie z.B. das in [Dö97] präsentierte Multiprozessorsystem, stellen hohe Anforderungen an die Simulation bzgl. Durchsatz, Debugmöglichkeiten und schneller Recompilation des Gesamtdesigns bei einer Modifikation eines Teilmodulen. Als Lösung wird in dieser Arbeit eine abstrakte Netzmaschine vorgestellt, die eine Spezialrechnerarchitektur für AADL Petri-Netze darstellt. Die AADL Module werden via der Petri-Netzzwischenrepräsentation modular in Code dieser abstrakten Maschine transformiert, welcher letztlich von einem normalen C-Compiler in Objektcode übersetzt. Modular bedeutet in diesem Zusammenhang, daß der erzeugte Objektcode unabhängig von seiner möglichen Zielumgebung, bestehend aus weiteren übersetzten AADL Modulen generiert werden kann. Hierdurch ist es möglich, komplexe, hierarchische AADL-Designs durch reines Linken von Objektmodulen der COMDES-Designdatenbasis schnell zu generieren.

Zur Spezifikation des temporalen Verhaltens von AADL Modulen werden im COMDES-System symbolische Zeitdiagramme verwendet. Um dem Designer frühzeitig einen Eindruck von seiner Spezifikation zu geben, wird in dieser Arbeit ein Ansatz für ihre Animation gegeben, welche sich wiederum auf die Simulation von AADL Petri-Netzen abstützt. Hierzu wird eine Übersetzung der Zeitdiagramme in eine Netzdarstellung angegeben. Die Simulation basiert jedoch nicht auf einem compilativen Verfahren, wie bei der Simulation der AADL-Module, sondern es wird ein interpretativen Ansatz vorgestellt, der sowohl den komplexen Simulationsalgorithmus als auch die Behandlung des symbolisch repräsentierten Datenraums effizient durchführt.

Die Verifikation von AADL Modulen gegen ihre temporallogische Spezifikation wird mittels symbolischem Model Checking durchgeführt. Bei diesem Verfahren wird das von einem AADL Modul abgeleitete AADL Petri-Netz in ein symbolisches, durch ROBDDs repräsentiertes Modell übersetzt. Als zentrale Struktur bei der Transformation der AADL Petri-Netze in ROBDDs ist die Struktur eines Kontrollautomaten eingeführt worden. Durch ihn es möglich, die Kontroll- und Datentransformationsaspekte zu trennen und ein jeweilig effizientes Transformationsschema anzugeben. Da trotz der Anwendung von symbolischen Verfahren zur Modellkodierung sehr große Strukturen aufgebaut werden können, die nicht mehr handhabbar sind, stehen dem Designer zwei vollautomatische Techniken zur Modellreduktion zur Verfügung, um seine spezifizierten Eigenschaften für ein gegebenes Modell doch noch nachweisen zu können:

1. Da Eigenschaften eines AADL Moduls nur Objekte des Interfaces referenzieren, können alle Zustände des Modells, die vom Interface her nicht unterscheidbar sind, durch Substitution eliminiert werden. Dieses Verfahren hat sich als besonders effektiv für VHDL und VHDL/S StateChart erwiesen, da bei diesen Sprachen nur Wertveränderungen im Interface zu ganz bestimmte Zeitpunkten, den wait-Konfigurationen, beobachtbar sind und somit alle Zustände, die keinen wait-Konfigurationen entsprechen eliminiert werden können. Dies hat zur Folge, daß nicht nur das Modell weniger Zustände hat sondern daß auch die Fixpunktberechnungen während des symbolischen Model Checkings schneller berechenbar sind.

2. Das symbolische Modell kann durch „Slicing“ auf die Anteile reduziert werden, die für die Verifikation einer gegebenen Eigenschaft notwendig ist. Hierbei handelt es sich um ein sichere Reduktion, d.h. sollte die temporallogische Eigenschaft für das reduzierte Modell wahr bzw. falsch sein, so gilt die gleiche Eigenschaft auch für das Originalmodell.

Durch die theoretischen und praktischen Erfahrungen, die im Rahmen des COMDES-Projektes gewonnen wurden, ergaben sich Kooperationen in weiterführende Projekten. Hier seien beispielhaft das ESPRIT-Projekt Nr. 6123 „FORMAT“ und das BMFT-Projekt „KorSo“ (Korrekte Software) angeführt. Das COMDES-System wurde desweiteren sehr erfolgreich auf vielen Konferenzen und Ausstellungen präsentiert. Hier seien beispielhaft die „EDAC 1992“ (Brüssel), „Cebit'93“ (Hannover), „CAV 1993“ (Elounda), „EURO-DAC/EURO-VHDL 1993“ (Hamburg) und die „EDAC 1994“ (Paris) angeführt.

