Christian-Albrechts-Universität zu Kiel

Diplomarbeit

Diplomarbeit

HW/SW Co-Design für einen reaktiven Prozessor

cand.-Ing. Sascha Gädtke

4. Mai 2007

Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

betreut durch:
Prof. Dr. Reinhard von Hanxleden,
Dipl.-Inf. Claus Traulsen

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Inhaltsverzeichnis

1. Einleitung

Reaktive Systeme müssen permanent auf Eingaben aus ihrer physischen Umgebung reagieren, wobei die Geschwindigkeit der Reaktion von der Umgebung bestimmt wird. Die Anforderungen an reaktive Systeme unterscheiden sich grundsätzlich von den Anforderungen an herkömmliche Softwaresysteme. Die Bedeutung von Rechenleistung tritt gegenüber funktional und zeitlich vorhersagbarem Verhalten in den Hintergrund. Genau wie in der physischen Welt Vorgänge parallel ablaufen, bestehen reaktive Systeme häufig aus mehreren nebenläufigen Threads, deren zeitliches Verhalten jeweils von der Umgebung bestimmt wird. Das Scheduling der Threads sollte statisch erfolgen und die Ausführungszeiten einen möglichst geringen Versatz aufweisen, um ein deterministisches Verhalten zu gewährleisten. In vielen eingebetteten Anwendungen spielt zudem ein geringer Energieverbrauch eine wichtige Rolle.

Herkömmliche Programmiersprachen und Entwicklungswerkzeuge sind nicht oder nur bedingt für die Entwicklung reaktiver Systeme geeignet, da sie den besonderen Anforderungen nicht gerecht werden. *Synchrone Sprachen* [4] sind an die Anforderungen von reaktiven Systemen angepasst und stellen insbesondere Konstrukte zum Ausdruck von Nebenläufigkeit und Preemption bereit. Synchrone Sprachen basieren auf der *Synchronizitäts-Hypothese*, in der angenommen wird, dass die Berechnung einer Reaktion keine Zeit kostet bzw. in einem definierten Zeitintervall erfolgt, das verglichen mit der Zeit der Signaländerungen klein ist. Dies erlaubt die Definition einer formalen Semantik.

Eine synchrone Sprache ist Esterel [6]. Esterel benutzt ein besonderes Ausführungsmodell, das die Ausführung in diskrete (*logische*) *Ticks* oder *Instanzen* unterteilt. Die Kommunikation zwischen mehreren Threads innerhalb eines Programms und die Kommunikation mit der Umwelt erfolgt über *Signale*. Es wird angenommen, dass alle Signale während eines Ticks genau einen Zustand haben. In der Syntax von Esterel sind Konstrukte zum Ausdruck von Preemption und Nebenläufigkeit enthalten. Esterel kann nicht direkt ausgeführt werden, sondern wird üblicherweise in eine andere Programmiersprache wie C oder Java zur Softwaresynthese oder in eine Hardwarebeschreibungssprache wie VHDL zur Hardwaresynthese übersetzt. Ausführungsplattform bei der Softwaresynthese ist im Allgemeinen ein herkömmlicher Prozessor. Neuere Ansätze für die Softwaresynthese benutzen *reaktive Prozessoren*, die im Gegensatz zu herkömmlichen Prozessoren reaktive Kontrollflussstrukturen direkt unterstützen und ein vorhersagbares zeitliches Verhalten aufweisen.

Ein spezieller reaktiver Prozessor ist der *Kiel Esterel Processor* (*KEP*) [35], dessen Instruktionssatz eng an die Esterel Syntax und Semantik angelehnt ist. Die meisten Esterel Statements werden direkt unterstützt und können innerhalb eines Instruktionszyklus abgearbeitet werden. Dies macht die Ausführung von Esterel

Programmen sehr effizient. Die Berechnung von komplexen Ausdrücken muss auf dem KEP, wie auch auf anderen Prozessoren, in mehrere Instruktionen sequenzialisiert werden und ist im Gegensatz zu einer Hardwareimplementierung als Schaltnetz aufwändig.

Das Ziel dieser Arbeit ist es, durch die Auslagerung der Berechnung von komplexen Ausdrücken in einen mit dem KEP verbundenen Logikblock die Anzahl der benötigten Instruktionen reduzieren zu können und damit eine beschleunigte Programmausführung zu erreichen. Bei dem vorgestellten HW/SW Co-Synthese-Ansatz werden alle komplexen Ausdrücke aus einem gegebenen ESTEREL Programm extrahiert und in Hardware synthetisiert. Die Ausgangssignale der Logik werden als zusätzliche Eingangssignale in den KEP geführt. Auf dem KEP wird eine modifizierte Programmversion ausgeführt, in der alle komplexen Ausdrücke durch die im Logikblock berechneten zusätzlichen Signale ersetzt sind. Die Partitionierung des gegebenen Programms in Software- und Hardwareteile erfolgt auf ESTEREL Ebene, um den Ansatz möglichst lange hardwareunabhängig zu halten und die Verwendung von herkömmlichen Verifikations- und Synthesewerkzeugen zu erlauben. In einem zweiten Schritt wird aus dem Hardwareteil die Logik synthetisiert und der Softwareteil auf dem KEP ausgeführt.

Durch das Co-Design gibt es keine negative Beeinflussung der Taktfrequenz des KEP, da die Laufzeit des kritischen Pfades im Logikblock auch für große Ausdrücke deutlich unter der Dauer für einen Instruktionszyklus liegt. Der zusätzliche Hardwareaufwand für den KEP ist gering und im Vergleich zur Größe des KEP zu vernachlässigen. In den Experimenten kann gezeigt werden, dass durch die Beseitigung von Berechnungen von komplexen Ausdrücken aus dem Programm der Energieverbrauch pro Tick gesenkt und die *Worst Case Reaction Time* (WCRT) deutlich verbessert werden können.

Ein älterer Stand dieser Arbeit [19] wurde im Rahmen einer Posterpräsentation für Studenten auf der Konferenz *Languages, Compilers, and Tools for Embedded Systems* (*LCTES'06*) veröffentlicht. Ein weiteres Paper mit dem Titel *HW/SW Co-Design for Reactive Processing* [20] wurde zur Veröffentlichung auf der *International Conference on Hardware - Software Codesign and System Synthesis* (*CODES+ISSS'07*) eingereicht.

Die Ausarbeitung unterteilt sich in folgende Kapitel: Kapitel 2 führt in die Grundlagen dieser Arbeit ein. Es gibt eine Einführung in die synchrone Sprache ESTEREL und stellt verschiedene Verfahren zur Ausführung von ESTEREL vor, darunter andere Co-Design-Ansätze und reaktive Prozessoren. Als spezieller reaktiver Prozessor wird der KEP, auf dessen Basis die Co-Synthese von ESTEREL implementiert werden soll, sowie der dazugehörige Compiler zur Übersetzung von ESTEREL in den KEP Instruktionssatz eingeführt. Darüber hinaus werden Algorithmen und Werkzeuge zur Logikminimierung und Hardwarebeschreibungssprachen vorgestellt, die für die Implementierung der Co-Synthese verwendet werden.

Kapitel 3 beschreibt das Verfahren zur Co-Synthese in den einzelnen Schritten Partitionierung, Logikminimierung und HW/SW Synthese und geht dabei auf verschiedene auftretende Probleme ein.

Kapitel 4 schließlich quantifiziert die Vor- bzw. Nachteile des Co-Synthese Ansatzes im Vergleich zur reinen Softwaresynthese eines ESTEREL Programms auf dem KEP.

2. Grundlagen

2.1. Reaktive Systeme

Rechnergestützte Systeme lassen sich in drei Kategorien einteilen [4]:

- *Transformative Systeme* berechnen Ausgabewerte aufgrund von Eingabewerten und stoppen dann (z.B. Compiler).

- *Interaktive Systeme* stehen in ständiger Interaktion mit der Umwelt. Das zeitliche Verhalten wird durch das System bestimmt. Es gibt keine Garantie für die Verfügbarkeit der Services, das zeitliche Verhalten oder die Determiniertheit. Beispiele sind Betriebssysteme, Datenbanksysteme oder das Internet.

- *Reaktive Systeme* stehen wie interaktive Systeme in ständiger Interaktion mit ihrer Umwelt. Die Umgebung stellt Anforderungen an die Verfügbarkeit, das zeitliche Verhalten und die Determiniertheit. Beispiele sind Prozesssteuerungen oder Signalprozessoren.

Große Systeme lassen sich oft nicht eindeutig zuordnen, sondern bestehen aus mehreren Subsystemen, die den verschiedenen Kategorien angehören.

Bei reaktiven Systemen unterscheidet man weiter zwischen *datenverarbeitenden* (*data handling*) und *steuerungsdominierten* (*control dominant*) reaktiven Systemen. Datenverarbeitende reaktive Systeme produzieren permanent Ausgabewerte zu ihren Eingabewerten. Ein typisches Beispiel sind Signalprozessoren, z.B. in Audio- und Videoanwendungen als Filter oder zum En- und Decodieren. Steuerungsdominierte reaktive Systeme produzieren diskrete Ausgabesignale zu Eingabesignalen. Man findet solche Systeme z.B. in der Prozesssteuerung, in Eingebetteten Systemen, bei der Überwachung von komplexen Systemen (z.B. als Sicherheitssystem um Fehlbedienungen und andere Anomalien zu erkennen) oder in Treibern und Controllern.

Genau wie in der physischen Welt verschiedene Ereignisse gleichzeitig ablaufen, müssen in Reaktiven Systemen oft mehrere Prozesse nebenläufig modelliert werden. Preemptionmechanismen werden benötigt, um einzelnen Prozessen dauerhaft oder temporär den Kontrollfluss zu entziehen. Um ein deterministisches Verhalten zu gewährleisten, sollte ein statisches Scheduling der Prozesse möglich sein, und die Ausführungszeit sollte möglichst geringe Schwankungen (Jitter) aufweisen.

Synchrone Programmiersprachen [4] sind speziell auf diese Bedürfnisse zugeschnitten.

2.2. Esterel

Esterel [6] ist eine *nebenläufige*, *synchrone* und *imperative* Programmiersprache, die auf die Implementierung von steuerungsdominierten *Reaktiven Systemen* zugeschnitten ist.

J.-P. Marmorat und J.-P. Rigault stellten 1982 bei der Entwicklung eines autonomen Roboterautos fest, dass herkömmliche Programmiersprachen keine zufriedenstellenden Ausdrucksmechanismen zur Darstellung zeitlicher Zusammenhänge bereitstellten [18]. Sie erkannten die Notwendigkeit von *Delay-* und *Preemption-*Mechanismen und forderten, dass Signale (der Eingabe- und Ausgabe-, sowie Kommunikationsmechanismus in Esterel) nur in diskreten Zeiteinheiten (*Instanzen*) wiederholt werden dürften. Daraufhin wurde von einer Forschergruppe unter Leitung von Gérard Berry am Centre de Mathématiques Appliqueés an der Ecole des Mines in Paris die Sprache Esterel, sowie eine formale Semantik, erste Verifikationswerkzeuge und Compiler entwickelt.

Der Name der Programmiersprache leitet sich ab von dem Namen des französischen Mittelgebirges „Esterel“, das zwischen Cannes und Saint-Raphaël an der Côte d'Azur gelegen ist. Der Name klingt ein wenig wie das französische Wort für Echtzeit: *temps-réel*. Esterel wird stetig weiterentwickelt und ist aktuell in der Version 7, welche zur Zeit als IEEE Standard bearbeitet wird. Die vorliegende Arbeit basiert auf dem älteren Esterel V5 Standard, da dieser sehr etabliert und durch eine Vielzahl von freien Werkzeugen unterstützt wird. Version 7 unterscheidet sich hauptsächlich in der besseren Unterstützung von Datentypen und einigen syntaktischen Erweiterungen.

Esterel wird herkömmlicherweise entweder nach C [8][29][16] oder eine Hardwarebeschreibung [5] in z.B. Verilog oder VHDL übersetzt. Im ersten Fall wird der C Code mit einem herkömmlichen Compiler für das Zielsystem (i.A. ein eingebetteter Mikrocontroller) übersetzt. Im zweiten Fall dient die Hardwarebeschreibung zur Konfiguration eines FPGA oder Synthese eines ASIC.

Weitere synchrone Programmiersprachen sind Lustre [22] und Signal [21], deren Entwicklung ebenfalls auf die achtziger Jahre zurückgeht und deren Entwickler eng mit der Esterel-Arbeitsgruppe zusammengearbeitet haben.

Eine grafische synchrone Sprache, die eng mit Esterel verwandt ist, ist SyncCharts [1].

2.2.1. Synchronizität in Esterel

Bei herkömmlichen Programmiersprachen ist es schwierig, wenn nicht unmöglich, das zeitliche Verhalten oder die Determiniertheit zu beweisen. Synchrone Sprachen wie Esterel bedienen sich eines besonderen Formalismus zur Modellierung des zeitlichen Verhaltens.

Definition 1 (Perfekte Synchronizität) *Ein System arbeitet* perfekt synchron, *wenn alle Reaktionen des Systems ohne Zeitverzögerung ausgeführt werden. d.h. Ausgaben werden zeitgleich zu den Eingaben produziert.*

Das Ideal der *perfekten Synchronizität* lässt sich in der Praxis nicht realisieren. Hier ist „ohne Zeitverzögerung“ durch „rechtzeitig“, d.h. vor der nächsten Eingabe zu ersetzen. Im synchronen Programmiermodell wird die kontinuierliche physische Zeit in eine logische Zeitrepräsentation diskretisiert. Diese besteht aus abzählbaren Interaktionen mit der Umwelt, den bereits in der Einleitung erwähnten Instanzen oder logischen Ticks. Innerhalb einer Instanz hat jedes Signal einen eindeutigen Status bzw. Wert und die Berechnung der Reaktion erfolgt – zumindest hypothetisch – unendlich schnell. Instanzen werden typischerweise in einer bestimmten Frequenz ausgeführt; die Intervallänge T zwischen zwei Ticks wird dabei von den Echtzeitanforderungen des Systems bestimmt. Jede Instanz besteht aus einer endlichen Anzahl von Statements, die keine datenabhängigen Schleifen enthalten dürfen. Dies ermöglicht eine Worst Case Execution Time Analyse (*WCET*) für die Dauer einer Instanz.

ESTEREL Statements, die in einer Instanz $t \in \mathbb{N}$ gestartet werden, enden in einer Instanz $t + \delta$. Ist $\delta = 0$, so spricht man von einem *instantanen* oder *transienten* Statement, ansonsten von einem *delayed* (verzögerten) oder *temporalen* Statement.

2.2.2. Syntax

In den folgenden Abschnitten wird eine kurze Beschreibung von ESTEREL V5 gegeben, eine ausführliche Darstellung gibt der *Esterel V5 Language Primer* [7].

Signale und Sensoren

Der Nachrichtenaustausch innerhalb des Programms und mit der Umwelt erfolgt über *Signale* und *Sensoren*. Signale lassen sich unterscheiden in *Pure Signals* und *Valued Signals*:

- *Pure Signals* besitzen nur einen *Status*, der in Abhängigkeit, ob das Signal in der jeweiligen Instanz emittiert wurde, *present* oder *absent* sein kann.
- *Valued Signals* besitzen zusätzlich zum Status noch einen Wert. Es gibt nur fünf vordefinierte Datentypen zwischen denen keine implizite Konvertierung möglich ist: `boolean`, `integer`, `float`, `double` und `string`. Für valued Signals kann zusätzlich eine *Combine-Funktion* angegeben werden, die kommutativ und assoziativ sein muss. Vordefiniert sind `and` und `or` für boolsche Signale und `+` und `*` für `integer`, `float` und `double`. Wird ein Signal ohne Combine Funktion mehrmals in einer Instanz emittiert, so ist dies ein Laufzeitfehler. Ist eine Combine Funktion angegeben, wird diese dazu genutzt, einen eindeutigen Wert zu ermitteln. Wird der Wert innerhalb einer Instanz mehrere Male abgefragt, ist der Wert immer der selbe.

Interface Signale können von der Art `input`, `output`, `inputoutput` oder `return` sein. Zu beachten ist, dass `input`-Signale emittiert und `output`-Signale getestet bzw. gelesen werden können. Ein `return` Signal ist ein spezielles Eingangssignal. Es signalisiert die Beendigung eines externen Tasks, der mit dem `exec`-Statement ausgeführt

wurde. Wie jedes andere `input` Signal können `return`-Signale einen Wert haben und es kann eine Combine-Funktion angegeben werden.

Im Interface können zusätzlich *Input Relations* angegeben werden. Input Relations stellen weitere Bedingungen an `input`- bzw. `return`- Signale. Die Relation `relation A # B` ist eine Inkompatibilitäts- oder Ausschlussrelation. Sie gibt an, dass die Signale `A` und `B` nicht gleichzeitig auftreten dürfen. Die Relation `relation A => B` ist eine Implikationsrelation. Das Signal `B` darf nur present sein, wenn `A` present ist.

Im Gegensatz zu valued Signals als Eingangssignal besitzen Sensoren nur einen Wert und keinen Status. Sie können jederzeit im Programm gelesen werden.

```
module EXAMPLE:
input A;                          % deklariert Pure Signal A als Eingangssignal
input B := 0 : integer;           % Eingangssignal B vom Typ 'integer' mit initialem Wert '0'
input C : combine integer with +; % Eingangssignal C vom Typ 'integer' mit Addition als Com-
output O : integer;               % Ausgangssignal O vom Typ 'integer'          \\ binefunktion

...

end module
```

Lokale Signale können überall innerhalb eines Programms deklariert werden. Dies geschieht ähnlich wie für Interface Signale. Ein lokales Signal ist nur innerhalb des Rumpfes der Deklaration gültig. Ein Signal kann innerhalb des Rumpfes erneut deklariert werden, dabei ist das Signal der äußeren Deklaration in diesem Bereich nicht sichtbar. Steht eine lokale Signaldeklaration innerhalb einer Schleife, kann diese mehrere Male in einer Instanz ausgeführt werden. Hierbei wird jedesmal eine neue Kopie des Signals erstellt. Dieses Verhalten wird *Reinkarnation* genannt.

```
...
loop
  signal S in
  present S then emit O end present;
  pause;
  emit S;
  end signal
end loop
...
```

Das vorstehende Programmfragment zeigt die lokale Deklaration des Signals `S` innerhalb einer Schleife. Der Status wird vor dem `pause`-Statement getestet und ist in der ersten Instanz *absent*. Nach dem `pause`-Statement wird `S` emittiert, die Schleife terminiert und startet den Rumpf instantan erneut. Obwohl `S` in der selben Instanz getestet wird, ist es aufgrund der Reinkarnation *absent*.

Variablen

Im Gegensatz zu Signalen haben Variablen nur einen Wert und keinen Status und können mehrere aufeinanderfolgende Werte innerhalb einer Instanz erhalten. Genau wie valued Signals können sie vom Typ `boolean`, `integer`, `float`, `double` oder `string` sein. Sie können überall innerhalb des Programms deklariert werden. Ihr Gültigkeitsbereich ist der Rumpf p der Deklaration. Wird innerhalb des Rumpfes die Variable

erneut deklariert, so ist die im äußeren Gültigkeitsbereich deklarierte Variable nicht sichtbar.

Im Gegensatz zu Signalen können sich mehrere Prozesse nicht eine Variable mit lesendem und schreibendem Zugriff teilen. Wird eine Variable in einem Thread verändert, so kann der andere weder lesend noch schreibend auf die Variable zugreifen. Deshalb können Variablen nicht zur Kommunikation zwischen Threads eingesetzt werden.

Expressions

Es gibt in ESTEREL drei Arten von Expressions: *Signal Expressions*, *Data Expressions* und *Delay Expressions*.

Signal Expressions sind Boolsche Ausdrücke über den Status mehrerer Signale. Sie sind das Argument eines `present` Signal Tests (s. Kap. 2.2.2) oder einer Delay Expression. Zur Verknüpfung sind die Operatoren `not`, `and` und `or` möglich und die Ausdrücke sind beliebig klammerbar. Auf den Status eines Signals oder einer Signal Expression in der vorigen Instanz kann mit `pre()` zugegriffen werden.

Data Expressions kombinieren die Werte von Signalen, Sensoren, Konstanten, Traps und Variablen. Auf Konstanten und Variablen kann direkt unter ihrem Namen zugegriffen werden. Den Wert eines Signals oder Sensors `S` erhält man über `?S` und den Wert einer Trap[1] `T` über `??T`. Operatoren sind die folgenden:

- *Boolsche Operatoren*: `not`, `and` und `or`
- *Vergleichsoperatoren*: `=` für Gleichheit, `<>` für Ungleichheit, `<`, `<=`, `>` und `>=`
- *Arithmetische Operatoren*: `+`, `-`, `*` und `/`.

Außerdem sind die Ausdrücke beliebig klammerbar. Auf ein Signal `S` kann mit `pre(?S)` auf den Wert in der vorigen Instanz zugegriffen werden.

Delay Expressions werden in *temporalen* Statements wie `await` oder `abort` eingesetzt. Es gibt drei Formen von Delays:

- *Standard Delays* werden durch eine einfache Signal Expression angegeben, z.B. `Meter and not Second`. Delay Expressions können nicht in der ersten Instanz erfüllt werden, sondern erst in der nächsten Instanz, in der die Signal Expression *present* ist.
- *Immediate Delays* können schon in der ersten Instanz erfüllt werden. Ein Immediate Delay startet mit dem Schlüsselwort `immediate` gefolgt von einem Signal Identifier oder von einer in eckigen Klammern `[]` geklammerten Signal Expression. Beispiel: `immediate[Meter and not Second]`.
- *Count Delays* werden definiert durch einen ganzzahligen Zähler gefolgt von einem Signal Identifier oder von einer in eckigen Klammern `[]` geklammerten Signal Expression.

[1] Traps sind der Exception-Handling Mechanismus in Esterel (s.Kap. 2.2.2)

Beispiel:

```
...
  await I1; emit O1
||
  await immediate I2; emit O2
||
  await 3 I3; emit O3
...
```

Der `pre`-Operator liefert den Status eines Signals oder einer Signal Expression bzw. Wert eines Signals in der vorigen Instanz:

- Den Status eines Signals `S` oder einer Signal Expression e in der vorigen Instanz erhält man mit `pre(S)` bzw. `pre(`e`)`.
- Den Wert eines Signals `S` in der vorigen Instanz erhält man mit `pre(?S)`

Eine Schachtelung von `pre`-Operatoren ist dabei nicht erlaubt. In der ersten Instanz eines Signals `S` ist der Status `pre(S)`=*absent*. Ist das Argument jedoch eine Signal Expression e, werden zur Definition zwei Hilfsoperatoren verwendet: pre_0(`S`) liefert den Wert von `S` in der vorigen Instanz mit dem Wert *absent* in der ersten Instanz und pre_1(`S`) liefert den Wert von `S` in der vorigen Instanz mit dem Wert *present* in der ersten Instanz. `pre(`e`)` ist dann wie folgt definiert:

$$\begin{aligned} \texttt{pre}(e) &= pre_0(e) \\ pre_i(e \texttt{ or } e') &= pre_i(e) \texttt{ or } pre_i(e') \\ pre_i(e \texttt{ and } e') &= pre_i(e) \texttt{ and } pre_i(e') \\ pre_i(\texttt{not } e) &= \texttt{not} pre_{\neg i}(e) \end{aligned}$$

Dabei gilt: $\neg 0 = 1$ und $\neg 1 = 0$.

Statements

Im Folgenden werden nur die wichtigsten Esterel Statements besprochen. Eine umfassende Dokumentation liefert der Esterel Primer [7].

- *Sequenz und Paralleloperator*
  - **`p;q`**: p und q werden sequenziell ausgeführt: Ist p beendet, wird instantan q gestartet.
  - **`p||q`**: Der Paralleloperator führt p und q als nebenläufige Threads aus. Er terminiert erst, wenn beide Threads terminiert sind.
- *Instantane Statements*
  - **`nothing`**: Macht nichts und terminiert sofort.
  - **`loop p end`**: Wiederholt den Rumpf p der Schleife unendlich. Der Rumpf selbst darf jedoch nicht instantan sein.

  - **`signal S in` *p* `end`**: Mit diesem Statement kann an jeder Stelle im Programm ein lokales Signal S deklariert werden. Der Gültigkeitsbereich ist der Rumpf p. Wie im Abschnitt Signale schon besprochen wurde, können Signale redeklariert werden. Das Signal im äußeren Gültigkeitsbereich ist im inneren Gültigkeitsbereich nicht sichtbar. Wird ein lokales Signal innerhalb einer Schleife deklariert, kann es zu mehreren *Reinkarnationen* des Signals kommen: D.h. es wird jedes mal eine neue Kopie des Signals instantiiert.
  - **`emit S`, `emit S(`*e*`)`**: Emittiert das Signal S, ein valued Signal ggf. mit dem Wert des Ausdrucks e.
  - **`present S then` *p* `else` *q* `end`**: Testet den Status der Signal Expression S. Ist S present geht der Kontrollfluss instantan nach p, anderenfalls nach q.
  - **`if E then` *p* `else` *q* `end`**: Ist die Data Expression E wahr, wird p unmittelbar gestartet, ansonsten q.

- *Delay Statements*

  - **`pause`**: Das `pause`-Statement terminiert in der nächsten Instanz.
  - **`await` *delay* `do` *p* `end`**: Das `await` verzögert bis die Delay Expression *delay* zutrifft. Dann wird der optionale Rumpf p ausgeführt.
  - **`every` *delay* `do` *p* `end`**: Das `every` Statement ist ein temporales Schleifenkonstrukt. Es wartet auf die Delay Expression *delay* und führt dann den Rumpf p aus. Tritt die Delay Expression erneut auf, bevor p terminiert ist, wird die Ausführung von p neu gestartet.
  - **`sustain S`, `sustain S(`*e*`)`**: Bleibt für immer aktiv und emittiert das Signal `S` (ggf. mit dem Wert der Expression e) in jeder Instanz. `sustain S` ist eine Kurzschreibweise für `loop emit S; pause end`.

- *Preemptive Statements*

  - **`suspend` *p* `when` *delay***: Der Rumpf p wird suspendiert, wenn die Delay Expression *delay* zutrifft, ansonsten wird er wieder fortgesetzt. *Dabei ist zu beachten:* Da die Zeit innerhalb von p angehalten wird, beginnt für ein innerhalb von p lokal definiertes Signal S nach dem Fortsetzen die nächste Instanz, egal wie viele Instanzen global vergangen sind. D.h. `pre(S)` bzw. `pre(?S)` hat den Status bzw. den Wert zur Zeit der Suspendierung.
  - **`[weak] abort` *p* `when` *delay***: Beendet den Rumpf p, wenn die Delay Expression zutrifft. Im Falle eines `weak abort` erhält der Rumpf ein letztes Mal die Kontrolle.

- *Exception Handling*

– **`trap T in` *p* `handle T do` *q* `end`**: Der Trap-Mechanismus ist der Exception-Handling Mechanismus von Esterel. `trap T in p` entspricht dem `try{}`-Block in C++ oder Java. Der Rumpf *p* wird instantan abgebrochen, wenn innerhalb von *p* ein `exit(T)` Statement ausgeführt wird. Dies entspricht dem `throw` Statement in C++ oder Java. Optional kann für jeden Trap ein bestimmter Handler aufgerufen werden. Dies entspricht dem `catch` in C++ oder Java. Als Erweiterung unterstützt ESTEREL Valued Traps mit denen zusätzlich ein Fehlercode geworfen werden kann.

Als *Pure Esterel* wird eine Teilsprache von ESTEREL bezeichnet. In Pure ESTEREL gibt es nur Pure Signals und keine valued Signals. Die Menge der Statements ist reduziert auf eine nicht-redundante Teilmenge aller ESTEREL Statements, aus denen sich die übrigen Statements ableiten lassen. Die Menge dieser so genannten Kernel Statements umfasst folgende Elemente: `nothing`, `emit S`, `pause`, `present S then p else q`, `suspend p when S`, `p;q`, `loop p end`, `p || q`, `trap T in p end`, `exit T` und `signal S in p end`.

2.2.3. Semantiken von Esterel

Bevor ein ESTEREL Programm kompiliert wird, muss zunächst eine semantische Analyse erfolgen, da ein syntaktisch korrektes ESTEREL Programm nicht in jedem Fall semantisch korrekt ist. Gründe hierfür sind die instantane Signal- und Kontrollübertragung innerhalb einer Instanz. Im Laufe der Entwicklung von ESTEREL wurden verschiedene Semantiken entwickelt: *Logical Behavioral Semantics*, *Constructive Behavioral Semantics*, *Logic/Constructive State Behavioral Semantics*, *Constructive Operational Semantics* und *Constructive Circuit Semantics*, wobei die konstruktiven Semantiken äquivalent sind. Eine formale Einführung in diese Thematik findet sich in [6].

Logische Korrektheit

Ein Programm ist *logisch reaktiv* bezüglich einer Eingabe, wenn mindestens ein logisch kohärenter globaler Zustand existiert. Ein Programm ist *logisch deterministisch* bezüglich einer Eingabe, wenn höchstens ein logisch kohärenter globaler Zustand existiert. Logische Kohärenz bedeutet, dass ein Signal genau dann den Status *present* hat, wenn es in der selben Instanz emittiert wurde.

Definition 2 *Ein Programm ist* logisch korrekt*, wenn es logisch reaktiv und deterministisch ist.*

Das bedeutet, dass es in jeder Instanz zu einer Eingabe genau eine mögliche Reaktion gibt. Das folgende Listing[2] zeigt ein logisch korrektes Programm:

```
module P2:
signal S in
```

[2] Quelle für die alle Listings im Abschnitt 2.2.3: *The Constructive Semantics of Pure Esterel* [6]

```
  emit S;
  present O then
    present S then
      pause
    end;
    emit O
  end
end signal
end module
```

Die einzig logisch kohärente Annahme ist `S` *present* und `O` *absent*: Würde man beispielsweise `O` als *present* annehmen, würden die Statustests `present O` und `present S` zutreffen und das `pause` Statement ausgeführt werden. Die Anweisung `emit O` würde erst in der nächsten Instanz ausgeführt werden, was im Widerspruch zu der Annahme `O` *present* steht.

Die folgenden beiden Programme sind nicht logisch korrekt:

```
module P3:
output O;
  present O else emit O end
end
```

```
module P4:
output O;
  present O then emit O end
```

Das erste Programm ist nicht reaktiv, da für alle Belegungen von `O` kein logisch kohärenter Zustand existiert; das zweite ist nicht deterministisch, da beide Annahmen `O` *absent* und `O` *present* logisch kohärent sind.

Nachteil der Analyse auf logische Korrektheit ist die wenig intuitive *self-justification*. Zum Beispiel möchte man, dass das im folgenden Listing gezeigte, logisch korrekte Programm, das eine Kombination aus *P3* und *P4* darstellt, abgelehnt wird:

```
module P9:
  present O1 then emit O1 end
||
  present O1 then
    present O2 else emit O2 end
  end
```

Hier gibt es genau eine logisch kohärente Annahme: `O1` und `O2` *absent*. Hier rechtfertigt z.B. die Annahme `O1` absent, dass `emit O1` nicht ausgeführt wird.

Konstruktivität

Bei der Konstruktivitätsanalyse wird die Kausalität des Informationsflusses berücksichtigt, es werden keine spekulativen Annahmen gemacht. Es wird für jedes Signal in einer Instanz statisch berechnet, ob es emittiert werden muss (*must*) oder nicht emittiert werden kann (*cannot*). Kann über ein Signal keine Aussage getroffen werden, wird das Programm abgelehnt. Konstruktivität stellt ein schärferes Kriterium zum Annehmen bzw. Ablehnen eines Programms dar, als die Logische Korrektheit. Dabei gilt: *Jedes konstruktive Programm ist auch logisch korrekt.*

Bei der Konstruktivitätsanalyse wird das Programm `P9` abgelehnt, da für `O1` und `O2` kein Status ermittelt werden kann.

Sowohl die Überprüfung von logischer Korrektheit als auch die Überprüfung von Konstruktivität sind sehr aufwändig. Der Nachweis der logischen Korrektheit ist NP-Vollständig, die Konstruktivitätsanalyse ist quadratisch zur Programmgröße. Daher wird häufig lediglich gefordert, dass das Programm zyklenfrei ist, d.h. dass keine zyklischen Signalabhängigkeiten existieren. Da die azyklischen Programme eine Teilmenge der konstruktiven Programme bilden, ist damit sicher gestellt, dass das Programm konstruktiv ist. Es existieren aber auch Programme mit zyklischen Signalabhängigkeiten, die konstruktiv sind. Ein Beispiel dafür ist der *Token Ring Arbiter* [7, S. 93]. Allerdings kann jedes konstruktive, zyklische Programm in ein äquivalentes azyklisches umgewandelt werden [25].

2.3. Synthese von Esterel

Ein ESTEREL Programm wird herkömmlicherweise in eine andere nicht-synchrone Sprache wie C oder VHDL übersetzt und dann in Software oder Hardware synthetisiert. *Co-Designs* sind hybride Ansätze aus Software- und Hardwaresynthese. Einen neueren Ansatz stellen *Reaktive Prozessoren* dar, deren Befehlssatz eine direkte Ausführung von ESTEREL oder einer anderen Synchronen Sprache erlauben.

2.3.1. Software- und Hardwaresynthese

Bei der *Softwaresynthese* [8][16][29] wird ein ESTEREL Programm zunächst in eine andere Programmiersprache wie C oder Java übersetzt und anschließend mit einem entsprechenden Compiler für das Zielsystem übersetzt. Das Zielsystem kann z.B. ein eingebetteter Mikrocontroller oder auch ein PC sein. Bei der *Hardwaresynthese* [5] wird ein ESTEREL Programm in die Beschreibung eines elektronischen Schaltkreises (z.B. in BLIF, Verilog oder VHDL) überführt. Aus der Hardwarebeschreibung kann ein ein anwendungsspezifischer Schaltkreis (ASIC) oder ein Schaltkreis auf einem Field Programmable Gate Array (FPGA) synthetisiert werden. Die Übersetzung erfolgt über *Automaten-*, *Netzlisten-* und *Kontrollflussgraphen*-basierte Ansätze.

- *Automaten-basiert*: Jedes Esterel Programm lässt sich als einzelner Zustandsautomat oder als eine Kombination aus mehreren Zustandsautomaten darstellen. In einem weiteren Schritt wird die Beschreibung des Zustandsautomaten zur Softwaresynthese in eine gewöhnliche Programmiersprache wie C, oder zur Hardwaresynthese in eine Hardwarebeschreibungssprache wie VHDL (vgl. VHDL-Synthese Kap.6) übersetzt. Implementiert wird dieses Verfahren beispielsweise durch den *Esterel V5* Compiler. Der erzeugte Code ist zwar sehr schnell, kann aber schon für kleine Programme sehr groß werden, da sich bei der Kombination von mehreren parallelen Automaten ihre Zustände multiplizieren können, was zu einem exponentiellen Wachstum führt, der sog. Zustandsexplosion.
- *Netzlisten-basiert*: Die Transformation eines ESTEREL Programms in eine Netzliste erfolgt analog zu der Constructive Circuit Semantics. Für jedes ESTEREL

Statement sind bestimmte Hardware-Templates definiert, so dass die Übersetzung in eine Netzliste „mechanisch“ erfolgen kann. Dieser Schaltkreis kann dann entweder simuliert oder in Hardware synthetisiert werden. Implementiert wird dieses Verfahren beispielsweise durch den *Esterel V5* Compiler. Der Code ist linear mit der Größe des ESTEREL Programms und damit wesentlich kompakter als bei dem Automaten-basierten Ansatz, aber auch deutlich langsamer.

- *Kontrollflussgraphen-basiert*: Aus dem ESTEREL Programm wird als Zwischenrepräsentation ein nebenläufiger Kontrollflussgraph generiert. Dieser wird zunächst in einen sequenziellen Kontrollflussgraphen transformiert, um ein Programm mit einem statischen Scheduling zu erzeugen. Der generierte Code ist kompakt und schnell, die Anforderungen an das ESTEREL Programm sind jedoch restriktiver. So werden beispielsweise Programme mit zyklischen Signalabhängigkeiten abgelehnt.

Softwaresynthese ist zwar sehr flexibel und kostengünstig, es ist jedoch sehr aufwändig, reaktive Kontrollflussstrukturen und Nebenläufigkeit für einen Sequentiellen Prozessor zu modellieren. Bei der Hardwaresynthese muss auch bei kleinen Änderungen am Programm die gesamte Hardware neu synthetisiert werden; datenintensive Berechnungen lassen sich in Hardware nur ungünstig implementieren. Für ESTEREL V5 lässt sich zudem nur Pure ESTEREL in Hardware implementieren.

2.3.2. Co-Design

Co-Design-Ansätze, wie z.B. POLIS oder Metropolis [2, 3], vereinen Ansätze für Software- und Hardwaresynthese. Die Ausführungsplattform besteht aus einem Mikrocontroller und einer mit ihm verbundenen, benutzerspezifischen Hardware. Ein ESTEREL Programm wird partitioniert in einen Hardware- und Softwareteil, sowie ein Interface zwischen Hard- und Software. Die Partitionierung richtet sich danach, ob bestimmte Programmmodule besser in Software oder in Hardware synthetisiert werden können. Dadurch lassen sich einige Nachteile der reinen Hardware- und Softwarelösungen beseitigen. Man erbt jedoch die geringe Flexibilität der Hardwarelösung und behält einen hohen Aufwand für die Implementierung von Kontrollstrukturen.

2.3.3. Reaktive Prozessoren

Bei der reinen Softwaresynthese erhält man durch die aufwändige Implementierung reaktiver Kontrollstrukturen einen sehr unhandlichen Programmcode. Mit so genannten *Gepatchten Reaktiven Prozessoren*, wie im RePIC oder ReFLIX [13, 31]-Ansatz, versucht man diesem Problem zu begegnen. Hierbei handelt es sich um gewöhnliche Mikrocontroller mit einem Patch. Der Patch ist ein externer Hardwareblock, der den Befehlssatz des Mikrocontrollers um reaktive Instruktionen erweitert. ReFLIX unterstützt jedoch nicht die direkte Ausführung von ESTEREL und hält auch nicht vollständig die ESTEREL Semantik ein. RePIC unterstützt nur eine Teilmenge von

ESTEREL, es fehlen Konstrukte für Suspendierung und Multithreading. Eine Multiprozessorerweiterung von RePIC stellt der EMPEROR [14] dar, der die Behandlung des Parallelоperators erlaubt.

Ein Reaktiver Prozessor, dessen Befehlssatz und Architektur speziell auf die Syntax und Semantik von ESTEREL ausgelegt sind, wird im folgenden Kapitel vorgestellt.

2.4. Kiel Esterel Prozessor (*KEP*)

Der Befehlssatz und die Architektur des *Kiel Esterel Processors KEP* [35, 23] sind im Gegensatz zu den gepatchten reaktiven Prozessoren speziell für die Syntax und Semantik von ESTEREL ausgelegt.

Zur Unterstützung von Preemptions verfügt der KEP über eine konfigurierbare Anzahl von Wächtern (*Watcher*). Diese überwachen das Auftreten von Signalen, die eine Preemption triggern, sofern sich der Programmzähler im Rumpf der Preemption befindet.

Zur Unterstützung von Nebenläufigkeit verwaltet der KEP für jeden Thread einen eigenen Programmzähler. Das Scheduling erfolgt prioritätsbasiert: In jedem Instruktionszyklus wird der Thread mit der höchsten Priorität ausgeführt. Prioritäten werden bei der Initialisierung von Threads vergeben und können über spezielle Statements für den aktiven Thread geändert werden (s. Kap. 2.4.2).

2.4.1. Architektur

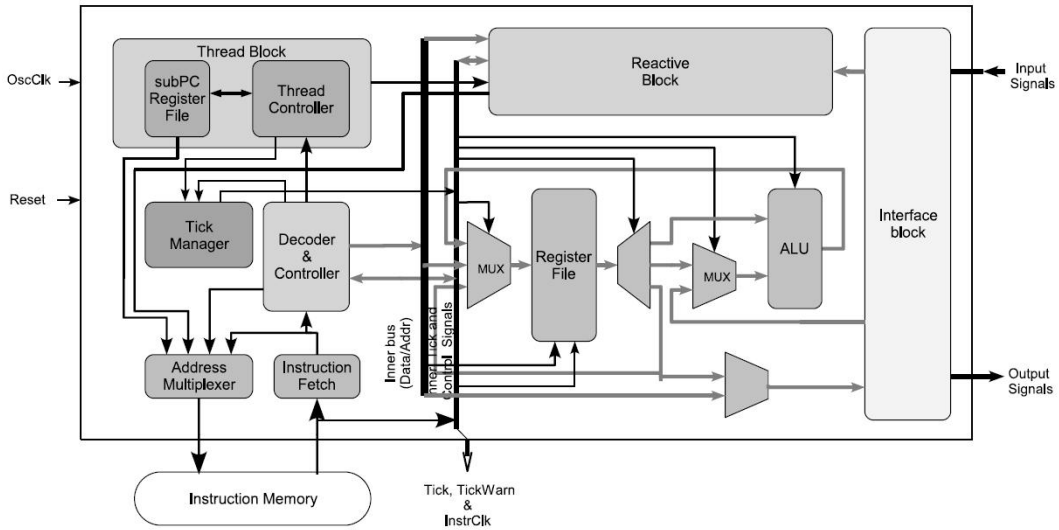


Abb. 2.1.: Schematische Darstellung der KEP Architektur (Quelle: [24])

In Anlehnung an die drei Ebenen (*interface layer*, *reactive kernel* und *data handling layer*), aus denen ein reaktives Programm besteht, bilden drei Blöcke die Haupt-

komponenten der KEP Architektur: Der *Data Handlig Block*, der *Reactive Multi-Threading Core* und der *Interface Block*. Eine schematische Darstellung der *KEP* Architektur ist in Abbildung 2.1 zu sehen.