Weiterführende Arbeiten sollten an der Verifikationskomponente des COMDES-Systems vorgenommen werden:

- Als wichtiges Kriterium für den Erhalt einer kompakten ROBDD-Darstellung des Modells ist das Auffinden einer optimalen Variablenordnung. Zwar kann man sich dieser durch Reorderingtechniken angenähern, eine Berechnung einer guten initialen Variablenordnung ist jedoch von elementarem Interesse.
- Die beiden vorgestellten Verfahren zur Modellreduktion führen zwar zu guten Resultaten, trotzdem sind hier weitergehende Forschungsarbeiten in dem Bereich der Abstraktionstechniken wünschenswert.

Literaturverzeichnis

[AB97] Telelogic AB. Information. *www: http://www.telelogic.se/company/pressrel/sdtpre.htm*, 1997.

[AHP96] R. Alur, G. Holzmann, and D. Peled. An Analyzer for Message Sequence Charts. *T. Margaria and B. Steffen (eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer LNCS*, 1996.

[AHU83] A. Aho, B. Hopcroft, and D. Ullmann. *Data Structures and Algorithms.* Addison-Wesly, 1983.

[Ake78] Sheldom B. Akers. Binary Decision Diagrams. In *Transactions on Computers*, volume 6, pages 509–516. IEEE, 1978.

[Arm89] James R. Armstrong. *Chip–Level Modeling with VHDL.* Series in Computer Engineering. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1989.

[Ash90a] Peter J. Ashenden. *The VHDL Cookbook.* Published by Electronic Mail, University of Adelaide, South Australia, first edition, jul 1990.

[Ash90b] P.J. Ashenden. The VHDL Cookbook. Technical report, University of Adelaide, Dept. of Computer Science, 1990.

[ASU87] Aho, Sethi, and Ullmann. *Compilers.* Addison-Wesly, 1987.

[AvT87] J.K. Annot and R.A.H. van Twist. A Novel Deadlock Free And Starvation Free Packet Switching Communication Processor. In G. Goos and J. Hartmanis, editors, *Proceedings, PARLE, Parallel Architectures and Languages Europe*, number 258 in Lecture Notes in Computer Science, pages 68–85, Philips Research Laboratories, Eindhoven, jun 1987.

[Bal88] Helmut Balzert. *Einführung in die Softwareergonomie.* Walter de Gryter, 1988.

[Bal89] Helmut Balzert. *Die Entwicklung von Software-Systemen.* BI-Wissenschaftverlag, 1989.

[BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. *ACM/IEEE Design Automation Conference*, 1991.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and Jim Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer science*, 1990.

[BCMD90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Sequential Cirsuit Verification Using Symbolic Model Checkingng. Technical report, Carnegie Mellon University, School of Computer Science, 1990.

[BEP93] D. Borrione, H. Eveking, and L. Pierre. Formal Proofs from HDL Descriptions. *Technical Report*, 1993.

[Bes96] Eike Best. Partial order verification with pep. *POMIV'96, Partial Order Methods in Verification, G. Holzmann, D. Peled and V. Pratt (Hrsg.), American Mathematical Society*, 1996.

[BF88] E. Best and C. Ferandez. *Nonsequential Processes - A Petri Net View.* EATS. Springer-Verlag, 1988.

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. Sdl with applications from protocol specification. *Prentice Hall International, Hertfordshire, UK*, 1991.

[BPS91] D. Borrione, L. Pierre, and A. Salem. PREVAIL: A Proof Environment for VHDL descriptions. *Proc. of the Advanced Research Workshop on Correct Hardware Design Methodologies*, 1991.

[Bra84] Wilfried Brauer. *Automatentheorie.* Teubner-Verlag, Stuttgart, 1984.

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. *Proceedings 27th DAC, Orlando, Florida*, 1990.

[Bro90] F.M. Brown. *Boolean Reasoning: The Logic of Boolean Equations.* Kluwer Akademic Publisher, 1990.

[Bro95] Udo Brockmeyer. Optimierungstechniken in der Modellgenerierung für automatische Verifikationstechniken im Hardwareentwurf. Master's thesis, Universität Oldenburg, 1995.