Der *Data Handling Block* ermöglicht die Berechnung von Expressions. Er besteht aus einem Register zum Speichern von Variablenwerten und Zwischenergebnissen und einer ALU.

Der *Reactive Multi-Threading Core* steuert den Kontrollfluss des Programms. Er stellt Strukturen zur Behandlung von Nebenläufigkeit, Preemption, Exceptions und Delays bereit.

Im *Interface Block* werden zum einen der Status und der Wert aller Signale, auch der lokalen Signale, gespeichert und zum anderen bildet er die Schnittstelle zur Umwelt. Zur Speicherung des Status der Signale dient das *SinoutReg* Register. Die Werte von valued Signals werden in einem RAM, dem *SDatReg*, gespeichert.

Der KEP führt Ticks (Instanzen) mit einer festen Frequenz aus. Bis zum Beginn des nachfolgenden Ticks müssen alle Instruktionen des aktuellen Ticks abgearbeitet sein. Alle KEP Assembler Instruktionen benötigen physikalisch bedingt eine bestimmte Zeit, können aber innerhalb eines Instruktionszyklus (entspricht drei Prozessortaktzyklen) ausgeführt werden. Der *TickManager* steuert die Initiierung neuer Ticks. Das reservierte Signal `_TICKLEN` gibt dem *TickManager* die Dauer eines Ticks in Prozessortaktzyklen an. Das Signal wird am Anfang des Programms über `emit _TICKLEN,x` gesetzt. Sind alle Instruktionen einer Instanz abgearbeitet, wartet der Prozessor bis zum Anfang der nächsten Instanz. Die minimale `_TICKLEN` ist die maximal mögliche Anzahl an KEP Assembler Instruktionen, die pro Instanz ausgeführt werden können. Eine konservative Näherung für diesen Wert liefert die Worst Case Reaction Time Analyse (WCRT) [9]; sie berechnet eine obere Schranke für die minimale `_TICKLEN`.

2.4.2. Befehlssatz

Der Befehlssatz des KEP umfasst die Esterel Kernel Statements sowie eine Auswahl an komplexeren Statements, um den resultierenden Code möglichst kompakt halten zu können. Hinzu kommt die Unterstützung von Esterel Expressions, insbesondere Delay Expressions (standard, immediate, count), sowie die Unterstützung von valued Signals und des `pre`-Operators. Da sich die meisten Esterel Statements direkt in KEP Assembler Instruktionen abbilden lassen, können die meisten Esterel Statements innerhalb eines Instruktionszyklus abgearbeitet werden.

ESTEREL Syntax	KEP Assembler	Anmerkungen

[
  p_1
  ||
  ⋮
  ||
  p_n
]
	PAR $prio_1$, $startAddr_1$, id_1
	  ...
	PAR $prio_n$, $startAddr_n$, id_n
	PARE *endAddr*
	$startAddr_1$:
	  ...
	$startAddr_n$:
	  ...
	endAddr:
	JOIN
		Die PAR Anweisung erzeugt einen neuen Thread mit der Priorität $prio_x$ und der Threadnummer id_x. Der Thread startet im Speicher bei der Adresse $startAddr_x$ und endet vor der Startadresse des nächsten Threads. PARE initialisiert das Ende des letzten Threads und JOIN wartet auf das terminieren aller in diesem Block erzeugten Threads.

	PRIO *prio*	Setzt die Priorität des aktuellen Threads.
signal S **in** ... **end**	**SIGNAL** S	Initialisiert ein lokales Signal S.
emit S[(*val*)]	**EMIT** S [, {*#data*|*reg*}]	Emittiert ein (valued) Signal S.
sustain S[(*val*)]	**SUSTAIN** S [, {*#data*|*reg*}]	Emittiert in jeder Instanz ein (valued) Signal S.
present S **then** ... **end**	**PRESENT** S, *elseAddr*	PRESENT ist implementiert als bedingter Sprung: Springt zu Adresse *elseAddr*, falls S absent ist.
nothing	**NOTHING**	Führt keine Operation aus.
halt	**HALT**	Wird für immer ausgeführt.
loop ... **end loop**	**GOTO** *addr*	Unbedingter Sprung zu *addr*.
pause	**PAUSE**	Wartet auf ein Signal. AWAIT TICK ist äquivalent zu PAUSE.
await [*count*] S	**AWAIT** [*count*] S	
await immediate S	**AWAITI** S	

await
  case [**immediate**] S_1 **do** p_1
  ...
  case [**immediate**] S_n **do** p_n
end await
	CAWAIT[I] S_1, $addr_1$
	  ...
	CAWAIT[I] S_n, $addr_n$
	CAWAITE
		Wartet gleichzeitig auf mehrere Signale.

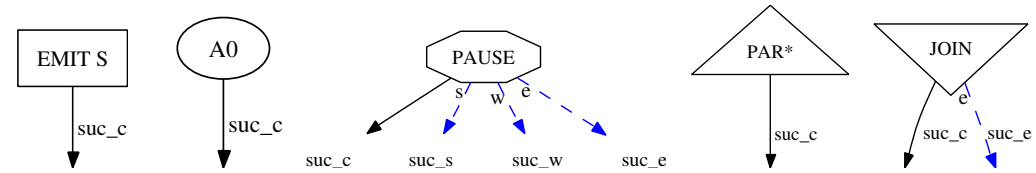
[**weak**] **abort**
  ...
when [*count*] S
	[W]**ABORT** [*count*] S *endAddr*
	  ...
	endAddr:
		Delayed und immediate Version des ABORT Statements. Der Body des ABORT Statements endet bei *endAddr*.

[**weak**] **abort**
  ...
when immediate S
	[W]**ABORTI** S *endAddr*
	  ...
	endAddr:	

suspend
  ...
when [{**immediate**|*count*}] S
	SUSPEND[{I| *count*}] S, *endAddr*
	  ...
	endAddr:
		Delayed und immediate Version des SUSPEND Statements. Der Body des SUSPEND Statements endet bei *endAddr*.

trap T **in**
  ...
  exit T
  ...
end trap
	startAddr:
	  ...
	  EXIT *exitAddr*, *startAddr*
	  ...
	exitAddr:
		EXIT beendet einen Trap Scope der bei *startAddr* beginnt und bei *exitAddr* endet. Im Gegensatz zu GOTO werden nebenläufige EXITs berücksichtigt und eingeschlossene || Operatoren beendet.

Tab. 2.1.: Einige wichtige ESTEREL Statements und ihre Entsprechungen in KEP(3a) Assembler Code



(a) transient (b) label (c) delay (d) fork (e) join

Abb. 2.2.: Nodes and edges of a Concurrent KEP Assembler Graph (CKAG).

Tabelle 2.1 gibt einen Überblick über wichtige ESTEREL Statements und ihren Entsprechungen in KEP Assembler Code. Für alle Statements, die sich nicht direkt in KEP Assembler Instruktionen abbilden lassen, gibt es wohldefinierte Übersetzungen in Kernel Statements bzw. vorhandene KEP Assembler Statements. Ein Beispiel für ein im KEP Instruktionssatz fehlendes ESTEREL Statement ist z.B. `every`. Das folgende Beispiel zeigt die Übersetzung des Statements `every` *S* `do` *p* `end`:

```
...
await S
loop
  p
each S
...
```

```
...
await S
loop
  abort
    p
    halt
  when S
end loop
...
```

```
...
abort
  loop
    pause
  end loop
when S;
loop
  abort
    p
    loop
      pause
    end loop
  when S
end loop
...
```

Im Prinzip werden alle im zweiten Übersetzungsschritt auftretenden Statements durch den KEP unterstützt. In einigen Fällen ist es für die Festlegung eines korrekten Schedulings bei Delay Statements, die über mehrere Instanzen hinweg gültig sein können, jedoch notwendig, diese noch weiter aufzulösen. Sollte ein solcher Fall nicht eintreten, so wird der entsprechende Code wieder kollabiert.

2.4.3. Esterel2KASM Compiler

Um ein ESTEREL Programm auf dem KEP ausführen zu können, muss dieses mit Hilfe des Esterel2KASM Compilers [23] zunächst in ein äquivalentes KEP Assembler Programm übersetzt werden. Der Esterel2KASM Compiler kann dabei nur eine bestimmte Teilmenge der zyklenfreien ESTEREL Programme übersetzen. Nach einer Expansion aller `run` Statements durch die entsprechenden Module als Vorverarbeitungsschritt werden zunächst alle ESTEREL Statements in die entsprechenden Assemblerinstruktionen aufgelöst.

Gleichzeitig wird der zu dem Programm gehörige ***C****oncurrent* ***K****EP* ***A****ssembler* ***G****raph* (*CKAG*) aufgebaut. Die CKAG Datenstruktur ist ein ungerichteter Graph,

die aus verschiedenen Knoten- und Kantentypen gebildet wird. Die Datenstruktur wird als Zwischenrepräsentation des ESTEREL Programms benutzt, die während des Kompiliervorgangs für die Berechnung des Schedulings (durch Zuweisung von Prozessprioritäten), für die Eliminierung von nicht erreichbarem Code, für Optimierungen des KEP Assembler Codes und für die Berechnung der Worst Case Reaction Time (WCRT) benutzt wird. *Transiente Knoten* (in der grafischen Darstellung repräsentiert durch ein Rechteck) repräsentieren instantane Statements, *Delay Knoten* (Achteck) repräsentieren Statements, die für eine oder mehrere Instanzen verzögern, *Fork-* und *Joinknoten* (Dreieck) repräsentieren Nebenläufigkeit und *Label Knoten* (Ellipse) repräsentieren Sprungmarken oder den Beginn bestimmter Blöcke. Zu jedem Knoten n gibt es eine Menge von Nachfolgeknoten $n.suc\_c$ im sequenziellen Programmfluss. Diese werden über durchgezogene Kanten in der grafischen Darstellung repräsentiert. Nachfolgeknoten beim Auftreten einer Preemption sind $n.suc\_s$ bei Strong Aborts, $n.suc\_w$ bei Weak Aborts und $n.suc\_t$ bei Exceptions, die als gestrichelte Kanten repräsentiert sind. Je nach Kontext werden bei der Erstellung von Delay Knoten zusätzlich Abort- und Exceptionkanten erzeugt. Ein weiterer Kantentyp sind *Dependency Kanten* (gepunktete Kanten), die Signalabhängigkeiten zwischen Threads darstellen. Eine Dependency Kante zeigt dabei vom Writer eines Signals (z.B. `emit` oder `sustain`) zum Reader eines Signals (z.B. `present` oder `every`). Reader-Writer Beziehungen werden benötigt, um bei der Berechnung des Schedulings die Threadprioritäten so zu vergeben, dass Writer eines Signals vor den Readern ausgeführt werden.

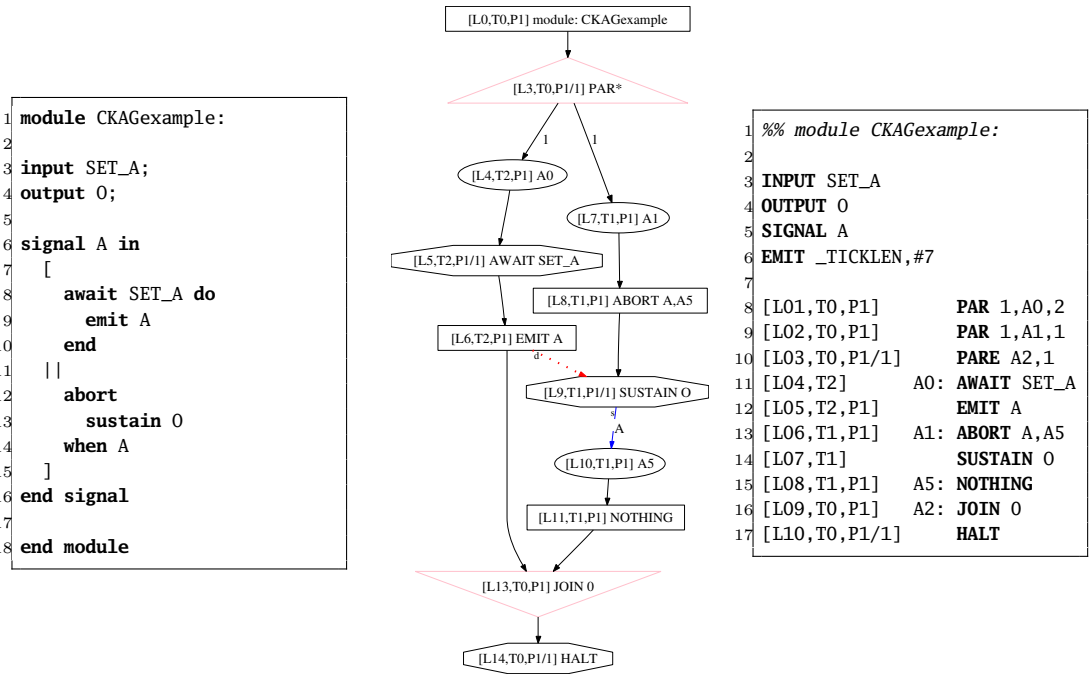


Abb. 2.3.: CKAG Beispiel

Abbildung 2.3 zeigt für ein einfaches Beispielprogramm den Aufbau des zugehörigen CKAG. Das ESTEREL Programm besteht aus zwei Threads. Der erste Thread wartet auf das Auftreten des Eingangssignals `SET_A` und emittiert dann das lokale Signal `A`. Das `sustain 0` Statement des zweiten Threads liegt innerhalb eines `abort` Rumpfes, der beendet wird, wenn `A` *present* ist.

Im CKAG wird der Kontrollfluss über einen Forkknoten in zwei Threads aufgeteilt und am Ende der Threads über einen Joinknoten wieder zusammengeführt. Der Code für den ersten Thread beginnt bei Label `A0` und für den zweiten bei Label `A1`. Der sequenzielle Programmfluss im ersten Thread geht vom Labelknoten `A0` über den Delay Knoten `await SET_A` und den transienten Knoten `emit A` in den Joinknoten. Im zweiten Thread deklariert der transiente Knoten `abort A,A5` den Beginn des abort Rumpfes an dieser Stelle, das Ende des Rumpfes bei Label `A5` und das Triggersignal `A`. Der Delay Knoten `sustain 0` liegt im Rumpf des `abort ... when A` Statements. Über eine Strong Abort Kante, die durch das Signal `A` getriggert wird, kann vom Knoten `sustain 0` der Labelknoten `A5` erreicht und damit der Rumpf des `abort` Statements beendet werden. Die Dependency Kante zwischen dem `emit A` Knoten und dem `sustain 0` Knoten im zweiten Thread zeigt an, dass der `sustain 0` Knoten ein Reader von `A` ist und dass infolgedessen `emit A` zuerst ausgeführt werden muss.

2.5. Logikminimierung

In der digitalen Schaltungstechnik unterscheidet man zwischen Schaltnetzen (*combinatorial circuit*) und Schaltwerken (*sequential circuit*). Schaltnetze verknüpfen mehrere Eingangssignale (Pegel logisch Null oder eins) zu einem oder mehreren Ausgangssignalen. Schaltnetze sind „gedächtnisfrei“; der Pegel der Ausgangssignale ergibt sich direkt aus den Eingangssignalen. Im Gegensatz zu Schaltnetzen haben Schaltwerke ein „Gedächtnis“. Dies erreicht man durch eine zeitlich verzögerte Rückkopplung der Ausgangssignale.

Mathematisch lässt sich das Ein-/Ausgabeverhalten von Schaltnetzen durch durch Boolsche Funktionen beschreiben. Eine Boolsche Funktion ist wie folgt definiert:

Definition 3 *$f : \{0,1\}^n \rightarrow \{0,1\}^m$ heißt Boolsche Funktion mit n Variablen und m Ausgängen.*

Jede Boolsche Funktion mit m Ausgängen lässt sich auch als ein m-dimensionaler Vektor $(f_1, ..., f_m)$ von Boolschen Funktionen mit n Variablen und einem Ausgang auffassen.

Als ON-Menge einer Boolschen Funktion f mit n Variablen und einem Ausgang bezeichnet man alle Elemente $\alpha \in \{0,1\}^n$ für die $f(\alpha) = 1$ gilt. Die Elemente von $ON(f)$ bezeichnet man als Minterme. Als OFF-Menge von f bezeichnet man alle Elemente $\alpha \in \{0,1\}^n$ für die $f(\alpha) = 0$ gilt.

Eine Beschreibungsmöglichkeit für Boolsche Funktionen sind Boolsche Ausdrücke.

Definition 4 *Die Menge der Boolschen Ausdrücke über der Variablenmenge $X =$*

$\{x_1, ..., x_n\}$ ist die kleinste Teilmenge der endlichen Folgen über dem Alphabet $X \cup \{0, 1, \cdot, +, (,), \overline{}\}$ für die gilt:

1. *Die Literale* 0 *und* 1 *sind Boolsche Ausdrücke*
2. *Die Variablen $x_1, ..., x_n$ sind Boolsche Ausdrücke*
3. *Sind $\omega_1, ..., \omega_k$ Boolsche Ausdrücke, dann auch die Konjunktion $(\omega_1 \cdot ... \cdot \omega_k)$ und die Disjunktion $(\omega_1 + ... + \omega_k)$*
4. *Ist ω ein Boolscher Ausdruck, dann auch die Negation bzw. das Komplement $\overline{\omega}$*

Es gibt 2^{2^n} verschiedene Boolsche Funktionen aber eine unendliche Anzahl verschiedener Ausdrücke. So geben z.B. die Ausdrücke $f(x, y) = x + y = x\overline{y} + xy + \overline{x}y = x\overline{x} + x\overline{y} + y$ die gleiche Boolsche Funktion f wieder.

Ziel der Logikminimierung ist es, eine möglichst günstige Repräsentation einer Boolschen Funktion zu finden. Kostenkriterien sind dabei der Verbrauch von Chipfläche, die Laufzeit des kritischen Pfades, der Stromverbrauch usw. In den folgenden Abschnitten werden verschiedene Verfahren für die zwei- und mehrstufige Logiksynthese vorgestellt. Eine gute Einführung in die Thematik liefern z.B. *Molitor* und *Scholl* [27].

2.5.1. Datenstrukturen und Algorithmen

Die Minimierungsstrategie ist abhängig von der eingesetzten Technologie auf der die Funktion implementiert werden soll.

Programmable Logical Arrays (PLA) erlauben zweistufige Realisierungen von Logikfunktionen. In der ersten Stufe (AND-Feld) werden Konjunktionen von den nichtnegierten und negierten Eingangssignalen gebildet. In der zweiten Stufe (OR-Feld) werden die Zwischenergebnisse durch Disjunktionen zusammengefasst. Zweistufige Realisierungen werden allgemein über *Polynome* beschrieben.

Definition 5 *Literal, Monom, Polynom Sei $X = \{x_1, ..., x_n\}$ die Variablenmenge.*

- *Ein Boolscher Ausdruck x_i ist ein positives und $\overline{x_i}$ ein negatives* Literal.
- *Ein* Monom *ist ein Produkt von Literalen. Die Länge eines Monoms Länge(m_j) ist gegeben durch die Anzahl der in diesem Produkt verwendeten Literale.*
- *Ein* Polynom *ist eine Summe von Monomen.*

Monome und Polynome lassen sich anschaulich als n-dimensionale Würfel darstellen. Der Grad n richtet sich nach der Anzahl der Eingangsvariablen. Jedem Eckknoten ist eine Variablenbelegung zugeordnet. Liegt die Variablenbelegung eines Knotens in der Menge $ON(f)$, so wird der Knoten markiert. Abbildung 2.4 zeigt die Darstellung der Funktion $f : B_3 \rightarrow B = \overline{x_1}x_2x_3 + \overline{x_3}$ als Würfel. Rechnerintern werden Polynome als Matrix dargestellt.

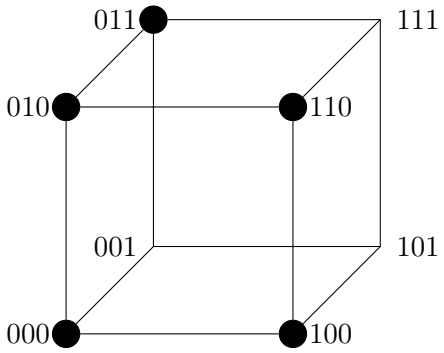


Abb. 2.4.: Würfeldarstellung der Funktion $f : B_3 \rightarrow B = \overline{x_1}x_2x_3 + \overline{x_3}$

Field Programmable Gate Arrays (FPGA) erlauben mehrstufige Realisierungen von Logikfunktionen. Sie bestehen aus einer Matrix von konfigurierbaren Logikblöcken (CLB), die über horizontale und vertikale Verdrahtungskanäle miteinander verdrahtet werden können. Je nach Hersteller sind die CLBs look-up table basiert (Xilinx) oder Multiplexer basiert (Actel).

Datenstrukturen, die zur Synthese mehrstufiger Realisierungen eingesetzt werden, sind *Logische Netzwerke* und *Binäre Entscheidungsgraphen* (*Binary Decision Diagram*, BDD).

Ein logisches Netzwerk besteht aus Eingangsknoten (den Variablen der Funktion), Ausgangsknoten (den Funktionswerten) und inneren Knoten, die eine Teilfunktion berechnen. Es gibt genau dann eine Kante zwischen vom Knoten v_1 zu v_2, wenn der Ausdruck in v_2 die Variable v_1 enthält. Logische Netzwerke werden im Rahmen von algebraischen Methoden zur mehrstufigen Logiksynthese eingesetzt.

Eine spezielle Form von BDDs, die reduzierten geordneten binären Entscheidungsgraphen, wurden 1986 von *Bryant* [11] eingeführt. BDDs finden in vielen Bereichen des logischen Entwurfs digitaler Schaltungen Anwendung, z.B. bei der formalen Verifikation oder der zwei- und der mehrstufigen Logiksynthese. Zudem lässt sich die BDD Darstellung einer boolschen Funktion fast eins-zu-eins auf Multiplexer basierten FPGAs implementieren. Bei Funktionen mit vielen Mintermen liefern BDDs oft eine wesentlich günstigere Repräsentation.

Minimierung zweistufiger Realisierungen

Bei der Zweistufigen Logikminimierung wird versucht, eine Überdeckung der in einem n-dimensionalen Würfel markierten Knoten zu finden, die aus möglichst wenigen maximalen Teilwürfeln besteht und keine nichtmarkierten Knoten enthalten darf. Die bekanntesten exakten Verfahren zur zweistufigen Logikminimierung sind Karnaugh-Veitch-Diagramme (KV-Diagramme) und das Verfahren von Quine und McCluskey. Exakte Verfahren zur Logikminimierung sind exponentiell [27, Abschnitt 3.3]. Wesentlich schnellere Berechnungen bei nicht garantierter Optimalität liefert das Espresso [10] Verfahren. Espresso II ist eines der bekanntesten und effizientesten

heuristischen Verfahren zur zweistufigen Logikminimierung. Es entstand 1979 aus einer gemeinsamen Arbeit vom *IBM Watson Research Center* und der *University of California Berkeley*.

Neben den Würfelbasierten Minimierungsverfahren gibt es auch exakte und heuristische zweistufige Minimierungsverfahren auf Basis von BDDs. Als heuristisches Verfahren ist das Verfahren von Minato [26] zu nennen. Insbesondere bei Funktionen mit vielen Mintermen stellt dieses Verfahren eine Alternative zu Espresso dar.

Minimierung mehrstufiger Realisierungen

Zweistufige Verfahren zur Logikminimierung sind sehr etabliert und gut erforscht. Zweistufige Realisierungen lassen sich zudem direkt auf PLAs implementieren. In vielen Fällen sind mehrstufige Realisierungen einer Boolschen Funktion jedoch kostengünstiger als ihre minimierte zweistufige Realisierung. Bei Boolschen Funktionen mit mehreren Ausgängen können beispielsweise gemeinsame Faktoren in der Berechnung mehrerer Ausgänge auftreten. Auch für Boolsche Funktionen mit nur einem Ausgang kann eine mehrstufige Realisierung wesentlich kostengünstiger sein als die zweistufige. Ein Beispiel hierfür ist die Paritätsfunktion $parity(x_1, ..., x_n) = (\sum_{i=1}^{n} x_i) \mod 2$. Das Minimalpolynom dieser Funktion besteht aus 2^{n-1} Primimplikanten. Bei einer Mehrstufigen Implementierung kommt man mit $n-1$ xor-Gattern mit je zwei Eingängen aus.

Das Hauptproblem bei der mehrstufigen Synthese ist es, eine geeignete Zerlegung der boolschen Funktion zu finden. Weit verbreitet sind algebraische Verfahren, wie sie von *Brayton*, *Hachtel* und *Sangiovanni Vincentelli* [10] eingeführt wurden. Das in dieser Arbeit benutzte Programm MV-SIS benutzt algebraische Verfahren zur funktionalen Zerlegung. Eine Funktion wird als logisches Netzwerk repräsentiert. Operationen, zur Optimierung eines logischen Netzwerkes sind:

- **Lokale Optimierung** Hierbei wird auf den einzelnen Knoten des Netzwerkes eine zweistufige Logikminimierung (z.B. Espresso) angewandt. In MV-SIS wird dies durch die Kommandos `simplify` oder `fullsimp` erreicht.
- **Eliminierung von Knoten** Hat ein innerer Knoten den Ausgangsgrad 1, d.h. ist nur ein anderer Knoten von ihm abhängig, kann es u.U. kostengünstiger sein, ihn zu eliminieren und im abhängigen Knoten zu substituieren.
- **Zerlegung eines Knotens** Statt der Eliminierung von Knoten kann es auch sinnvoll sein, einen Knoten in zwei Teilknoten zu zerlegen.
- **Extraktion gemeinsamer Teilausdrücke** Enthalten mehrere Knoten gemeinsame Faktoren, ergeben sich durch die Zerlegung große Kosteneinsparungspotentiale. In MV-SIS findet z.B. die Funktion `fxu` (*fast extract unate*) gute Gemeinsame Faktoren in einem logischen Netz. Die Funktion `decomp` führt eine vollständige Faktorisierung des Netzwerkes durch.

In der letzten Zeit gewinnen auf Binary Decision Diagrams basierte Verfahren zur funktionalen Zerlegung an Bedeutung.

2.5.2. Softwarepakete zur Logikminimierung (und -synthese)

MV-SIS

MV-SIS wurde von der MVSIS Arbeitsgruppe unter der Leitung von Robert K. Brayton an der University of California in Berkeley entwickelt und ist eine Weiterentwicklung des SIS Paketes (ebenfalls UC Berkeley). MV-SIS ist ein interaktives System zur mehrstufigen, mehrwertigen Logiksynthese. MV-SIS war ursprünglich als Softwarepaket zur Minimierung mehrwertiger Logik gedacht, hat sich jedoch darüber hinaus zu einem vollwertigen Werkzeug zur Synthese und Verifikation entwickelt und enthält die meisten der in der Arbeitsgruppe entwickelten Algorithmen.

Das Ein- und Ausabeformat für binäre bzw. mehrwertige logische Netze ist BLIF bzw. MV-BLIF (s. Abschnitt 2.6.1). Zur zwei- bzw.mehrstufigen Minimierung kombinatorischer Netzwerke stellt MV-SIS-2.0 verschiedene Würfel- BDD- und Netzlisten basierte Verfahren zur Verfügung (s. Abschnitt 2.5.1): z.B. Espresso oder das Verfahren von Minato zur zweistufigen oder algebraische Methoden zur mehrstufigen Logikminimierung. Verfahren zur Minimierung von sequentiellen Schaltkreisen oder zur Abbildung auf eine Zielplattform werden ab Version 3.0 unterstützt.

Die Steuerung von MV-SIS erfolgt über eine Eingabeaufforderung, über Kommandozeilenparameter oder per Skript.

2.6. Hardwarebeschreibungssprachen

Mit Hilfe von Hardwarebeschreibungssprachen lassen sich komplexe integrierte Schaltkreise auf eine formale Art und Weise beschreiben. Ihre Syntax und Semantik beinhalten Konstrukte zur Beschreibung von Nebenläufigkeit und zur Festlegung des zeitlichen Verhaltens.

VHDL und Verilog HDL sind verbreitete Hochsprachen zur Hardwarebeschreibung, die ein hohes Abstraktionsniveau bieten. BLIF ist eine sehr hardwarenahe Sprache zur textuellen Beschreibung von Netzlisten. Sie wurde ursprünglich zur Beschreibung von PLAs entwickelt und dient den Logikminimierungs- und Logiksyntheseprogrammen *espresso*, *SIS* und *MV-SIS* als Ein- und Ausgabeformat. BLIF und VHDL werden im Folgenden kurz eingeführt.

2.6.1. BLIF

Das *Berkeley Logic Interchange Format* (BLIF) [34] wurde an der University of California, Berkeley entwickelt. Es dient zur textuellen Beschreibung von hierarchisch aufgebauten Schaltnetzen oder Schaltwerken auf Logikebene. Ein Schaltkreis wird dabei als gerichteter Graph von kombinatorischen und sequenziellen Logikknoten betrachtet.

Ein *Modell* ist das Hauptstrukturelement in BLIF. Ein Modell ist die Beschreibung eines hierarchischen Schaltkreises. Es besteht aus einer Deklaration des Interfaces zur Umwelt und der funktionalen Beschreibung.

```
.model <Modell-Name>
.inputs <Liste der Eingangssignale>
.outputs <Liste der Ausgangssignale>
.clock <Liste der verwendeten Clocks>
...
<commands>
...
.end
```

Mit den `.input`-, `.output`- und `.clock`-Anweisungen wird das Interface des Modells deklariert. Nach jeder dieser Anweisungen steht eine Liste von Signalen. Die Elemente der Listen werden durch Leerzeichen getrennt. Ein Backslash '\' konkateniert zwei aufeinanderfolgende Zeilen. Bei der Deklaration mehrerer Eingangs-, Ausgangs- oder Clock-Listen werden diese miteinander konkateniert.

Die Verhaltensbeschreibung erfolgt auf einem sehr hardwarenahen Niveau. BLIF stellt Konstrukte zur Implementierung von Logikgattern, Auffangregistern (Latches) und endlichen Automaten zur Verfügung. Endliche Automaten werden im *KISS-Format* beschrieben. Zusätzlich lassen sich Modelle aus der selben oder aus einer anderen Datei innerhalb eines Modells referenzieren, so dass eine hierarchische Entwicklung der Schaltungsbeschreibung möglich ist. Über *Clock-Constraints* wird das Verhalten simulierter Clocks gesetzt und das Verhalten von Clock-Ereignissen relativ zueinander beschrieben. Über *Delay-Constraints* lassen sich Signallaufzeiten festlegen.

Die Deklaration von Logikgattern erfolgt über die `.names`-Anweisung. Ein Logikgatter assoziiert eine Logikfunktion mit einem Signal im Modell. Hinter der `.names`-Anweisung folgt in der selben Zeile eine Liste der n Eingangssignale und abschließend das Ausgangssignal. Die Funktion wird durch eine Wahrheitstabelle repräsentiert. Jede Zeile ist eine mögliche Ein-/Ausgabebelegung. Die n Bit weite Eingabebelegung wird durch ein Leerzeichen von der ein Bit weiten Ausgabe getrennt. Alle Zeilen mit einer `1` in der Ausgabe gehören zum *ON-Set* der Funktion, die mit einer `0` zu *OFF-Set* der Funktion. Eine '`1`' bedeutet, dass die Eingabe nicht komplementiert wird, eine '`0` bedeutet, dass die Eingabe komplementiert wird und ein'-' bedeutet, dass die Eingabe nicht benutzt wird. Die Elemente einer Eingabebelegung werden `und`-Verknüpft und die Eingabebelegungen werden `oder`-Verknüpft. Das folgende Beispiel zeigt die Implementierung der Funktion `O = A or B or not C`:

```
.model EXAMPLE
.inputs A B C
.outputs O
.name A B C O
1-- 1
-1- 1
--0 1
.end
```

2.6.2. VHDL

VHDL steht für ***V****ery High Speed Integrated Circuit* ***H****ardware* ***D****escription* ***L****anguage*. Ihre Entwicklung begann Anfang der 80er Jahre initiiert durch das US-Verteidigungs-

ministerium. In den Jahren 1987 und 1993 wurde VHDL durch den Standard IEEE-1076 als herstellerunabhängige Hochsprache definiert und wird seitdem regelmäßig überarbeitet.

VHDL ermöglicht die Beschreibung und Simulation von digitalen Schaltungen auf einer sehr hohen Abstraktionsebene. Entwurfsziel können dabei verschiedene rekonfigurierbare Logikbausteine wie beispielsweise ein FPGA (Field Programmable Gate Array) oder CPLD (Complex Programmable Logic Device), ein ASIC (Application Specific Integrated Circuit) oder ganze Platinen sein. Durch ihr hierarchisches Konzept ist es innerhalb kurzer Zeit möglich, auch komplexe Schaltkreise wie Prozessoren mit einigen Millionen Gattern zu entwickeln.

Nicht der volle Sprachstandard von VHDL ist synthetisierbar, sondern nur eine bestimmte Teilmenge. Man unterscheidet zwischen *synthetisierbarem* und *funktionalem* Code. Letzterer ist ausschließlich simulierbar und dient der Spezifikation von Verhaltensmodellen oder für Simulations- und Debuggingzwecke. Der praktisch häufigste Fall von nicht-synthetisierbarem Code ist die Erzeugung von Testumgebungen (sog. *Testbenches*). Welcher Code synthetisierbar ist und welcher nicht, hängt aber auch von den Fähigkeiten des verwendeten Synthesetools und der Zielplattform ab. Die Hardwaresynthese aus VHDL entspricht im Wesentlichen einer Mustererkennung. So ist es teilweise notwendig, bestimmte Implementationsmuster *Templates* zu benutzen, um die Logik auf bestimmte Hardwareressourcen der Zielplattform abzubilden. Ein Beispiel hierfür ist der *Ripple Carry Adder*, der speziell für die effiziente Implementierung auf FPGAs zugeschnitten ist.

Syntax

Im Folgenden wird eine kurze Einführung in VHDL gegeben. Diese beschränkt sich auf die Sprachteile, die zur Implementierung von Schaltnetzen benutzt werden. Eine gute Einführung geben z.B. *Reichhardt* und *Schwarz* [30].

Einbinden von Paketen Mit der `library` Anweisung können in einem VHDL Programm zusätzliche Bibliotheken eingebunden werden. Bibliotheken sind eine meist vorkompilierte Sammlung von VHDL Einheiten. Sie können eigene Deklarationen, aber auch Teilentwürfe oder ganze Komponenten (*Intellectual Property (IP)*) enthalten. Bibliotheken zur Erweiterung des Sprachstandards werden von den Syntheseswerkzeugherstellern oder von Arbeitsgruppen der IEEE zur Verfügung gestellt. Die Bibliothek `ieee` umfasst u.a. folgende Packages:

- **`std_logic_1164`** Definiert die Datentypen `std_logic` bzw. `std_ulogic` und `std_logic_vector` bzw. `std_ulogic_vector` für mehrwertige Logik, sowie einige Funktionen auf diesen Datentypen.
- **`std_logic_signed und std_logic_unsigned`**: Definiert vorzeichenlose und vorzeichenbehaftete Operationen für den Datentyp `std_(u)logic_vector`. Es ist keine Mischung von vorzeichenloser mit vorzeichenbehafteter Arithmetik erlaubt.

- **`numeric_std` bzw. `std_logic_arith`**: Definiert die Datentypen `signed` und `unsigned`, arithmetische Operationen und Vergleichsoperationen auf diesen Datentypen sowie Konvertierungsfunktionen.

Mit der `library ieee;` Anweisung wird die IEEE-Bibliothek deklariert. Ein bestimmtes Paket der Bibliothek muss vor jeder Entity, in der es benutzt wird, mit Hilfe der `use` Anweisung eingebunden werden (z.B. `use ieee.std_logic_1164.all;`).

Datenobjekte und Datentypen Als Objektklassen für Daten stellt VHDL *Konstanten*, *Variablen* und *Signale* zur Verfügung:

- *Konstanten* wird bei der Initialisierung einmal ein fester Wert zugewiesen.

- *Variablen* werden bei der Verhaltensbeschreibung mit sequentiellen Anweisungen innerhalb von Prozessen, die der algorithmischen Struktur von Programmiersprachen entsprechen, eingesetzt.

- *Signale* dienen zur Kommunikation zwischen Prozessen oder als Datenleitungen zwischen Funktionsblöcken. Ähnlich wie bei ESTEREL wird bei der Simulation von VHDL die zeitliche Ordnung von Interaktionen über Signale geregelt.

VHDL ist stark typisiert. Signale und Variablen erhalten bei ihrer Deklaration einen festen Typ. Bei der Wertzuweisung von Signalen oder Variablen unterschiedlichen Typs muss eine explizite Typkonvertierungsfunktion angegeben werden. Die wichtigsten Datentypen, die z.T. nicht zum Standardsprachumfang gehören sondern über zusätzliche Pakete eingebunden werden müssen, sind:

- `bit`/`bit_vector`: Der Datentyp `bit` besteht aus den logischen Werten '0' oder '1'. Nach der Deklaration hat ein Signal standardmäßig den Wert '0'. Der Typ `bit_vector` ist ein vordefinierter Array-Typ zu `bit`.

- `std_[u]logic` und `std_[u]logic_vector`: Der IEEE-Standard 1164-1993 führt die Datentypen `std\_logic` und `std_ulogic` mit einer neunwertigen Logik ein, die folgende Werte umfasst: 'U' (nicht initialisiert), 'X' (undefiniert, mehrere aktive Signaltreiber), '0' (starke logische '0' (vgl. bit)), '1' (starke logische '1'(vgl bit)), 'Z' (hochohmig (Tri-State Ausgang)), 'W' (schwach unbekannt), 'L' (schwache logische '0'), 'H' (schwache logische '1') und '-' (don't care (kann für Logikminimierung verwendet werden)). Im Gegensatz zum Typ `std_ulogic` sind für Signale vom Typ `std_logic` mehrere Treiber erlaubt, d.h. es dürfen mehrere nebenläufige Signalzuweisungen erfolgen. In diesem Fall wird bei der Simulation der resultierende Wert über eine Auflösungsfunktion bestimmt. Der Datentyp `std_logic` findet insbesondere bei bidirektionalen Bussen Verwendung. Der Typ `std_[u]logic_vector` ist ein vordefinierter Array-Typ zu `std_[u]logic`.

- `boolean`: Der Datentyp `boolean` kann die Werte *true* oder *false* annehmen. Im Gegensatz zu `bit` dient er in der Praxis hauptsächlich dazu, architekturspezifische, logische Ausdrücke zu erzeugen, weniger dazu, die elektrischen Pegel von Signalen zu erzeugen.
- `integer`: Der Datentyp `integer` wird in den meisten CAE-Systemen durch 32 Bit repräsentiert. Er sollte nur für den indizierten Zugriff auf Vektor- oder Feldkomponenten und innerhalb von Verhaltensmodellen oder Testumgebungen eingesetzt werden.
- `signed` und `unsigned`: Der ergänzende IEEE 1076.3 Standard spezifiziert die Datentypen `signed` und `unsigned`. Sie sind eine arithmetische Interpretation des Typs `std_logic_vector`. Negative Zahlen werden im Zweierkomplement dargestellt.

Zusätzlich lassen sich komplexe Datentypen, wie *Arrays* oder *Records*, oder aber Subtypen von bestehenden Typen definieren.

Signalzuweisungen erfolgen über den Signalzuweisungsoperator `<=`:

```
   A <= '0';     -- Zuweisung an ein Signal vom Typ bit bzw. std_logic
   B <= true;    -- Zuweisung an ein Signal vom Typ boolean
   C <= "1010";  -- Zuweisung an einen bit_vector oder std_logic_vector mit 4 Bit Breite
D(0) <= '0';     -- Zuweisung an Bit 0 in einem Vektor
```

Eine Wertzuweisung an Variablen erfolgt mit dem Zuweisungsoperator `:=`.

Ausdrücke Für die Datentypen `boolean`, `bit` und `std_logic` stehen die logischen Operatoren `not`, `and`, `nand`, `or`, `nor`, `xor` und `xnor` zur Verfügung. Der Operator `not` hat die höchste Priorität, die Priorität der übrigen Operatoren muss durch Klammerung festgelegt werden.

Für die oben vorgestellten ganzzahligen Typen definieren die entsprechenden Bibliotheken Vergleichsoperatoren `=`,`/=` (ungleich),`<`,`<=`,`>` und `>=` (alle synthetisierbar) und arithmetische Operatoren `+`,`-`,`abs`,`*`,`/` (Division meist nicht synthesefähig),`**` (Zweierpotenz), `mod` und `rem` (Divisionsrest nur in Spezialfällen synthetisierbar).

Strukturelemente Neben dem schon erwähnten `package` sind `entity` und `architecture` grundlegende Strukturelemente einer VHDL-Beschreibung.