[Bry86] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *Transaction on Computers*, C-35, 1986.

[CBM90] Oliver Coudert, Christian Berthet, and Jean Christophe Madre. Formal Boolean Manipulations for the Verification of Sequential Machines. Technical report, Bull Research Center, 1990.

[CP87] P. Camurati and P. Prinetto. Formal verification of hardware correctness. In *CHDL*, pages 225–247, 1987.

[CP89] P. Camurati and P. Prinetto. Formal verification of hardware correctness: introduction and survey of current research. In *IEEE Computer*, pages 8–19, 1989.

[CPo89] V. Claus, H. Pargmann, and other. Abschlußbereicht der Projektgruppe NESSI. Technical report, Universität Oldenburg, 1989.

[CZJ+92] Ney Calazans, Qinhai Zhang, Ricardo Jacobi, Bruno Yernaux, and Anne-Marie Trullemans. Advanced Ordering and Manipulation Techniques for Binary Decision Diagrams. In *Proceedings, The European Conference on Design Automation*, Brussels, Belgium, 1992.

[Dam90] W. Damm. *Multiprozessorsysteme.* Skript zur Vorlesung. Universität Oldenburg, 1990.

[DD88] W. Damm and G. Döhmen. A Compositional Petri-Net Semantics for AADL. Technical report, ESPRIT 415 report No 5, University of Oldenburg, 1988.

[DD89a] Werner Damm and Gert Döhmen. A Net-Based Specification Method for Computer Architecture Design. In Jaco W. de Bakker, editor, *Languages for Parallel Architectures: Design, Semantics, and Implementation Models*, pages 51–109 ,chapter 2. Wiley & Sons, Chichester, 1989.

[DD89b] Werner Damm and Gert Döhmen. Specifying Distributed Computer Architectures in AADL. *Parallel Computing*, 1989.

[DD90] Werner Damm and Gert Döhmen. Using Formal Specification and Verification Methods in the Design of Parallel Architectures. In H. Kersten, editor, *Sichere Software: formale Spezifikation und Verifikation vertrauenswürdiger Systeme*, Heidelberg, 1990. Hüthig Buch Verlag.

[DDG+89] W. Damm, G. Döhmen, V. Gerstner, J. Helbig, B. Josko, F. Korf, and T. Peikenkamp. AADL Language Document. Technical report, Universiät Oldenburg, 1989.

[DDG+94] D. Dams, G. Döhmen, R. Gerth, R. Herrmann, P. Kelb, and H. Pargmann. Design of a VHDL/S Model Checker Based on Adaptive State and Data Abstraction. *Computer Aided Verification*, 1994.

[DH95] G. Döhmen and R. Herrmann. A Deterministic Finite-State Model for VHDL. In Carlos Delgado Kloos and Peter T. Breuer, editors, *Formal Semantics for VHDL*, pages 170–204. Kluwer Academic Publisher, 1995.

[DJ93] W. Damm and B. Josko. A Net-Based Semantics for VHDL. In *Proc. EURO-DAC'93 with EURO-VHDL'93, Hamburg*, pages 514–519, Los Alamitos, California, September 1993. IEEE Computer Society Press.

[DJS93a] W. Damm, B. Josko, and R. Schlör. A Net-Based Semantics for VHDL. *EURO-DAC with EURO-VHDL*, 1993.

[DJS93b] W. Damm, B. Josko, and R. Schlör. Linking VHDL with Formal Verification Tools: How to Generate Finite State Models out of VHDL Designs. Technical report, University of Oldenburg, 1993.

[DJS93c] W. Damm, B. Josko, and R. Schlör. Specification and Verification of VHDL-based System-Level Hardware Designs. Technical report, LIPA-RI Summerschool, 1993.

[Dö97] Gert Döhmen. *Abstraction in Hardware Description and Design: The Design, Semantics and Implementation of a Hardware Description Language*. PhD dissertation, University of Oldenburg, Oldenburg, Germany, 1997. (in Preparation).

[Döh94] G. Döhmen. Petri Nets as Intermediate Representation Between VHDL and Symbolic Transition Systems. In *Proc. EURO-DAC'94 with EURO-VHDL'94, Grenoble*, pages 572–577, Los Alamitos, California, September 1994. IEEE Computer Society Press.

[EC81] E.A. Emerson E.M. Clarke. Design and synthesis of synchronization skeletons using Branching Time Temporal Logic. *LNCS, Proceedings of Workshop on Logics of Programs*, pages 52–71, 1981.