Eine `entity` beschreibt die Schnittstelle eines VHDL-Funktionsblocks nach außen. Die Deklaration der Anschlüsse erfolgt mit der `port` Anweisung:

```
entity MUX2x1 is
  port(
    S: in std_logic;                      -- Zur Selektion des Eingangs
    I: in std_logic_vector(1 downto 0);   -- 2 Bit Eingangsvektor
    O: out std_logic;                     -- Ausgangssignal
end MUX2x1;
```

Innerhalb einer `architecture` wird die Funktionalität eines Blocks beschrieben. Jeder `entity` muss mindestens eine solche Implementierung zugeordnet sein. Eine `architecture` besteht aus einem *Deklarationsteil*, sowie *konkurrenten Anweisungen* zur Verhaltensbeschreibung:

```
architecture MUX of MUX2x1 is
  -- Deklarationsteil
begin
  -- konkurrente Anweisungen
end MUX;
```

Im Deklarationsteil erfolgen u.a. *Typdeklarationen*, *Konstantendeklarationen*, *lokale Signaldeklarationen* oder *Komponentendeklarationen*. Konkurrente Anweisungen sind z.B. Signalzuweisungen, Prozesse oder Instanzen von Komponenten. Signalzuweisungen können unbedingt (`A <= "1010";`) oder bedingt erfolgen:

```
architecture MUX of MUX2x1 is
begin
  with S select
  O <= I(0) when '0',
       I(1) when '1';
end MUX;
```

Innerhalb von Prozessen erfolgt die Funktionsbeschreibung über sequenzielle Anweisungen, ähnlich einer gewöhnlichen Programmiersprache:

```
architecture MUX of MUX2x1 is
begin
MUXPROC:  process (S)
          begin
            case S is
              when '0' => O <= I(0);
              when '1' => O <= I(1);
            end case;
          end process MUXPROC;
end MUX;
```

VHDL erlaubt den hierarchischen Aufbau der Sytembeschreibung. Dazu lassen sich `entity`/`architecture`-Paare in einem übergeordneten Funktionsblock instantiieren. Eine zu nutzende Komponente (`component`) wird im Deklarationsteil der `architecture` zunächst mit ihrer Schnittstelle (`port`) deklariert und anschließend im Architekturrumpf instantiiert. Existieren mehrere Implementierungen für eine `entity`, so kann mit Hilfe der `for ... use entity ...`-Anweisung eine bestimmte Implementierung ausgesucht werden.

```
architecture MUX4x1_impl of MUX4x1 is
component MUX2x1
  port(                                                 --        _ MUX_0
    S: in std_logic;                                    --  I(0)-| \
    I: in std_logic_vector(1 downto 0);                 --       |  |--+
    O: out std_logic;                                   --  I(1)-|_/   |   _ MUX_2
end component;                                          --        |    +--| \
signal O_INT: std_logic_vector(1 downto 0);             --  S(0)--+       |  |-- O
begin                                                   --        |    +--|_/
MUX_0: MUX2x1 port map(S(0), I(1 downto 0), O_INT(0));  --  I(2)-| \   |   |
MUX_1: MUX2x1 port map(S(0), I(3 downto 2), O_INT(1));  --       |  |--+  S(1)
MUX_2: MUX2x1 port map(S(1), O_INT, O);                 --  I(3)-|_/
end MUX4x1_impl;                                        --          MUX_1
```

3. HW/SW Co-Synthese

In diesem Kapitel wird ein Verfahren zur HW/SW Co-Synthese von ESTEREL Programmen vorgestellt, das komplexe Expressions aus dem gegebenen Programm extrahiert und diese in einem mit dem KEP verbundenen Logikblock berechnet. Zusätzlich wird eine modifizierte Programmversion erzeugt, die dann wie gewöhnlich auf dem erweiterten KEP ausgeführt werden kann. In Kapitel 4 wird für verschiedene Programme untersucht, wie sich das vorgestellte Verfahren auf die Ticklänge, den Ressourcenverbrauch und den Energieverbrauch auswirkt.

3.1. Problemstellung

Die wichtigsten ESTEREL Statements werden durch den Befehlssatz des KEP direkt unterstützt oder lassen sich leicht auf diesen zurückführen. Die Berechnung von komplexen Expressions ist vergleichsweise aufwändig, da diese nicht innerhalb einer Instruktion ausgewertet werden können, sondern in mehrere KEP Assembler Instruktionen zerlegt werden müssen. Die Berechnung von komplexen Expressions in KEP Assembler ist exemplarisch in den oberen beiden Listings in Abbildung 3.1 dargestellt.

Das Listing *Original Esterel Programm* enthält den ESTEREL Programmcode und das Listing *KEP Assembler Code* den resultierenden KEP Assembler Code. Das `present (A or B) and C then ... end present` Statement in Zeile 10 im Programmcode wird zerlegt in die KEP Assembler Instruktionen in Zeilen 13 bis 16 im KEP Assembler Code. `PRESENT S, Addr` ist in KEP Assembler ein bedingter Sprung, der als „*Springe zu* `Addr`*, wenn* `S` *absent ist*“ zu interpretieren ist. `PRESENT C,A0` testet das Signal `C`. Ist es *absent*, ist die ganze Signal Expression *absent* und das Programm wird bei `A1` fortgeführt. Ist es jedoch *present*, ist noch `(`A or B) auszuwerten. Ist nun beispielsweise `A` *present*, wird die folgende `GOTO A1` Instruktion ausgeführt und `O1` emittiert. In den Zeilen 18 bis 27 im KEP Assembler Code wird der if-Test `if (?D or (?E and ?F)) then ...` berechnet. In Zeile 18 wird zunächst der Wert von Signal `D` ins Register `REG0` geladen. `CMPS REG0,#1` vergleicht den Inhalt des Registers mit der Konstanten `1` (true). `JW EE,A5` ist ein bedingter Sprung. Ergab der letzte Vergleich *equal* (`EE`), wird `GOTO A4` ausgeführt und schließlich `O2` mit dem Wert `1` (*true*) emittiert, ansonsten wird zu `A5` gesprungen. Die Interpretation der übrigen Instruktionen bleibt dem Leser überlassen.

Wie das Beispiel zeigt, werden für die Berechnung der Signal Expression vier und für die Berechnung der valued Expressions sogar zehn Instruktionen benötigt, da zum Laden eines Signalwerts in ein Register und für den Vergleich des Registerin-

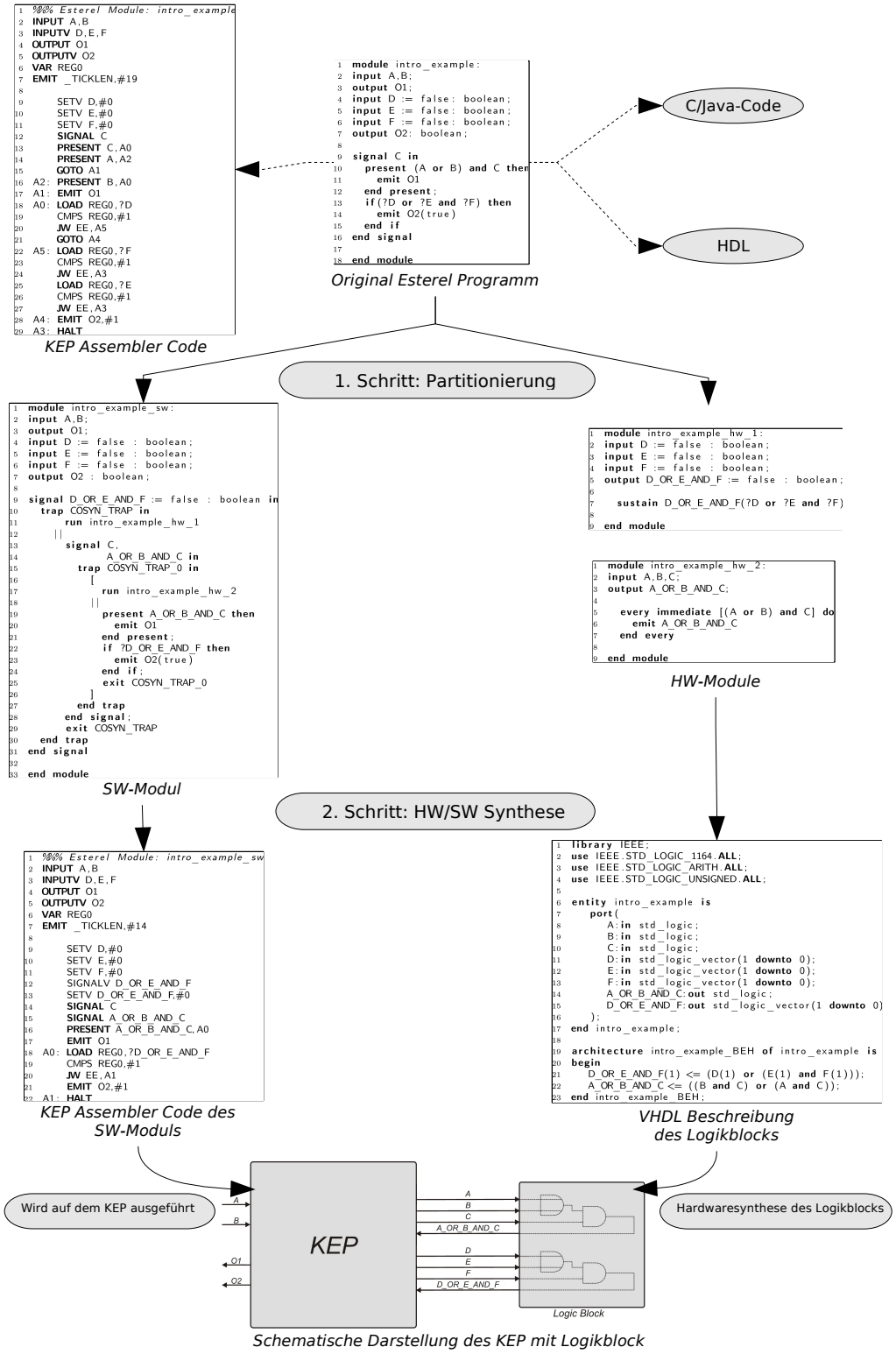


Abb. 3.1.: Überblick über den Co-Synthese Prozess: Die gestrichelten Pfade zeigen die bisherigen ESTEREL Syntheseansätze. Dies sind zum einen die traditionelle Software- und Hardwaresynthese, als auch die reine Softwaresynthese für den KEP (KEP Assembler). Die durchgezogenen Pfade veranschaulichen den in dieser Arbeit vorgestellten Co-Synthesefluss.

halts mit einer Konstanten zwei zusätzliche Instruktionen benötigt werden, bevor aufgrund des Vergleichsergebnisses ein bedingter Sprung ausgeführt wird. Die Anzahl der Instruktionen im KEP Assembler Code wird also erheblich von der Anzahl und der Komplexität der in einem Programm auftretenden Expressions beeinflusst, was sich üblicherweise auch negativ auf die WCRT des Programmes und damit auf die minimale Ticklänge auswirkt.

Im Folgenden wird ein Lösungsansatz für die effizientere Berechnung von komplexen Expressions vorgestellt.

3.2. Lösungsansatz

Die Grundidee besteht darin, die Berechnung von komplexen Expressions in einen mit dem KEP verbundenen Logikblock auszulagern, der für jedes ESTEREL Programm individuell generiert wird. Über ein spezielles Interface erhält der Logikblock Zugriff auf alle für die Berechnung der Expressions benötigten Status- und Signalwerte und gibt die Ergebnisse der Berechnung als zusätzliches Eingangssignal an den KEP zurück. Auf dem KEP wird eine modifizierte Programmversion ausgeführt, in der alle komplexen Expressions durch das entsprechende Hilfssignal ersetzt sind.

Diese Hardware/Software Co-Synthese eines ESTEREL Programms erfolgt in zwei Schritten mit einem optionalen Zwischenschritt:

- Schritt 1 (Partitionierung): Das gegebene ESTEREL Programm wird zunächst in ein semantisch äquivalentes ESTEREL Programm transformiert, das die Software- und Hardwareteile in unterschiedlichen Modulen modelliert. Die Partitionierung auf ESTEREL Basis soll den Syntheseprozess möglichst lange hardwareunabhängig halten und eine vollständige Verifizierbarkeit dieses Schrittes mit vorhandenen Werkzeugen, wie z.B. *EsterelStudio* [17], ermöglichen.

- Optionaler Zwischenschritt (Logikminimierung): Durch die Logikminimierung sollen einzelne Ausdrücke im Logikblock vereinfacht und gemeinsame Teilausdrücke identifiziert werden, um den resultierenden Logikblock möglichst klein zu halten.

- Schritt 2 (HW/SW Synthese): Der Softwareteil des partitionierten Programmes wird wie gewohnt mit dem Esterel2KASM Compiler für die Ausführung auf dem KEP kompiliert, der Hardwareteil wird in die VHDL Beschreibung des Logikblocks konvertiert, in Hardware synthetisiert und mit dem KEP verbunden.

Für die Implementierung des oben skizzierten Ansatzes sind diverse, nicht offensichtliche Schwierigkeiten zu lösen auf die im Folgenden kurz eingegangen werden soll. Eine erste Schwierigkeit besteht darin, ein Verfahren für die korrekte Partitionierung des ESTEREL Programms in Software- und Hardwaremodule zu finden. Um eine komplexe Expression auswerten zu können, müssen alle Signale, die zur Berechnung notwendig sind, im Hardwaremodul bekannt sein. Als Kommunikationsmechanismus zwischen den Modulen kommen nur Signale in Frage, was die Modellierung

einer korrekten Kommunikation erschwert, wenn die zu berechnende Expression lokale Signale enthält. Dies hängt damit zusammen, dass lokale Signale schizophren sein können, d.h. dass sie während einer Instanz verschiedene Werte haben können. Dieses als Reinkarnation bekannte Problem, sowie ein weiteres Problem mit lokalen Signaldeklarationen innerhalb von `suspend`-Blöcken werden im folgenden Abschnitt detailliert erörtert. Darüber hinaus werden zwei verschiedene Partitionierungsansätze vorgestellt und im Bezug auf die Lösung der o.g. Probleme gegeneinander abgewogen.

Eine weitere Schwierigkeit ergibt sich bei der Softwaresynthese, da der Esterel2KASM Compiler das Scheduling für das modifizierte Programm nicht korrekt berechnen kann. Pro Instanz müssen in konkurrierenden Threads die schreibenden Statements eines Signals vor den lesenden ausgeführt werden. Der Compiler ist nicht in der Lage festzustellen, von welchen Signalen die Berechnung eines Hilfssignals abhängt, da die Berechnung nicht mehr durch das Programm selbst, sondern durch den für den Compiler unbekannten Logikblock durchgeführt wird. Abschnitt 3.5.1 zeigt, wie dieses Problem durch eine Modifikation des Compilers gelöst werden kann.

3.3. Partitionierung

Wie im vorigen Abschnitt bereits angesprochen wurde, muss eine Methode gefunden werden, mit der auch aus Programmen mit komplexen Expressions, die lokale Signale enthalten, ein äquivalentes partitioniertes Programm erzeugt werden kann. Anhand einer einfachen Partitionierugsmethode werden zunächst die auftretenden Probleme erörtert. Da diese einfache Methode auch durch verschiedene Workarounds keine zufriedenstellenden Ergebnisse liefert, wird anschließend ein enger an der Semantik von Signalen in Esterel ausgerichtetes Verfahren vorgestellt. Abschließend wird ausführlich auf die Implementierung der Transformation von Signal- und valued Expressions eingegangen.

3.3.1. Schwierigkeiten bei der Partitionierung

Ein erster naheliegender Partitionierungsansatz wäre die Aufteilung des Esterel Programms in drei Module, ein *Softwaremodul*, ein *Hardwaremodul* und ein *Mainmodul*. Das Softwaremodul entspricht im Wesentlichen dem Originalprogramm, es wurden jedoch auftretende komplexe Expressions durch Hilfssignale ersetzt, die im Hardwaremodul berechnet werden. Das Mainmodul startet das Software- und das Hardwaremodul parallel. Ein Beispiel für diese Partitionierung ist in Abbildung 3.2 dargestellt.

Das Softwaremodul enthält alle Statements des Originalprogramms. Die komplexen Expressions `A and C or B and C` und `A or B` wurden hier durch die beiden Hilfssignale `A_and_C_or_B_and_C` und `A_or_B` ersetzt, die im Interface als neue `input`-Signale deklariert werden. Die Berechnung der Hilfssignale erfolgt im Hardwaremodul in parallelen `every immediate` *SigExp* `do emit AuxSig end every` Statements. Diese emittieren die Hilfssignale ***`AuxSig`*** in jeder Instanz, in der die Signal Expression ***`SigExp`*** *present* ist. Alle für die Berechnung notwendigen Signale werden

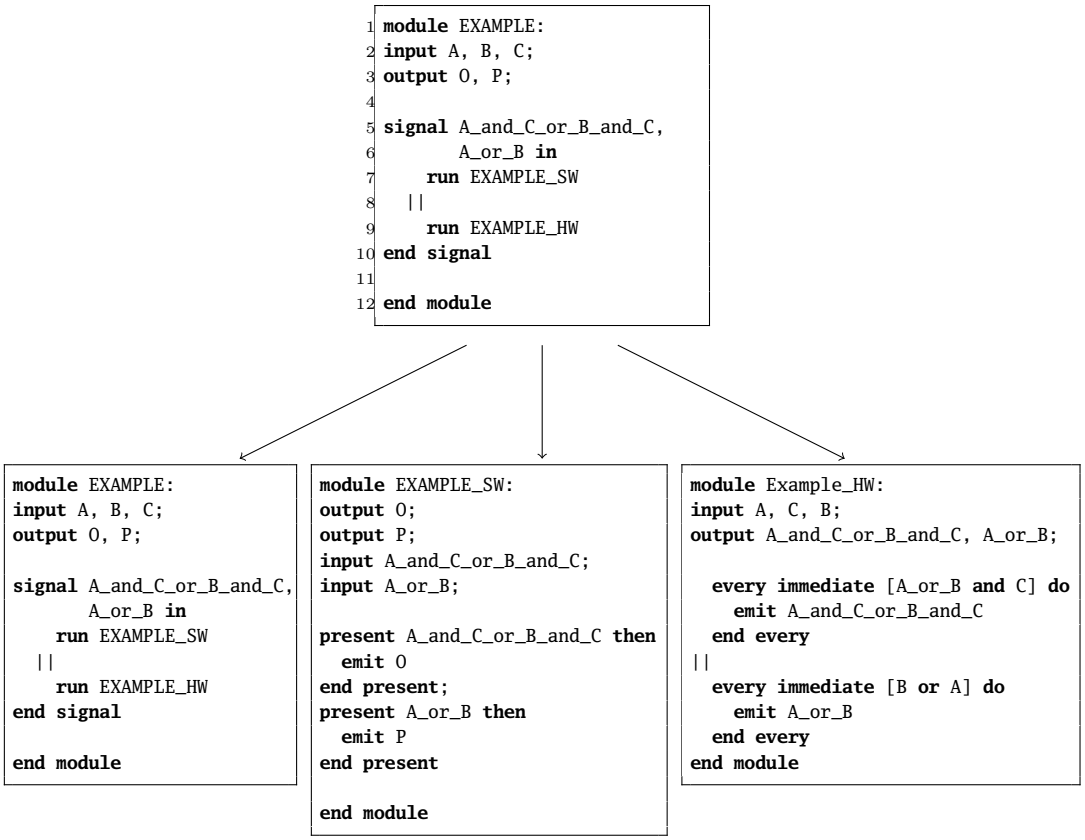


Abb. 3.2.: Aufteilung eines (Esterel) Programms in Main-, SW- und HW-Modul

im Interface des Hardwaremoduls als Eingang und die Hilfssignale als Ausgang deklariert. Das Mainmodul führt das Software- und das Hardwaremodul parallel aus und verfügt, da es als neue Schnittstelle des partitionierten Programms zur Umwelt fungiert, über das gleiche Interface wie das Originalprogramm. Hilfssignale, die zwischen dem Soft- und dem Hardwaremodul geteilt werden, werden im Mainmodul lokal deklariert.

Dieser Ansatz erlaubt es im Allgemeinen nicht, Expressions mit lokalen Signalen zu behandeln, da ihr Status nicht korrekt an das Hardwaremodul übermittelt werden kann. Das lokale Signal einfach global im Interface zu deklarieren, führt in vielen Fällen nicht zum gewünschten Ergebnis, da dies die Semantik des Programms verändern kann. Gründe hierfür sind das mögliche Auftreten von Schizophrenie durch Reinkarnation des lokalen Signals oder die Suspendierung des Blocks, in dem das lokale Signal deklariert ist.

Nach der Semantik von Esterel haben alle Signale in einem Programm pro Instanz genau einen Status. Wird jedoch ein lokales Signal innerhalb eines Schleifenrumpfes deklariert, so wird es in jedem Schleifenzyklus neu instantiiert. Auf diese Art und Weise können zwei simultane Inkarnationen eines Signals pro Instanz existieren.

Das folgende Beispiel illustriert dieses Verhalten:

```
...
loop
  signal S in
    present S then ...
    pause;
    emit S;
  end signal
end loop
...
```

Das Signal S wird innerhalb einer Schleife lokal deklariert. In der ersten Instanz ist es *absent* und der Signaltest fällt negativ aus. In der folgenden Instanz wird das Signal emittiert und der Rumpf der Schleife instantan neu gestartet. Die neue Inkarnation von S hat wieder den Status absent. Abbildung 3.3 zeigt die offensichtlich falsche Partitionierung des Programms *LocalSig*, in der das Hilfssignal ab der zweiten Instanz immer den falschen Status *present* hat.

Ein weiteres Problem mit lokalen Signalen tritt bei der Deklaration innerhalb eines

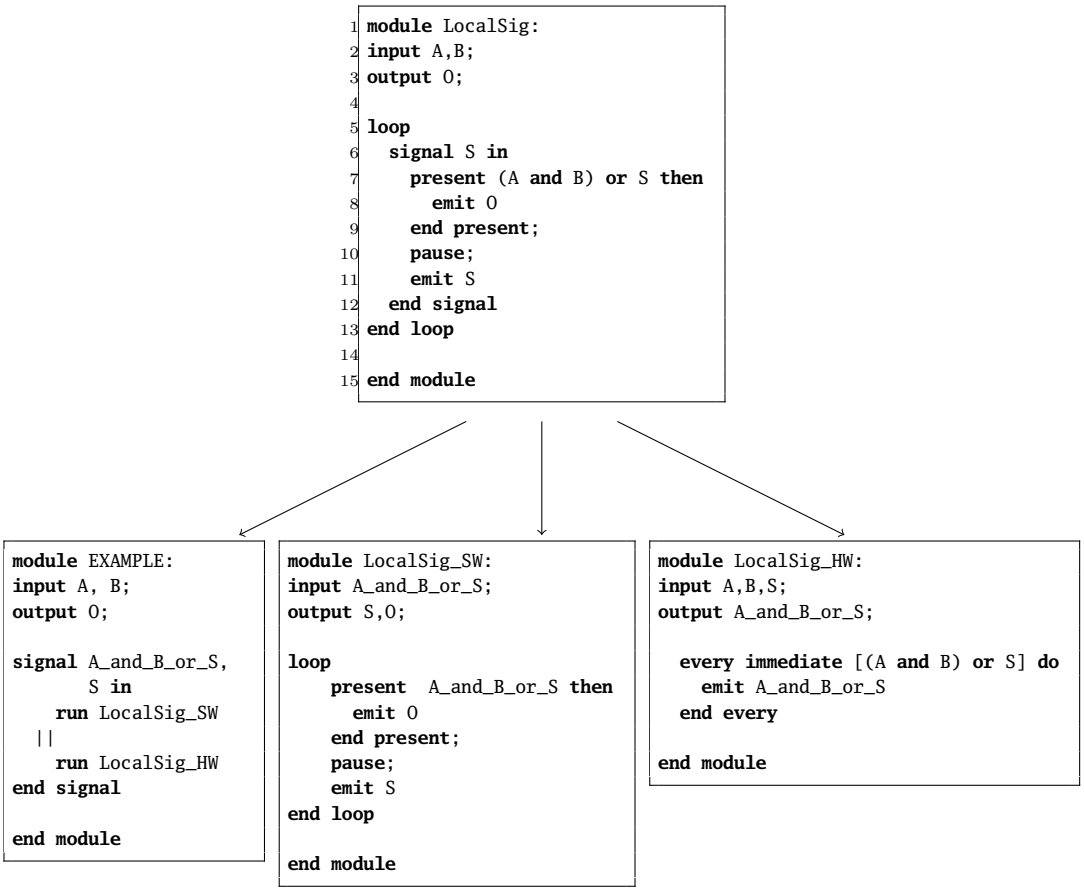


Abb. 3.3.: Fehlerhafte Partitionierung in HW-, SW- und Mainmodule

`suspend` Statements auf. Im folgenden Beispiel liefert etwa der Ausdruck `pre(S)` den Status von S in der vorigen Instanz, in der das `signal` Statement aktiv war.

```
...
suspend
  signal S in
    ...
    present pre(S) then ...
    ...
  end signal
when I
...
```

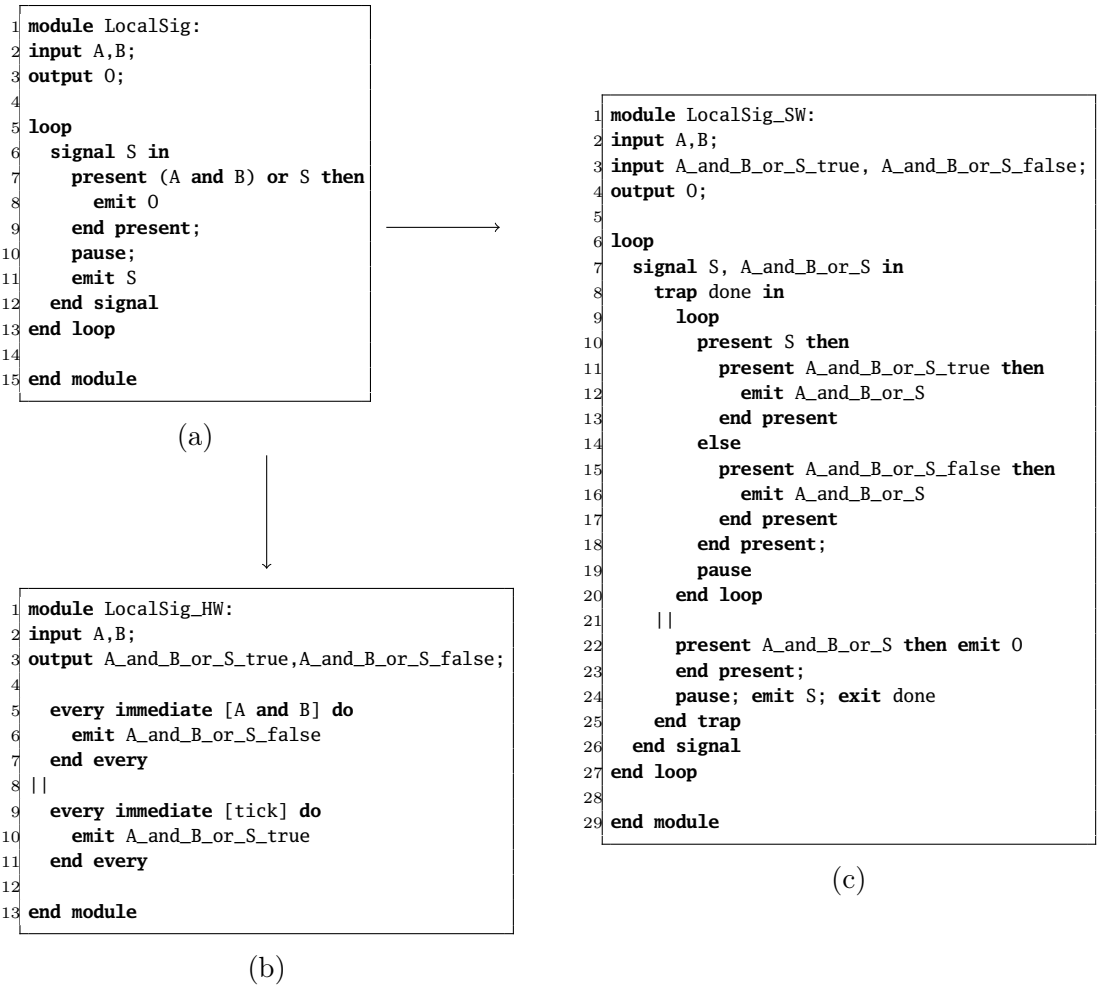
Im Hardwaremodul kann bei der Berechnung des Ausdrucks `pre(S)` nicht entschieden werden, ob das Signal `S` suspendiert ist. Dies ließe sich jedoch durch die Einführung eines weiteren Signals `LocalTick` lösen, das parallel zum Rumpf des `suspend`-Blocks in jeder Instanz (in der der Block aktiv ist) emittiert wird. Es kann aber auch hier bei einer eventuellen Reinkarnation von `S` der korrekte Wert in der vorigen Instanz nicht berechnet werden.

Das eigentliche Problem besteht also in der korrekten Behandlung von schizophrenen Signalen. Dazu wurde ein Workaround für diesen Ansatz eingeführt, bei dem zwischen den Modulen keine Daten von lokalen Signalen ausgetauscht werden müssen. Die eigentliche Berechnung des Teilausdrucks mit dem lokalen Signal verbleibt im Softwaremodul, das Hardwaremodul berechnet für alle hypothetischen Signalbelegungen der lokalen Signale, die in der Expression vorkommen, ein Hilfssignal.

Die Idee für die Hilfskonstruktion wird anhand des in Abbildung 3.4 gezeigten Beispiels `LocalSig` skizziert. Das lokale Signal `S` wird innerhalb einer Schleife instantan redeklariert. Sein Status bzw. Wert kann daher nicht über ein Interfacesignal an das Hardwaremodul übermittelt werden. Stattdessen werden im Hardwaremodul zwei Hilfssignale `A_and_B_or_S_true` und `A_and_B_or_S_false` für beide möglichen Werte von `S`, *present* oder *absent*, berechnet. Die Fallunterscheidung für `S` *present* oder *absent* erfolgt im Softwaremodul: Ist `S` *present* wird das Hilfssignal `A_and_B_or_S` emittiert, wenn `A_and_B_or_S_true` *present* ist, ist `S` *absent* wird das Hilfssignal `A_and_B_or_S` emittiert, wenn `A_and_B_or_S_false` *present* ist. Die Verwendung von Variablen zur Kommunikation zwischen den Modulen ist nicht möglich, da ein schreibender Zugriff auf Variablen in konkurrierenden Threads nicht erlaubt ist.

Die Repräsentation von lokalen Signalen und Interfacesignalen ist im KEP identisch. Vom Logic Block aus kann also ohne Probleme auf den Status- bzw. Signalwert eines lokalen Signals zugegriffen werden. Bei der Hardwaresynthese kann daher die Berechnung der beiden Hilfssignale `A_and_B_or_S_true` und `A_and_B_or_S_false` durch die Berechnung des eigentlichen Hilfssignals `A_and_B_or_S` substituiert werden. Bei der Softwaresynthese kann der komplette `present S then ... else ... end` Block weggelassen werden.

Der Aufwand für die Hilfskonstruktion steigt jedoch quadratisch mit der Anzahl der verwendeten lokalen Signale in der Expression und ist ungeeignet für valued Signals. Eine klare Partitionierung in Software- und Hardwareteile findet außerdem nur unvollständig statt, da die Berechnung von Teilausdrücken mit lokalen Signalen im Softwaremodul verbleibt und die endgültige Partitionierung bei der HW/SW



```
module LocalSig:
input A,B;
output O;

loop
  signal S in
    present (A and B) or S then
      emit O
    end present;
    pause;
    emit S
  end signal
end loop

end module
```

(a)

```
module LocalSig_HW:
input A,B;
output A_and_B_or_S_true,A_and_B_or_S_false;

  every immediate [A and B] do
    emit A_and_B_or_S_false
  end every
||
  every immediate [tick] do
    emit A_and_B_or_S_true
  end every

end module
```

(b)

```
module LocalSig_SW:
input A,B;
input A_and_B_or_S_true, A_and_B_or_S_false;
output O;

loop
  signal S, A_and_B_or_S in
    trap done in
      loop
        present S then
          present A_and_B_or_S_true then
            emit A_and_B_or_S
          end present
        else
          present A_and_B_or_S_false then
            emit A_and_B_or_S
          end present
        end present;
        pause
      end loop
    ||
      present A_and_B_or_S then emit O
      end present;
      pause; emit S; exit done
    end trap
  end signal
end loop

end module
```

(c)

Abb. 3.4.: Hilfskonstruktion für lokale Signale: (a) Programmbeispiel mit lokalem Signal S, (b) Hilfskonstruktion im Hardwaremodul, (c) Hilfskonstruktion im Softwaremodul

Synthese vollzogen werden muss.

Eine weitere Möglichkeit besteht darin, Quellen von Schizophrenie in einem vorgeschalteten Schritt aufzulösen. Die hierzu betrachteten Verfahren arbeiten jedoch entweder nicht auf Esterel Ebene [6, Kapitel 12] oder basieren auf Erweiterungen bzw. Dialekten von Esterel [32, 33]. Die Beseitigung von Schizophrenie geht mit einem Anwachsen des Codes einher, das im schlimmsten Fall quadratisch sein kann. Ein Anwachsen des Codes steht jedoch dem gesetzten Ziel einer Laufzeitverbesserung entgegen.

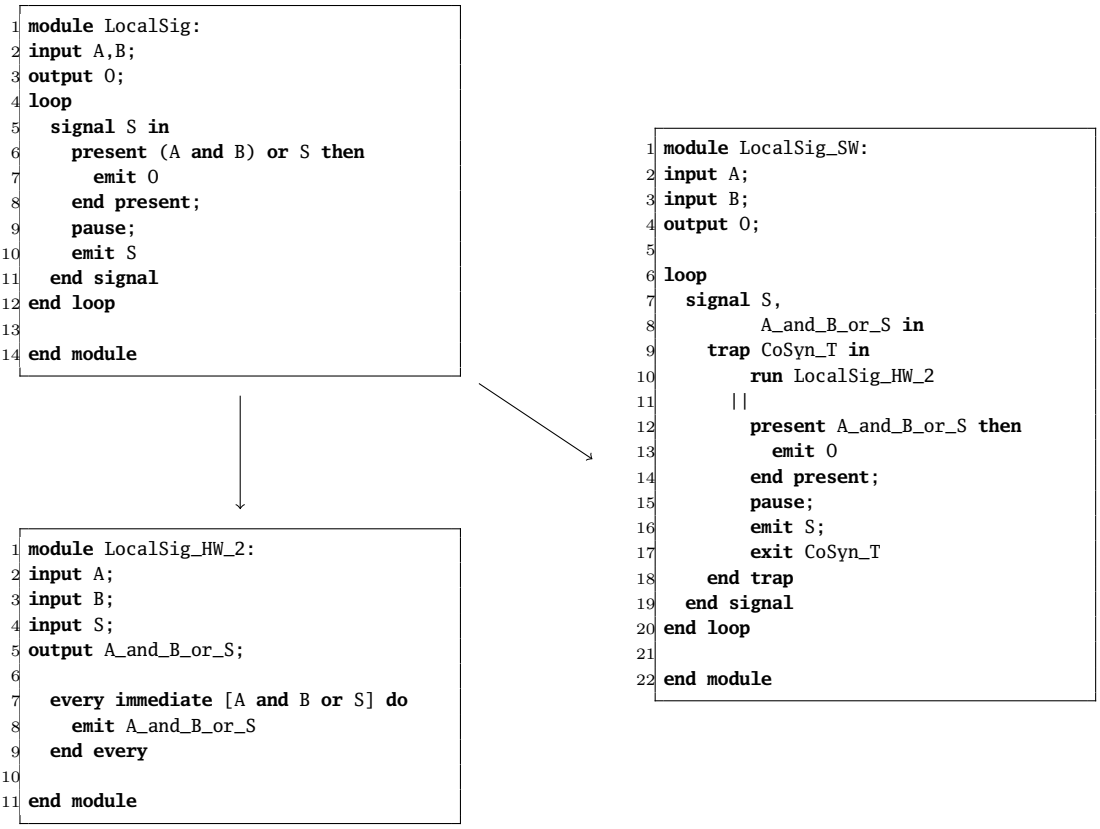
Aufgrund der mit diesem Ansatz verbundenen Schwierigkeiten ist der vorgestellte Ansatz nicht praktikabel. Ein an der Semantik von lokalen Signalen in Esterel ausgerichtetes Verfahren wird im folgenden Kapitel vorgestellt.

3.3.2. Partitionierungsansatz mit lokalen Hardwaremodulen

Der in der Arbeit verwendete Lösungsansatz verwendet eine Partitionierung des ESTEREL Programms in ein Software- und mehrere Hardwaremodule. Für jeden Gültigkeitsbereich eines Signals, d.h. für die globale und für jede lokale Signaldeklaration, existiert ein eigenes Hardwaremodul. Jedes Hardwaremodul wird parallel zum Rumpf des jeweiligen `signal`-Statements ausgeführt. Das Hilfssignal zu einer Expression wird in dem hierarchisch niedrigsten Signal-Gültigkeitsbereichs deklariert und in dem zugehörigen Hardwaremodul berechnet, in dem alle Signale der Expression bekannt sind. Werden beispielsweise drei lokale Signale `A`, `B` und `C` durch drei ineinandergeschachtelte `signal`-Statements deklariert, so wird eine Expression `A or B` in dem zur Signaldeklaration von `B` gehörigen Hardwaremodul berechnet. Damit soll erreicht werden, dass das Hilfssignal in einem möglichst großen Gültigkeitsbereich sichtbar ist und eine ggf. häufiger vorkommende Expression mehrmals ersetzen kann. Alle in der Expression verwendeten Signale werden als Eingangssignale des Hardwaremoduls deklariert, das Hilfssignal als Ausgangssignal. Im Gegensatz zu der im vorigen Abschnitt vorgestellten Partitionierungsmethode mit nur einem parallel zum Softwaremodul ausgeführten Hardwaremodul, können in diesem Ansatz das Software- und die Hardwaremodule über lokale Signale miteinander kommunizieren, da eine Signalreinkarnation immer auch eine Reinkarnation des in dem Gültigkeitsbereich ausgeführten Hardwaremoduls zur Folge hat.

Abbildung 3.5 zeigt die Partitionierung des Beispiels *LocalSig* mit einer lokalen Signaldeklaration. Das Hardwaremodul *LocalSig_HW_2* wird innerhalb des Gültigkeitsbereichs des lokalen Signals `S` parallel zum ursprünglichen Rumpf der lokalen Signaldeklaration gestartet. Alle zur Berechnung notwendigen Signale können nun direkt im Interface des Hardwaremoduls deklariert werden. Das Hilfssignal `A_and_B_or_C`, das die Expression `A and B or C` im `present` Statustest ersetzt, wird im gleichen Gültigkeitsbereich deklariert. Da die Hilfssignale im Hardwaremodul in einer Endlosschleife berechnet werden, terminiert die Ausführung des Hardwaremoduls und damit das Parallelstatement nicht. Die Terminierung wird daher durch ein `trap` Statement, das das Parallelstatement umgibt, erzwungen. Ist das Ende der ursprünglichen Signaldeklaration erreicht, initiiert eine `exit` Anweisung die Beendigung der `trap` Anweisung.

Da die Repräsentation von lokalen Signalen und Interfacesignalen im KEP identisch ist, können alle Hardwaremodule in *einen* Logikblock synthetisiert werden. Das Softwaremodul könnte fast direkt in ein KEP Assembler Programm kompiliert werden, lediglich die `run` Anweisungen müssten durch z.B. `nothing` ersetzt werden. Die neu eingeführten Parallel-, `trap`- und `exit`- Statements sind dann jedoch überflüssig und werden vor der Softwaresynthese ebenfalls entfernt. Alle bei der Partitionierung eingeführten `trap`- und `exit`-Statements beginnen mit dem reservierten Präfix „COSYN_TRAP“, über das sie später identifiziert werden. Das `run`-Statement ist das erste Statement nach dem `trap` und wird ebenfalls entfernt.



```
module LocalSig:
input A,B;
output O;
loop
  signal S in
    present (A and B) or S then
      emit O
    end present;
    pause;
    emit S
  end signal
end loop

end module
```

```
module LocalSig_HW_2:
input A;
input B;
input S;
output A_and_B_or_S;

  every immediate [A and B or S] do
    emit A_and_B_or_S
  end every

end module
```

```
module LocalSig_SW:
input A;
input B;
output O;

loop
  signal S,
         A_and_B_or_S in
    trap CoSyn_T in
        run LocalSig_HW_2
      ||
        present A_and_B_or_S then
          emit O
        end present;
        pause;
        emit S;
        exit CoSyn_T
    end trap
  end signal
end loop

end module
```

Abb. 3.5.: Partitionierungsansatz mit lokalen Hardwaremodulen

3.3.3. Transformationsregeln

Grundvoraussetzung für die Transformation ist, dass das ESTEREL Programm in einem Modul vorliegt. Alle im Programm verwendeten Symbolnamen müssen eindeutig sein, damit im KEP alle Signale eineindeutig einer bestimmten Adresse zugeordnet werden können. Zudem ist zu beachten, dass der KEP Compiler bei Symbolnamen nicht zwischen Groß- und Kleinschreibung unterscheidet, bzw. keine Kleinbuchstaben verwendet werden dürfen. Dazu wird das Programm entsprechend vorverarbeitet: Mit dem CEC [15] Modul *cec-expandmodules* wird das Programm zunächst in ein Modul expandiert, ein weiterer Vorverarbeitungsschritt ersetzt in allen vorkommenden Symbolnamen Kleinbuchstaben durch Großbuchstaben und benennt alle Namen durch Anhängen eines nummerierten Postfixes global eindeutig.

Signal Expressions

Signal Expressions werden wie im einführenden Beispiel behandelt. Zusätzlich zu den Operatoren `and`, `or` und `not` kann zusätzlich noch der `pre` Operator verwendet werden. In Signal Expressions ist das Argument des `pre` Operators entweder ein ein-

faches Signal oder eine Signal Expression. Eine Schachtelung von `pre` Operatoren ist nicht erlaubt. Der `pre` Operator ist distributiv gegenüber `and` und `or`, jedoch nicht gegenüber `not`. Für die erste Instanz der Gültigkeit eines Signals `S` gilt nämlich `pre(not S)` = *absent*. Es gelten die Regeln aus Abschnitt 2.2.2 auf Seite 10. Der `pre` Wert eines Signals kann nicht innerhalb des mit dem KEP verbundenen Logik Blocks berechnet werden. Es ist auf Hardwareebene nicht feststellbar, ob das Signal reinkarniert wurde, oder ob die Signaldeklaration innerhalb eines `suspend` Blocks liegt. Gemäß des Esterel Primers wird daher parallel zum Rumpf der Signaldeklaration ein Hilfssignal mit dem Status des Signals in der vorigen Instanz berechnet.