[EC83] A.P. Sistla E.M. Clarke, E.A. Emerson. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications: A Practical Approach. *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[EC86] A.P. Sistla E.M. Clarke, E.A. Emerson. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications: A Practical Approach. *ACM Transactions on Programming Languages and Systems*, pages 244–263, 1986.

[EFT91] Reinhard Enders, Thomas Filkorn, and Dirk Taubner. Generating BDD's for Symbolic Modelchecking in CCS. In *Proceedings Third Workshop on Computer Aided Verification*, pages 263–278, Aalborg, 1991.

[Eve91] Hans Eveking. *Verifikation digitaler Systeme.* Leitfaeden und Monographien der Informatik. Teubner-Verlag, Stuttgart, Darmstadt, 1991.

[FDT95] J. Fischer, E. Dimitrov, and U. Taubert. Analysis and Formal Verification of SDL'92 Specifications using Extended Petri Nets. *Research Report, Dep. of Comp. Sci., Humboldt-University*, 1995.

[FJ91] F. Feldbrugge and K. Jensen. Computer Tools for High-Level Petri Nets. In *High-Level Petri Nets – Theory and Application*, pages 691–717, Berlin, Germany, 1991. K. Jensen and G. Rozenberg, Springer Verlag.

[FL88] Fischer and LeBlanc. *Crafting a Compiler.* The Benjamin/Cummings Company, 1988.

[FYBSV93] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli. Dynamic Variable Reordering for BDD Minimization. *Proceedings of EURO-DAC*, 1993.

[Gra95] Bernd Grahlmann. Pep: Ein werkzeug zur modellierung, simulation, analyse und verifikation paralleler systeme. *W. Reisig J. Desel, A. Oberweis, editor,* Algorithmen und Werkzeuge für Petrinetze*, number 309, pages 20-25. AIFB Universität Karlsruhe*, 1995.

[Gul93] Jürgen Gulbins. *UNIX*. Springer Verlag, 1993.

[Har87] David Harel. StateCharts: A Visual Formalism for Complex Systems. *Sience of Computer Programming 8*, 1987.

[Her92] Ronald Herrmann. Übersetzung hardwareorientierter Datenstrukturen in Binary Decision Diagrams. Master's thesis, Universität Oldenburg, 1992.

[HK94] J. Helbig and P. Kelb. On OBDD-Representation of StateCharts. *Proceedings European Design Automation Conference*, 1994.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 8(8), 1978.

[Hoe91] Stefan Hoereth. Improving the Performance of a BDD–based Tautology–Checker. In *Proceedings Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 387–396, Turin, jun 1991.

[HP94] R. Herrmann and H. Pargmann. Computing Binary Decision Diagrams for VHDL Data Types. Technical Report 2, Universität Oldenburg, 1994.

[HPSS87] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the Formal Semantics of StateCharts. *Proceeding First IEEE, Symposium on Logic in Computer Science*, 1987.

[HR76] L. Hyafil and R.L. Rivest. Constructing Optimal Binary Decision Trees is NP–Complete. In *IPL 5*, pages 15–17, 1976.

[HS66] J. Hartmanis and R.E. Stearns. *Algebrais Structure Theory of Sequential Machine*. Prentice Hall, 1966.

[HU90] J. E. Hopcroft and J. D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, München, 1990.

[HW91] B. Heinemann and K. Weihrauch. *Logik für Informatiker*. Teubner-Verlag, Stuttgart, 1991.

[IEE87] IEEE. VHDL IEEE standard 1076 LRM. *IEEE*, 1987.

[Ill85] J.A. Illik. *Erfogreich Programmieren mit C.* Sybex Verlag, 1985.

[JAB+92] Jawahar Jain, Magdy Abadir, James Bitner, Donald S. Fussell, and Jacob A. Abraham. IBDDs: An Efficient Functional Representation for Digital Circuits. In *Proceedings, The European Conference on Design Automation*, Brussels, Belgium, 1992.

[Jan90a] Geert L.J.M. Janssen. Progress in Verification with PTL. Final bra 3281 (asicis) deliverable, tue / a2, Eindhoven University of Technology, 1990.

[Jan90b] Geert L.J.M. Janssen. Verification of Finite State Machines using Temporal Logic. Technical report, Eindhoven University of Technology, 1990.