Signal Expressions sind Bestandteil eines `present` Signaltests oder einer Delay Expression. Während der Partitionierung wird beim Auftreten von Signal Expressions eine Funktion aufgerufen, die folgende Transformation durchführt:

1. Führe ein neues Hilfssignal s mit einem eindeutigen Namen ein und ersetze die Signal Expression e durch dieses Signal.

2. Multipliziere alle in der Expression e enthaltenen `pre` Operatoren entsprechend der o.g. Regeln aus, bis sie nur noch auf ein Signal angewendet werden.

3. Ersetze alle in der Expression e vorkommenden Anwendungen `pre(S)` bzw. `pre_0(S)` und `pre_1(S)` durch ein Hilfssignal `pre_S` bzw. `pre0_S` und `pre1_S`. Wenn dieses Signal nicht schon berechnet wird, deklariere es auf der gleichen Ebene, wie das Signal `S` und berechne seinen Status pre_0 oder pre_1 parallel zum Rumpf der jeweiligen Deklaration.

4. Finde den hierarchisch niedrigsten Gültigkeitsbereich, in dem alle Signale der Signal Expression e bekannt sind. Deklariere in dem Gültigkeitsbereich zusätzlich das Hilfssignal s. Deklariere in dem korrespondierenden Hardwaremodul alle in der Signal Expression vorkommenden Signale als `input` und das Hilfssignal s als `output`.

5. Füge zum Rumpf des Hardwaremoduls einen Thread hinzu, der das Hilfssignal in jeder Instanz emittiert, in der die Signal Expression e *present* ist.

Data Expressions

Data Expressions werden in `if` Anweisungen, bei der Emittierung von Signalen mit Wert (valued Signals), bei Wertzuweisungen an Variablen oder beim Behandeln von Exceptions benutzt. Eine Data Expression wird gemäß *Esterel Primer* [7, Abschnitt 7.4.1] nach den folgenden Regeln produziert:

```
DataExpression :=
{
    Constant |
    ?SignalIdentifier |
    ??ExceptionIdentifier |
```

```
    pre(?SignalIdentifier) |
    Variable |
    (DataExpression) |
    DataExpression ∘ DataExpression |
    not DataExpression |
    - DataExpression |
    FunctionCall
}
```

Als Verknüpfungsoperator (∘) zwischen zwei Data-Expressions sind boolsche (`and`,`or`), arithmetische (`+`,`-`,`*`,`/`,`mod`) und Vergleichsoperatoren (`=`,`<>`,`<`,`<=`,`>`,`>=`) erlaubt. *FunctionCalls*, d.h. der Aufruf einer externen Funktion in nativem Programmcode, werden vom KEP z.Zt. nicht unterstützt und werden daher nicht berücksichtigt.

Die Implementierung von Addition und Subtraktion sowie von Vergleichsoperatoren ist in sequenzieller Logik zwar möglich, aber mit einem hohen Hardwareaufwand verbunden, da jedes verwendete Signal (32 Bit) mit der Hardware verbunden werden muss, und da die Implementierung der Schaltfunktionen mit einem enorm hohen Platzbedarf verbunden ist. Die Berechnung von arithmetischen und Vergleichsfunktionen sollte daher weiter mit Hilfe der ALU des KEP stattfinden.

Variablen stellen ein zusätzliches Problem dar: Nach der Spezifikation von Esterel sind sie im Gegensatz zu Signalen nicht zur Kommunikation zwischen Threads geeignet und können ihren Wert während einer Instanz mehrmals ändern. Verschiedene Threads können sie zwar lesen aber nicht schreiben. Auch die Implementierung im KEP unterscheidet sich grundlegend von Signalen. Während Signale mit gleichem Namen eindeutig einer Speicherstelle zugeordnet sind, werden Variablenwerte nur temporär innerhalb von Registern gespeichert.

Hat eine Data Expression ein nicht boolsches Ergebnis, wird sie nicht von der HW/SW Co-Synthese berücksichtigt. Anderenfalls werden Teilberechnungen, die Vergleichs- und arithmetische Operatoren oder Variablen enthalten, extrahiert und durch ein Hilfssignal ersetzt. Diese Hilfssignale werden unmittelbar vor der Data Expression mit dem entsprechenden Wert emittiert und sind pro Instanz und Data Expression eindeutig.

Der `pre(?S)` Operator liefert den Wert des Signals in der vorigen Instanz und darf nur auf einfache Signale angewandt werden.

Beim Auftreten von Data Expressions wird die folgende Transformation durchgeführt:

1. Ist das Ergebnis der Data Expression nicht boolsch, lasse sie unverändert.
2. Ansonsten führe ein neues Hilfssignal s mit einem eindeutigen Namen ein und ersetze die Data Expression e durch dieses Signal.
3. Ersetze alle in der Expression e vorkommenden Anwendungen `pre(?S)` durch ein Hilfssignal `preV_S`. Wenn dieses Signal nicht schon berechnet wird, deklariere es auf der gleichen Ebene, wie das Signal `S` und berechne seinen Status und Wert parallel zum Rumpf der jeweiligen Deklaration.

4. Extrahiere alle Teilausdrücke p_x, die auf oberster Ebene eine Vergleichsoperation ausführen oder eine boolsche Variable sind und ersetze sie durch ein Hilfssignal a_x. Das Hilfssignal muss pro Instanz und Data Expression eindeutig sein. Emittiere die (boolschen) Hilfssignale mit dem entsprechenden Wert unmittelbar vor der Data Expression (`emit` a_x`(`p_x`)`). Ist die Data Expression Bestandteil eines `sustain` Statements, muss es in diesem Fall *dismantled* werden.

5. Finde den hierarchisch niedrigsten Gültigkeitsbereich, in dem alle Signale der Data Expression e bekannt sind. Deklariere in dem Gültigkeitsbereich zusätzlich die Hilfssignale s und a_x. Deklariere in dem korrespondierenden Hardwaremodul alle in der Data Expression vorkommenden Signale als `input` und das Hilfssignal s als `output`.

6. Füge zum Rumpf des Hardwaremoduls einen Thread hinzu, der das Hilfssignal s mit dem Wert der Data Expression e in jeder Instanz emittiert. Initialisiere außerdem alle nicht initialisierten Signale aus e mit dem Wert *false*.

3.4. Zwischenschritt: Logikminimierung

Um aus den Hardwaremodulen einen möglichst minimalen Logikblock synthetisieren zu können, werden in einem optionalen Zwischenschritt vor der Software- und Hardwaresynthese die in den Hardwaremodulen berechneten Expressions minimiert. Die Logikminimierung basiert auf dem MV-SIS Logiksynthesesystem, das als Ein- und Ausgabeformat das BLIF Format benutzt.

Anhand des Beispiels in Abbildung 3.6 sollen die einzelnen Schritte der Logikminimierung veranschaulicht werden.

Das Hardwaremodul berechnet den Status der Hilfssignale `A_OR_B_AND_B_OR_C` und `A_OR_B`. In der BLIF Datei werden zunächst alle Eingangs- und Ausganssignale deklariert. Danach wird mit dem Schlüsselwort `.names` die Funktion `A_OR_B_AND_B_OR_C` mit den Parametern `A`, `B` und `C` deklariert. Auf Basis der Signal Expression `(A or B) and (C or D)` wird die Implementierung der Funktion in Form einer Wertetabelle berechnet und im Anschluss an die Funktionsdeklaration in die Datei geschrieben. Das Verfahren hierzu ist sehr einfach gehalten und testet sämtliche Signalbelegungen durch. Der Code für die zweite Funktion wird analog erzeugt. Anschließend wird MV-SIS mit der erzeugten BLIF Datei als Eingabe und Parametern zur zweistufigen Logikminimierung und zur Extraktion von gemeinsamen Teilausdrücken gestartet. Das Ergebnis ist in der BLIF Datei rechts unten in Abbildung 3.6 zu sehen. Man kann erkennen, dass `A or B` als gemeinsamer Teilausdruck beider Funktionen erkannt wurde, da der Parameter `A` in der ersten Funktion durch den Funktionswert `A_or_B` der zweiten Funktion ersetzt wurde.

Die Rücktransformation in ein Hardwaremodul erfolgt analog zur Transformation mit folgenden Erweiterungen: Werden in der BLIF Datei lokale Funktionen (als extrahierte Teilausdrücke) erzeugt, so werden lokale Signale mit den Funktionsnamen

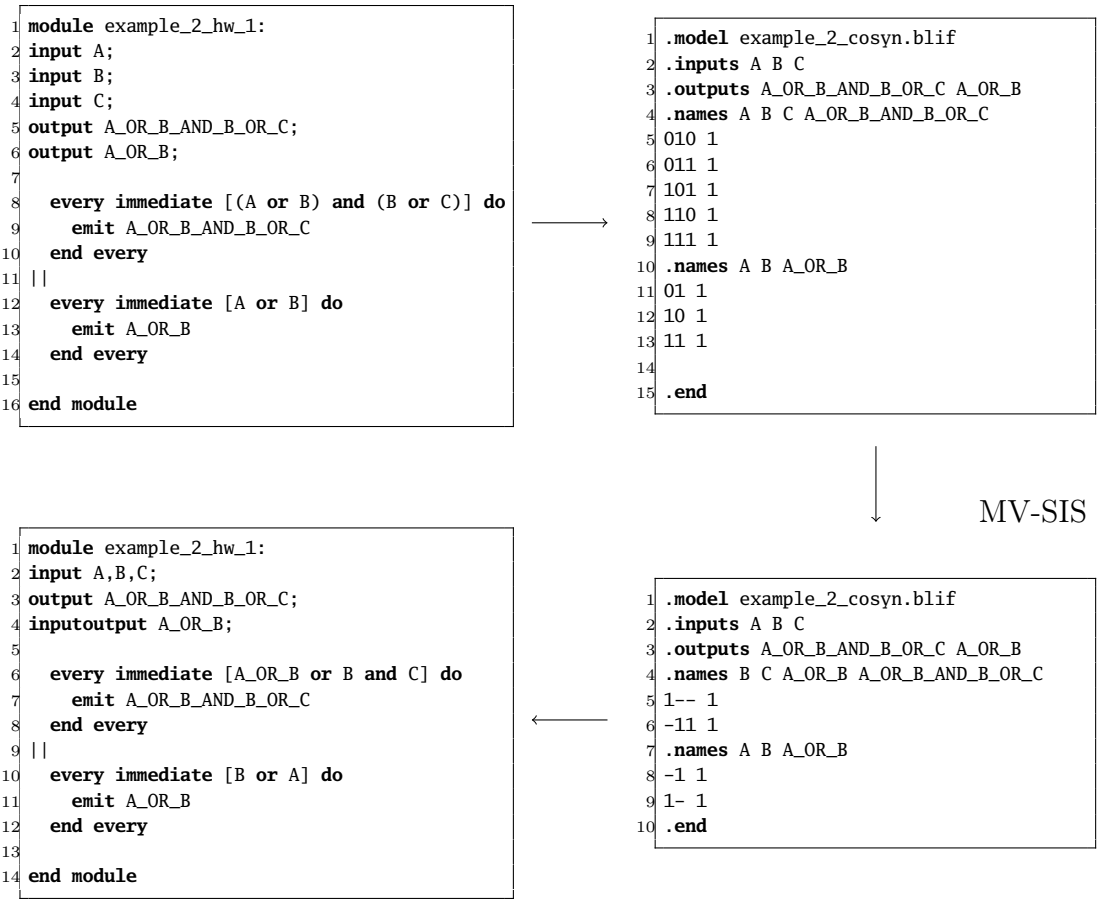


Abb. 3.6.: Logikminimierung

im Hardwaremodul deklariert. In dem rücktransformierten Hardwaremodul wurde das Signal `A_OR_B` als `inputoutput` deklariert, da es innerhalb des Hardwaremoduls gelesen wird. Das Lesen eines Ausgangssignals ist zwar in `Esterel` erlaubt, in VHDL jedoch nicht. Das Signal wird daher auf diese Art und Weise markiert, um es bei der Softwaresynthese gesondert identifizieren zu können.

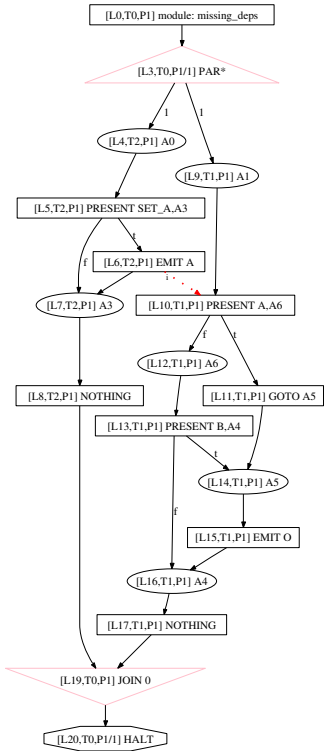
3.5. Software- und Hardwaresynthese

3.5.1. Softwaresynthese

Zur Ausführung des Softwaremoduls auf dem KEP muss dieses zunächst wie andere Esterel Programme mit dem Esterel2KASM Compiler [23] von Esterel nach KEP Assembler Code (KASM) übersetzt werden. Der KEP Assember Compiler generiert danach aus dem KASM den maschinenlesbaren Operationscode.

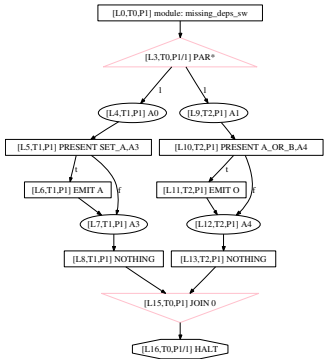
Durch die Partitionierung gehen jedoch Informationen, die der Esterel2KASM

```
module missing_deps:

input SET_A, B;
output O;

signal A in

  present SET_A then
    emit A
  end present
||
  present A or B then
    emit O
  end present

end signal

end module
```




```
%%% Esterel Module: missing_deps

INPUT SET_A,B
OUTPUT O
SIGNAL A

EMIT _TICKLEN,#12

[L01,T0,P1]        PAR 1,A0,2
[L02,T0,P1]        PAR 1,A1,1
[L03,T0,P1/1]      PARE A2,1
[L04,T2,P1]    A0: PRESENT SET_A,A3
[L05,T2,P1]        EMIT A
[L06,T2,P1]    A3: NOTHING
[L07,T1,P1]    A1: PRESENT A,A6
[L08,T1,P1]        GOTO A5
[L09,T1,P1]    A6: PRESENT B,A4
[L10,T1,P1]    A5: EMIT O
[L11,T1,P1]    A4: NOTHING
[L12,T0,P1]    A2: JOIN 0
[L13,T0,P1/1]      HALT
```

```
module missing_deps_sw:
input SET_A;
input B;
output O;

signal A,
       A_OR_B in

[
  present SET_A then
    emit A
  end present
||
  present A_OR_B then
    emit O
  end present
];


end signal

end module
```




```
%%% Esterel Module: missing_deps_sw

INPUT SET_A,B
OUTPUT O
SIGNAL A,A_OR_B

EMIT _TICKLEN,#11

[L01,T0,P1]        PAR 1,A0,1
[L02,T0,P1]        PAR 1,A1,2
[L03,T0,P1/1]      PARE A2,1
[L04,T1,P1]    A0: PRESENT SET_A,A3
[L05,T1,P1]        EMIT A
[L06,T1,P1]    A3: NOTHING
[L07,T2,P1]
A1: PRESENT A_OR_B,A4
[L08,T2,P1]        EMIT O
[L09,T2,P1]    A4: NOTHING
[L10,T0,P1]    A2: JOIN 0
[L11,T0,P1/1]      HALT
```

Abb. 3.7.: Oben: Esterel Programm mit Reader-/Writer Abhängigkeit zwischen zwei Threads, daneben der resultierende CKAG mit Abhängigkeitskante (gestrichelt) und der resultierende KASM Code; Unten: Das aus der Partitionierung resultierende SW-Modul, daneben der resultierende CKAG mit fehlender Abhängigkeitskante und das resultierende KASM Programm mit falschem Scheduling

Compiler zur Berechnung von Abhängigkeiten zwischen Threads benötigt, verloren. Dies soll mit Hilfe des in Abbildung 3.7 gezeigten Beispiels erläutert werden. Das Programm `missing_deps` besteht aus zwei Threads. Im ersten Thread wird das lokale Signal `A` emittiert, wenn das Eingangssignal `SET_A` in der ersten Instanz *present* ist. In dem parallelen Thread wird in der ersten Instanz der Status der Signal Expression `A or B` getestet und für den *present* Fall das Ausgangssignal `O` emittiert. Das Statement `emit A` im ersten Thread ist ein sog. *Writer* von `A`, `present A or B then ...` ist ein sog. *Reader* von `A`. Ein Writer eines Signals muss vor dem Reader ausgeführt werden, was bei der Berechnung des statischen Schedulings berücksichtigt werden muss. In Abbildung 3.7 Mitte oben ist der zum Programm korrespondierende Concurrent KEP Assembler Graph (CKAG) dargestellt, auf dessen Basis der Esterel2KASM Compiler das Scheduling berechnet. Die gestrichelte Kante stellt die Abhängigkeit des `present` Knotens von dem `emit` Knoten dar. Der KEP führt bei gleicher Priorität den Thread mit der höchsten ThreadID zuerst aus, anderenfalls den Thread mit der höchsten Priorität. In dem Beispiel hat Thread 1 die Priorität 2 und Thread 2 die Priorität 1. Der erste Thread wird also vor dem zweiten ausgeführt.

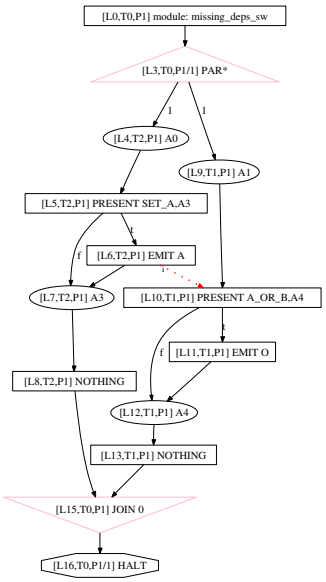
Bei der HW/SW Co-Synthese wird die Signal Expression `A or B` durch das Hilfssignal `A_OR_B` ersetzt. Das Hilfssignal wird innerhalb des mit dem KEP verknüpften Logikblocks berechnet. Dem Esterel2KASM Compiler fehlt die Information, dass `A_OR_B` von `A` abhängt, d.h. dass `present A_OR_B then ...` Reader von `A`, bzw. dass `emit A` Writer von `A_OR_B` ist. Abbildung 3.7 Mitte unten zeigt den zum SW Modul korrespondierenden CKAG, der keine Abhängigkeit zwischen den Threads enthält. Das Scheduling kann nun nicht mehr korrekt berechnet werden, es resultiert der in Abbildung 3.7 rechts unten gezeigte KASM Code, der für Thread 1 und 2 die gleiche Priorität aufweist. Der Thread mit der höheren ThreadID, also Thread 2 wird nun fälschlicherweise zuerst ausgeführt.

Um das Scheduling trotzdem korrekt berechnen zu können, müssen die fehlenden Abhängigkeiten im *CKAG*) ergänzt werden. Dazu werden die Abhängigkeiten der Hilfssignale während der Transformation in eine Datei geschrieben, die vom Esterel2KASM Compiler zur Rekonstruktion der fehlenden Abhängigkeiten eingelesen wird. Abbildung 3.8 zeigt den CKAG mit den rekonstruierten Abhängigkeiten und den korrekten KASM Code.

3.5.2. Hardwaresynthese

Prinzipiell könnte man mit Hilfe des CEC den Esterel Code jedes Hardwaremoduls in eine Hardwarebeschreibung synthetisieren. Dabei würde zunächst ein endlicher Automat aus den Modulen erzeugt werden, der dann in Hardware synthetisiert werden würde. Der dabei entstehende Overhead für die Implementierung des Automaten wäre jedoch viel zu groß. Stattdessen genügt es, aus den Expressions ein einfaches Schaltnetz zu erzeugen, über das die Hilfssignale berechnet werden. Die Implementierung wird anhand des Beispiels aus der Übersichtsgrafik in Abbildung 3.1 beschrieben, das der Übersichtlichkeit halber in Abbildung 3.9 noch einmal abgebildet ist.

```
module missing_deps_sw:
input SET_A;
input B;
output O;

signal A,
       A_OR_B in

[
  present SET_A then
    emit A
  end present
||
  present A_OR_B then
    emit O
  end present
];


end signal

end module
```




```
%%% Esterel Module: missing_deps_sw

INPUT SET_A,B
OUTPUT
SIGNAL A,A_OR_B

EMIT _TICKLEN,#11

[L01,T0,P1]        PAR 1,A0,2
[L02,T0,P1]        PAR 1,A1,1
[L03,T0,P1/1]      PARE A2,1
[L04,T2,P1]    A0: PRESENT SET_A,A3
[L05,T2,P1]        EMIT A
[L06,T2,P1]    A3: NOTHING
[L07,T1,P1]
A1: PRESENT A_OR_B,A4
[L08,T1,P1]        EMIT O
[L09,T1,P1]    A4: NOTHING
[L10,T0,P1]    A2: JOIN 0
[L11,T0,P1/1]      HALT
```

Abb. 3.8.: CKAG des SW Moduls mit rekonstruierter Abhängigkeit zwischen den Threads und resultierender KASM Code mit korrektem Scheduling

```
module intro_example_hw_1:
input D := false : boolean;
input E := false : boolean;
input F := false : boolean;
output D_OR_E_AND_F := false : boolean;

  sustain D_OR_E_AND_F(?D or ?E and ?F)

end module
```

```
module intro_example_hw_2:
input A,B,C;
output A_OR_B_AND_C;

  every immediate [(A or B) and C] do
    emit A_OR_B_AND_C
  end every

end module
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity intro_example is
   port(
      A:in std_logic;
      B:in std_logic;
      C:in std_logic;
      D:in std_logic_vector(1 downto 0);
      E:in std_logic_vector(1 downto 0);
      F:in std_logic_vector(1 downto 0);
      A_OR_B_AND_C:out std_logic;
      D_OR_E_AND_F:out std_logic_vector(1 downto 0)
   );
end intro_example;

architecture intro_example_BEH of intro_example is
begin
   D_OR_E_AND_F(1) <= (D(1) or (E(1) and F(1)));
   A_OR_B_AND_C <= A or (B and C);
end intro_example_BEH;
```

Abb. 3.9.: Hardwaresynthese

Zunächst einmal ist festzustellen, dass aus den zwei Hardwaremodulen ein einziger Logikblock erzeugt werden kann. Im Gegensatz zur ESTEREL-Ebene gibt es auf KEP-Ebene keine semantischen Probleme mit lokalen Signalen, da Schnittstellensignale und lokale Signale im KEP in der gleichen Datenstruktur repräsentiert sind. Da alle Symbolnamen im Programm global eindeutig sind (was als Vorbedingung zur Transformation gefordert war und die neu eingeführten Hilfssignale ebenfalls eindeutig benannt wurden), sind alle Signale eindeutig adressierbar.

Die Anweisungen zum Einbinden von Bibliotheken sind vor der Deklaration einer neuen Entwurfseinheit obligatorisch. Der Name der `entity` bildet sich aus dem Originalnamen des Programms.

Alle Interfacesignale der Logikblöcke werden in der `port`-Deklaration der neuen Entwurfseinheit deklariert. Die Signale `A`, `B` und `C` werden vom Typ `std_logic` als Eingang deklariert, da sie als *pure signals* nur einen Statuswert besitzen. Das Signal `A_OR_B_AND_C` wird analog als Ausgang deklariert. Die boolschen Signale `D`, `E` und `F` besitzen zusätzlich zu ihrem Status einen Signalwert. Sie werden vom Typ `standard_logic_vector(1 downto 0)` als Eingang mit zwei Bit Breite deklariert. Bit 0 trägt den Statuswert und Bit 1 den boolschen Wert. Das Ausgangssignal `D_OR_E_AND_F` wird analog als Ausgang deklariert.

In der anschließenden Verhaltensimplementierung wird ein einfaches Schaltnetz beschrieben. Das Statement `every immediate SigExp do emit AuxSig end` implementiert eine permanente Statuszuweisung des Ergebnisses der Signal Expression `SigExp` an das Hilfssignal `AuxSig`. Stark vereinfacht kann dies in VHDL als schlichte Signalzuweisung `AuxSig <= SigExp` implementiert werden. Dabei ist auf eine vollständige Klammerung der Signal Expression zu achten, da `and` und `or` in VHDL die gleiche Präzedenz haben. In dem Beispiel ergibt sich für `every immediate [A or (B and C)] do emit A_OR_B_AND_C end` die Signalzuweisung `A_OR_B_AND_C <= (A or B) and C`. Eine `sustain AuxSig(ValExpr)` Anweisung emittiert in jeder Instanz das Signal `AuxSig` mit dem Wert der Expression `ValExp`. Stark vereinfacht ergibt sich in VHDL die Signalzuweisung `AuxSig(1) <= ValExpr`. Der Wert jedes Signals in `ValExp` wird über den Index 1 indiziert. Es ist ebenfalls auf eine vollständige Klammerung zu achten. In dem Beispiel ergibt sich für `sustain D_OR_E_AND_F(?D or ?E and ?F)` in VHDL die Signalzuweisung `D_OR_E_AND_F(1) <= D(1) or (E(1) and F(1))`.

Im Falle einer mehrstufigen Logikminimierung eingeführte lokale Signale werden in der VHDL Architekturbeschreibung ebenfalls als lokale Signale deklariert.

3.6. Schnittstelle zwischen Logikblock und KEP

Der bei der Co-Synthese generierte Logikblock ist als einfaches Schaltnetz implementiert. Bei der Verknüpfung mit dem KEP muss von außen sicher gestellt werden, dass alle Eingänge des Logikblocks korrekt belegt sind und dass dem KEP die Ausgänge wieder zugänglich gemacht werden. Alle Daten für lokale und Interfacesignale werden in bestimmten Hardwarestrukturen innerhalb des Interfaceblocks des KEP gespei-

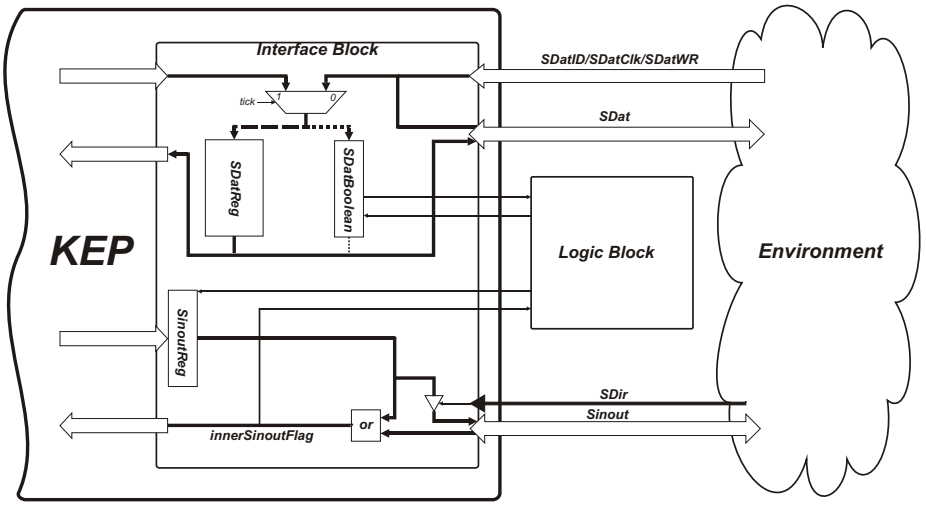


Abb. 3.10.: Schematische Darstellung des erweiterten Interfaceblocks des KEP ohne `pre`-Unterstützung mit angeschlossenem Logikblock

chert. Im Folgenden wird zunächst der Aufbau des Interfaceblocks beschrieben und anschließend notwendige Erweiterungen für eine Verknüpfung mit dem Logikblock vorgestellt.

Abhängig von seiner Konfiguration unterstützt der KEP eine feste maximale Anzahl von n Signalen. Die Statuswerte von lokalen und Ausgangssignalen werden in dem n Bit weiten *SinoutReg* Register gespeichert. Je nach Konfiguration des KEP werden zusätzlich die Statuswerte der vorigen Instanz im *preSinoutReg* vorgehalten. Die Speicherung der Signalwerte erfolgt in einem Dual-Port Block RAM mit 32 Bit Datenwortbreite. Die Schnittstelle zur Umgebung ist folgendermaßen aufgebaut: Über den n Bit weiten bidirektionalen Datenbus *Sinout* kann lesend auf die Statuswerte zugegriffen werden bzw. der Signalstatus von Eingangssignalen gesetzt werden. Die Richtung des Busses wird über die Eingangsleitung *SDir* gesteuert. Im Gegensatz zu den Statuswerten kann auf die 32 Bit weiten Signalwerte nicht parallel zugegriffen werden, da dies einen enormen Hardwarebedarf zur Folge hätte. Für den sequenziellen Zugriff auf den RAM steht ein bidirektionaler 32 Bit weiter Datenbus *SDat*, eine Adressleitung *SDatID*, eine Taktleitung *SDatClk* und eine Leitung zur Initiierung des Schreibvorgangs *SDatWR* zur Verfügung. Die Umwelt erhält nur zwischen zwei Ticks Zugriff auf die Signaldaten. Während des Assemblierens durch den KEP Assembler Compiler erfolgt eine eindeutige Zuordnung der Signalnamen auf eine Speicheradresse. Sind alle Signalnamen eindeutig gewählt, ist diese Abbildung bijektiv.

Um den Logikblock mit dem Interfaceblock verknüpfen zu können, muss dieser geeignet erweitert werden und die Abbildung von Signalnamen auf Adressen bekannt sein. Eingangs- und Ausgangssignale des Logikblocks sind vom Typ `std_logic` bzw.

`std_logic_vector(1 downto 0)`. Der Typ `std_logic` wird für pure Signals verwendet und der Typ `std_logic_vector(1 downto 0)` für valued Signals. Dabei enthält Bit 0 den Status und Bit 1 den boolschen Wert des Signals. Alle Ausgänge der Logik, die einen Statuswert führen, werden mit dem Eingang des *SinoutReg* verbunden. Das Register übernimmt den Wert am Ende jedes Instruktionszyklus. Der Ausgang des *SinoutReg* wird konjunktiv verknüpft mit dem Datenbus *Sinout* und auf den internen Datenbus *innerSinoutFlag* gelegt. Der Bus ermöglicht damit einen Zugriff auf die Statuswerte aller Signale. Die Statuseingänge des Logikblocks werden mit den entsprechenden Signalleitungen des *innerSinoutFlag* verknüpft. Um auf die boolschen Werte parallel zugreifen zu können, wird das zusätzliche *SDatBoolean* Register eingeführt. In diesem Register wird parallel zum RAM eine Kopie des nullten Bits aller Signale (entspricht bei boolschen Signalen dem boolschen Wert) abgelegt. Auf die Ausgänge des Registers kann wie im *SinoutReg* parallel zugegriffen werden. Die Ein- und Ausgänge des Logikblocks, die einen boolschen Ausdruck berechnen, werden mit den entsprechenden Aus- und Eingängen des *SDatBoolean* Registers verknüpft.

3.6.1. Inkrementelle Entwicklung und Partielle Rekonfiguration

Ein Nachteil dieses Co-Synthese Ansatzes ist, dass für jedes *Esterel* Programm ein individueller Logikblock generiert werden muss. Da der KEP zur Zeit als VHDL Code vorliegt und mit Hilfe eines Synthesewerkzeugs für eine FPGA basierte Zielplattform synthetisiert wird, ist die Flexibilität dieses Ansatzes gewährleistet. Soll nicht der gesamte KEP bei jeder Änderung des Logikblocks neu synthetisiert werden, bietet die Xilinx ISE Entwicklungsumgebung [38] spezielle Designmethoden an:

- Bei der *inkrementellen Entwicklung* [37, Kapitel 3] wird das gesamte Design in unterschiedliche Entwurfseinheiten unterteilt. Diese werden auf oberster Ebene zusammengeführt. In diesem Fall ist nur einmalig eine Synthese des gesamten Designs nötig, anschließend werden nur die Entwurfseinheiten neu synthetisiert, in denen eine Veränderung aufgetreten ist. Voraussetzung für die inkrementelle Synthese ist ein festes Interface zwischen den einzelnen Entwurfseinheiten. Zusätzlich wird für ein inkrementelles Platzieren und Verdrahten (*place and route*) gefordert, dass alle Entwurfseinheiten festen Bereichen auf dem FPGA zugeordnet werden (über *area constraints*), die sich nicht überschneiden dürfen, und dass die Größe der Entwurfseinheiten nur begrenzt variiert.

- Bei der *Partiellen Rekonfiguration* [37, Kapitel 5] ist es auf bestimmten FPGA Typen möglich, nur bestimmte Bereiche des FPGA neu zu konfigurieren. Dies ist sogar im laufenden Betrieb möglich. Die Anforderungen in Bezug auf Interface und Fläche sind jedoch restriktiver, als bei der inkrementellen Entwicklung: Die Fläche eines partiell rekonfigurierbaren Moduls hat immer die volle Bausteinhöhe und die Breite ist ein Vielfaches von 4 Slices. Module können nur auf einer geraden 4 Slice Grenze platziert werden. Die Kommunikation über die Grenzen eines partiell rekonfigurierbaren Moduls hinweg mit anderen partiell rekonfigurierbaren oder statischen Modulen muss über festgelegte, statische

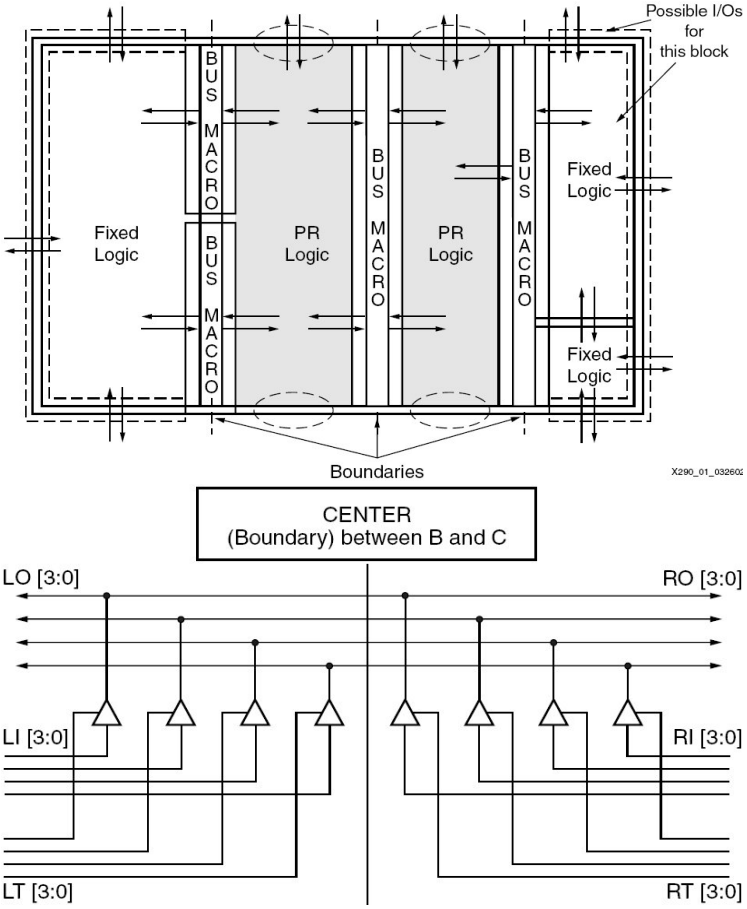


Abb. 3.11.: Oben: Aufteilung in rekonfigurierbare und statische Module, Unten: Implementierung des Bus Makros (Beide Graphiken aus [37])

Routingressourcen erfolgen, die über sog. Bus-Makros instantiiert werden. Ein Bus-Makro ist ein bereits geroutetes Hard-Makro, das eine feste Routingbrücke zwischen zwei Modulen spezifiziert. Jede Zeile eines partiell rekonfigurierbaren FPGAs (z.B. VirtexII oder VirtexII Pro) besitzt vier sog. TBUF Longlines, über die ein Signal von rechts nach links oder umgekehrt wandern kann. Jede Slice verfügt über zwei Tri-State Buffer (TBUF), über die auf eine der vier Longlines geschrieben werden kann. Das Standard Bus-Makro benutzt acht Tri-State Buffer, so dass pro Zeile vier Bit über eine Modulgrenze hinweg geroutet werden können. Daraus ergibt sich eine Breite von vier Slices pro Bus-Makro.

Die verwendete ISE Entwicklungsumgebung unterstützt jedoch den Designfluss für die partielle Rekonfiguration nur unvollständig; für eine vollständige Unterstützung wären weitere Tools notwendig. Für den inkrementellen Designfluss ist eine Anpassung des KEP Designs nötig, da nur Module auf oberster Ebene getrennt synthetisiert werden können. In einer festen Konfiguration des KEP in Form eines ASIC wäre es

denkbar, diesen mit einem rekonfigurierbaren Logikbaustein (z.B. FPGA, PLA) zu verbinden, oder eine rekonfigurierbare Logik im Design des KEP zu integrieren.

3.7. Implementierung

Die Implementierung der HW/SW Co-Synthese basiert auf dem quelloffenen *Columbia Esterel Compiler* (CEC) [15], der von der *Languages and Compilers Group* um *S.A. Edwards* an der Columbia University entwickelt wurde. Der CEC unterstützt eine Teilmenge von ESTEREL V5 und generiert wahlweise C Code für die Softwaresynthese oder eine Verilog bzw. BLIF Beschreibung eines Schaltkreises. Der CEC ist in C++ programmiert und durch seinen modularen Aufbau einfach zu erweitern. Jeder Verarbeitungsschritt erfolgt durch ein eigenständiges Programm. Die einzelnen Module des CEC repräsentieren ein *Esterel* Programm in einer baumartigen Datenstruktur, dem *Abstract Syntax Tree* (AST). Der *Esterel* Parser und die Klasse AST werden mit Hilfe des Parsergenerators ANTLR automatisch erzeugt. Um auf dem AST zu arbeiten, wird das Visitor Entwurfsmuster benutzt. Dazu muss die abstrakte Klasse *Visitor* geeignet implementiert werden.

Eine Unterstützung des PRE Operators fehlt im CEC; der Operator kann weder geparst werden, noch ist ein entsprechendes Feld im AST vorgesehen. Daher wurde die zugrunde liegende Grammatik um den `pre` Operator erweitert und der AST und der ESTEREL Parser mit Hilfe von ANTLR neu erzeugt. Auf eine weitere Dokumentation wird hier verzichtet, weil es sich um minimale Änderungen in der Grammatik handelt.

Die HW/SW Co-Synthese setzt nach den Schritten *Parsen* (cec-strlxml) und *Modulexpansion* (cec-expandmodules) an. Die einzelnen Module im- und exportieren den AST als XML Datei.

4. Experimentelle Auswertung

In diesem Kapitel soll quantifiziert werden, welche Auswirkungen sich durch den Co-Syntheseansatz ergeben. Die Vorteile dieses Ansatzes variieren von Programm zu Programm und sind üblicherweise vor allem dort zu erwarten, wo viele komplexe Ausdrücke vorkommen. Daher wurde für die Versuche ein Satz von Programmen als Benchmark ausgewählt, die eine entsprechende Anzahl an komplexen Ausdrücken aufweisen. Programme ohne Signalausdrücke können auf dem KEP bereits effizient ausgeführt werden.

Den Vorteil der Reduktion von Instruktionszyklen pro Tick und einem damit verbundenen verringerten Stromverbrauch oder einer verbesserten WCRT erkauft man sich mit einem leicht erhöhten Ressourcenverbrauch auf dem KEP und der Ausführungsplattform.

Ein weiterer interessanter Punkt ist die Laufzeit des längsten Pfades im Logikblock. Jedes Schaltnetz hat physikalisch bedingt eine bestimmte Gatterlaufzeit, die mit der Schachtelungstiefe ansteigt. Die Eingangssignale des KEP werden jeweils zu Beginn eines neuen Ticks abgetastet und stehen ab diesem Zeitpunkt auch dem Logikblock zur Verfügung. Die Berechnung des Logikblocks muss zu Beginn des ersten Instruktionszyklus vorliegen, der um einen Systemtakt verzögert nach dem Tick beginnt. Ist der zu ersetzende Ausdruck tief verschachtelt, kann dies in Extremfällen zu einer Verschlechterung der maximalen Taktfrequenz des KEP führen. Dies ist im Normalfall aber nicht zu erwarten und tritt bei den untersuchten Beispielen nicht auf.

Bei den Experimenten werden die folgenden Werte für die normale Ausführung eines `Esterel` Programms und für den Co-Design Ansatz miteinander verglichen:

- *Ticklänge:* Die „WCRT“, die vom Esterel2KASM Compiler durchgeführt wird, ermittelt analytisch eine konservative obere Schranke für den Wert für `_TICKLEN`. Die „AVE“ und „MAX“ Werte geben die gemessenen Durchschnitts- und Maximalwerte an, die mit automatisch und manuell generierten Testszenarien als Eingabevektoren ermittelt wurden. Mit Hilfe von *EsterelStudio* [17] lassen sich Testszenarien für ein ESTEREL Programm automatisch erzeugen. Diese bestehen aus einem oder mehreren Tests, die wiederum für eine oder mehrere Instanzen verschiedene Eingangssignalbelegungen enthalten. Die Testszenarien werden so erzeugt, dass alle Zustände in dem jeweiligen ESTEREL Programm erreicht werden.

- *Signale:* Diese Spalte gibt an, wie viele Hilfssignale benötigt wurden, um die Expressions zu ersetzen. Die Menge an maximal zur Verfügung stehenden Signalen ist von der Konfiguration des KEP abhängig. Die Leistungsaufnahme

steigt mit der in der Konfiguration des KEP unterstützten Anzahl von Signalen und sollte daher möglichst klein gehalten werden. Die Anzahl der für das Co-Design zur Verfügung stehenden Hilfssignale ist also begrenzt.

- *FPGA Ausnutzung:* Für die Logik und insbesondere für die Erweiterung des Interfaces zum KEP steigt die Gesamtgröße des Designs. Für den FPGA als Zielplattform wird die FPGA Ausnutzung durch die Anzahl der belegten Slices und der 4 Input Look Up Tables (LUTs) angegeben.

- *Energieverbrauch:* Die Leistungsaufnahme P des KEP und des Co-Designs wird mit Hilfe des Xilinx WEB Power Tools [36] berechnet, das die Leistungsaufnahme auf Basis des verwendeten FPGA Typs, der verwendeten FPGA Ressourcen und der Taktfrequenz der Logik approximiert. Jeder Instruktionszyklus hat eine Dauer von drei Prozessortakten. Dazu kommt die Zeit zum Lesen und Schreiben der Signale. Der Energieverbrauch pro Tick ergibt sich aus der Leistungsaufnahme P multipliziert mit der Dauer eines Ticks , $T_{tick} = (3V\_{\text{TICKLEN}} + 2)T_{\text{osc}}$. Für die Berechnung der Spitzenleistungsaufnahme P_{peak} wird eine Taktfrequenz der Logik von 40 MHz angenommen. Für die Berechnung der Leistungsaufnahme im unbeschäftigten Zustand P_{idle}, d.h. in der Zeit vom Ende der Abarbeitung der Instruktionen eines Ticks bis zum Start des nachfolgenden Ticks, wird eine Frequenz von 0 MHz angenommen. Die Leistungsaufnahme verringert sich in diesem Fall auf die Ruheleistung des FPGA. „WCRT“ gibt den im Extremfall größten zu erwartenden Wert für den Energieverbrauch pro Tick an, wenn die Ticklänge gleich der WCRT ist. „AVE“ und „MAX“ basieren auf den gemessenen Durchschnitts- und Maximalwerten für die Ticklänge.

- *Delay:* Die Signallaufzeit des längsten Pfades der Logik in Nanosekunden wird mit Hilfe des ISE Entwicklungsumgebung berechnet.

Ausführungsplattform für die Experimente ist ein KEP der bis zu 85 Signale und 60 Threads unterstützt. Der KEP bzw. KEP und Logikblock wurden in allen Experimenten für den Xilinx Virtex II Pro (xc2vp30-6ff896) FPGA synthetisiert und auf diesem ausgeführt. Der KEP in dieser Größe benötigt in etwa ein Drittel der auf dem FPGA vorhandenen Ressourcen.

In den Experimenten werden folgende fünf Benchmarks miteinander verglichen: Das TCINT Programm aus der Estbench [12], zwei azyklische Versionen des Token Ring Arbiters mit 3 und 10 Stationen, das Programm SyntheticExample [28], das einen Filter modelliert und das BACKHOE Programm, welches für die Simulation eines Baggers benutzt wird.

Die Ergebnisse der Experimente sind in Tabelle 4.1 gezeigt. Der Co-Design Ansatz reduziert verglichen mit der normalen Ausführung auf dem KEP sowohl die analytisch berechnete WCRT zwischen 17.6% und 58.9%, als auch die experimentell ermittelten Durchschnitts- und Maximalwerte zwischen 9.8% und 43.7% bzw. zwischen 11.9% bzw. 31.2%.

Andererseits erkauft man sich die aufgezeigten Vorteile durch einen erhöhten Ressourcenverbrauch in Form von Signalen auf dem KEP und in Form eines erhöhten

Benchmark	Design	Ticklength			Signals			FPGA Utilization		Energy/Tick [μWs]			Delay [ns]
		WCRT	AVE	MAX	In	Out	Local	# slices	# 4 input LUTs	WCRT	AVE	MAX	
Tcint	KEP only	155	59	101	19	20	12	5144	7591	9.574	7.237	8.270	-
	Co-Design	115	51	89	19	20	25	5247	7751	7.147	5.591	6.546	5.285
	Diff	-40	-8	-12		+13		+103	+160	-2.399	-1.646	-1.724	-
	% Diff	-25.8	-13.6	-11.9		+25.5		+2.0	+2.1	-25.062	-22.741	-20.849	-
Token Ring Arbiter (3 stations)	KEP only	74	51	61	3	3	16	5144	7591	4.592	4.051	4.297	-
	Co-Design	61	46	50	3	3	19	5254	7735	3.820	3.469	3.569	11.852
	Diff	-13	-5	-11		+3		+110	+144	-0.772	-0.582	-0.728	-
	% Diff	-17.6	-9.8	-18.0		+13.6		+2.1	+1.9	-16.806	-14.360	-16.931	-
Token Ring Arbiter (10 stations)	KEP only	256	172	201	10	10	51	5144	7591	17.785	13.743	14.457	-
	Co-Design	208	148	155	10	10	61	5214	7762	12.943	11.460	11.636	18.850
	Diff	-48	-24	-46		+10		+70	+171	-2.842	-2.284	-2.821	-
	% Diff	-18.75	-14.0	-22.9		+19.6		+1.4	+2.3	-18.007	-16.616	-19.514	-
Synthetic Example	KEP only	133	94	101	16	7	14	5144	7591	8.221	7.286	7.458	-
	Co-Design	90	77	83	16	7	40	5217	7762	5.624	5.322	5.472	5.774
	Diff	-43	-17	-18		+26		+73	+170	-2.592	-1.964	-1.986	-
	% Diff	-32.3	-18.1	-17.8		+70.2		+1.4	+2.2	-31.591	-26.959	-26.624	-
Backhoe	KEP only	209	16	32	12	15	3	5144	7591	12.895	8.171	8.565	-
	Co-Design	86	9	22	12	15	5	5254	7734	5.369	3.465	3.790	4.402
	Diff	-123	-7	-10		+2		+110	+143	-7.526	-4.707	-4.775	-
	% Diff	-58.9	-43.7	-31.2		+6.7		+2.1	+1.9	-58.362	-57.599	-55.745	-

Tab. 4.1.: Versuchsergebnisse

Hardwareaufwands für den Logikblock und seine Anbindung an den KEP. Der zusätzliche Hardwareaufwand des Co-Designs verglichen mit dem KEP liegt zwischen 1.4% bis 2.1% für die Anzahl der Slices und zwischen 1.9% bis 2.2% für die Anzahl der 4 Input LUTs und fällt damit kaum ins Gewicht. Die Leistungsaufnahme des FPGA im Ruhezustand beträgt für beide Designs $P_{\text{idle}} = 492\text{mW}$ und im beschäftigten Zustand $P_{\text{peak}} = 820\text{mW}$ für den KEP und ca. $P_{\text{peak}} = 827\text{mW}$ für das Co-Design (Werte nicht in der Tabelle ersichtlich). Der zusätzliche Hardwarebedarf hat somit einen zu vernachlässigenden Einfluss auf den Energieverbrauch pro Tick. Hier überwiegen die Einsparungen, die aus der Reduzierung der WCRT, als auch aus der Reduzierung der Durchschnitts- und Maximalwerte für die Ticklänge resultieren. Für den analytisch ermittelten maximalen Energieverbrauch ergeben sich Einsparungen zwischen 16.8% und 58.4%. Im experimentell ermittelten Durchschnitt reduziert sich der Energieverbrauch um 14.3% bis 57.6%.

Signale sind eine begrenzte Ressource; ihre maximale Anzahl hängt von der Konfiguration des KEP ab (hier 85 Signale). In der Praxis würde man eine KEP Konfiguration wählen, die bestmöglich zu der Anwendung passt. Der Signalbedarf steigt je nach Benchmark z.T. erheblich zwischen 6.7% für den BACKHOE-Benchmark und 70.2%.für den SyntheticExample-Benchmark. Während bei allen Beispielen von einer statischen KEP Konfiguration ausgegangen wurde, soll für das SyntheticExample sowohl für die reine Softwaresynthese, als auch für das Co-Design eine minimale Konfiguration des KEP für die Berechnung des Energieverbrauchs zugrunde gelegt werden. Der KEP unterstützt in beiden Fällen 12 Threads. Für die Softwaresynthese werden 37 Signale benötigt und für das Co-Design 63. Es wurden folgende Werte Ermittelt:

- *nur KEP:* #slices: 2913, #4 Input LUTs: 4390, Energy/Tick [μWs]: 6,687 (WCRT), 6,199 (AVE), 6,291 (MAX).

- *Co-Design:* #slices: 3174,#4 Input LUTs: 4633, Energy/Tick [μWs]: 4,638 (WCRT), 4,477 (AVE), 4,562 (MAX).

Damit egibt sich eine Verbesserung des durchschnittlichen Energieverbrauchs um 27,78 % (gegenüber 26,95 % in der festen Konfiguration), da eine Variation der durch die KEP-Konfiguration unterstützten Signale nur minimale Änderungen im Stromverbrauch bewirkt und der Basisstromverbrauch des FPGA sehr hoch ist.

Die höchste Laufzeit für den Logikblock von 18.85 ns wurde für das azyklische Token Ring Arbiter Programm mit zehn Stationen ermittelt. Die Periodendauer eines Systemtaktes ist die Zeit, in der das Berechnungsergebnis des Logikblocks spätestens vorliegen muss. Dies entspricht genau der Dauer zwischen dem Beginn eines neuen Ticks (zu dem alle Eingangssignale gelesen werden) und dem Beginn des ersten Instruktionszyklus. Die Dauer zwischen den einzelnen Instruktionszyklen ist größer/-gleich dieser Dauer. Die Periodendauer für eine Taktfrequenz von 40 MHz beträgt 25 ns und liegt damit 6.15 ns über der höchsten ermittelten Laufzeit. Das Ergebnis liegt also für alle getesteten Beispiele rechtzeitig vor. Der Token Ring Arbiter ist

ein bekanntes Beispiel für ein konstruktives ESTEREL Programm, das zyklische Signalabhängigkeiten enthält. Bei der Konvertierung solcher Programme in azyklische entstehen jedoch sehr tief verschachtelte Signalausdrücke; so ergibt sich für dieses Beispiel eine Schachtelung von 30 Gatterebenen. Auch wenn in der Praxis vergleichbare Anwendungen eher selten sind, kann die Laufzeit der Logik in einzelnen Fällen ein limitierender Faktor sein.

Aus den experimentellen Ergebnissen kann man ablesen, dass die Effektivität des Co-Design Ansatzes stark von dem Programm abhängt. Es konnte in allen Fällen eine Verkürzung der WCRT und eine Verringerung des Energieverbrauchs nachgewiesen werden.

5. Zusammenfassung

Der KEP ist sehr effizient in der Ausführung von ESTEREL Statements, da diese größtenteils direkt in KEP Assembler Instruktionen abgebildet werden können. Komplexe Expressions, die auf dem KEP in mehrere sequenzielle Instruktionen sequenzialisiert werden müssen, lassen sich sehr effizient in Form von Schaltnetzen in Hardware implementieren. Es wurde ein Ansatz vorgestellt, mit dem sich Expressions aus ESTEREL Programmen extrahieren lassen, ohne dabei die Semantik des Programms zu verändern. Dazu wird das Programm zunächst auf ESTEREL Code Basis in Software- und Hardwarebestandteile zerlegt. Das eigentliche Programm mit den durch Hilfssignale ersetzten Expressions bildet das neue Softwaremodul. Die Hilfssignale werden in einem bzw. mehreren Hardwaremodulen berechnet. Im zweiten Schritt wird aus den Hardwaremodulen eine kombinatorische Logik gebildet, die mit dem KEP verbunden wird. Das Softwaremodul wird nach KEP Assembler übersetzt und auf dem KEP mit Logikblock ausgeführt.

Arithmetische Ausdrücke können nicht effizient in der vorgestellten Weise in Hardware abgebildet werden. Sowohl für die Bereitstellung der Signalwerte, als auch für die Implementierung einzelner Berechnungen würde sich ein enormer Ressourcenverbrauch ergeben.

In den Experimenten konnte nachgewiesen werden, dass für Programme, die komplexe Ausdrücke enthalten, z.T. deutliche Verbesserungen der Ticklänge und des Energieverbrauchs erzielt werden können. Der zu erreichende Vorteil hängt maßgeblich davon ab, an welchen Stellen im Programm die komplexen Ausdrücke auftreten und ob durch ihre Beseitigung der längste Pfad bei der WCRT Analyse verkürzt werden kann. Ein Nachteil Co-Design Ansatzes ist die hohe Zahl von zusätzlich benötigten Hilfssignalen. Hier ist es denkbar, Hilfssignale, die in unterschiedlichen Instanzen auftreten, wiederzubenutzen. Eine andere Möglichkeit besteht darin, bevorzugt lange Ausdrücke oder günstige, gemeinsame Teilausdrücke ausfindig zu machen und nur diese in der Logik zu behandeln. Es ist auch denkbar, mit Hilfe der WCRT Analyse iterativ Ausdrücke in den Instanzen zu ersetzen, die die WCRT maßgeblich negativ beeinflussen.

Synthetisiert man KEP und Logik für die Ausführung auf einem FPGA, ergeben sich im Gegensatz zu den anderen Co-Design Ansätzen keine Nachteile in Bezug auf Flexibilität. Nutzt man inkrementelle Syntheseflüsse, so müssen nur geänderte Module neu synthetisiert werden. Nutzt man die Möglichkeit vieler FPGAs zur partiellen Rekonfiguration, lässt sich gezielt der Logikblock überschreiben. FPGAs werden jedoch eher im Entwicklungsprozess eingesetzt; in der Praxis wird der KEP in einer statischen Form z.B. als ASIC vorliegen. Um das Co-Design auch hier nutzen zu können, ist es denkbar, den KEP mit einer vorgeschalteten rekonfigurierbaren

Logik in Form eines PLA oder eines kleinen FPGA zu kombinieren. Eine andere Möglichkeit besteht darin, im KEP selbst eine rekonfigurierbare Logik einzubetten, die z.B. über bestimmte Leitungen oder Instruktionen konfigurierbar ist.

6. Literaturverzeichnis

[1] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. `http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf`.

[2] Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen M. Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciono Lavagno, Alberto Sangiovanni-Vincentelli, and Kei Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, April 1997.

[3] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Paserone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

[5] Gérard Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

[6] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999.

[7] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.

[8] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. In *Proceedings of the Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P07)*, Braga, Portugal, March 2007.

[10] R.K. Brayton, G.D. Hachtel, , and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. The Kluwer International Series in Engineering and Computer Science, Kluwer Academic Publishers, 1984.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[12] Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[13] C.M.Edmund Chow, Joyce S.Y.Tong, M.W.Sajeewa Dayaratne, Partha S Roop, and Zoran Salcic. RePIC - A New Processor Architecture Supporting Direct Esterel Execution. School of Engineering Report No. 612, University of Auckland, 2004.

[14] M. W. Sajeewa Dayaratne, Partha S. Roop, and Zoran Salcic. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, April 2005.

[15] Stephen A. Edwards. CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/~sedwards/cec/`.

[16] Stephen A. Edwards. Compiling Esterel into Sequential Code. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES 99)*, May 1999.

[17] Esterel Technologies. Esterel studio. `http://www.esterel-technologies.com/products/esterel-studio/overview.html`.

[18] Esterel.org. Esterel history. `http://www-sop.inria.fr/esterel.org/Html/History/History.htm`.

[19] Sascha Gädtke, Xin Li, Marian Boldt, and Reinhard von Hanxleden. HW/SW Co-Design for a Reactive Processor. In *Proceedings of the Student Poster Session at the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.

[20] Sascha Gädtke, Claus Traulsen, and Reinhard von Hanxleden. Hw/sw co-design for reactive processing. Submitted.

[21] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9), September 1991.

[22] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[23] Xin Li, Marian Boldt, and Reinhard von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, October 21–25 2006.

[24] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.

[25] Jan Lukoschus. *Removing Cycles in Esterel Programs*. PhD thesis, Christian-Albrechts-Universität Kiel, Department of Computer Science, July 2006. `http://e-diss.uni-kiel.de/tech-fak.html`.

[26] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. *Design Automaton Conference*, pages 52–57, 1990.

[27] Paul Molitor and Christoph Scholl. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. Teubner, Stuttgart, Leipzig, 1999.

[28] Markus Nilsson. A tool for automatic formal analysis of fault tolerance. Master's thesis, Linköping University, 2005.

[29] Dumitru Potop-Butucaru and Robert de Simone. *Optimization for faster execution of Esterel programs*, pages 285–315. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[30] Jürgen Reichardt and Bernd Schwarz. *VHDL-Synthese*. Oldenbourg Wissenschaftsverlag, München, 2003.

[31] Z. Salcic, P. S. Roop, M. Biglari-Abhari, and A. Bigdeli. REFLIX: A Processor Core with Native Support for Control Dominated Embedded Applications. *Elsevier Journal of Microprocessors and Microsystems*, 28:13–25, 2004.

[32] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, November 2001. ACM.

[33] Olivier Tardieu and Robert de Simone. Curing schizophrenia by program rewriting in Esterel. In *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign*, San Diego, CA, USA, 2004.

[34] University of California, Berkeley. *Berkeley Logic Interchange Format*, 2005. `http://www.cs.uic.edu/~jlillis/courses/cs594/spring05/blif.pdf`.

[35] Reinhard von Hanxleden. The Kiel Esterel Processor Homepage. `http://www.informatik.uni-kiel.de/rtsys/kep/`.

[36] Xilinx, Inc. Xilinx virtex-ii pro web power tool version 8.1.01. `http://www.xilinx.com/cgi-bin/power_tool/power_Virtex2p`.

[37] Xilinx, Inc. *Development System Reference Guide*, 2005.

[38] Xilinx, Inc. *Xilinx ISE 8 Software Manuals and Help - PDF Collection*, 2005.

A. Kommentierter Programmcode

A.1. HW/SW Co-Synthese

A.1.1. Partitionierung des Esterel Programms

Der erste Schritt der HW/SW Co-Synthese ist die Partitionierung des gegebenen Programms auf ESTEREL Codebasis in ein äquivalentes Programm bestehend aus einem Softwaremodul und mehreren Hardwaremodulen. Dabei werden aus dem Originalprogramm Signal- und Datenausdrücke extrahiert und durch ein Hilfssignal ersetzt. Die Hilfssignale werden in den Hardwaremodulen berechnet.

Das Programm *CoDesign-partitioner* erfüllt diese Aufgabe. Es erwartet zwei und einen dritten optionalen Parameter:

1. Den Dateinamen einer AST Datei, die das bereits expandierte *Esterel* Programm enthält
2. Einen Dateinamen für das partitionierte Programm
3. Einen Dateinamen für das partitionierte Programm mit minimierten Hardwaremodulen

Zusätzlich wird eine weitere Datei **_.dep* geschrieben. Diese gibt für alle während der Partitionierung erzeugten Hilfssignale an, von welchen Signalen der Ausdruck, den sie ersetzen, abhängt. Diese Datei wird vom Esterel2KASM Compiler benötigt, um während der Übersetzung des Softwaremoduls verloren gegangene Signalabhängigkeiten wiederherzustellen.

In den Dateien `CoDesignDataStructures.{cpp,hpp}` werden Datenstrukturen implementiert, die vom `CoDesignEsterelPrinter` benötigt werden. Die Klasse `ExtSymbolTable` erweitert den `AST::SymbolTable` um zwei nützliche Funktionen. Die Klasse `HWModuleTree` erlaubt es, alle Gültigkeitsbereiche von Signalen in einer Baumstruktur zu repräsentieren. Jeder Knoten enthält u.a. einen Pointer auf die `scope` Anweisung und das zugehörige Hardwaremodul. Die Klasse erlaubt den direkten Zugriff auf den aktuellen Gültigkeitsbereich und auf den hierarchisch niedrigsten Gültigkeitsbereich, in dem alle Symbole einer gegebenen Menge von Symbolen bekannt sind. Letzteres erlaubt es, ein Hilfssignal für einen möglichst großen Gültigkeitsbereich sichtbar zu machen. Dies kann sinnvoll sein, wenn das Programm mehrere äquivalente Ausdrücke enthält.

Die Klassen `CoDesignEsterelPreprocessor` und `CoDesignEsterelPrinter` implementieren die Klasse `AST::Visitor` und sind Modifikationen des `EsterelPrinter`s. Mit Hilfe der Klasse `CoDesignEsterelPreprocessor` wird das gegebene Programm

vorverarbeitet: Die Namen aller im Programm verwendeten Symbole werden in Großbuchstaben konvertiert und global eindeutig umbenannt. Alle verwendeten Symbole werden in einer Liste im Hilfsobjekt `CoDesignData` gespeichert. Mit Hilfe der Klasse `CoDesignEsterelPrinter` erfolgt die eigentliche Partitionierung.

A.1.2. CoDesign-partitioner.cpp

```
#include "IR.hpp"
#include "AST.hpp"
#include "EsterelPrinter.hpp"
#include "CoDesignEsterelPrinter.hpp"
#include "CoDesignEsterelPreprocessor.hpp"
#include "CoDesignDataStructures.hpp"
#include "CoDesignLogicMinimization.hpp"
#include <iostream>
#include <stdlib.h>
#include <fstream>

void usage() {
       std::cerr << "Usage:\n";
       std::cerr << "CoDesign-partitioning InputFile(*.exp)\n"
                 << "                      OutputFile(*_cosyn.strl)\n"
                 << "                      [minimized OutputFile(*_cosyn_min.strl)]\n";
}
using namespace CoDesign;


/* This program reads the AST of the original Esterel program from XML file and performs the
 * partitioning in sw- and hw-modules and writes the result as Esterel file. The information on
 * which signals the introduced auxiliary signals depend on is writtem to a '*_sw.dep' file.
 * This file is read by the Esterel2KEP Compiler in order to restore signal dependencies lost during
 * the partitioning.
 * Additionally an optional logic minimization step can be performed. The result is also written
 * to an Esterel file.
 *
 * @param name of the AST file
 * @param name of the partitioned Esterel program
 * @param name of the partitioned Esterel program with minimized hw modules
 *        (this parameter is optional)
 */
int main(int argc, char* argv[])
{
  try {
    if(3 <= argc <= 4) {
      std::string basefilename(argv[1]);
      std::ifstream infile(argv[1]);
      std::ofstream outfile(argv[2]);
      if(!infile) {std::cerr << "Can't open input file " << argv[1] << "\n";}
      if(!outfile) {std::cerr << "Can't open output file " << argv[2] << "\n";}
      IR::XMListream r(infile);
      IR::Node *n;
      r >> n;
      AST::ASTNode *an = dynamic_cast<AST::ASTNode*>(n);
      if (!an) throw IR::Error("Root node is not an AST node");

      CoDesignData* csd = new CoDesignData();
      CoDesignEsterelPreprocessor pp(csd);
      an->welcome(pp);

      CoDesign::CoDesignEsterelPrinter p(outfile,csd);
      an->welcome(p);
      outfile.flush();
      outfile.close();

      basefilename = basefilename.erase(basefilename.size()-4,basefilename.size());
      std::ofstream depfile((basefilename + "_sw.dep").c_str());
      if(!depfile) {std::cerr << "Can't open output file " << basefilename << "_sw.dep\n";}
      depfile << csd->signalDependencies.str();
      depfile.flush();
      depfile.close();

      if(argc == 4) {
        std::ofstream minoutfile(argv[3]);
        if(!minoutfile) {std::cerr << "Can't open file for minimized output " << argv[3] << "\n";}
        LogicMinimization::MVSIS mvsis;
        basefilename += "_cosyn";
        mvsis.minimize(csd,basefilename);
        AST::EsterelPrinter p(minoutfile);
        (csd->convModules).welcome(p);
        minoutfile.flush();
        minoutfile.close();
      }
    } else {
      std::cerr << "Wrong number of arguments!\n\n";
      usage();
    }
  }
  catch (IR::Error &e) {
    std::cerr << e.s << std::endl;
    exit(-1);
  }

  return 0;
}
```

A.1.3. CoDesignDataStructures.hpp

```
#ifndef _CODESIGNDATASTRUCTURES_HPP_
#define _CODESIGNDATASTRUCTURES_HPP_
#include "AST.hpp"
#include <iostream>
#include <sstream>
#include <vector>
#include <map>


using namespace AST;
using std::vector;
using std::map;
using std::string;

namespace CoDesign {

  /* The class ExtSymbolTable extends the class AST::SymbolTable by two new functions
   * for adding a complete SymbolTable and for expanding a string with a postfix if one
   * symbol in the table has the same name
   */
  class ExtSymbolTable : public AST::SymbolTable {
  public:
    void addSymbols(SymbolTable*);
    string getUniqueSigName(string);
  };


  // This class stores a (HW)Module and the local Signal-/Variable Scope it belongs to
  class HWModuleTreeNode {
  public:
    HWModuleTreeNode* parent;           // Pointer to the parent element
    vector<HWModuleTreeNode*> children; // Pointer to the child elements

    ScopeStatement* scopeStatement;     // Signal- or Variable-scope the module
                                        // belongs to
    Module* hwModule;                   // HWModule corresponding to the scope

    ParallelStatementList* preFunctions;

    bool hasIntersection(SymbolTable*);
  };

  /* This class is used for storing a tree of all local signal scopes used in the program
   * It gives easy access to the current scope that is currently processed by the partitioner
   * or to the lowest scope in that all signals of an expression are declared
   */
  class HWModuleTree {
  public:
    int treeSize;                           // number of nodes in the tree
    HWModuleTreeNode *currentNode;          // the last node entered or set by setCurrentNode
       ()
    std::vector<HWModuleTreeNode*> hwModules; // vector allows easier iteration through the
       tree

    HWModuleTree(){
      treeSize = 0;
      currentNode = NULL;
    }
    // add a child to current node and make it current
    HWModuleTreeNode* addChild(std::string, ScopeStatement*);

    // Get HWModule in the lowest possible scope-level in which all Symbols are declared
    HWModuleTreeNode* getLowestNode(SymbolTable*);

    // set the current scope level
    void setCurrentNode(HWModuleTreeNode *n) {
      currentNode=n;
    }

    // get the current scope level
    HWModuleTreeNode* getCurrentNode() {
      return currentNode;
    }

    // returns pointer to the root node
    HWModuleTreeNode* getRootNode() {
      return dynamic_cast<HWModuleTreeNode*>(*hwModules.begin());
    }

  };

  /* This data structure is used for storing global data during the CoSynthesis process
   *
   */
  class CoDesignData {
  public:
    Module* swModule;                 // pointer to the SW-Module
    HWModuleTree hwModuleTree;        // stores several HW-Modules, one for each scope
    std::string hwModuleNamePrefix;

    Modules convModules;              // Pointer to the converted AST
    ExtSymbolTable *usedSymbols;      // stores all symbols used in the esterel program
    ExtSymbolTable *usedPreSymbols;   // stores all auxiliary symbols for pre expressions

    std::stringstream signalDependencies; // stores signal dependencies for the introduced
       auxiliary
                                          // signals

    CoDesignData() {
      usedSymbols = new ExtSymbolTable();
      usedPreSymbols = new ExtSymbolTable();
    }
  };
}
#endif //_CODESIGNDATASTRUCTURES_HPP_
```

A.1.4. CoDesignDataStructures.cpp

```
#include"CoDesignDataStructures.hpp"

namespace CoDesign {

  // Adds symbols in the given SymbolTable 'source' to 'dest'
  void ExtSymbolTable::addSymbols(SymbolTable *source){
    if (source!=NULL) {
      for(unsigned int x=0;x<source->size();x++) {
        AST::Symbol *sym;
        sym = *(source->symbols.begin()+x);
        if (!this->local_contains(sym->name)) {
          this->enter(sym);
        }
      }
    }
  }

  // Tests if 'name' is unique in 'st'. In this case 'name' is returned,
  // otherwise a postfix, consisting of an underscore and a number, is appended to 'name'.
  string ExtSymbolTable::getUniqueSigName(string name){
    string postfix="";
    int x=0;
    while(this->local_contains(name+postfix)){
      std::ostringstream s;
      s << "_" << x;
      postfix = s.str();
      x++;
    }
    return name+postfix;
  }


  /* Returns true if 'st' and the set of Symbols in the scope declaration intersect
   */
  bool HWModuleTreeNode::hasIntersection(SymbolTable* st){
    SymbolTable* scopeSymbols = scopeStatement->symbols;
    vector<Symbol*>::iterator i = scopeSymbols->symbols.begin();
    for(;i!=scopeSymbols->symbols.end();i++) {
      vector<Symbol*>::iterator j = st->symbols.begin();
      for(;j!=st->symbols.end();j++) {
        if(((Symbol*)(*i))->name.compare(((Symbol*)(*j))->name)==0) return true;
      }
    }
    return false;
  }

  /* Adds a new child to the tree
   * @param namePrefix Prefix for the hw-module name
   * @param scope A pointer to the local signal declaration
   * @return Returns a pointer to the new child node
   */
  HWModuleTreeNode* HWModuleTree::addChild(std::string namePrefix, ScopeStatement* scope){
    HWModuleTreeNode* newNode = new HWModuleTreeNode();
    std::ostringstream s;
    treeSize++;
    s << treeSize;
    newNode->hwModule = new Module(new ModuleSymbol(namePrefix+"_hw_"+s.str()));
    newNode->hwModule->constants = new SymbolTable();
    newNode->hwModule->functions = new SymbolTable();
    newNode->hwModule->procedures = new SymbolTable();
    newNode->hwModule->tasks = new SymbolTable();
    newNode->hwModule->types = new SymbolTable();
    newNode->hwModule->variables = new SymbolTable();
    newNode->hwModule->signals = new ExtSymbolTable();
    newNode->hwModule->body = new ParallelStatementList();
    newNode->preFunctions = new ParallelStatementList();
    newNode->scopeStatement = scope;
    newNode->parent = currentNode;
    if(hwModules.size()==0) {
      currentNode = newNode;
    }
    else {
      currentNode->children.push_back(newNode);
      currentNode = newNode;
    }
    hwModules.push_back(newNode);
    return newNode;
  }

  /* Get HWModuleTreeNode in the lowest possible level in which all Symbols are declared
   * (This is the Node in the lowest order scope, where the set of used Symbols
   * in the expression and the set of declared symbols in the scope intersect.)
   */
  HWModuleTreeNode* HWModuleTree::getLowestNode(SymbolTable* st){
    HWModuleTreeNode* node = currentNode;
    while((node != this->getRootNode()) && (!node->hasIntersection(st)))
      node = node->parent;
    return node;
  }

}
```

A.1.5. CoDesignEsterelPreprocessor.hpp

```
#ifndef CODESIGNESTERELPREPROCESSOR_HPP_
#define CODESIGNESTERELPREPROCESSOR_HPP_

#include "AST.hpp"
#include <iostream>
#include <sstream>
#include <vector>
#include <map>
#include <cassert>
#include <algorithm>
#include "CoDesignDataStructures.hpp"

using namespace AST;
using std::vector;
using std::map;
using std::string;

/* This class implements a vistor for the AST, that is used to preprocess the Esterel program
      before
 * the CoSynthesis. While traversing through the AST it does the following:
 * - make all symbol names upper case and unique
 * - store all found symbols in the CoDesignData datastructure
 */
namespace CoDesign {
  class CoDesignEsterelPreprocessor : public AST::Visitor {
    unsigned int indentlevel;

    std::vector<int> precedence;
    std::map<string, int> level;
    std::map<string, int> unarylevel;

                CoDesignData* cosyndata;
  public:

    CoDesignEsterelPreprocessor(CoDesignData*);
    virtual ~CoDesignEsterelPreprocessor() {
    }

    using Visitor::visit; // Bring the Visitor's visit method into scope

    void process(ASTNode* n) { assert(n); n->welcome(*this); }
    void statement(Statement *);
    void expression(Expression *);
    void sigexpression(Expression *);

    static const int sequentialPrecedence = 2;
    static const int parallelPrecedence = 1;

    bool push_precedence(int);
    void pop_precedence();

    Status visit(ModuleSymbol&);
    Status visit(VariableSymbol&);
    Status visit(BuiltinConstantSymbol&);
    Status visit(BuiltinSignalSymbol&);
    Status visit(BuiltinTypeSymbol&);
    Status visit(BuiltinFunctionSymbol&);
    Status visit(TypeRenaming &);
    Status visit(ConstantRenaming &);
    Status visit(FunctionRenaming &);
    Status visit(ProcedureRenaming &);
    Status visit(SignalRenaming &);
    Status visit(PredicatedStatement &);
    Status visit(TaskCall &);

    Status visit(Module&);
    Status visit(Exclusion&);
    Status visit(Implication&);
    Status visit(Modules&);
    Status visit(SymbolTable&);

    Status visit(TypeSymbol&);
    Status visit(ConstantSymbol&);
    Status visit(SignalSymbol&);
    Status visit(FunctionSymbol&);
    Status visit(ProcedureSymbol&);
    Status visit(TaskSymbol&);

    Status visit(StatementList&);
    Status visit(ParallelStatementList&);

    Status visit(Nothing&);
    Status visit(Pause&);
    Status visit(Halt&);
    Status visit(Emit&);
    Status visit(Sustain&);
    Status visit(Assign&);
    Status visit(StartCounter&);
    Status visit(ProcedureCall&);
    Status visit(Exec&);
    Status visit(Present&);
    Status visit(If&);
    Status visit(Loop&);
    Status visit(Repeat&);
    Status visit(Abort&);
    Status visit(Await&);
    Status visit(LoopEach&);
    Status visit(Every&);
    Status visit(Suspend&);
    Status visit(DoWatching&);
    Status visit(DoUpto&);
    Status visit(Trap&);
    Status visit(Exit&);
    Status visit(Var&);
```

```
    Status visit(Signal&);
    Status visit(Run&);

    Status visit(UnaryOp&);
    Status visit(BinaryOp&);
    Status visit(LoadVariableExpression&);
    Status visit(LoadSignalExpression&);
    Status visit(LoadSignalValueExpression&);
    Status visit(Literal&);
    Status visit(FunctionCall&);
    Status visit(Delay&);

    Status visit(CheckCounter&);

    Status visit(IfThenElse&);

    void indent() { indentlevel += 2; }
    void unindent() { indentlevel -= 2; }
  };
}
#endif /*CODESIGNESTERELPREPROCESSOR_HPP_*/
```

A.1.6. CoDesignEsterelPreprocessor.cpp

```
#include "CoDesignEsterelPreprocessor.hpp"
#include <cassert>

namespace CoDesign {
  CoDesignEsterelPreprocessor::CoDesignEsterelPreprocessor(CoDesignData* csd) :
    indentlevel(0), cosyndata(csd) {

    precedence.push_back(0);

    level["or"] = 1;
    level["and"] = 2;
    unarylevel["not"] = 3;
    unarylevel["pre"] = 3;

    level["="] = 4;
    level["<>"] = 4;
    level["<"] = 4;
    level[">"] = 4;
    level["<="] = 4;
    level[">="] = 4;

    level["+"] = 5;
    level["-"] = 6;

    level["*"] = 7;
    level["/"] = 8;
    level["mod"] = 8;

    unarylevel["-"] = 9;
  }

  void CoDesignEsterelPreprocessor::statement(Statement *s) {
    assert(s);
    s->welcome(*this);
  }

  void CoDesignEsterelPreprocessor::expression(Expression *e) {
    precedence.push_back(0);
    process(e);
    precedence.pop_back();
  }

  void CoDesignEsterelPreprocessor::sigexpression(Expression *e) {
    precedence.push_back(0);
    process(e);
    precedence.pop_back();
  }

  bool CoDesignEsterelPreprocessor::push_precedence(int p) {
    bool needBrackets = (p < precedence.back()) || ((p == precedence.back()) && ((p==level["-"
      ]) || (p==level["/"])));
    precedence.push_back(p);
    return needBrackets;
  }

  void CoDesignEsterelPreprocessor::pop_precedence() {
    precedence.pop_back();
  }

  Status CoDesignEsterelPreprocessor::visit(ModuleSymbol &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(VariableSymbol &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(TypeRenaming &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(ConstantRenaming &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(FunctionRenaming &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(ProcedureRenaming &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(SignalRenaming &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(PredicatedStatement &) { assert(0); }
  Status CoDesignEsterelPreprocessor::visit(TaskCall &) { assert(0); }

  Status CoDesignEsterelPreprocessor::visit(BuiltinConstantSymbol &) { return Status(); }
  Status CoDesignEsterelPreprocessor::visit(BuiltinSignalSymbol &) { return Status(); }
  Status CoDesignEsterelPreprocessor::visit(BuiltinTypeSymbol &) { return Status(); }
  Status CoDesignEsterelPreprocessor::visit(BuiltinFunctionSymbol &) { return Status(); }

  Status CoDesignEsterelPreprocessor::visit(Module &m) {
    assert(m.symbol);
    // ackquire all used global symbols in 'usedSymbols'
    ExtSymbolTable usedSymbols;
    cosyndata->usedSymbols->addSymbols(m.types);
    usedSymbols.addSymbols(m.constants);
    cosyndata->usedSymbols->addSymbols(m.functions);
    cosyndata->usedSymbols->addSymbols(m.procedures);
    cosyndata->usedSymbols->addSymbols(m.tasks);
    usedSymbols.addSymbols(m.signals);
    usedSymbols.addSymbols(m.variables);
    // make symbol names upper case and unique and write them
    // to cosyndata->usedSymbols
    for(unsigned int x=0;x<usedSymbols.size();x++) {
      AST::Symbol *sym;
      std::string symName;
      sym = *(usedSymbols.symbols.begin()+x);
      symName = sym->name;
      std::cerr << symName << "\n";
      transform (symName.begin(), symName.end(),    // source
                 symName.begin(),                   // destination
                 toupper);                          // operation
      symName = cosyndata->usedSymbols->getUniqueSigName(symName);
      if(!((sym->name == "tick") ||(sym->name == "true") ||(sym->name == "false") )) sym
       ->name = symName;
      cosyndata->usedSymbols->enter(sym);
    }
    process(m.body);
    return Status();
  }
```

```
Status CoDesignEsterelPreprocessor::visit(Modules &m) {
  vector<Module*>::iterator i = m.modules.begin();
  while ( i != m.modules.end() ) {
    assert(*i);
    assert((*i)->symbol);
    process(*i);
    i++;
  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Exclusion& e) {
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Implication& e) {
  assert(e.predicate);
  assert(e.implication);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(SymbolTable &t) {
  for ( SymbolTable::const_iterator i = t.begin() ; i != t.end() ; i++ ) {
    process(*i);
  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(TypeSymbol &s) {
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(ConstantSymbol &s) {
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(SignalSymbol &s) {
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(FunctionSymbol &s) {
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(ProcedureSymbol &s) {
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(TaskSymbol &s) {
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(StatementList &l) {
  push_precedence(sequentialPrecedence);
  vector<Statement*>::const_iterator i = l.statements.begin();
  while ( i != l.statements.end() ) {
    process(*i);
    i++;
  }
  pop_precedence();
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(ParallelStatementList &l) {
  bool needBrackets = push_precedence(parallelPrecedence);

  vector<Statement*>::const_iterator i = l.threads.begin();
  while ( i != l.threads.end() ) {
    process(*i);
    i++;
  }

  pop_precedence();
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Nothing &) { return Status(); }
Status CoDesignEsterelPreprocessor::visit(Pause &) { return Status(); }
Status CoDesignEsterelPreprocessor::visit(Halt &) { return Status(); }

Status CoDesignEsterelPreprocessor::visit(Emit &e) {
  assert(e.signal);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(StartCounter &s) {
  assert(s.counter);
  expression(s.count);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Sustain &e) {
  assert(e.signal);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Assign &a) {
  assert(a.variable);
  assert(a.value);

  return Status();
}

Status CoDesignEsterelPreprocessor::visit(ProcedureCall &c) {
  assert(c.procedure);

  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Exec &e) {
```

```
  for ( vector<TaskCall*>::const_iterator i = e.calls.begin() ;
        i != e.calls.end() ; i++) {
    assert(*i);
    assert((*i)->procedure);
    assert((*i)->signal);
    if ((*i)->body) {
      process((*i)->body);
    }
  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Present &p) {
  if (p.cases.size() == 1) {
    // Simple if-then-else
    PredicatedStatement *ps = p.cases[0];
    assert(ps);
    sigexpression(ps->predicate);
    if (ps->body) {
      statement(ps->body);
    }
  } else {
    // Multiple cases
    for (vector<PredicatedStatement*>::const_iterator i = p.cases.begin() ;
         i != p.cases.end() ; i++) {
      assert(*i);
      assert((*i)->predicate);
      sigexpression((*i)->predicate);
      if ((*i)->body) {
        process((*i)->body);
      }
    }
  }
  if (p.default_stmt) {
    statement(p.default_stmt);
  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(If &s) {
  vector<PredicatedStatement*>::const_iterator i = s.cases.begin();
  assert(i != s.cases.end());
  expression((*i)->predicate);
  if ((*i)->body) {
    statement((*i)->body);
  }
  for ( i++ ; i != s.cases.end() ; i++ ) {
    assert(*i);
    expression((*i)->predicate);
    statement((*i)->body);
  }
  if (s.default_stmt) {
    statement(s.default_stmt);
  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Loop &l) {
  statement(l.body);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Repeat &r) {
  expression(r.count);
  statement(r.body);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Abort &a) {
  statement(a.body);
  if (a.cases.size() == 1) {
    // Simple abort condition
    PredicatedStatement *ps = a.cases[0];
    assert(ps);
    sigexpression(ps->predicate);
    if (ps->body) {
      statement(ps->body);
    }
  } else {
    // Abort cases

    for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
          i != a.cases.end() ; i++ ) {
      assert(*i);
      sigexpression((*i)->predicate);
      if ((*i)->body) {

        process((*i)->body);

      }
    }

  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Await& a) {
  if (a.cases.size() == 1) {
    // Simple abort condition
    PredicatedStatement *ps = a.cases[0];
    assert(ps);
    sigexpression(ps->predicate);
    if (ps->body) {
      statement(ps->body);
    }
  } else {

    for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
          i != a.cases.end() ; i++ ) {
      assert(*i);
      sigexpression((*i)->predicate);
      if ((*i)->body) {
```

```
        process((*i)->body);
      }
    }

  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(LoopEach &l) {
  statement(l.body);
  sigexpression(l.predicate);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Every &e) {
  sigexpression(e.predicate);
  statement(e.body);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Suspend &s) {
  statement(s.body);
  sigexpression(s.predicate);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(DoWatching &d) {
  statement(d.body);
  sigexpression(d.predicate);
  if (d.timeout) {
    statement(d.timeout);
  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(DoUpto &d) {
  statement(d.body);
  sigexpression(d.predicate);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Trap &t) {
  assert(t.symbols);
  // make symbol names upper case and unique and write them
  // to cosyndata->usedSymbols
  for(unsigned int x=0;x<t.symbols->size();x++) {
    AST::Symbol *sym;
    std::string symName;
    sym = *(t.symbols->symbols.begin()+x);
    symName = sym->name;
    transform (symName.begin(), symName.end(),    // source
               symName.begin(),                   // destination
               toupper);                          // operation
    symName = cosyndata->usedSymbols->getUniqueSigName(symName);
    sym->name = symName;
    cosyndata->usedSymbols->enter(sym);
  }

  statement(t.body);
  for (vector<PredicatedStatement*>::const_iterator i = t.handlers.begin() ;
       i != t.handlers.end() ; i++ ) {
    assert(*i);
    expression((*i)->predicate);
    statement((*i)->body);
  }

  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Exit &e) {
  assert(e.trap);
  assert(e.trap->kind == SignalSymbol::Trap);

  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Var &v) {
  // make symbol names upper case and unique and write them
  // to cosyndata->usedSymbols
  for(unsigned int x=0;x<v.symbols->size();x++) {
    AST::Symbol *sym;
    std::string symName;
    sym = *(v.symbols->symbols.begin()+x);
    symName = sym->name;
    transform (symName.begin(), symName.end(),    // source
               symName.begin(),                   // destination
               toupper);                          // operation
    symName = cosyndata->usedSymbols->getUniqueSigName(symName);
    sym->name = symName;
    cosyndata->usedSymbols->enter(sym);
  }

  statement(v.body);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Signal &s) {
  // make symbol names upper case and unique and write them
  // to cosyndata->usedSymbols
  for(unsigned int x=0;x<s.symbols->size();x++) {
    AST::Symbol *sym;
    std::string symName;
    sym = *(s.symbols->symbols.begin()+x);
    symName = sym->name;
    transform (symName.begin(), symName.end(),    // source
               symName.begin(),                   // destination
               toupper);                          // operation
    symName = cosyndata->usedSymbols->getUniqueSigName(symName);
    sym->name = symName;
    cosyndata->usedSymbols->enter(sym);
```

```
  }

  statement(s.body);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Run &r) {
      /* The run statement should not appear in the program anymore because
       * it is beeing expanded before preprocessing.
       */
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(UnaryOp &u) {
  assert(unarylevel.find(u.op) != unarylevel.end());
  process(u.source);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(BinaryOp &b) {
  assert(level.find(b.op) != level.end());
  process(b.source1);
  process(b.source2);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(LoadVariableExpression &e) {
  assert(e.variable);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(LoadSignalExpression &e) {
  assert(e.signal);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(LoadSignalValueExpression &e) {
  assert(e.signal);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Literal &l) {
  assert(l.type);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(FunctionCall &e) {
  assert(e.callee);
  vector<Expression*>::const_iterator i = e.arguments.begin();
  while (i != e.arguments.end()) {
    expression(*i);
    i++;
  }
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(Delay &e) {
  if (e.is_immediate) {
    assert(e.count == NULL);
  } else {
    expression(e.count);
  }
  sigexpression(e.predicate);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(CheckCounter &e) {
  assert(e.counter);
  expression(e.predicate);
  return Status();
}

Status CoDesignEsterelPreprocessor::visit(IfThenElse &s) {
  expression(s.predicate);
  if (s.then_part) {
    statement(s.then_part);
  }
  if (s.else_part) {
    statement(s.else_part);
  }
  return Status();
}

}
```