[Jos90] Bernhard Josko. Verifying the correctness of AADL modules using model checking. In Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Lecture Notes in Computer Science 430*, pages 386–400. Stepwise Refinement of Distributed Systems; Models, Formalisms, Correctness, Springer-Verlag, 1990.

[Jos93] Bernhard Josko. Modular Specification and Verification of Reactive Systems. Habilitationsschrift, Universität Oldenburg, 1993.

[JPHS92] S.-W. Jeong, B. Plessier, G.D. Hachtel, and F. Somenzi. Variable Ordering for Binary Decision Diagrams. In *Proceedings, The European Conference on Design Automation*, Brussels, Belgium, 1992.

[Kor90] Franz Korf. Simulation modularer AADL Petri-Netze. Technical report, Universität Oldenburg, 1990.

[Kor91] Franz Korf. Net-Based Efficient Simulation of AADL Specifications. Technical report, Universität Oldenburg, 1991.

[Kor97] Franz Korf. *System-Level Synthesewerkzeuge: Von der Theorie zur Anwendung.* PhD dissertation, University of Oldenburg, Oldenburg, Germany, 1997. (in Preparation).

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, New Jersey, 1978.

[Led92] Leda C. A., France. *VHDL IEEE procedural interface*, 1992.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusik, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addisson Wesley, 1989.

[LSU89] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1989.

[McC86] E.J. McCluskey. *Logic Design Principles.* Prentice Hall, 1986.

[McM93] K. L. McMillan. Symbolic Model Checking. *Kluwer Academic Publishers*, 1993.

[MH92] McGraw-Hill. ADA - The Complete Reference. *McGraw-Hill*, 1992.

[MIY90] Shinichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *Proceedings 27th Design Automation Conference*, pages 52–57, Orlando, Florida, 1990. ACM/IEEE.

[Mor82] B.M.E. Moret. *Decision Trees and Diagrams.* ACM Computing Survey, 1982.

[Nut89] G.J. Nutt. A Flexible, Distributed Simulation System. *Application and Theory of Petri Nets*, 1989.

[OC93] S. Olcoz and J.M. Colom. Toward a Formal Semantics of IEEE Std. VHDL 1076. *EURO-VHDL*, 1993.

[Old91a] E.-R. Olderog. *Nets, Terms and Formulas.* Cambridge Tracts in Theoretical Computer science 23, 1991.

[Old91b] E.-R. Olderog. *Programmverifikation – Sequentielle, parellele und verteilte Programme.* Springer Verlag, Berlin, 1991.

[Par91] Hergen Pargmann. Effiziente Simulation von AADL Petrinetzen. Master's thesis, Universität Oldenburg, 1991.

[Par95] Hergen Pargmann. COMDES: Simulation and Verification of System-Level Hardware Designs. *AIS-Report, Universität Oldenburg*, 1995.

[Par96] Hergen Pargmann. Where no man has been before! Simulation and Verification of High-Level Petri Nets. *Petrinetz-Workshop, Universität Oldenburg*, 1996.

[Pau86] Paul. Ein Verfahren zur Effizienten Simulation modifizierter Petri-Netze. Technical report, Universität Dortmund, 1986.

[Pau91] L.C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[PH94] H. Pargmann and R. Herrmann. Efficient Symbolic Transition Systems for VHDL. Technical Report 1/94, Universität Oldenburg, 1994.

[PH95] H. Pargmann and R. Herrmann. Reduced Symbolic Transition Systems for VHDL. *Proceedings GI/ITG Workshop, Passau*, 1995.

[Pil88] R. Piloty. *Schaltwerktechnik.* Vorlesungsscript, Technische Hochschule Darmstadt, 1988.

[Pö95] Renee Pörschke. Ermittlung von Variablenordnungen für VHDL-Transitionssysteme. Master's thesis, Universität Oldenburg, 1995.

[PS91] A. Pnueli and M. Shalev. What is in a Step: On the Semantics on StateCharts. *LNCS, Theoretical Aspects of Computer Science*, 526, 1991.

[Pup88] F. Puppe. *Einführung in Expertensysteme.* Springer-Verlag, Berlin, 1988.

[PW91] P. Paeppinghaus and P. Warkentin. Implementation of a Symbolic Circuit Simulation Tool – A Note on Structure Sharing for Graphs representing Boolean Functions. Internal report, Siemens AG, Research Laboratories for Applied Computer Science and Software, Otto–Hahn–Ring 6, W– 8000 Munich, 1991.

[Ram78] F.J. Rammig. Mixed Level Modelling and Simulation of VLSI Systems. *Logic Design and Simulation*, 1978.