A.1.7. CoDesignEsterelPrinter.hpp

```
//#line 27 "EsterelPrinter.nw"
#ifndef _CODESIGN_ESTEREL_PRINTER_HPP
#define _CODESIGN_ESTEREL_PRINTER_HPP

#include "AST.hpp"
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include "CoDesignEsterelPreprocessor.hpp"
#include "CoDesignDataStructures.hpp"
#include <algorithm>

using namespace AST;

namespace CoDesign {
  class CoDesignEsterelPrinter : public Visitor {
    std::ostream &o;
    unsigned int indentlevel;

    std::vector<int> precedence;
    std::map<string, int> level;
    std::map<string, int> unarylevel;

    CoDesignData* cosyndata;       //
    Statement** currentNode;    // Pointer to the current statement in the AST
  public:
    /*
     * @param ostream Output stream for pretty printing the converted program
     * @param CoDesignData
     */
    CoDesignEsterelPrinter(std::ostream &, CoDesignData*);
    virtual ~CoDesignEsterelPrinter() {}

    using Visitor::visit; // Bring the Visitor's visit method into scope

    // Member functions are commented in the .cpp file
    void process(ASTNode* n) { assert(n); n->welcome(*this); }
    void statement(Statement *);
    void getUsedSymbols(AST::Expression*, SymbolTable&);
    Expression* getDelayExp(Expression*);
    StatementList* decomposeExpression(Expression*,Expression**);
    void expression(Expression *,Expression **);
    void sigexpression(Expression *,Expression **);
    string getCompoundSigName(Expression *);
    string getUniqueSigName(string,SymbolTable*);

    Expression* applyPre(Expression*,bool);
    void substPre(Expression*, Expression**);

    static const int sequentialPrecedence = 2;
    static const int parallelPrecedence = 1;

    bool push_precedence(int);
    void pop_precedence();

    Status visit(ModuleSymbol&);
    Status visit(VariableSymbol&);
    Status visit(BuiltinConstantSymbol&);
    Status visit(BuiltinSignalSymbol&);
    Status visit(BuiltinTypeSymbol&);
    Status visit(BuiltinFunctionSymbol&);
    Status visit(TypeRenaming &);
    Status visit(ConstantRenaming &);
    Status visit(FunctionRenaming &);
    Status visit(ProcedureRenaming &);
    Status visit(SignalRenaming &);
    Status visit(PredicatedStatement &);
    Status visit(TaskCall &);

    Status visit(Module&);
    Status visit(Exclusion&);
    Status visit(Implication&);
    Status visit(Modules&);
    Status visit(SymbolTable&);

    Status visit(TypeSymbol&);
    Status visit(ConstantSymbol&);
    Status visit(SignalSymbol&);
    Status visit(FunctionSymbol&);
    Status visit(ProcedureSymbol&);
    Status visit(TaskSymbol&);

    Status visit(StatementList&);
    Status visit(ParallelStatementList&);

    Status visit(Nothing&);
    Status visit(Pause&);
    Status visit(Halt&);
    Status visit(Emit&);
    Status visit(Sustain&);
    Status visit(Assign&);
    Status visit(StartCounter&);
    Status visit(ProcedureCall&);
    Status visit(Exec&);
    Status visit(Present&);
    Status visit(If&);
    Status visit(Loop&);
    Status visit(Repeat&);
    Status visit(Abort&);
    Status visit(Await&);
    Status visit(LoopEach&);
    Status visit(Every&);
    Status visit(Suspend&);
```

```
    Status visit(DoWatching&);
    Status visit(DoUpto&);
    Status visit(Trap&);
    Status visit(Exit&);
    Status visit(Var&);
    Status visit(Signal&);
    Status visit(Run&);

    Status visit(UnaryOp&);
    Status visit(BinaryOp&);
    Status visit(LoadVariableExpression&);
    Status visit(LoadSignalExpression&);
    Status visit(LoadSignalValueExpression&);
    Status visit(Literal&);

    Status visit(FunctionCall&);
    Status visit(Delay&);
    Status visit(CheckCounter&);

    Status visit(IfThenElse&);

    void indent() { indentlevel += 2; }
    void unindent() { indentlevel -= 2; }
    void tab() { for (unsigned int i = 0 ; i < indentlevel ; i++) o << ' '; }
  };
}
#endif
```

A.1.8. CoDesignEsterelPrinter.cpp

```
#include "CoDesignEsterelPrinter.hpp"
#include "EsterelPrinter.hpp"
#include <cassert>

namespace CoDesign {

  CoDesignEsterelPrinter::CoDesignEsterelPrinter(std::ostream &oo,CoDesignData* csd) : o(oo),
      indentlevel(0),
                                                                                       cosyndata(csd) {
    precedence.push_back(0);
    cosyndata->convModules=Modules();

    level["or"] = 1;
    level["and"] = 2;
    unarylevel["not"] = 3;
    unarylevel["pre"] = 3;
    level["="] = 4;
    level["<>"] = 4;
    level["<"] = 4;
    level[">"] = 4;
    level["<="] = 4;
    level[">="] = 4;

    level["+"] = 5;
    level["-"] = 6;

    level["*"] = 7;
    level["/"] = 8;
    level["mod"] = 8;

    unarylevel["-"] = 9;
  }

  void CoDesignEsterelPrinter::statement(Statement *s) {
    assert(s);
    s->welcome(*this);
  }

  /* Gets all used SignalSymbols in Expression 'e' and stores them in 'usedSymbols'
   * by recursing over the Expression 'e'
   */
  void CoDesignEsterelPrinter::getUsedSymbols(AST::Expression* e, SymbolTable &usedSymbols){
    if (dynamic_cast<AST::LoadSignalExpression*>(e)) {
      AST::SignalSymbol* Sig = dynamic_cast<AST::LoadSignalExpression*>(e)->signal;
      if (!usedSymbols.local_contains(Sig->name)) {
        usedSymbols.enter(Sig);
      }
    }
    if (dynamic_cast<AST::LoadSignalValueExpression*>(e)) {
      AST::SignalSymbol* Sig = dynamic_cast<AST::LoadSignalValueExpression*>(e)->signal;
      if (!usedSymbols.local_contains(Sig->name)) {
        usedSymbols.enter(Sig);
      }
    }
    if (dynamic_cast<AST::LoadVariableExpression*>(e)) {
      AST::VariableSymbol* Var = dynamic_cast<AST::LoadVariableExpression*>(e)->variable;
      if (!usedSymbols.local_contains(Var->name)) {
        usedSymbols.enter(Var);
      }
    }
    if (dynamic_cast<AST::BinaryOp*>(e)) {
      getUsedSymbols(dynamic_cast<AST::BinaryOp*>(e)->source1, usedSymbols);
      getUsedSymbols(dynamic_cast<AST::BinaryOp*>(e)->source2, usedSymbols);
    }
    if (dynamic_cast<AST::UnaryOp*>(e)) {
      getUsedSymbols(dynamic_cast<AST::UnaryOp*>(e)->source, usedSymbols);
    }
  }

  /* returns a disjunction term of all SignalSymbols in 'e'
   */
  Expression* CoDesignEsterelPrinter::getDelayExp(Expression *e){
    Expression* delay=
      new LoadSignalExpression(NULL,
                               new SignalSymbol("tick",NULL,AST::SignalSymbol::Input,NULL,
      NULL));
    SymbolTable signalTable;
    getUsedSymbols(e,signalTable);
    for(unsigned int i = 0; i < signalTable.size(); i++) {
      if(i==0) delay = new LoadSignalExpression(NULL,(SignalSymbol*)signalTable.symbols[i]);
      else delay = new BinaryOp(NULL,
                                "or",
                                delay,
                                new LoadSignalExpression(NULL,(SignalSymbol*)signalTable.
      symbols[i])
                                );
    }
    return delay;
  }

  /* Decomposes the given expression e into boolean and non boolean parts:
   * Boolean variables, function calls with boolean return values and subexpressions that map
      from
   * (non-boolean,non-boolean) to boolean are replaced by boolean auxiliary signals. A list of
      emit
   * statements that emit these auxiliary signals with the according value is returned
   *
   * This is achieved by recursively walking through the expression
   *
   * @param e Pointer to the expression to be decomposed; the expression must evaluate to
      boolean!
   * @param pe Double Pointer used to modify the expression
   * @return A list of emit statements
   */
```

```
StatementList* CoDesignEsterelPrinter::decomposeExpression(Expression *e,Expression **pe) {
  StatementList* sl = new StatementList();              // stores assignments of auxiliary
    signals
  // if the root node is a binary operator
  if (dynamic_cast<AST::BinaryOp*>(e)) {
    // if the binary operator is a boolean operator {and, or}
    if(level[dynamic_cast<BinaryOp*>(e)->op] < level["="]){
      // apply this function recursively to the two operands ..
      StatementList* sl1;
      StatementList* sl2;
      sl1 = decomposeExpression(dynamic_cast<AST::BinaryOp*>(e)->source1,
                                &dynamic_cast<AST::BinaryOp*>(e)->source1);
      sl2 = decomposeExpression(dynamic_cast<AST::BinaryOp*>(e)->source2,
                                &dynamic_cast<AST::BinaryOp*>(e)->source2);
      // .. and store the found auxiliary signal assignments
      if(sl1->statements.size()>0) {
        *sl << sl1;
      }
      if(sl2->statements.size()>0) {
        *sl << sl2;
      }
    }
    // if the binary operator is comparision operator
    else {
      if(level[dynamic_cast<BinaryOp*>(e)->op] == level["="]) {
        // introduce new auxiliary signal
        SignalSymbol* s =
          new SignalSymbol(cosyndata->usedSymbols->getUniqueSigName(getCompoundSigName(e)),
                           new TypeSymbol("boolean"),SignalSymbol::Local,NULL,NULL);
        // declare auxiliary signal in the current local signal scope
        cosyndata->hwModuleTree.getCurrentNode()->scopeStatement->symbols->enter(s);
        cosyndata->usedSymbols->enter(s);
        // emit the signal just before the expression
        *sl << new Emit(s,e);
        // substitute the subexpression with the new signal
        *pe = new LoadSignalValueExpression(s);
      }
    }
  }
  // if the root node is an unary operator
  else if (dynamic_cast<AST::UnaryOp*>(e)) {
    // if unary operator is out of {pre, not}
    if(unarylevel[dynamic_cast<UnaryOp*>(e)->op] < unarylevel["-"]) {
      // apply this function recursively to the operand ..
      StatementList* sl1;
      sl1 = decomposeExpression(dynamic_cast<AST::UnaryOp*>(e)->source,
                                &dynamic_cast<AST::UnaryOp*>(e)->source);
      // and store the found auxiliary signal assignments
      if(sl1->statements.size()>0) {
        *sl << sl1;
      }
    }
    // if the unary operator is minus (this shouldn't occur)
    else {
      assert(0);  // minus operator not allowed!
    }
  }
  // if e is a function call (actually not supported by the KEP)
  else if (dynamic_cast<AST::FunctionCall*>(e)) {
    SignalSymbol* s
      = new SignalSymbol(cosyndata->usedSymbols
                         ->getUniqueSigName(dynamic_cast<AST::FunctionCall*>(e)->callee->
    name),
                       new TypeSymbol("boolean"),SignalSymbol::Local,NULL,NULL);
    cosyndata->hwModuleTree.getCurrentNode()->scopeStatement->symbols->enter(s);
    cosyndata->usedSymbols->enter(s);
    *sl << new Emit(s,e);
    *pe = new LoadSignalValueExpression(s);
  }
  // if e is a variable (or constant)
  else if (dynamic_cast<AST::LoadVariableExpression*>(e)) {
    VariableSymbol* var = dynamic_cast<AST::LoadVariableExpression*>(e)->variable;
    if (!dynamic_cast<AST::ConstantSymbol*>(var)) {
      // enter new boolean auxiliary signal in the current signal scope
      SignalSymbol* s =
        new SignalSymbol(cosyndata->usedSymbols->getUniqueSigName(getCompoundSigName(e)),
                         new TypeSymbol("boolean"),SignalSymbol::Local,NULL,NULL);
      cosyndata->hwModuleTree.getCurrentNode()->scopeStatement->symbols->enter(s);
      cosyndata->usedSymbols->enter(s);
      // emit this new signal with the variable value
      *sl << new Emit(s,e);
      // substitute the variable with the new signal
      *pe = new LoadSignalValueExpression(s);
    }
  }
  return sl;
}

/* Handles an expression:
 * - terminates if expression is non-boolean
 * - substitutes all occurrences of pre(?S) with an auxiliary signal preV_S and declares and
 *   calculates it within the signal scope of S
 * - substitutes all comparision expressions and boolean variables in e with auxiliary
    signals
 *   a_x and emit them with the according value immediately before the expression
 * - substitute the corresponding expression e with an auxiliary signal s
 * - declare the auxiliary signals a_x and s in the lowest signal scope, in which all signals
 *   are known. Declare all signals in e as inputs and s as output to the hardware module
 * - corresponding to the scope.
 * - Sustain s with the value of e within a new thread in the HW-Module and initialize all
    non
 *   initialized signals out of e with false
*/
void CoDesignEsterelPrinter::expression(Expression *e, Expression **pe) {
  process(e);
  if (dynamic_cast<UnaryOp*>(e) || dynamic_cast<BinaryOp*>(e)){
    // get new signal name for the evaluated expression
    StatementList *sl;
    bool isBool = false;
    // consider only complex expressions for decomposition
    if (dynamic_cast<UnaryOp*>(e)) {
      // Only boolean (sub-)expressions can be calculated in HW
```

```
  if(unarylevel[dynamic_cast<UnaryOp*>(e)->op] < unarylevel["-"])
    sl = decomposeExpression(e,pe);
  isBool = true;
}
else if(dynamic_cast<BinaryOp*>(e)) {
  // Only boolean (sub-)expressions can be calculated in HW
  if(level[dynamic_cast<BinaryOp*>(e)->op] < level["=="])
    sl = decomposeExpression(e,pe);
  isBool=true;
}
if(isBool) {
  if(sl->statements.size()>0) {
    Sustain* sust = dynamic_cast<Sustain*>(*currentNode);
    if(sust != NULL) {
      *sl << new Emit(sust->signal,e);
      *currentNode = new Loop(sl);
    }
    else {
      *sl << *currentNode;
      *currentNode = (Statement*) sl;
    }
  }
  // substitute all pre operators in e
  substPre(*pe,pe);
  e=*pe;
  // get new signal name for the evaluated expression
  string compSigName = cosyndata->usedSymbols->getUniqueSigName(getCompoundSigName(e));
  // store all symbols used in e in the symbol table st
  SymbolTable *st = new SymbolTable();
  getUsedSymbols(e,*st);
  // find corresponding scope
  HWModuleTreeNode* hwm = cosyndata->hwModuleTree.getLowestNode(st);
  Module* hwModule = hwm->hwModule;
  // diterate over all symbols in the expression
  std::vector<Symbol*>::iterator i = st->symbols.begin();
  while (i != st->symbols.end()) {
    // add constant declaration to hw-module
    if (ConstantSymbol* c = dynamic_cast<ConstantSymbol*>(*i)) {
      if(!hwModule->constants->local_contains(c->name)) {
        hwModule->constants->enter(c);
      }
    }
    if (SignalSymbol* s = dynamic_cast<SignalSymbol*>(*i)) {
      // add initializer to non initialized signals
      if(!s->initializer) {
        s->initializer = new LoadVariableExpression(
                           new ConstantSymbol(
                                          "false",
                                          new TypeSymbol("boolean"),
                                          new Literal("0",new TypeSymbol("boolean
"))
                                          )
                           );
      }
      // add input signal declaration to hw-module
      if(!hwModule->signals->local_contains(s->name)) {
        std::cerr << s->name << "␣" << s->type->name << "␣" << s->kind << "\n";
        hwModule->signals->enter(
                                 new SignalSymbol(s->name,
                                                  s->type,
                                                  (s->kind==SignalSymbol::Sensor)?(
SignalSymbol::Sensor):(SignalSymbol::Input),
                                                  s->combine,
                                                  s->initializer)
                                 );
      }

      // store signal dependenies
      cosyndata->signalDependencies << compSigName << "␣" << s->name << "\n";
    }
    i++;
  }
  // declare the auxiliary signal that substitutes e as output to the hw-module ...
  SignalSymbol* compHWSig =
    new SignalSymbol(compSigName,
                     new TypeSymbol("boolean"),
                     AST::SignalSymbol::Output,
                     NULL,
                     NULL
                     );
  compHWSig->initializer =
    new LoadVariableExpression(
                               new ConstantSymbol(
                                                  "false",
                                                  new TypeSymbol("boolean"),
                                                  new Literal("0",new TypeSymbol("
boolean"))
                                                  )
                               );
  (*hwModule->signals).enter(compHWSig);
  cosyndata->usedSymbols->enter(compHWSig);
  // ... and sustain it in a new thread within the hw-module
  Sustain* sustain = new Sustain(compHWSig,e);
  dynamic_cast<ParallelStatementList*>(hwModule->body)
    ->threads.push_back(sustain);

  // Substitute the SignalExpression in the software module by
  // the newly generated signal
  SignalSymbol* compSWSig =
    new SignalSymbol(compSigName,
                     new TypeSymbol("boolean"),
                     AST::SignalSymbol::Local,
                     NULL,
                     NULL
                     );
  compSWSig->initializer =
    new LoadVariableExpression(
                               new ConstantSymbol(
                                                  "false",
                                                  new TypeSymbol("boolean"),
                                                  new Literal("0",new TypeSymbol("
boolean"))
```

```
                                                    )
                                    );

        hwm->scopeStatement->symbols->enter(compSWSig);
        *pe = new LoadSignalValueExpression(compSWSig);
      }
    }
  }

  /* Generates a valid Esterel SignalIdentifier from a given SignalExpression
   * @param e An expression
   * @return A string built from the expression by substituting operators with alphabetic
      characters
   *         and spaces with an underscore. The string is cropped to 40 characters
   */
  string CoDesignEsterelPrinter::getCompoundSigName(Expression *e){
    std::ostringstream comp;
    EsterelPrinter ep(comp);
    ep.print(e);
    string compStr = comp.str();
    for(unsigned int x=0; x<compStr.size();x++) {
      if ((compStr.substr(x,1)=="␣")) compStr.replace(x,1,"_");
      else if ((compStr.substr(x,1)==")")||(compStr.substr(x,1)=="(")){
        compStr.replace(x,1,"");x--;}
      else if ((compStr.substr(x,1)=="=")) compStr.replace(x,1,"EQ");
      else if ((compStr.substr(x,2)=="<>")) compStr.replace(x,2,"NEQ");
      else if ((compStr.substr(x,1)=="<")) compStr.replace(x,1,"LT");
      else if ((compStr.substr(x,2)=="<=")) compStr.replace(x,2,"LET");
      else if ((compStr.substr(x,1)==">")) compStr.replace(x,1,"GT");
      else if ((compStr.substr(x,2)==">=")) compStr.replace(x,2,"GET");
      else if ((compStr.substr(x,1)=="+")) compStr.replace(x,1,"ADD");
      else if ((compStr.substr(x,1)=="-")) compStr.replace(x,1,"SUB");
      else if ((compStr.substr(x,1)=="*")) compStr.replace(x,1,"MUL");
      else if ((compStr.substr(x,1)=="?")) {compStr.replace(x,1,""); x--;}
      else if ((compStr.substr(x,2)=="??")) {compStr.replace(x,2,""); x--;}
    }
    compStr=compStr.substr(0,40);
    transform (compStr.begin(), compStr.end(),    // source
               compStr.begin(),                   // destination
               toupper);                          // operation
    if(compStr[compStr.size()-1]=='_') compStr.erase(compStr.size()-1);
    return compStr;
  }

  /* Applies 'pre' to SignalExpression 'e' rekursively according to the rules
   * in the Esterel Primer Chapter 4.6.2. The flag 'i' is the index of the
   * current pre operator pre0 or pre1 (see Esterel Primer chapter 4.6.2).
   * @return Expression with expanded pre() operators
   */
  Expression* CoDesignEsterelPrinter::applyPre(Expression* e, bool i){
    BinaryOp *b;
    UnaryOp *u;
    if ((dynamic_cast<LoadSignalExpression*>(e)!=NULL) && !i) return new UnaryOp(NULL,"pre_0"
      ,e);
    else if ((dynamic_cast<LoadSignalExpression*>(e)!=NULL) && i) return new UnaryOp(NULL,"
      pre_1",e);
    else if ((u = dynamic_cast<UnaryOp *>(e))!=NULL)
      return new UnaryOp(NULL,"not",applyPre(u->source,!i));
    else {
      b = dynamic_cast<BinaryOp*>(e);
      if(b->op.compare("and")==0)
        return new BinaryOp(NULL,"and",applyPre(b->source1,i),applyPre(b->source2,i));
      if(b->op.compare("or")==0)
        return new BinaryOp(NULL,"or",applyPre(b->source1,i),applyPre(b->source2,i));
    }
    return NULL;
  }

  /* Substitute all 'pre(S)' and 'pre(?S)' Expressions in 'e' with new Signal pre_S and preV_S
   * respectively. New Signals pre_S and preV_S are calculated in the HW-module of the same
      scope
   * where S is declared.
   * Valued Expressions (see Primer chapter 4.6):
   *  - pre(?S) gives the value of S in the previous instance, S is SignalIdentifier
   * Signal Expressions:
   *  - argument of pre is a signal expression
   *  - rules in Esterel Primer Chapter 4.6.2 hold
   */
  // void CoDesignEsterelPrinter::substPre(Expression *e, Expression **pe, Module* hwModule){
  void CoDesignEsterelPrinter::substPre(Expression *e, Expression **pe){
    if(dynamic_cast<UnaryOp*> (e)) {
      UnaryOp *u = dynamic_cast<UnaryOp*> (e);
      string compSigName;
      // if operator is the pre() operator ...
      if(u->op.compare(0,3,"pre")==0) {
        LoadSignalValueExpression* lsve=NULL;
        string compSigName;
        SignalSymbol* preSig;
        // if pre is applied to valued signal
        if((lsve = dynamic_cast<LoadSignalValueExpression*> (u->source))!=NULL) {
          // get unique signal name consisting of a prefix "preV_" the signal name and an
          // optional number
          compSigName = cosyndata->usedSymbols->getUniqueSigName("preV_" + lsve->signal->name
    );
          // reuse preV_S if it is already calculated
          if (cosyndata->usedPreSymbols->local_contains(compSigName)) {
            preSig = (SignalSymbol*) cosyndata->usedPreSymbols->get(compSigName);
            //if(!hwModule->signals->local_contains(compSigName))
            //  hwModule->signals->enter(preSig);
            // if preV_S is not calculated yet
          } else {
            // introduce new signal named 'preV_S'
            preSig =
              new SignalSymbol(compSigName,lsve->signal->type,AST::SignalSymbol::Local,NULL,
    NULL);
            // get the scope where S was declared
            SymbolTable* st = new SymbolTable();
            getUsedSymbols(u->source,*st);
            HWModuleTreeNode* topHWModule = cosyndata->hwModuleTree.getLowestNode(st);
            // declare signal 'preV_S' in that scope
            topHWModule->scopeStatement->symbols->enter(preSig);
```

```
      // claculate 'preV_S' in the corresponding Scope
      VariableSymbol *var = new VariableSymbol("preval",lsve->signal->type, u->source)
;
      Var *lvd = new Var();
      lvd->symbols = new SymbolTable();
      lvd->symbols->enter(var);
      StatementList *sl = new StatementList();
      LoadSignalExpression* lse = new LoadSignalExpression(lsve->signal->type,lsve->
signal);
      Every* est =  new Every(lvd, new Delay(NULL,lse,NULL,true,NULL));
      lvd->body = sl;
      sl->statements.push_back(new Pause());
      sl->statements.push_back(new Emit(preSig,new LoadVariableExpression(var)));
      topHWModule->preFunctions->threads.push_back(est);

      // declare signal 'preV_S' as input in the HWModule where it is used
      preSig =
        new SignalSymbol(compSigName,lsve->signal->type,AST::SignalSymbol::Input,NULL,
NULL);
      //hwModule->signals->enter(preSig);
      // enter signal 'preV_S' in usedPreSymbols
      cosyndata->usedPreSymbols->enter(preSig);
    }
    *pe = new LoadSignalValueExpression(preSig);
    // if pre is applied to a pure signal
  } else if (dynamic_cast<LoadSignalExpression*> (u->source)) {
    if ((u->op.compare("pre_0")==0)||(u->op.compare("pre")==0)) {
      // get unique signal name consisting of a prefix "pre0_" the signal name and an
      // optional number
      compSigName =
        cosyndata->usedSymbols->getUniqueSigName("pre0_" + getCompoundSigName(u->source
));
      // reuse pre0_S if it is already calculated
      if (cosyndata->usedPreSymbols->local_contains(compSigName)) {
        preSig = (SignalSymbol*) cosyndata->usedPreSymbols->get(compSigName);
        // if pre0_S is not calculated yet
      } else {
        // introduce new signal named 'pre0_S'
        preSig = new SignalSymbol(compSigName,NULL,AST::SignalSymbol::Local,NULL,NULL);
        // get the scope where S was declared
        SymbolTable* st = new SymbolTable();
        getUsedSymbols(u->source,*st);
        HWModuleTreeNode* topHWModule = cosyndata->hwModuleTree.getLowestNode(st);
        // declare signal 'pre0_S' in that scope
        topHWModule->scopeStatement->symbols->enter(preSig);
        // claculate 'pre0_S' in the corresponding HWModule
        StatementList *sl = new StatementList();
        sl->statements.push_back(new Pause());
        sl->statements.push_back(new Emit(preSig,NULL));
        topHWModule
          ->preFunctions
          ->threads.push_back(new Every(sl, new Delay(NULL,u->source,NULL,true,NULL)));
        // declare signal 'pre0_S' as input in the HWModule where it is used
        preSig = new SignalSymbol(compSigName,NULL,AST::SignalSymbol::Input,NULL,NULL);
        //hwModule->signals->enter(preSig);
        // enter signal 'pre0_S' in usedPreSymbols
        cosyndata->usedPreSymbols->enter(preSig);
      }
    } else {
      // get unique signal name consisting of a prefix "pre1_" the signal name and an
      // optional number
      compSigName =
        cosyndata->usedSymbols->getUniqueSigName(
                                                 "pre1_" + getCompoundSigName(u->source
));
      // reuse pre1_S if it is already calculated
      if (cosyndata->usedPreSymbols->local_contains(compSigName)) {
        preSig = (SignalSymbol*) cosyndata->usedPreSymbols->get(compSigName);
        // if pre1_S is not calculated yet
      } else {
        // introduce new signal named 'pre1_S'
        preSig = new SignalSymbol(compSigName,NULL,AST::SignalSymbol::Local,NULL,NULL);
        // get the scope where S was declared
        SymbolTable* st = new SymbolTable();
        getUsedSymbols(u->source,*st);
        HWModuleTreeNode* topHWModule = cosyndata->hwModuleTree.getLowestNode(st);
        // declare signal 'pre1_S' in that scope
        topHWModule->scopeStatement->symbols->enter(preSig);
        // claculate 'pre1_S' in the corresponding HWModule
        StatementList *sl_0 = new StatementList();
        StatementList *sl_1 = new StatementList();
        sl_0->statements.push_back(new Emit(preSig,NULL));
        sl_1->statements.push_back(new Pause());
        sl_1->statements.push_back(new Emit(preSig,NULL));
        sl_0->statements.push_back(new Every(sl_1, new Delay(NULL,u->source,NULL,true,
NULL)));
        topHWModule->preFunctions->threads.push_back(sl_0);
        // declare signal 'pre1_S' as input in the HWModule where it is used
        preSig = new SignalSymbol(compSigName,NULL,AST::SignalSymbol::Input,NULL,NULL);
        //hwModule->signals->enter(preSig);
        // enter signal 'pre1_S' in usedPreSymbols
        cosyndata->usedPreSymbols->enter(preSig);
      }
    }
    *pe = new LoadSignalExpression(NULL,preSig);
  } else {
    // if pre is applied to a signal expression
    e = *pe = applyPre(u->source,false);
    //substPre(e,&e,hwModule);
    substPre(e,&e);
  }
}
// else substPre(u->source,&(u->source),hwModule);
else substPre(u->source,&(u->source));
}
else if(dynamic_cast<BinaryOp*> (e)) {
  BinaryOp *b = dynamic_cast<BinaryOp*> (e);
           //substPre(b->source1,&(b->source1),hwModule);
  substPre(b->source1,&(b->source1));
  //substPre(b->source2,&(b->source2),hwModule);
  substPre(b->source2,&(b->source2));
}
```

```
}

/*
 * - Substitutes the signal expression e with an auxiliary signal s
 * - expand all pre() operators in the expression until they are applied to just one signal
 * - substitute all pre(), pre_0() and pre_1() with an auxiliary signal pre_S, pre0_S and
    pre1_S
 *   and declare and calculate these signals within parallel to the scopes body that declares
     S
 * - Find the hierarchically lowest scope in that all signals in e are known and declare the
 *   auxiliary signal s within this scope. Declare in the hw-module correspondig to that
    scope
 *   all signals used in e as input and s as output
 * - Add a new thread to the hw-module that emits s everytime the expression e is present
*/
void CoDesignEsterelPrinter::sigexpression(Expression *e,Expression **pe) {
  // complex signal expression or not?
  if (dynamic_cast<Delay*>(e) || dynamic_cast<LoadSignalExpression*>(e)){
    process(e);
  }
  else {
    // expand all pre operators and substitiute them with auxiliary signals
    substPre(*pe,pe);
    // get new signal name for the evaluated expression
    string compSigName = cosyndata->usedSymbols->getUniqueSigName(getCompoundSigName(e));
    // store all signals used in the expression in st
    SymbolTable *st = new SymbolTable();
    getUsedSymbols(e,*st);
    // get the corresponding signal scope
    HWModuleTreeNode* hwm = cosyndata->hwModuleTree.getLowestNode(st);
    Module* hwModule = hwm->hwModule;
    std::vector<Symbol*>::iterator i = st->symbols.begin();
    // declare used symbols in the sigexpression as input to the HWModule
    // and write signal dependencies
    while (i != st->symbols.end()) {
      SignalSymbol* s = dynamic_cast<SignalSymbol*>(*i);
      if(s->kind != SignalSymbol::Input) {
        *i = new SignalSymbol(s->name,s->type,SignalSymbol::Input,s->combine,s->initializer);
      }
      i++;
      cosyndata->signalDependencies << compSigName << "␣" << s->name << "\n";
    }
    ((ExtSymbolTable*) hwModule->signals)->addSymbols(st);
    // declare the auxiliary signal as output to the hw-module
    SignalSymbol* compHWSig =
      new SignalSymbol(compSigName,NULL,AST::SignalSymbol::Output,NULL,NULL);
    (*hwModule->signals).enter(compHWSig);
    cosyndata->usedSymbols->enter(compHWSig);
    // The hardware module emits the new signal everytime
    // the SignalExpression is true
    dynamic_cast<ParallelStatementList*>(hwModule->body)
      ->threads.push_back(new Every(new Emit(compHWSig,NULL),new Delay(NULL,e,NULL,true,NULL
    ));
    // Substitute the SignalExpression in the software module by
    // the newly generated signal
    SignalSymbol* compSWSig =
      new SignalSymbol(compSigName,NULL,AST::SignalSymbol::Local,NULL,NULL);
    hwm->scopeStatement->symbols->enter(compSWSig);
    *pe = new LoadSignalExpression(NULL,compSWSig);
  }
}

bool CoDesignEsterelPrinter::push_precedence(int p) {
  bool needBrackets = p < precedence.back();
  precedence.push_back(p);
  return needBrackets;
}

void CoDesignEsterelPrinter::pop_precedence() {
  precedence.pop_back();
}

Status CoDesignEsterelPrinter::visit(ModuleSymbol &) { assert(0); }
Status CoDesignEsterelPrinter::visit(VariableSymbol &) { assert(0); }
Status CoDesignEsterelPrinter::visit(TypeRenaming &) { assert(0); }
Status CoDesignEsterelPrinter::visit(ConstantRenaming &) { assert(0); }
Status CoDesignEsterelPrinter::visit(FunctionRenaming &) { assert(0); }
Status CoDesignEsterelPrinter::visit(ProcedureRenaming &) { assert(0); }
Status CoDesignEsterelPrinter::visit(SignalRenaming &) { assert(0); }
Status CoDesignEsterelPrinter::visit(PredicatedStatement &) { assert(0); }
Status CoDesignEsterelPrinter::visit(TaskCall &) { assert(0); }

Status CoDesignEsterelPrinter::visit(BuiltinConstantSymbol &) { return Status(); }
Status CoDesignEsterelPrinter::visit(BuiltinSignalSymbol &) { return Status(); }
Status CoDesignEsterelPrinter::visit(BuiltinTypeSymbol &) { return Status(); }
Status CoDesignEsterelPrinter::visit(BuiltinFunctionSymbol &) { return Status(); }

/*
 * Performs the partitioning of the original program into sw- and hw-modules
 */
Status CoDesignEsterelPrinter::visit(Module &m) {
  assert(m.symbol);
  // First create the sw-module as a copy of the original module
  cosyndata->swModule=new Module(new ModuleSymbol(m.symbol->name+"_sw"));
  cosyndata->hwModuleNamePrefix = m.symbol->name;
  Signal* defaultScope = new Signal();
  defaultScope->symbols = new SymbolTable();
  cosyndata->hwModuleTree.addChild(cosyndata->hwModuleNamePrefix,defaultScope);
  /* Data Objects */
  // functions, procedures, tasks
  cosyndata->swModule->functions = m.functions;
  cosyndata->swModule->procedures = m.procedures;
  cosyndata->swModule->tasks = m.tasks;
  // types (must be declared where they are used)
  cosyndata->swModule->types = m.types;
  // constants (must be declared where they are used)
  cosyndata->swModule->constants = m.constants;

  /* Signals, Sensors and Relations */
  // the interface signals must be declared in the main (i.e SW-) module
  // and where they are used
  cosyndata->swModule->signals = m.signals;
```

```
// relations (must be declared in the main module)
cosyndata->swModule->relations = m.relations;

/* Body */
// The body of the software module is similar to the original module body.
// Only SignalExpressions will be substituted, every by a new auxiliary signal.
// This auxiliary signal is emitted in the hardware module everytime the
// original expression is true (see sigexpression())
cosyndata->swModule->body = m.body;

// start the partitioning
process(cosyndata->swModule->body);

// Run HW- and SW-Modules parallel to the body of their corresponding signal scopes.
// The newly generated body looks like this:
//  trap COSYN_TRAP in
//    run HW-Module
//  ||
//    Calculation of auxiliary signals substituting pre operators
//  ||
//    original body;
//    exit COSYN_TRAP
//  end trap
vector<HWModuleTreeNode*>::iterator i = cosyndata->hwModuleTree.hwModules.begin();
i = cosyndata->hwModuleTree.hwModules.end() ) {
  HWModuleTreeNode* hwModule = ((HWModuleTreeNode*) (*i));
  if ((dynamic_cast<ParallelStatementList*>(hwModule->hwModule->body))->threads.size()>0) {
    SignalSymbol* trapSymbol;

    ParallelStatementList *psl = new ParallelStatementList();
    psl->threads.push_back(new Run(hwModule->hwModule->symbol->name,NULL));
    if(hwModule->preFunctions->threads.size()>0) {
      psl->threads.push_back(hwModule->preFunctions);
      trapSymbol =
        new SignalSymbol(cosyndata->usedSymbols->getUniqueSigName("COSYN_TRAP_PRE"),
                         NULL,
                         SignalSymbol::Trap,
                         NULL,
                         NULL);
      cosyndata->usedSymbols->enter(trapSymbol);
    }
    else {
      trapSymbol =
        new SignalSymbol(cosyndata->usedSymbols->getUniqueSigName("COSYN_TRAP"),
                         NULL,
                         SignalSymbol::Trap,
                         NULL,
                         NULL);
      cosyndata->usedSymbols->enter(trapSymbol);
    }
    Trap* trap = new Trap();
    trap->symbols = new SymbolTable();
    trap->symbols->enter(trapSymbol);
    StatementList* sl = new StatementList();
    if(i == cosyndata->hwModuleTree.hwModules.begin()) {
      sl->statements.push_back((Statement*)cosyndata->swModule->body);
      sl->statements.push_back(new Exit(trapSymbol,NULL));
      psl->threads.push_back(sl);
      trap->body = psl;

      hwModule->scopeStatement->body = trap;
      cosyndata->swModule->body = hwModule->scopeStatement;
      cosyndata->convModules.add(hwModule->hwModule);
    } else {
      sl->statements.push_back(hwModule->scopeStatement->body);
      sl->statements.push_back(new Exit(trapSymbol,NULL));
      psl->threads.push_back(sl);
      trap->body = psl;
      hwModule->scopeStatement->body = trap;
      cosyndata->convModules.add(hwModule->hwModule);
    }
  }
  i++;
}
cosyndata->convModules.add(cosyndata->swModule);
return Status();
}

Status CoDesignEsterelPrinter::visit(Modules &m) {
  // preprocess the code and get table of used symbols
  vector<Module*>::iterator i = m.modules.begin();
  while ( i != m.modules.end() ) {
    assert(*i);
    assert((*i)->symbol);
    process(*i);
    i++;
  }

  // PrettyPrint the converted Esterel code
  AST::EsterelPrinter p(o);
  cosyndata->convModules.welcome(p);

  return Status();
}

Status CoDesignEsterelPrinter::visit(Exclusion& e) {
  return Status();
}

Status CoDesignEsterelPrinter::visit(Implication& e) {
  return Status();
}

Status CoDesignEsterelPrinter::visit(SymbolTable &t) {
  for ( SymbolTable::const_iterator i = t.begin() ; i != t.end() ; i++ ) {
    process(*i);
  }
  return Status();
}
```

```
Status CoDesignEsterelPrinter::visit(TypeSymbol &s) {
  return Status();
}

Status CoDesignEsterelPrinter::visit(ConstantSymbol &s) {
  if (s.initializer) {
    expression(s.initializer,NULL);
  }
  assert(s.type);
  return Status();
}

Status CoDesignEsterelPrinter::visit(SignalSymbol &s) {
  return Status();
}

Status CoDesignEsterelPrinter::visit(FunctionSymbol &s) {
  return Status();
}

Status CoDesignEsterelPrinter::visit(ProcedureSymbol &s) {
  return Status();
}

Status CoDesignEsterelPrinter::visit(TaskSymbol &s) {
  return Status();
}

Status CoDesignEsterelPrinter::visit(StatementList &l) {
  vector<Statement*>::const_iterator i = l.statements.begin();
  while ( i != l.statements.end() ) {
    currentNode = (Statement**) &(*i);
    process(*i);
    i++;
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(ParallelStatementList &l) {
  vector<Statement*>::const_iterator i = l.threads.begin();
  while ( i != l.threads.end() ) {
    currentNode = (Statement**) &(*i);
    process(*i);
    i++;
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(Nothing &) { return Status(); }
Status CoDesignEsterelPrinter::visit(Pause &) { return Status(); }
Status CoDesignEsterelPrinter::visit(Halt &) { return Status(); }

Status CoDesignEsterelPrinter::visit(Emit &e) {
  assert(e.signal);
  if (e.value) {
    expression(e.value,&(e.value));
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(StartCounter &s) {
  assert(s.counter);
  expression(s.count,&(s.count));
  return Status();
}

Status CoDesignEsterelPrinter::visit(Sustain &e) {
  assert(e.signal);
  if (e.value) {
    expression(e.value,&(e.value));
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(Assign &a) {
  assert(a.variable);
  assert(a.value);
  expression(a.value,&(a.value));
  return Status();
}

Status CoDesignEsterelPrinter::visit(ProcedureCall &c) {
  assert(c.procedure);
  vector<Expression*>::const_iterator j = c.value_args.begin();
  while (j != c.value_args.end()) {
    expression(*j,(Expression**)&(*j));
    j++;
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(Exec &e) {
  for ( vector<TaskCall*>::const_iterator i = e.calls.begin() ;
        i != e.calls.end() ; i++) {
    assert(*i);
    assert((*i)->procedure);
    vector<VariableSymbol*>::const_iterator k = (*i)->reference_args.begin();
    while (k != (*i)->reference_args.end()) {
      k++;
    }
    vector<Expression*>::const_iterator j = (*i)->value_args.begin();
    while (j != (*i)->value_args.end()) {
      expression(*j,(Expression**)&(*j));
      j++;
    }
    assert((*i)->signal);
    if ((*i)->body) {
      currentNode = (Statement**) &((*i)->body);
      process((*i)->body);
    }
  }
  return Status();
```

```
}

Status CoDesignEsterelPrinter::visit(Present &p) {
  if (p.cases.size() == 1) {
    PredicatedStatement *ps = p.cases[0];
    assert(ps);
    sigexpression(ps->predicate,&(ps->predicate));
    if (ps->body) {
      currentNode = (Statement**) &(ps->body);
      statement(ps->body);
    }
  } else {
    for (vector<PredicatedStatement*>::const_iterator i = p.cases.begin() ;
         i != p.cases.end() ; i++) {
      assert(*i);
      assert((*i)->predicate);
      sigexpression((*i)->predicate,&((*i)->predicate));
      if ((*i)->body) {
        currentNode = (Statement**) &((*i)->body);
        process((*i)->body);
      }
    }
  }
  if (p.default_stmt) {
    currentNode = (Statement**) &(p.default_stmt);
    statement(p.default_stmt);
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(If &s) {
  vector<PredicatedStatement*>::const_iterator i = s.cases.begin();
  assert(i != s.cases.end());
  expression((*i)->predicate,&((*i)->predicate));
  if ((*i)->body) {
    currentNode = (Statement**) &((*i)->body);
    statement((*i)->body);
  }
  for ( i++ ; i != s.cases.end() ; i++ ) {
    assert(*i);
    expression((*i)->predicate,&((*i)->predicate));
    currentNode = (Statement**) &((*i)->body);
    statement((*i)->body);
  }
  if (s.default_stmt) {
    currentNode = (Statement**) &(s.default_stmt);
    statement(s.default_stmt);
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(Loop &l) {
  currentNode = (Statement**) &(l.body);
  statement(l.body);
  return Status();
}

Status CoDesignEsterelPrinter::visit(Repeat &r) {
  expression(r.count,&(r.count));
  currentNode = (Statement**) &(r.body);
  statement(r.body);
  return Status();
}

Status CoDesignEsterelPrinter::visit(Abort &a) {
  statement(a.body);
  if (a.cases.size() == 1) {
    PredicatedStatement *ps = a.cases[0];
    assert(ps);
    sigexpression(ps->predicate,&(ps->predicate));
    if (ps->body) {
      currentNode = (Statement**) &(ps->body);
      statement(ps->body);
    }
  } else {
    // Abort cases
    for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
          i != a.cases.end() ; i++ ) {
      tab();
      assert(*i);
      sigexpression((*i)->predicate,&((*i)->predicate));
      if ((*i)->body) {
        currentNode = (Statement**) &((*i)->body);
        process((*i)->body);
      }
    }
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(Await& a) {
  if (a.cases.size() == 1) {
    // Simple abort condition
    PredicatedStatement *ps = a.cases[0];
    assert(ps);
    sigexpression(ps->predicate,&(ps->predicate));
    if (ps->body) {
      currentNode = (Statement**) &(ps->body);
      statement(ps->body);
    }
  } else {
    for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
          i != a.cases.end() ; i++ ) {
      assert(*i);
      sigexpression((*i)->predicate,&((*i)->predicate));
      if ((*i)->body) {
        currentNode = (Statement**) &((*i)->body);
        process((*i)->body);
      }
    }
  }
  return Status();
```

```
}

Status CoDesignEsterelPrinter::visit(LoopEach &l) {
  currentNode = (Statement**) &(l.body);
  statement(l.body);
  sigexpression(l.predicate,&(l.predicate));
  return Status();
}

Status CoDesignEsterelPrinter::visit(Every &e) {
  sigexpression(e.predicate,&(e.predicate));
  currentNode = (Statement**) &(e.body);
  statement(e.body);
  return Status();
}

Status CoDesignEsterelPrinter::visit(Suspend &s) {
  currentNode = (Statement**) &(s.body);
  statement(s.body);
  sigexpression(s.predicate,&(s.predicate));
  return Status();
}

Status CoDesignEsterelPrinter::visit(DoWatching &d) {
  currentNode = (Statement**) &(d.body);
  statement(d.body);
  sigexpression(d.predicate,&(d.predicate));
  if (d.timeout) {
    currentNode = (Statement**) &(d.timeout);
    statement(d.timeout);
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(DoUpto &d) {
  currentNode = (Statement**) &(d.body);
  statement(d.body);
  sigexpression(d.predicate,&(d.predicate));
  return Status();
}

Status CoDesignEsterelPrinter::visit(Trap &t) {
  assert(t.symbols);
  SymbolTable::const_iterator i = t.symbols->begin();
  currentNode = (Statement**) &(t.body);
  statement(t.body);
  for (vector<PredicatedStatement*>::const_iterator i = t.handlers.begin() ;
       i != t.handlers.end() ; i++ ) {
    assert(*i);
    expression((*i)->predicate,&((*i)->predicate));
    currentNode = (Statement**) &((*i)->body);
    statement((*i)->body);
  }

  return Status();
}

Status CoDesignEsterelPrinter::visit(Exit &e) {
  assert(e.trap);
  assert(e.trap->kind == SignalSymbol::Trap);
  if (e.value) {
    expression(e.value,&(e.value));
  }
  return Status();
}

Status CoDesignEsterelPrinter::visit(Var &v) {
  cosyndata->hwModuleTree.addChild(cosyndata->hwModuleNamePrefix,&v);
  currentNode = (Statement**) &(v.body);
  statement(v.body);
  // store the current scope
  cosyndata->hwModuleTree.setCurrentNode(cosyndata->hwModuleTree.getCurrentNode()->parent);
  return Status();
}

Status CoDesignEsterelPrinter::visit(Signal &s) {
  cosyndata->hwModuleTree.addChild(cosyndata->hwModuleNamePrefix,&s);
  currentNode = (Statement**) &(s.body);
  statement(s.body);
  // store the current scope
  cosyndata->hwModuleTree.setCurrentNode(cosyndata->hwModuleTree.getCurrentNode()->parent);
  return Status();
}

Status CoDesignEsterelPrinter::visit(Run &r) {
  if ( r.types.size() + r.constants.size() + r.functions.size() +
       r.procedures.size() + r.tasks.size() + r.signals.size() > 0 ) {

    bool more = false;

    if (r.types.size() > 0) {
      vector<TypeRenaming*>::const_iterator i = r.types.begin();
      while (i != r.types.end()) {
        assert(*i);
        assert((*i)->new_type);
        i++;
      }
      more = true;
    }

    if (r.constants.size() > 0 ) {
      vector<ConstantRenaming*>::const_iterator i = r.constants.begin();
      while (i != r.constants.end()) {
        assert(*i);
        assert((*i)->new_value);
        expression((*i)->new_value,&((*i)->new_value));
        i++;
      }
      more = true;
    }

    if (r.functions.size() > 0) {
```

```
    vector<FunctionRenaming*>::const_iterator i = r.functions.begin();
    while (i != r.functions.end()) {
      assert(*i);
      assert((*i)->new_func);
      i++;
    }
    more = true;
  }

  if (r.procedures.size() > 0) {
    vector<ProcedureRenaming*>::const_iterator i = r.procedures.begin();
    while (i != r.procedures.end()) {
      assert(*i);
      assert((*i)->new_proc);
      i++;
    }
    more = true;
  }

  if (r.tasks.size() > 0) {
    vector<ProcedureRenaming*>::const_iterator i = r.tasks.begin();
    while (i != r.tasks.end() ) {
      assert(*i);
      assert((*i)->new_proc);
      i++;
    }
    more = true;
  }

  if (r.signals.size() > 0) {
    vector<SignalRenaming*>::const_iterator i = r.signals.begin();
    while (i != r.signals.end() ) {
      assert(*i);
      assert((*i)->new_sig);
      i++;
    }
  }
 }
 return Status();
}

Status CoDesignEsterelPrinter::visit(UnaryOp &u) {
 assert(unarylevel.find(u.op) != unarylevel.end());
 process(u.source);
 return Status();
}

Status CoDesignEsterelPrinter::visit(BinaryOp &b) {
 assert(level.find(b.op) != level.end());
 process(b.source1);
 process(b.source2);
 return Status();
}

Status CoDesignEsterelPrinter::visit(LoadVariableExpression &e) {
 assert(e.variable);
 return Status();
}

Status CoDesignEsterelPrinter::visit(LoadSignalExpression &e) {
 assert(e.signal);

 return Status();
}

Status CoDesignEsterelPrinter::visit(LoadSignalValueExpression &e) {
 assert(e.signal);

 return Status();
}

Status CoDesignEsterelPrinter::visit(Literal &l) {
 assert(l.type);
 return Status();
}

Status CoDesignEsterelPrinter::visit(FunctionCall &e) {
 assert(e.callee);
 vector<Expression*>::const_iterator i = e.arguments.begin();
 while (i != e.arguments.end()) {
   expression(*i,(Expression**)&(*i));
   i++;
 }
 return Status();
}

Status CoDesignEsterelPrinter::visit(Delay &e) {
 if (e.is_immediate) {
   assert(e.count == NULL);
 } else {
   expression(e.count,&e.count);
 }
 sigexpression(e.predicate,&(e.predicate));
 return Status();
}

Status CoDesignEsterelPrinter::visit(CheckCounter &e) {
 assert(e.counter);
 expression(e.predicate,&(e.predicate));
 return Status();
}

Status CoDesignEsterelPrinter::visit(IfThenElse &s) {
 expression(s.predicate,&(s.predicate));
 if (s.then_part) {
   statement(s.then_part);
 }
 if (s.else_part) {
   statement(s.else_part);
 }
 return Status();
}
```

}

A.1.9. Zwischenschritt: Logikminimierung

Mit Hilfe der Klasse `MVSIS` lassen sich zweistufige und mehrstufige Logikminimierungsverfahren auf die zu berechnenden Ausdrücke in den Hardwaremodulen anwenden. Die Klasse implementiert dazu keine eigenen Algorithmen zur Logikminimierung, sondern bedient sich des externen Tools MVSIS. MVSIS unterstützt zur Ein- und Ausgabe von Logikbeschreibungen das BLIF Format. Die Funktion `writeBLIF` konvertiert alle in einem Hardwaremodul vorkommenden Ausdrücke in eine BLIF Datei. Die Funktion `readBLIF` liest die BLIF Datei mit der minimierten Logik wieder ein und konvertiert die Logikbeschreibung zurück nach ESTEREL.

A.1.10. CoDesignLogicMinimization.hpp

```
#ifndef _LOGICMINIMIZATION_HPP_
#define _LOGICMINIMIZATION_HPP_

#include "AST.hpp"
#include "CoDesignDataStructures.hpp"
#include <cmath>

namespace LogicMinimization {

  class SortedSymbolTable : public AST::SymbolTable
  {
  public:
    SortedSymbolTable(){}
    // enter symbol alphabetically sorted
    void enter(AST::Symbol *);
    // returns position in the list
    int getPos(std::string);
  };

  /**
   * An object of this class is thrown if an error occurs during logic minimization
   */
  class MVSISError {
  public:
    std::string s;
    MVSISError(std::string ss) : s(ss) {}
  };

  /*  Performs a multi-level logic minimization on each hw-module by calling the function "
      minimize"
   */
  class MVSIS
  {
  private:
    // Member functions are commented in the .cpp implementation file
    bool evaluate(AST::Expression*, SortedSymbolTable&, char* term);
    void getUsedSymbols(AST::Expression*, SortedSymbolTable&);
    Expression* getDelayExp(Expression*);
    int writeOnSet(AST::Expression*, std::ostream&);
    int writeBlif(Module*, std::string);
    void readBlif(Module*, std::string);
  public:
    MVSIS();
    void minimize(CoDesign::CoDesignData*, std::string);
  };
}
#endif
```

A.1.11. CoDesignLogicMinimization.cpp

```
#include "CoDesignLogicMinimization.hpp"
#include <iostream>
#include <cstring>
#include <fstream>
#include <sstream>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
#include <stdlib.h>

namespace LogicMinimization{

  /* Inserts a Symbol alphabetically into the sorted symbol table
   */
  void SortedSymbolTable::enter(AST::Symbol * s) {
    assert(s);
    assert(!local_contains(s->name));
    for(int x=0; abs(x) < size(); x++) {
      if ((*(symbols.begin()+x))->name.compare(s->name)>=0) {
        symbols.insert(symbols.begin()+x,s);
        return;
      }
    }
    symbols.push_back(s);
  }

  /* Gives the position in the symbol table
   */
  int SortedSymbolTable::getPos(std::string s) {
    for(int x=0; abs(x) < size(); x++) {
      if ((*(symbols.begin()+x))->name==s) return x;
    }
    return -1;
  }


  MVSIS::MVSIS() {
  }

  /* Calculates the result of a Boolean Function (given in expression e)
   * depending on the input assignment (given in term)
   */
  bool MVSIS::evaluate(AST::Expression* e, SortedSymbolTable &usedSymbols, char* term){
    if (dynamic_cast<AST::LoadSignalExpression*>(e)) {
      AST::SignalSymbol* Sig = dynamic_cast<AST::LoadSignalExpression*>(e)->signal;
      if (term[usedSymbols.getPos(Sig->name)]=='0') return false;
      else return true;
    }
    if (dynamic_cast<AST::LoadSignalValueExpression*>(e)) {
      AST::SignalSymbol* Sig = dynamic_cast<AST::LoadSignalValueExpression*>(e)->signal;
      if (term[usedSymbols.getPos(Sig->name)]=='0') return false;
      else return true;
    }
    if (dynamic_cast<AST::LoadVariableExpression*>(e)) {
      LoadVariableExpression* lve = dynamic_cast<AST::LoadVariableExpression*>(e);
      ConstantSymbol* cs = dynamic_cast<AST::ConstantSymbol*>(lve->variable);
      assert(cs);
      Literal* l;
      if(l = dynamic_cast<AST::Literal*>(cs->initializer)) ;
      else if(lve = dynamic_cast<AST::LoadVariableExpression*>(cs->initializer)) {
        cs = dynamic_cast<AST::ConstantSymbol*>(lve->variable);
        assert(cs);
        l = dynamic_cast<AST::Literal*>(cs->initializer);
      }
      assert(l);
      //std::cerr << cs->name << " " << l->value << "\n";
      if(l->value == "1") return true;
      else return false;
    }

    if (dynamic_cast<AST::BinaryOp*>(e)) {
      bool e1,e2;
      e1 = evaluate(dynamic_cast<AST::BinaryOp*>(e)->source1, usedSymbols, term);
      e2 = evaluate(dynamic_cast<AST::BinaryOp*>(e)->source2, usedSymbols, term);
      if(dynamic_cast<AST::BinaryOp*>(e)->op=="or") return (e1||e2);
      if(dynamic_cast<AST::BinaryOp*>(e)->op=="and") return (e1&&e2);
    }
    if (dynamic_cast<AST::UnaryOp*>(e)) {
      return !evaluate(dynamic_cast<AST::UnaryOp*>(e)->source, usedSymbols, term);
    }
    return false;
  }

  /* Gets all Symbols expression 'e' depends on and stores them in 'usedSymbols'
   */
  void MVSIS::getUsedSymbols(AST::Expression* e, SortedSymbolTable &usedSymbols){
    if (dynamic_cast<AST::LoadSignalExpression*>(e)) {
      AST::SignalSymbol* Sig = dynamic_cast<AST::LoadSignalExpression*>(e)->signal;
      if (!usedSymbols.local_contains(Sig->name)) {
        usedSymbols.enter(Sig);
      }
    }
    if (dynamic_cast<AST::LoadSignalValueExpression*>(e)) {
      AST::SignalSymbol* Sig = dynamic_cast<AST::LoadSignalValueExpression*>(e)->signal;
      if (!usedSymbols.local_contains(Sig->name)) {
        usedSymbols.enter(Sig);
      }
    }
    if (dynamic_cast<AST::BinaryOp*>(e)) {
      getUsedSymbols(dynamic_cast<AST::BinaryOp*>(e)->source1, usedSymbols);
      getUsedSymbols(dynamic_cast<AST::BinaryOp*>(e)->source2, usedSymbols);
    }
    if (dynamic_cast<AST::UnaryOp*>(e)) {
      getUsedSymbols(dynamic_cast<AST::UnaryOp*>(e)->source, usedSymbols);
```

```
  }
}

/* returns a disjunction term of all SignalSymbols in 'e'
 */
Expression* MVSIS::getDelayExp(Expression *e){
  Expression* delay =
    new LoadSignalExpression(NULL,
                             new SignalSymbol("tick",NULL,AST::SignalSymbol::Input,NULL,
    NULL));
  SortedSymbolTable signalTable;
  getUsedSymbols(e,signalTable);
  for(unsigned int i = 0; i < signalTable.size(); i++) {
    if(i==0) delay = new LoadSignalExpression(NULL,(SignalSymbol*)signalTable.symbols[i]);
    else delay =
      new BinaryOp(NULL,
                   "or",
                   delay,
                   new LoadSignalExpression(NULL,(SignalSymbol*)signalTable.symbols[i]));
  }
  return delay;
}

/* Calclates the minterms of a function given in expression 'e' and writes them to the
 * output file stream 'ofs'
 * @return -1 if calculation fails
 */
int MVSIS::writeOnSet(AST::Expression* e, std::ostream& os) {
  int z,n,numberOfOnes;
  SortedSymbolTable usedSymbols;

  getUsedSymbols(e,usedSymbols);
  if(usedSymbols.size() > 20) return -1;    // too many symbols
  for (int minterm=0; minterm < pow(2,usedSymbols.size()); minterm++) {
    z=minterm;
    numberOfOnes = 0;
    char* term = new char[usedSymbols.size()+1];
    term[usedSymbols.size()]='\0';
    for (int x=usedSymbols.size()-1; x>=0; x--) {
      n=z%2;
      z=z/2;
      if(n==0) {
        term[x]='0';
      }
      else {
        term[x]='1';
        numberOfOnes++;
      }
    }

    if(evaluate(e, usedSymbols, term)) {
      os << term << " 1\n";
    }
  }
  return 0;
}


/*
 * Writes all expressions in the hw-module to a file in blif format using the following
    pattern:
 * - sustain s(e): s is the functions name and e the function
 * - every e do emit s end: s is the function name and e is the function
 *
 * @param m The hw-module
 * @param file The destination file name
 * @return Returns -1 if writing fails, 0 otherwise
*/
int MVSIS::writeBlif(AST::Module* m ,std::string file){
  std::ostringstream inputs;
  std::ostringstream outputs;
  std::ostringstream truthtable;
  SortedSymbolTable inputSymbols;

  AST::ParallelStatementList* psl;
  AST::StatementList* ssl;
  AST::Every* est;
  AST::Sustain* sust;

  // bring hw.module into a uniform structure
  // (this is necessary when the hw-module has just one thread)
  if ((psl=dynamic_cast<AST::ParallelStatementList*>(m->body))) ;
  else if ((est=dynamic_cast<AST::Every*>(m->body))) {
    psl=new AST::ParallelStatementList();
    psl->threads.push_back(est);
    m->body=psl;
  }
  else if ((sust=dynamic_cast<AST::Sustain*>(m->body))) {
    psl=new AST::ParallelStatementList();
    psl->threads.push_back(ssl);
    m->body=psl;
  }
  else if (dynamic_cast<AST::VariableSymbol*>(m->body)) ;
  else throw MVSISError("MVSISError: Error in HWModule structure");

  // write combinatorial logic functions
  for(unsigned int x=0; x < psl->threads.size(); x++) {
    // i.e. if signal expression is calculated
    if ((est = dynamic_cast<AST::Every*>(psl->threads[x]))){
      AST::Delay* de = dynamic_cast<AST::Delay*>(est->predicate);

      SortedSymbolTable usedSymbols;
      AST::Expression *e = de->predicate;
      getUsedSymbols(e, usedSymbols);

      // declare the new function ...
      truthtable << ".names ";
      for(unsigned int y=0;y<usedSymbols.size();y++){
        // ... by first writing a list of parameters ...
        truthtable << dynamic_cast<AST::SignalSymbol*>(*(usedSymbols.symbols.begin()+y))->
    name
                   << " ";
        if (!inputSymbols.local_contains(
```

```
          dynamic_cast<AST::SignalSymbol*>(*(usedSymbols.symbols.begin()+y))->name)) {
        inputSymbols.enter(dynamic_cast<AST::SignalSymbol*>(*(usedSymbols.symbols.begin()+y
))));
      }
    }
    // ... followed by the function name
    truthtable << dynamic_cast<AST::Emit*>(est->body)->signal->name;
    // declare function name as output
    outputs << dynamic_cast<AST::Emit*>(est->body)->signal->name << " ";
    // write the truthtable of the function
    truthtable << "\n";
    if (writeOnSet(e, truthtable) == -1) return -1;
  }
  // if data expression is calculated
  else {
    sust = dynamic_cast<AST::Sustain*>(psl->threads[x]);

    SortedSymbolTable usedSymbols;
    AST::Expression *e = sust->value;
    getUsedSymbols(e, usedSymbols);

    // declare new function ...
    truthtable << ".names ";
    // ... add parameter names beginning with an underscore to identify data expressions
during
    // reading ...
    for(unsigned int y=0;y<usedSymbols.size();y++){
      SignalSymbol* argument = dynamic_cast<AST::SignalSymbol*>(*(usedSymbols.symbols.begin
()+y));
      SignalSymbol* boolArgument =
        new SignalSymbol("_"+argument->name,
                         argument->type,
                         (AST::SignalSymbol::kinds)argument->kind,
                         NULL,
                         NULL);
      truthtable << boolArgument->name << " ";
      if (!inputSymbols.local_contains(boolArgument->name)) {
        inputSymbols.enter(boolArgument);
      }
    }
    // ... followed by the fuction name (also beginning with an underscore)
    truthtable << "_" << sust->signal->name;
    outputs << "_" << sust->signal->name << " ";
    truthtable << "\n";
    if (writeOnSet(e, truthtable) == -1) return -1;
  }
}


for(unsigned int y=0;y<inputSymbols.size();y++){
  inputs << dynamic_cast<AST::SignalSymbol*>(*(inputSymbols.symbols.begin()+y))->name << " "
  ";
}


// Open output file and write data
  std::ofstream ofs(file.c_str());
  if(!ofs) throw MVSISError("MVSISError: Can't open output file: "+file);
  ofs << ".model " << file << "\n";
  ofs << ".inputs ";
  ofs << inputs.str();
  ofs << "\n.outputs ";
  ofs << outputs.str();
  ofs << "\n" << truthtable.str();
  ofs << "\n.end\n";
  ofs.flush();
  ofs.close();
  return 0;
}

void MVSIS::readBlif(Module* m,std::string file){
  std::string line,longline,token;
  SortedSymbolTable funcArgs;
  AST::SignalSymbol *localSignal, *funcSymbol;
  AST::Expression *expression, *term, *var;
  AST::LoadVariableExpression *constTrue,*constFalse;
  bool isPure = true;           // true if function is pure

  AST::ParallelStatementList* body = new AST::ParallelStatementList();
  AST::Signal *scope = NULL;

  // Declare constants true and false
  constTrue = new LoadVariableExpression(
              new ConstantSymbol(
                                 "true",
                                 new TypeSymbol("boolean"),
                                 new Literal("1",new TypeSymbol("boolean"))
                                 ));
  constFalse = new LoadVariableExpression(
               new ConstantSymbol(
                                  "false",
                                  new TypeSymbol("boolean"),
                                  new Literal("0",new TypeSymbol("boolean"))
                                  )
               );

  // open input file
  std::ifstream ifs(file.c_str());
  if(!ifs) throw MVSISError("MVSISError: Can't open input file: "+file);

  std::getline(ifs,line,'\n');                                  // read first input line
  // while end of file isn't reached
  while((line.compare(".end")!=0)&&(line.compare(".e")!=0)){
    // reassemble wrapped lines
    while (line[line.size()-1]=='\\') {
      longline = line;
      std::getline(ifs,line,'\n');
      longline.replace(longline.size()-1,longline.size()-1,line);
      line = longline;
    }
    std::istringstream lineStr(line);
```

```
lineStr >> token;                                    // read first token from input
 line

// read in function declaration (parameters and name)
if (token.compare(".names")==0) {
  while(lineStr >> token){
    if(token[0]=='_') {
      isPure = false;
      token = token.erase(0,1);
      std::cerr << token << "\n";
    }
    // if signal is not local
    if (!((token[0]=='[')&&(token[token.size()-1]==']'))){
      if (m->signals->local_contains(token)) {
        // the last token is the function name
        if(!lineStr.eof()) {
          funcArgs.enter(m->signals->get(token));
          SignalSymbol *s;
          if ((s = dynamic_cast<AST::SignalSymbol*>(m->signals->get(token)))->kind ==
              SignalSymbol::Output) s->kind = SignalSymbol::Inputoutput;
        }
        else funcSymbol = dynamic_cast<AST::SignalSymbol*>(m->signals->get(token));
      } else {
        funcArgs.enter(scope->symbols->get(token));
      }
    }
    // else (if signal is local i.e. if a common subexpression was extracted)
    else {
      if(scope==NULL) {
        scope = new AST::Signal();
        scope->symbols = new AST::SymbolTable();
        scope->body=body;
      }
      // convert signal name into a valid esterel signal name
      token.erase(0,1);
      token.erase(token.size()-1,1);
      token="temp_"+token;
      // make signal name unique within the module
      std::string postfix="";
      int x=0;
      while(m->signals->local_contains(token+postfix)){
        std::ostringstream s;
        s << "_" << x;
        postfix = s.str();
        x++;
      }
      token+=postfix;

      if(scope->symbols->local_contains(token)) {
        localSignal = (SignalSymbol*) scope->symbols->get(token);
      }   else {
        if(isPure) {
          localSignal =
            new AST::SignalSymbol(token,NULL,AST::SignalSymbol::Local,NULL,NULL);
        } else {
          localSignal =
            new AST::SignalSymbol(token,new TypeSymbol("boolean"),AST::SignalSymbol::
Local,NULL,NULL);
        }
        // if not already included, add to 'signal'-statement
        if (!scope->symbols->local_contains(token)){scope->symbols->enter(localSignal);}
      }
      // the last signal name is the fuction name
      if(!lineStr.eof()) {
        funcArgs.enter(localSignal);
      }
      else funcSymbol = localSignal;
    }
  }
  if(scope==NULL) {
    m->body=body;
  } else {
    m->body=scope;
  }

  // if function is constant
  if (funcArgs.size()==0) {
    std::getline(ifs,line,'\n');  // get first cube from truth table
    if(line.compare("1")==0) {      // if function is constant false
      if(funcSymbol->type == NULL){ // if function is signal expression
        body->threads.push_back(new AST::Sustain(funcSymbol,NULL));
      } else {                      // if function is boolean
        body->threads.push_back(new AST::Sustain(funcSymbol,constTrue));
      }
      std::getline(ifs,line,'\n');
    }
    else {                          // if function is constant true
      if(!funcSymbol->type == NULL){// if function is signal expression
        body->threads.push_back(new AST::Sustain(funcSymbol,constFalse));
      }
      std::getline(ifs,line,'\n');
    }
  }
  // else build SoP from the truth table
  else {
    std::getline(ifs,line,'\n');  // get first cube from truth table
    int x=0;
  // while there are cubes on the truth table of the current function
  while((line.compare(".end")!=0)
        &&(line.compare(".e")!=0)
        &&(line.find(".names")==std::string::npos))
    {
      term=NULL;
      for (int pos=0; abs(pos)<funcArgs.size();pos++) {
        var=NULL;
        // if parameter is negated for this product term
        if(line[pos]=='0') {
          if (funcSymbol->type == NULL) { // if function is signal expression
            var=new AST::UnaryOp(NULL,
                                 "not",
                                 new AST::LoadSignalExpression(
```

```
                                        NULL,(AST::SignalSymbol*)*(funcArgs.symbols.begin()+pos))
                                );
          } else {                        // if function is valued expression
            var=new AST::UnaryOp(NULL,
                                "not",
                                new AST::LoadSignalValueExpression(
                                    (AST::SignalSymbol*)*(funcArgs.symbols.begin()+pos))
                                );
          }
        }
        // if parameter is negated for this product term
        if(line[pos]=='1') {
          if (funcSymbol->type == NULL) {  // if function is signal expression
            var=new AST::LoadSignalExpression(
                        NULL,(AST::SignalSymbol*)*(funcArgs.symbols.begin()+pos)
                                            );
          }
          else {                           // if function is valued expression
            var=new AST::LoadSignalValueExpression(
                        (AST::SignalSymbol*)*(funcArgs.symbols.begin()+pos)
                                                );
          }
        }

        // add parameter to the product term
        if (var!=NULL){
          if(!((dynamic_cast<AST::BinaryOp*>(term))||
               (dynamic_cast<AST::LoadSignalExpression*>(term))||
               (dynamic_cast<AST::LoadSignalValueExpression*>(term))||
               (dynamic_cast<AST::UnaryOp*>(term))))
            {
              term = var;
            } else {
            term = new AST::BinaryOp(NULL,"and",term,var);
            }
          }

      }
      // sum up terms
      if(x==0) {
        expression = term;
      }
      else {
        expression = new AST::BinaryOp(NULL,"or",expression,term);
      }
      std::getline(ifs,line,'\n');
      x++;
    }
  // if function symbol is pure
  if(funcSymbol->type == NULL){
    body->threads.push_back(
       new AST::Every(new AST::Emit(funcSymbol,NULL),
                      new AST::Delay(NULL,expression,NULL,true,NULL))
       );
  }
        // if function symbol is valued
        else {
          Sustain* sustain = new Sustain(funcSymbol,expression);
          body->threads.push_back(sustain);
        }

        funcArgs.clear();
        isPure = true;
        }
      }
      else std::getline(ifs,line,'\n');
    }
  }

  /*
   * Performs a multi-level logic minimization on each hw-module
   * @param csd
   * @param filename_prefix A prefix to name the blif files
   */
  void MVSIS::minimize(CoDesign::CoDesignData* csd, std::string filename_prefix){
    // Get the module list constituting the partitioned program from the shared data structure
    AST::Modules modules = csd->convModules;
    AST::Module* m;
    std::vector<AST::Module*>::iterator i;
    // iterate over HW-modules
    for(i = modules.modules.begin();i != (modules.modules.end()-1); i++) {
      m = dynamic_cast<AST::Module*>(*i);
      // write hw-module to blif file
      if (writeBlif(m ,filename_prefix+".blif") == 0) {

        // call mvsis in order to minimize the blif file via the execvp system call
        pid_t cpid;
        int status;
        char* args[]=
        {"mvsis\0","-t\0","blif\0","-T\0","blif\0","-c\0","fullsimp;fxu\0",
       "-o\0","\0","\0",NULL};
        std::string mvsisExe="mvsis";
        std::string fn1,fn2,pwd;

        pwd = getenv("PWD");
        pwd += "/";
        fn1 = pwd+filename_prefix+".blif";
        fn2 = pwd+filename_prefix+"_min.blif";

        args[8] = const_cast<char*>(fn2.c_str());
        args[9] = const_cast<char*>(fn1.c_str());

        switch (cpid = fork())
        {
        case -1:
          throw MVSISError("MVSISError: Execution of mvsis failed");
        case  0:
          execvp(mvsisExe.c_str(),args);
          break;
        default:
          if (waitpid(cpid, &status, 0) == -1) throw
```

```
                readBlif(m,filename_prefix+"_min.blif");
              }
              else std::cerr << "Module " << m->symbol->name << " not minimized.\n";
            }
            std::cerr << "minimize() finished.\n";
          }
        }
      MVSISError("MVSISError: Execution of mvsis failed");
      if (!WIFEXITED(status)) throw
        MVSISError("MVSISError: Execution of mvsis failed. Status is: "+
                                                                        550
    status);
      }
      std::cerr << "MVSIS finished. Reading...\n";
      // read minimized hw-module from blif file
```

A.1.12. HW/SW Synthese

Im zweiten Schritt der Co-Synthese wird aus den Hardwaremodulen des partitionierten ESTEREL Programms eine VHDL Beschreibung des Logikblocks erzeugt. Aus dem Softwaremodul werden zunächst einige überflüssige Statements entfernt (vgl. Abschnitt 3.3.2) bevor es direkt ausgegeben wird.

A.1.13. CoDesign-hwswsynthesis.cpp

```
#include "IR.hpp"
#include "AST.hpp"
#include "CoDesignHWSWPrinter.hpp"
#include <iostream>
#include <stdlib.h>
#include <fstream>

/* This program reads in the AST of the partitioned Esterel program from an XML file and writes
 * the modified Esterel program for sw synthesis to 'stdout' and a VHDL description of the
      logic
 * block for hw synthesis to a file
 * @param name for the VHDL file
 */
int main(int argc, char* argv[])
{
  try {

    IR::XMListream r(std::cin);
    IR::Node *n;
    r >> n;

    if (argc != 2) throw IR::Error("Wrong number of arguments");

    std::ofstream hwOut(argv[1]);
    if (!hwOut) throw IR::Error("Can't open output file");

  AST::ASTNode *an = dynamic_cast<AST::ASTNode*>(n);
  if (!an) throw IR::Error("Root node is not an AST node");

  CoDesign::CoDesignHWSWPrinter p(std::cout,hwOut);
  an->welcome(p);

  hwOut.close();

 } catch (IR::Error &e) {
  std::cerr << e.s << std::endl;
  exit(-1);
 }

 return 0;
}
```

A.1.14. CoDesignHWSWPrinter.hpp

```
#ifndef _CODESIGN_VHDL_PRINTER_HPP
#  define _CODESIGN_VHDL_PRINTER_HPP

#  include "AST.hpp"
#  include <iostream>
#  include <vector>
#  include <map>
#  include <sstream>

namespace CoDesign {

  using std::vector;
  using std::map;
  using std::string;
  using namespace AST;


  class VHDLLogicBlock{
  public:
    VHDLLogicBlock(){};
    ~VHDLLogicBlock(){};
    string name;
    SymbolTable inputSignals;
    SymbolTable valuedInputSignals;
    SymbolTable outputSignals;
    SymbolTable valuedOutputSignals;
    SymbolTable localSignals;
    std::ostringstream implementation;
    friend std::ostream& operator<<(std::ostream &o, const VHDLLogicBlock &lb);
  };

  /* This class is used in the second step for sw- and hw-synthesis
   * Given the AST of the partitioned program to the 'process' function, the sw module is
   * prited to the output stream o and the VHDL description of the logic block is printed to
   * the hwModule out stream. During printing the sw-module superfluous statements are omitted.
   * The logic block is generated from the hw-modules.
   *
  */
  class CoDesignHWSWPrinter : public Visitor {
    std::ostream &o;                    // OutputStream for SW Module
    std::ostream &hwModuleOut;          // OutputStream for HW Module

    unsigned int indentlevel;

    std::vector<int> precedence;
    std::map<string, int> level;
    std::map<string, int> unarylevel;

    VHDLLogicBlock vhdlCode;
    bool isSWModule;                    // flag indicates if SW- or HW-Module is currently
      processed,
                                        // for printing the sw-module, the visitor methods are
                                        // similar to the CEC's EsterelPrinter except some
    changes
                                        // for removing some superfluous statements
    SymbolTable *usedSymbols;           // stores all in the Esterel program used symbols
  public:
    CoDesignHWSWPrinter(std::ostream&, std::ostream &);
    virtual ~CoDesignHWSWPrinter() {}
    string getUniqueSigName(string,SymbolTable*);

    using Visitor::visit;               // Bring the Visitor's visit method into scope

    void process(ASTNode* n) { assert(n); n->welcome(*this); }
    void statement(Statement *);
    void expression(Expression *);
    void sigexpression(Expression *);

    static const int sequentialPrecedence = 2;
    static const int parallelPrecedence = 1;

    bool push_precedence(int);
    void pop_precedence();

    Status visit(ModuleSymbol&);
    Status visit(VariableSymbol&);
    Status visit(BuiltinConstantSymbol&);
    Status visit(BuiltinSignalSymbol&);
    Status visit(BuiltinTypeSymbol&);
    Status visit(BuiltinFunctionSymbol&);
    Status visit(TypeRenaming &);
    Status visit(ConstantRenaming &);
    Status visit(FunctionRenaming &);
    Status visit(ProcedureRenaming &);
    Status visit(SignalRenaming &);
    Status visit(PredicatedStatement &);
    Status visit(TaskCall &);

    Status visit(Module&);
    Status visit(Exclusion&);
    Status visit(Implication&);
    Status visit(Modules&);
    Status visit(SymbolTable&);

    Status visit(TypeSymbol&);
    Status visit(ConstantSymbol&);
    Status visit(SignalSymbol&);
    Status visit(FunctionSymbol&);
    Status visit(ProcedureSymbol&);
    Status visit(TaskSymbol&);

    Status visit(StatementList&);
    Status visit(ParallelStatementList&);
```

```
    Status visit(Nothing&);
    Status visit(Pause&);
    Status visit(Halt&);
    Status visit(Emit&);
    Status visit(Sustain&);
    Status visit(Assign&);
    Status visit(StartCounter&);
    Status visit(ProcedureCall&);
    Status visit(Exec&);
    Status visit(Present&);
    Status visit(If&);
    Status visit(Loop&);
    Status visit(Repeat&);
    Status visit(Abort&);
    Status visit(Await&);
    Status visit(LoopEach&);
    Status visit(Every&);
    Status visit(Suspend&);
    Status visit(DoWatching&);
    Status visit(DoUpto&);
    Status visit(Trap&);
    Status visit(Exit&);
    Status visit(Var&);

    Status visit(Signal&);
    Status visit(Run&);

    Status visit(UnaryOp&);
    Status visit(BinaryOp&);
    Status visit(LoadVariableExpression&);
    Status visit(LoadSignalExpression&);
    Status visit(LoadSignalValueExpression&);
    Status visit(Literal&);
    Status visit(FunctionCall&);
    Status visit(Delay&);
    Status visit(CheckCounter&);

    Status visit(IfThenElse&);

    void indent() { indentlevel += 2; }
    void unindent() { indentlevel -= 2; }
    void tab() { for (unsigned int i = 0 ; i < indentlevel ; i++) o << ' '; }
  };
}

#endif
```