[Ram89] F.J. Rammig. *Systematischer Entwurf digitaler Systeme.* Teubner Verlag, 1989.

[Rei85] Wolfgang Reisig. *Petri Nets. An Introduction.* EATCS Monographs on Computer Science, Springer Verlag, 1985.

[Rei91] U. Reimer. *Einführung in die Wissensrepräsentation.* Teubner-Verlag, Stuttgart, 1991.

[RT88a] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator : A System for Constructing Language-based Editors.* Texts and Monographs in Computer Science. Springer-Verlag, 1988.

[RT88b] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer-Verlag, third edition, 1988.

[Rud93] R. Rudell. Dynamic Variable Ordering for ordered Binary Decision Diagrams. *IEEE/ACM International Conference on CAD*, 1993.

[Sch86] Schreiner. *System-Programmierung in UNIX (I und II).* Teubner-Verlag, 1986.

[Sch91] H.-J. Schneider. *Lexikon der Informatik und Datenverarbeitung.* Oldenbourg-Verlag, München, 1991.

[SD93] Rainer Schlör and Werner Damm. Specification and Verification of System-Level Hardware-Designs using Timing Diagrams. *The European Conference on Design Automation with the European Event in ASIC Design*, pages 518–524, 1993.

[SSW94] S. Schöf, M. Sonnenschein, and R. Wieting. Sequentielle und verteilte Simulation von THOR-Netzen. In *Proc. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 61–66, Berlin, Germany, October 1994. Forschungsbereicht 309, Universiät Karlsruhe.

[SSW95] S. Schöf, M. Sonnenschein, and R. Wieting. Effecient Simulation of THOR Nets. In *Proc. of the 16th International Conference on Application and Theory of Petri Nets*, pages 61–66, Berlin, Germany, June 1995. LNCS 935, Springer Verlag.

[Ste88] Franz Stetter. *Grundbegriffe der Theoretischen Informatik.* Springer Verlag, 1988.

[Ste92] W.R. Stevens. *Programmieren von UNIX-Netzen.* Hanser Verlag, 1992.

[Sto88] H. Stoyan. *Programmiermethoden der Künstlichen Intelligenz.* Springer-Verlag, Berlin, 1988.

[SUN] SUN Microsystems. *SUNOS 4.1.3 Operating System Manuals 1-13.*

[uRS94] F. Korf und R. Schlör. Interface controller synthesis from requirement specifications. In *Proceedings, The European Conference n Design Automation*, pages 385–394, Paris, France, 1994. IEEE, Computer Society Press.

[vT90] J. van Tassel. The semantics of VHDL with VAL and HOL: Towards Practical Verification Tools. *Technical Report 196, University of Cambridge, UK*, 1990.

[WG84] Waite and Goos. *Compiler Construction.* Springer Verlag, 1984.

[Wir86] Niklaus Wirth. *Algorithmen und Datenstrukturen.* Teubner Verlag, 1986.

[Yoe90] M. Yoeli. Formal Verification of hardware Design. In *IEEE Computer Society Press Tutorial*, pages 10–45, 1990.

[Zeh87] C.A. Zehnder. *Informationssysteme und Datenbanken.* Teubner Verlag, 1987.

[Zel89] Stephan Zelewski. *Komplexitätstheorie.* Vieweg Verlag, 1989.

Lebenslauf

WS 1995/86	Beginn des Informatik Studiums an der Universität Oldenburg
1986 - 1988	Tätigkeiten als Wissenschaftliche Hilfskraft: ◦ Projekt *Moby* bei Prof. Dr. V. Claus ◦ Übungsgruppenleiter am Fachbereich Informatik für: Pascal, Smalltalk-80 und Software Praktikum
1989	*Vordiplom*
1988 - 1991	Wissenschaftliche Hilfskraft im Projekt *COMDES* bei Prof. Dr. W. Damm
1991	*Diplom* Thema der Diplomarbeit: Effizienente Simulation von AADL Petrinetzen
seit 1991	Mentor für Technische Informatik und Wirtschaftsinformatik an der Fernuniversität Hagen
1991 - 1997	Wissenschaftlicher Mitarbeiter bei Prof. Dr. W. Damm an der Universität Oldenburg und OFFIS
1997 - 1999	SAP-Berater bei der UMC GmbH
seit 1.4.1999	Leiter der Entwicklung der AS-Inpro GmbH - eine Tochter der Lufthansa Systems AG -