A.1.15. CoDesignHWSWPrinter.cpp

```
#include "CoDesignHWSWPrinter.hpp"
#include <cassert>

namespace CoDesign {

  /* This function overloads the stream output operator for writing an object of type
      VHDLLogicBlock
   */
  std::ostream& operator<<(std::ostream &o,const VHDLLogicBlock &lb){
    bool prev = false;

    // include some VHDL libraries
    o << "\nlibrary IEEE;\nuse IEEE.STD_LOGIC_1164.ALL;\nuse IEEE.
      STD_LOGIC_ARITH.ALL;\nuse IEEE.STD_LOGIC_UNSIGNED.ALL;\n\n";
    // declare the logicblock as new entity
    o << "entity " << lb.name << " is\n";
    o << "   port(\n";
    prev = false;
    // declare pure input signals
    for (unsigned int x=0; x<lb.inputSignals.size(); x++){
      if(prev) {prev = false; o << ";\n";}
      o << "      " << (dynamic_cast<Symbol*> (lb.inputSignals.symbols[x]))->name << ": in
       std_logic";
      prev = true;
    }
    // declare boolean input signals as vector of size 2:
    // s[0] is the state, s[1] is the boolean value
    for (unsigned int x=0; x<lb.valuedInputSignals.size(); x++){
      if(prev) {prev = false; o << ";\n";}
      o << "      "
        << (dynamic_cast<Symbol*> (lb.valuedInputSignals.symbols[x]))->name
        << ": in std_logic_vector(1 downto 0)";
      prev = true;
    }
    // declare pure output signals
    for (unsigned int x=0; x<lb.outputSignals.size(); x++){
      if(prev) {prev = false; o << ";\n";}
      o << "      "
        << (dynamic_cast<Symbol*> (lb.outputSignals.symbols[x]))->name
        << ": out std_logic";
      prev = true;
    }
    // declare boolean output signals as vector of size 2:
    // s[0] is the state, s[1] is the boolean value
    for (unsigned int x=0; x<lb.valuedOutputSignals.size(); x++){
      if(prev) {prev = false; o << ";\n";}
      o << "      "
        << (dynamic_cast<Symbol*> (lb.valuedOutputSignals.symbols[x]))->name
        << ": out std_logic_vector(1 downto 0)";
      prev = true;
    }
    o << "\n   );\n";
    o << "end " << lb.name << ";\n\n";
    // write the implementation of the entity
    o << "architecture " << lb.name << "_BEH of " << lb.name << " is\n";
    // write the declaration of local signals
    for (unsigned int x=0; x<lb.localSignals.size(); x++){
      if((dynamic_cast<SignalSymbol*> (lb.localSignals.symbols[x]))->type) {
        o << "   signal "
          << (dynamic_cast<Symbol*> (lb.localSignals.symbols[x]))->name
          << ": std_logic_vector)(1 downto 0);\n";
      }
      else {
        o << "   signal "
          << (dynamic_cast<Symbol*> (lb.localSignals.symbols[x]))->name
          << ": std_logic;\n";
      }
    }
    o << "begin\n";
    // write the functional behaviour
    o << lb.implementation.str();
    o << "end " << lb.name << "_BEH;\n";
    return o;
  }

  /* The VHDL Printer synthesizes a a VHDL Description of the logic block from the hw-modules
   * and writes out a simplified version of the sw-module for execution on the KEP
   * in the HW/SW Co-Design
   *
   * @param oo Output stream for the sw-module
   * @param hwModuleOut Output stream for the Logic Block
  */
  CoDesignHWSWPrinter::CoDesignHWSWPrinter(std::ostream &oo, std::ostream &hwMod) :
    o(oo), hwModuleOut(hwMod), indentlevel(0) {
    precedence.push_back(0);

    usedSymbols = new SymbolTable();
    level["or"] = 1;
    level["and"] = 2;
    unarylevel["not"] = 3;
    unarylevel["pre"] = 3;

    level["="] = 4;
    level["<>"] = 4;
    level["<"] = 4;
    level[">"] = 4;
    level["<="] = 4;
    level[">="] = 4;

    level["+"] = 5;
    level["-"] = 6;

    level["*"] = 7;
    level["/"] = 8;
```

```
  level["mod"] = 8;

  unarylevel["-"] = 9;
}

/* If a symbol with the name 'name' is contained in the Symbol table, the function searches for a
 * numbered postfix that makes 'name' unique.
 *
 * @param name
 * @param st
*/
string CoDesignHWSWPrinter::getUniqueSigName(string name,SymbolTable *st){
  string postfix="";
  int x=0;
  while(st->local_contains(name+postfix)){
    std::ostringstream s;
    s << "_" << x;
    postfix = s.str();
    x++;
  }

  return name+postfix;
}

void CoDesignHWSWPrinter::statement(Statement *s) {
  assert(s);
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << '\n';
    indent();
    tab();
    s->welcome(*this);
    o << '\n';
    unindent();
    tab();
  }
  else {
    s->welcome(*this);
  }
}

void CoDesignHWSWPrinter::expression(Expression *e) {
  precedence.push_back(0);
  process(e);
  precedence.pop_back();
}

void CoDesignHWSWPrinter::sigexpression(Expression *e) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    precedence.push_back(0);
    // write [ ] or not ?
    if (dynamic_cast<Delay*>(e) || dynamic_cast<LoadSignalExpression*>(e))
      process(e);
    else {
      o<<'[';
      process(e);
      o<<']';
    }
    precedence.pop_back();
  }
  else {
    precedence.push_back(0);
    process(e);
    precedence.pop_back();
  }
}

bool CoDesignHWSWPrinter::push_precedence(int p) {
  bool needBrackets =
    (p < precedence.back()) || ((p == precedence.back()) && ((p==level["-"]) || (p==level["/"])));
  precedence.push_back(p);
  return needBrackets;
}

void CoDesignHWSWPrinter::pop_precedence() {
  precedence.pop_back();
}

Status CoDesignHWSWPrinter::visit(ModuleSymbol &) { assert(0); }
Status CoDesignHWSWPrinter::visit(VariableSymbol &) { assert(0); }
Status CoDesignHWSWPrinter::visit(TypeRenaming &) { assert(0); }
Status CoDesignHWSWPrinter::visit(ConstantRenaming &) { assert(0); }
Status CoDesignHWSWPrinter::visit(FunctionRenaming &) { assert(0); }
Status CoDesignHWSWPrinter::visit(ProcedureRenaming &) { assert(0); }
Status CoDesignHWSWPrinter::visit(SignalRenaming &) { assert(0); }
Status CoDesignHWSWPrinter::visit(PredicatedStatement &) { assert(0); }
Status CoDesignHWSWPrinter::visit(TaskCall &) { assert(0); }

Status CoDesignHWSWPrinter::visit(BuiltinConstantSymbol &) { return Status(); }
Status CoDesignHWSWPrinter::visit(BuiltinSignalSymbol &) { return Status(); }
Status CoDesignHWSWPrinter::visit(BuiltinTypeSymbol &) { return Status(); }
Status CoDesignHWSWPrinter::visit(BuiltinFunctionSymbol &) { return Status(); }

Status CoDesignHWSWPrinter::visit(Module &m) {
  assert(m.symbol);
  // if current module is a hw-module
  std::cerr << "Processing module \"" << m.symbol->name << "\"\n";
  if(((m.symbol->name.find("_hw")!=std::string::npos)
      && (m.symbol->name.find("_sw")==std::string::npos))
     || (m.symbol->name.find_last_of("_hw")>m.symbol->name.find_last_of("_sw"))) {
    isSWModule = false;
    assert(m.signals);
    process(m.signals);        // add interface signals to the logic block
    assert(m.body);
    process(m.body);           // convert body to combinatorial functions
  }
  // if current module is the sw module use the original CEC Printers method
  if(((m.symbol->name.find("_sw")!=std::string::npos)
      && (m.symbol->name.find("_hw")==std::string::npos))
```

```
     || (m.symbol->name.find_last_of("_sw")>m.symbol->name.find_last_of("_hw"))) {
    isSWModule = true;
    assert(m.symbol);
    tab();
    o << "module " << m.symbol->name << ":\n";
    process(m.types);
    process(m.constants);
    process(m.functions);
    process(m.procedures);
    process(m.tasks);
    process(m.signals);
    for ( vector<InputRelation*>::const_iterator i = m.relations.begin() ;
          i != m.relations.end() ; i++ ) {
      assert(*i);
      process(*i);
    }
    o << '\n';
    tab();
    process(m.body);
    o << "\n\n";
    tab();
    o << "end module\n";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Modules &m) {
  vector<Module*>::iterator i = m.modules.begin();
  // print the sw-module first
  while ( i != m.modules.end() ) {
    assert(*i);
    assert((*i)->symbol);
    if((((*i)->symbol->name.find("_sw")!=std::string::npos)
        && ((*i)->symbol->name.find("_hw")==std::string::npos))
       || ((*i)->symbol->name.find_last_of("_sw")>(*i)->symbol->name.find_last_of("_hw")))
      {
        // take the sw-modules name without the postfix '_sw' as name for the logic
        vhdlCode.name = (*i)->symbol->name.substr(0,(*i)->symbol->name.find_last_of("_sw")
     -2);
        process(*i);
      }
    i++;
  }

  i = m.modules.begin();
  // convert hw-modules to VHDL code of the logic block
  while ( i != m.modules.end() ) {
    assert(*i);
    assert((*i)->symbol);
    if((((*i)->symbol->name.find("_hw")!=std::string::npos)
        && ((*i)->symbol->name.find("_sw")==std::string::npos))
       || ((*i)->symbol->name.find_last_of("_hw")>(*i)->symbol->name.find_last_of("_sw")))
      process(*i);
    i++;
    if ( i != m.modules.end() ) o << '\n';
  }

  hwModuleOut << vhdlCode;         // print the VHDL code of the logic block
  return Status();
}

Status CoDesignHWSWPrinter::visit(Exclusion& e) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "relation ";
    vector<SignalSymbol*>::const_iterator i = e.signals.begin();
    while ( i != e.signals.end() ) {
      assert(*i);
      o << (*i)->name;
      i++;
      if ( i != e.signals.end() ) o << " # ";
    }
    o << ";\n";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Implication& e) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    assert(e.predicate);
    assert(e.implication);
    o << "relation ";
    o << e.predicate->name << " => " << e.implication->name << ";\n";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(SymbolTable &t) {
  for ( SymbolTable::const_iterator i = t.begin() ; i != t.end() ; i++ ) {
    if (*i != NULL) {
      assert(*i);
      process(*i);}assert(*i);
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(TypeSymbol &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "type " << s.name << ";\n";
  }
  // collect used symbols
  if(!usedSymbols->local_contains(s.name)) usedSymbols->enter(&s);
  return Status();
}

Status CoDesignHWSWPrinter::visit(ConstantSymbol &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "constant " << s.name;
    if (s.initializer) {
      o << " = ";
```

```
      expression(s.initializer);
    }
    assert(s.type);
    o << "␣:␣" << s.type->name << ";\n";
  }
  // collect used symbols
  if(!usedSymbols->local_contains(s.name)) usedSymbols->enter(&s);
  return Status();
}

Status CoDesignHWSWPrinter::visit(SignalSymbol &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    switch (s.kind) {
    case SignalSymbol::Input: o << "input"; break;
    case SignalSymbol::Output: o << "output"; break;
    case SignalSymbol::Inputoutput: o << "inputoutput"; break;
    case SignalSymbol::Sensor: o << "sensor"; break;
    case SignalSymbol::Return: o << "return"; break;
    default: assert(0); break;
    }
    o << ' ' << s.name;
    if (s.initializer) {
      o << "␣:=␣";
      expression(s.initializer);
      assert(s.type);
    }
    if (s.type) {
      o << "␣:␣";
      if (s.combine) o << "combine␣";
      o << s.type->name;
      if (s.combine) o << "␣with␣" << s.combine->name;
    }
    o << ";\n";
  }
  else {
    // add signal declaration to the interface of the logic block
    switch (s.kind) {
    case SignalSymbol::Input:
      if (!s.type) {
        if(!vhdlCode.inputSignals.local_contains(s.name))
          vhdlCode.inputSignals.enter(&s);
      } else {
        if(!vhdlCode.valuedInputSignals.local_contains(s.name))
          vhdlCode.valuedInputSignals.enter(&s);
      }
      break;
    case SignalSymbol::Output:
      if (!s.type) {
        if(!vhdlCode.outputSignals.local_contains(s.name))
          vhdlCode.outputSignals.enter(&s);
      } else {
        if(!vhdlCode.valuedOutputSignals.local_contains(s.name))
          vhdlCode.valuedOutputSignals.enter(&s);
      }
      break;
    case SignalSymbol::Inputoutput:
      if (!s.type) {
        if(!vhdlCode.outputSignals.local_contains(s.name))
          vhdlCode.outputSignals.enter(&s);
      } else {
        if(!vhdlCode.valuedOutputSignals.local_contains(s.name))
          vhdlCode.valuedOutputSignals.enter(&s);
      }
      break;
    default: assert(0); break;
    }
  }
  // collect used symbols
  if(!usedSymbols->local_contains(s.name)) usedSymbols->enter(&s);
  return Status();
}

Status CoDesignHWSWPrinter::visit(FunctionSymbol &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "function␣" << s.name << '(';
    vector<TypeSymbol*>::const_iterator i = s.arguments.begin();
    while (i != s.arguments.end()) {
      assert(*i);
      o << (*i)->name;
      i++;
      if (i != s.arguments.end()) o << ",␣";
    }
    assert(s.result);
    o << ")␣:␣" << s.result->name << ";\n";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(ProcedureSymbol &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "procedure␣" << s.name << '(';
    vector<TypeSymbol*>::const_iterator i = s.reference_arguments.begin();
    while (i != s.reference_arguments.end()) {
      assert(*i);
      o << (*i)->name;
      i++;
      if (i != s.reference_arguments.end()) o << ",␣";
    }
    o << ")(";
    i = s.value_arguments.begin();
    while (i != s.value_arguments.end()) {
      assert(*i);
      o << (*i)->name;
      i++;
      if (i != s.value_arguments.end()) o << ",␣";
    }
    o << ");\n";
  }
  return Status();
```

```
}

Status CoDesignHWSWPrinter::visit(TaskSymbol &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "task " << s.name << '(';
    vector<TypeSymbol*>::const_iterator i = s.reference_arguments.begin();
    while (i != s.reference_arguments.end()) {
      assert(*i);
      o << (*i)->name;
      i++;
      if (i != s.reference_arguments.end()) o << ", ";
    }
    o << ")(";
    i = s.value_arguments.begin();
    while (i != s.value_arguments.end()) {
      assert(*i);
      o << (*i)->name;
      i++;
      if (i != s.value_arguments.end()) o << ", ";
    }
    o << ");\n";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(StatementList &l) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    push_precedence(sequentialPrecedence);
    vector<Statement*>::const_iterator i = l.statements.begin();
    while ( i != l.statements.end() ) {
      process(*i);
      i++;
      if (i != l.statements.end() ) { o << ";\n"; tab(); }
    }
    pop_precedence();
  } else {
    // deprecated: a hw modules body should be a ParallelStatemntList or a single
    // every or sustain statement
    push_precedence(sequentialPrecedence);
    vector<Statement*>::const_iterator i = l.statements.begin();
    while ( i != l.statements.end() ) {
      process(*i);
      i++;
    }
    pop_precedence();
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(ParallelStatementList &l) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    bool needBrackets = push_precedence(parallelPrecedence);
    if (needBrackets) {
      o << "[\n";
      tab();
    }

    vector<Statement*>::const_iterator i = l.threads.begin();
    while ( i != l.threads.end() ) {
      indent();
      o << "  "; // indent doesn't take effect until the next tab()
      process(*i);
      unindent();
      i++;
      if (i != l.threads.end() ) { o << '\n'; tab(); o << "||\n"; tab(); }
    }

    if (needBrackets) {
      o << '\n';
      tab();
      o << ']';
    }

    pop_precedence();
  }
  else {
    // the common body of a hw module is a ParallelStatementList consisting
    // of sustain and every statements
    vector<Statement*>::const_iterator i = l.threads.begin();
    while ( i != l.threads.end() ) {
      process(*i);
      i++;
    }

  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Nothing &) { o << "nothing"; return Status(); }
Status CoDesignHWSWPrinter::visit(Pause &) { o << "pause"; return Status(); }
Status CoDesignHWSWPrinter::visit(Halt &) { o << "halt"; return Status(); }

Status CoDesignHWSWPrinter::visit(Emit &e) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    assert(e.signal);
    o << "emit " << e.signal->name;
    if (e.value) {
      o << '(';
      expression(e.value);
      o << ')';
    }
  }
  else {
    // NOTE: For HW-Modules the emit statement is handled in the every visitor
  }
  return Status();
}
```

```
Status CoDesignHWSWPrinter::visit(StartCounter &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    assert(s.counter);
    o << "StartCounter ";
    expression(s.count);
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Sustain &e) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    assert(e.signal);
    o << "sustain " << e.signal->name;
    if (e.value) {
      o << '(';
      expression(e.value);
      o << ')';
    }
  }
  // NOTE: For HW-Modules the sustain visitor is called for boolean signals only!
  else {
    assert(e.signal);
    if (e.value) {
      vhdlCode.implementation << "   ";
      // if a signal is an output of the logic but also accessed internally,
      // it is declared as input/output. While accessing an output signal in esterel is
     allowed,
      // this is not allowed in VHDL. So a new local VHDL signal is introduced,
      // the combinatorial function is assigned to the local signal and finally assign the
     local
      // signal to the sustained signal
      if(e.signal->kind == SignalSymbol::Inputoutput) {
        vhdlCode.implementation << getUniqueSigName(e.signal->name + "_tmp",usedSymbols);
        vhdlCode.implementation << "(1) <= ";
        expression(e.value);
        vhdlCode.implementation << ";\n";
        vhdlCode.implementation << "   ";
        vhdlCode.implementation << e.signal->name;
        vhdlCode.implementation << " <= ";
        vhdlCode.implementation << getUniqueSigName(e.signal->name + "_tmp",usedSymbols) <<
       ";\n";
        vhdlCode.localSignals.enter(new SignalSymbol(
                                    getUniqueSigName(e.signal->name + "_tmp",usedSymbols)
     ,
                                    e.signal->type, SignalSymbol::Local,NULL,NULL));
      } else {
        // if the sustained signal is an output signal
        vhdlCode.implementation << e.signal->name;
        vhdlCode.implementation << "(1) <= ";
        expression(e.value);
        vhdlCode.implementation << ";\n";
      }
    }
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Assign &a) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    assert(a.variable);
    assert(a.value);
    o << a.variable->name << " := ";
    expression(a.value);
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(ProcedureCall &c) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    assert(c.procedure);
    o << "call " << c.procedure->name << '(';
    vector<VariableSymbol*>::const_iterator i = c.reference_args.begin();
    while (i != c.reference_args.end()) {
      o << (*i)->name;
      i++;
      if (i != c.reference_args.end()) o << ", ";
    }
    o << ")(";
    vector<Expression*>::const_iterator j = c.value_args.begin();
    while (j != c.value_args.end()) {
      expression(*j);
      j++;
      if (j != c.value_args.end()) o << ", ";
    }
    o << ')';
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Exec &e) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "exec";
    indent();
    for ( vector<TaskCall*>::const_iterator i = e.calls.begin() ;
          i != e.calls.end() ; i++) {
      assert(*i);
      assert((*i)->procedure);
      o << '\n';
      tab();
      o << "case " << (*i)->procedure->name << '(';
      vector<VariableSymbol*>::const_iterator k = (*i)->reference_args.begin();
      while (k != (*i)->reference_args.end()) {
        o << (*k)->name;
        k++;
        if (k != (*i)->reference_args.end()) o << ", ";
      }
```

```
      o << ")(";
      vector<Expression*>::const_iterator j = (*i)->value_args.begin();
      while (j != (*i)->value_args.end()) {
        expression(*j);
        j++;
        if (j != (*i)->value_args.end()) o << ",␣";
      }
      assert((*i)->signal);
      o << ")␣return␣" << (*i)->signal->name;
      if ((*i)->body) {
        o << "␣do\n";
        indent();
        tab();
        process((*i)->body);
        unindent();
      }
    }
    o << '\n';
    unindent();
    tab();
    o << "end␣exec";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Present &p) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "present␣";
    if (p.cases.size() == 1) {
      // Simple if-then-else
      PredicatedStatement *ps = p.cases[0];
      assert(ps);
      sigexpression(ps->predicate);
      o << ' ';
      if (ps->body) {
        o << "then";
        statement(ps->body);
      }
    } else {
      // Multiple cases
      indent();
      for (vector<PredicatedStatement*>::const_iterator i = p.cases.begin() ;
           i != p.cases.end() ; i++) {
        o << '\n';
        tab();
        o << "case␣";
        assert(*i);
        assert((*i)->predicate);
        sigexpression((*i)->predicate);
        if ((*i)->body) {
          o << "␣do\n";
          indent();
          tab();
          process((*i)->body);
          unindent();
        }
      }
      unindent();
      o << '\n';
      tab();
    }
    if (p.default_stmt) {
      o << "else";
      statement(p.default_stmt);
    }
    o << "end␣present";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(If &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "if␣";
    vector<PredicatedStatement*>::const_iterator i = s.cases.begin();
    assert(i != s.cases.end());
    expression((*i)->predicate);
    o << ' ';
    if ((*i)->body) {
      o << "then";
      statement((*i)->body);
    }
    for ( i++ ; i != s.cases.end() ; i++ ) {
      assert(*i);
      o << "elsif␣";
      expression((*i)->predicate);
      o << "␣then";
      statement((*i)->body);
    }
    if (s.default_stmt) {
      o << "else";
      statement(s.default_stmt);
    }
    o << "end␣if";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Loop &l) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "loop";
    statement(l.body);
    o << "end␣loop";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Repeat &r) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
```

```
    if (r.is_positive) o << "positive␣";
    o << "repeat␣";
    expression(r.count);
    o << "␣times";
    statement(r.body);
    o << "end␣repeat";
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Abort &a) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    if (a.is_weak) o << "weak␣";
    o << "abort";
    statement(a.body);
    o << "when";
    if (a.cases.size() == 1) {
      // Simple abort condition
      PredicatedStatement *ps = a.cases[0];
      o << ' ';
      assert(ps);
      sigexpression(ps->predicate);
      if (ps->body) {
        o << "␣do";
        statement(ps->body);
        goto PrintEnd;
      }
    } else {
      // Abort cases
      indent();
      for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
            i != a.cases.end() ; i++ ) {
        o << '\n';
        tab();
        assert(*i);
        o << "case␣";
        sigexpression((*i)->predicate);
        if ((*i)->body) {
          o << "␣do\n";
          indent();
          tab();
          process((*i)->body);
          unindent();
        }
      }
      unindent();
      o << '\n';
      tab();
    PrintEnd:
      o << "end";
      if (a.is_weak) o << "␣weak";
      o << "␣abort";
    }
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Await& a) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "await␣";
    if (a.cases.size() == 1) {
      // Simple abort condition
      PredicatedStatement *ps = a.cases[0];
      assert(ps);
      sigexpression(ps->predicate);
      if (ps->body) {
        o << "␣do";
        statement(ps->body);
        o << "end␣await";
      }
    } else {
      indent();
      for ( vector<PredicatedStatement*>::const_iterator i = a.cases.begin() ;
            i != a.cases.end() ; i++ ) {
        o << '\n';
        tab();
        assert(*i);
        o << "case␣";
        sigexpression((*i)->predicate);
        if ((*i)->body) {
          o << "␣do\n";
          indent();
          tab();
          process((*i)->body);
          unindent();
        }
      }
      unindent();
      o << '\n';
      tab();
      o << "end␣await";
    }
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(LoopEach &l) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "loop";
    statement(l.body);
    o << "each␣";
    sigexpression(l.predicate);
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Every &e) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
```

```
    o << "every ";
    sigexpression(e.predicate);
    o << " do";
    statement(e.body);
    o << "end every";
  }
  else {
    std::cerr << "every\n";
    Emit *est = dynamic_cast<Emit*>(e.body);
    assert(est);
    // if a signal is an output of the logic but also accessed internally,
    // it is declared as input/output. While accessing an output signal in esterel is allowed
    ,
    // this is not allowed in VHDL. So a new local VHDL signal is introduced,
    // the combinatorial function is assigned to the local signal and finally the local
    // signal is assigned to the emitted signal
    if(est->signal->kind == SignalSymbol::Inputoutput) {
      vhdlCode.implementation << "   ";
      vhdlCode.implementation << getUniqueSigName(est->signal->name + "_tmp",usedSymbols);
      vhdlCode.implementation << " <= ";
      sigexpression(e.predicate);
      vhdlCode.implementation << ";\n";
      vhdlCode.implementation << "   ";
      vhdlCode.implementation << est->signal->name;
      vhdlCode.implementation << " <= ";
      vhdlCode.implementation << getUniqueSigName(est->signal->name + "_tmp",usedSymbols)
                              << ";\n";
      vhdlCode.localSignals.enter(new SignalSymbol(
                                  getUniqueSigName(est->signal->name + "_tmp",usedSymbols
    ),
                                  est->signal->type, SignalSymbol::Local,NULL,NULL));
    } else {
      // if the emitted signal is an output signal
      vhdlCode.implementation << "   " << est->signal->name << " <= ";
      sigexpression(e.predicate);
      vhdlCode.implementation << ";\n";
    }

  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Suspend &s) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "suspend";
    statement(s.body);
    o << "when ";
    sigexpression(s.predicate);
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(DoWatching &d) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "do";
    statement(d.body);
    o << "watching ";
    sigexpression(d.predicate);
    //    o << ']';
    if (d.timeout) {
      o << " timeout";
      statement(d.timeout);
      o << "end timeout";
    }
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(DoUpto &d) {
  // for printing the sw-module, the original CEC EsterelPrinter method is used
  if (isSWModule) {
    o << "do";
    statement(d.body);
    o << "upto ";
    sigexpression(d.predicate);
  }
  return Status();
}

Status CoDesignHWSWPrinter::visit(Trap &t) {
  if (isSWModule) {
    assert(t.symbols);
    SymbolTable *tst = t.symbols;
    SignalSymbol *ts = dynamic_cast<SignalSymbol*>(*(tst->begin()));
    assert(ts);
    assert(ts->kind == SignalSymbol::Trap);
    // if the trap was created during the partitionig ..
    if ((ts->name.substr(0,10)=="COSYN_TRAP")) {
      ParallelStatementList *psl = dynamic_cast<ParallelStatementList*>(t.body);
      assert(psl);
      // .. remove run statement that runs the hw-module
      assert(dynamic_cast<Run*>(*(psl->threads.begin())));
      psl->threads.erase(psl->threads.begin());
    }
    // if no pre operators were substituted in this scope ...
    if ((ts->name.substr(0,10)=="COSYN_TRAP") && (ts->name.substr(0,14)!="
    COSYN_TRAP_PRE")) {
      // ... remove the trap
      statement(t.body);
    }
    else {
      // ... otherwise keep the trap and print it as common
      o << "trap ";
      SymbolTable::const_iterator i = t.symbols->begin();
      while ( i != t.symbols->end() ) {
        Symbol *s = *i;
        assert(s);
        SignalSymbol *ts = dynamic_cast<SignalSymbol*>(s);
        assert(ts);
        assert(ts->kind == SignalSymbol::Trap);
```

```
      o << ts->name;
      if (ts->initializer) {
        o << "␣:=␣";
        expression(ts->initializer);
      }
      if (ts->type) {
        o << "␣:␣" << ts->type->name;
      }
      i++;
      if ( i != t.symbols->end() ) {
        o << ",\n";
        tab();
        o << "␣␣␣␣␣";
      }
    }
    o << "␣in";
    statement(t.body);
    for (vector<PredicatedStatement*>::const_iterator i = t.handlers.begin() ;
         i != t.handlers.end() ; i++ ) {
      assert(*i);
      o << "handle␣";
      expression((*i)->predicate);
      o << "␣do";
      statement((*i)->body);
    }
    o << "end␣trap";
  }
 }
 return Status();
}

Status CoDesignHWSWPrinter::visit(Exit &e) {
 if (isSWModule) {
  // don't print the exit statement if it was introduced during partitioning
  // and no pre operators were substituted in this scope
  if (!((e.trap->name.substr(0,10)=="COSYN_TRAP")
        && (e.trap->name.substr(0,14)!="COSYN_TRAP_PRE"))) {
   o << "exit␣";
   assert(e.trap);
   assert(e.trap->kind == SignalSymbol::Trap);
   o << e.trap->name;
   if (e.value) {
    o << '(';
    expression(e.value);
    o << ')';
   }
  }
 }
 return Status();
}

Status CoDesignHWSWPrinter::visit(Var &v) {
 // for printing the sw-module, the original CEC EsterelPrinter method is used
 if (isSWModule) {
  o << "var␣";
  SymbolTable::const_iterator i = v.symbols->begin();
  while ( i != v.symbols->end() ) {
   Symbol *s = *i;
   assert(s);
   VariableSymbol *vs = dynamic_cast<VariableSymbol*>(s);
   assert(vs);
   o << vs->name;
   if (vs->initializer) {
    o << "␣:=␣";
    expression(vs->initializer);
   }
   assert(vs->type);
   o << "␣:␣" << vs->type->name;
   i++;
   if ( i != v.symbols->end() ) {
    o << ",\n";
    tab();
    o << "␣␣␣␣";
   }
  }
  o << "␣in";
  statement(v.body);
  o << "end␣var";
 }
 return Status();
}

Status CoDesignHWSWPrinter::visit(Signal &s) {
 // for printing the sw-module, the original CEC EsterelPrinter method is used
 if (isSWModule) {
  o << "signal␣";
  SymbolTable::const_iterator i = s.symbols->begin();
  while ( i != s.symbols->end() ) {
   Symbol *sy = *i;
   assert(sy);
   SignalSymbol *ss = dynamic_cast<SignalSymbol*>(sy);
   assert(ss);
   o << ss->name;
   if (ss->initializer) {
    o << "␣:=␣";
    expression(ss->initializer);
   }
   if (ss->type) {
    o << "␣:␣";
    if (ss->combine) o << "combine␣";
    o << ss->type->name;
    if (ss->combine) o << "␣with␣" << ss->combine->name;
   }
   i++;
   if ( i != s.symbols->end() ) {
    o << ",\n";
    tab();
    o << "␣␣␣␣␣␣␣";
       }
   if(!usedSymbols->local_contains(ss->name)) usedSymbols->enter(ss);
  }
  o << "␣in";
```

```
      statement(s.body);
      o << "end signal";
    }
    else {
      // Enter local signals declared in the hw-module to the local signal declaration of the
       VHDL
      // code. Rename the signal if necessary
      SymbolTable::const_iterator i = s.symbols->begin();
      while ( i != s.symbols->end() ) {
        Symbol *sy = *i;
        assert(sy);
        SignalSymbol *ss = dynamic_cast<SignalSymbol*>(sy);
        ss->name = getUniqueSigName(ss->name,usedSymbols);
        if(!usedSymbols->local_contains(ss->name)) usedSymbols->enter(ss);
        vhdlCode.localSignals.enter(ss);
        i++;
      }
      statement(s.body);
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(Run &r) {
    assert(0);            // run statement shouldn't occur in the sw module
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(UnaryOp &u) {
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    if (isSWModule) {
      assert(unarylevel.find(u.op) != unarylevel.end());
      push_precedence(unarylevel[u.op]); // Highest: parenthesis never needed
      o << u.op;
      if (u.op.compare("pre")==0) o << "(" ; else o << ' ';
      process(u.source);
      if (u.op.compare("pre")==0) o << ")" ;
      pop_precedence();
    }
    else {
      assert(unarylevel.find(u.op) != unarylevel.end());
      push_precedence(unarylevel[u.op]); // Highest: parenthesis never needed
      vhdlCode.implementation << u.op;
      if (u.op.compare("not")==0) vhdlCode.implementation << ' ';
      process(u.source);
      pop_precedence();
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(BinaryOp &b) {
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    if (isSWModule) {
      assert(level.find(b.op) != level.end());
      bool needParen = push_precedence(level[b.op]);
      if (needParen) o << '(';
      process(b.source1);
      o << ' ' << b.op << ' ';
      process(b.source2);
      if (needParen) o << ')';
      pop_precedence();
    }
    else {
      assert(level.find(b.op) != level.end());
      bool needParen = true; // Parentheses always needed because 'and' and 'or' have the same
                             // precedence in VHDL
      if (needParen) vhdlCode.implementation << '(';
      process(b.source1);
      vhdlCode.implementation << ' ' << b.op << ' ';
      process(b.source2);
      if (needParen) vhdlCode.implementation << ')';
      pop_precedence();
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(LoadVariableExpression &e) {
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    if (isSWModule) {
      assert(e.variable);
      o << e.variable->name;
    }
    else {
      // print a constant true or false
      assert(e.variable);
      if(e.variable->name == "true") vhdlCode.implementation << "1";
      else vhdlCode.implementation << "0";
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(LoadSignalExpression &e) {
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    if (isSWModule) {
      assert(e.signal);
      o << e.signal->name;
    }
    else {
      // print a signal name to the combinatorial function
      assert(e.signal);

      if(e.signal->kind == SignalSymbol::Inputoutput) {
        vhdlCode.implementation << getUniqueSigName(e.signal->name + "_tmp",usedSymbols);
      } else {
        vhdlCode.implementation << e.signal->name;
      }
      // if signal is boolean then s(0) is the state value
      if(e.signal->type) vhdlCode.implementation << "(0)";
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(LoadSignalValueExpression &e) {
```

```
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    if (isSWModule) {
      assert(e.signal);
      if (e.signal->kind == SignalSymbol::Trap) o << '?';
      o << '?' << e.signal->name;
    }
    else {
      // print a signal name to the combinatorial function
      assert(e.signal);
      if(e.signal->kind == SignalSymbol::Inputoutput) {
        vhdlCode.implementation << getUniqueSigName(e.signal->name + "_tmp",usedSymbols);
      } else {
        vhdlCode.implementation << e.signal->name;
      }
      // if signal is boolean then s(1) is the boolean value
      vhdlCode.implementation << "(1)";
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(Literal &l) {
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    if (isSWModule) {
      assert(l.type);
      if (l.type->name == "string") {
        o << '\" ';
␣␣␣␣␣␣␣␣for␣(string::iterator␣i␣=␣l.value.begin()␣;␣i␣!=␣l.value.end()
      ␣;␣i++)␣{
␣␣␣␣␣␣␣␣␣␣if␣(*i␣==␣'\"') o << '\" ';
␣␣␣␣␣␣␣␣␣␣o␣<<␣*i;
␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣o␣<<␣'\"';
      }
      else o << l.value;
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(FunctionCall &e) {
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    // function calls not supported by KEP
    if (isSWModule) {
      assert(e.callee);
      o << e.callee->name;
      o << '(';
      vector<Expression*>::const_iterator i = e.arguments.begin();
      while (i != e.arguments.end()) {
        expression(*i);
        i++;
        if ( i != e.arguments.end()) o << ",␣";
      }
      o << ')';
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(Delay &e) {
    // for printing the sw-module, the original CEC EsterelPrinter method is used
    if (isSWModule) {
      if (e.is_immediate) {
        assert(e.count == NULL);
        o << "immediate␣";
      } else {
        expression(e.count);
        o << ' ';
      }
      //    o << '[';
      sigexpression(e.predicate);
      //    o << ']';
    }
    else{
      sigexpression(e.predicate);
    }
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(CheckCounter &e) {
    assert(e.counter);
    o << "check(";
    expression(e.predicate);
    o << ')';
    return Status();
  }

  Status CoDesignHWSWPrinter::visit(IfThenElse &s) {
    o << "if␣(";
    expression(s.predicate);
    o << ")";
    if (s.then_part) {
      o << "␣{";
      statement(s.then_part);
      o << "}";
    }
    if (s.else_part) {
      o << "␣else␣{";
      statement(s.else_part);
      o << "}";
    }
    return Status();
  }
}
```

A.2. Wiederherstellung von Signalabhängigkeiten im CKAG

Durch die Ersetzung von Ausdrücken durch Hilfssignale während der HW/SW Co-Synthese gehen Abhängigkeiten zwischen Readern und Writern eines Signals verloren (vgl. Abschnitt 3.5.1). Während der Partitionierung wird daher eine Datei erstellt, die eine Sicherung dieser Abhängigkeiten enthält. Auf Basis dieser Datei kann der Esterel2KASM Compiler die fehlenden Signalabhängigkeiten wieder rekonstruieren.

Um dies zu erreichen wurde die Klasse `DependencyHandler` des Esterel2KASM Compilers geeignet erweitert. Die Klasse `ExtDependencyHandler` liest die Datei mit den gesicherten Abhängigkeiten ein und fügt diese zum CKAG hinzu.

Um fehlende Signalabhängigkeiten wieder herzustellen, muss das Programm *cec-astkep* aus der Esterel2KASM Compiler Toolkette mit der zusätzlichen Option `-c` ***`depfilename`*** gestartet werden, wobei ***`depfilename`*** der Name der Datei mit den gesicherten Abhängigkeiten ist. Normalerweise geschieht dies automatisch über das Makefile, indem man das Target `*.cosyn.kep.kasm` oder `*.opt.cosyn.kep.kasm` angibt.

A.2.1. ExtDependencyHandler.cpp

```
/* The class ExtDependencyHandler extends the original KEP class DependencyHandler.
 * Normally, the  Esterel2KASM compiler cannot reconstruct from which signals the
 * auxiliary signals, that substitute complex expressions, depend on. This in-
 * formation was stored in a file during the partitioning process. In case of
 * co-synthesis this class has to be used that restores the signal dependencies
 * in the CKAG
 */

 class ExtDependencyHandler : public KEP::DependencyHandler {
 protected:
   std::string ext_deps_file;
 public:
   ExtDependencyHandler(KEP::Debug *d =new KEP::Debug(),std::string filename = "")
     : KEP::DependencyHandler(d), ext_deps_file(filename)
   {}
 protected:
    void add_dependencies(bool=false);
 };

 void ExtDependencyHandler::add_dependencies(bool all) {
   debug->print("start KepDependencies::add_dependencies:");
   debug->indent();

   visit_nodes();
   std::map<KEP::SignalSymbol*,SignalNodes>::iterator m,n;
   std::set<KEP::KepNode*>::const_iterator w,r;

   std::ifstream ifs(ext_deps_file.c_str());
   std::string line, reader_signal, writer_signal;
   std::getline(ifs,line,'\n');

   while(!ifs.eof()) {
     std::istringstream lineStr(line);
     lineStr >> reader_signal;
     lineStr >> writer_signal;

     // The dependencies object stores for each signal all reader and writer nodes
     // in the CKAG.

     // Iterate over all dependencies
     for (m=this->dependencies.begin();m!=this->dependencies.end();m++) {
       KEP::SignalSymbol *symbol = m->first;
       assert(symbol);
       // if element for the dependencies of an auxiliary signal (reader_signal)
       // is found ...
       if(reader_signal == symbol->getName()) {
         // ... find all writers of (writer_signal), that is a signal the auxiliary
         // signal depends onn ...
         for (n=this->dependencies.begin();n!=this->dependencies.end();n++) {
           KEP::SignalSymbol *symbol = n->first;
           const SignalNodes &sn = n->second;
           assert(symbol);
           if(writer_signal == symbol->getName()) {
             if (!n->second.writers.empty()) {
               for (w=sn.writers.begin();w!=sn.writers.end();w++) {
                 // ... and add them to the data structure
                 m->second.writers.insert(*w);
               }
             }
           }
         }
       }
     }
     std::getline(ifs,line,'\n');
   }
   ifs.close();


   for (m=this->dependencies.begin();m!=this->dependencies.end();m++) {
     const SignalNodes &sn = (*m).second;

     if (!sn.writers.empty() && !sn.readers.empty()) {
       for (w=sn.writers.begin();w!=sn.writers.end();w++) {
         assert(*w);
         KepThreadId *id_w = (*w)->getThreadId();
         assert(id_w);

         for (r=sn.readers.begin();r!=sn.readers.end();r++) {
           assert(*r);
           KepThreadId *id_r = (*r)->getThreadId();
           assert(id_r);

           if (!all && id_w->concurrent(id_r)) {
             (*w)->addReader(*r,m->first);
           }
         }
       }
     }
   }

   debug->unindent();
   debug->print("end KepDependencies::add_dependencies.");
 }
```

A.3. Interfacemodul des KEP

Als Ausführungsplattform für das Co-Design dient der KEP mit dem dazugehörigen KEP-Compiler in der Version 4.16. Die unter Microsoft Windows ausführbare Datei `kepcmpv4.16.exe` enthält sowohl einen Compiler von KASM in KEP Operationscode, als auch einen VHDL Codegenerator. Mit Hilfe des Codegenerators kann eine VHDL Beschreibung des KEP in einer individuell angepassten Konfiguration erzeugt werden. Eine spezielle Option erlaubt es, die VHDL Beschreibung des Logikblocks zu importieren und automatisch in den Code des Interfacemoduls zu integrieren. Der nachfolgende Code zeigt die Implementierung des Generators für den Logikblock. Der Programmcode innerhalb von `if (CoDesignFlag==true) { }` Blöcken ist relevant für die Integration des Logikblocks in das Interfacemodul.

A.3.1. blkinterface.cpp

```
int generate_blkinterface()
{
  int i,tmpdat1,tmpdat2,tmpdat3;
  char tmpchar1[2];
  char tmpchar2[8];
  char tmpchars[MaxChars]="";
  string tmpstr1,tmpstr2,tmpstr3,tmpstr4,tmpstr5;
  string tmpEntityName;

  string tmpstr[MaxStatements];
  string tmpSignalName[MaxStatements][2];              // tmpSignalName[x][0]: name
  // tmpSignalName[x][1]: dir and type     (IN | INV | OUT |OUTV)



  int tmpflag1;


  ofstream outfile1("blkinterface.vhd",ios_base::out);
  if(!outfile1){return -1;}
  outfile1<<"
      ----------------------------------------------------------------------------------
      \n";
  outfile1<<"-- Generated by "<<version<<"\n";
  outfile1<<"
      ----------------------------------------------------------------------------------
      \n";
  outfile1<<"-- Xin Li\n";
  outfile1<<"-- Realtime and Embedded System Groups\n";
  outfile1<<"-- University of Kiel , Germany\n";
  outfile1<<"-------------------------------------------------------------- \n";
  outfile1<<"-- Design Name: Kiel Esterel Processor 4\n";
  outfile1<<"-- Component Name: blkinterface\n";
  outfile1<<"-- Description:\n";
  outfile1<<"--    Address Width = "<<AddressWidth<<"\n";
  outfile1<<"--    I/O Signals = "<<SignalNum<<"\n";
  //outfile1<<"--    Valued I/O Signals = "<<ValuedNum<<"\n";
  outfile1<<"--    Datapath Width (bit) = "<<DataWidth<<"\n";
  outfile1<<"
      ----------------------------------------------------------------------------------
      \n";
  tmpdat1=1;

  // Reads the hardware description of the logic block from file
  // and adds enters it into the code for the interface block.
  // Further the entity name is stored in 'tmpEntityName' and the interface
  // declaration is written to the string 'tmpstr'
  if(CoDesignFlag==true)
    {
      //cite hw_vhdl.vhd
      ifstream inFile(HWKEPFileName.c_str());
      tmpflag1=1;
  tmpdat1=0;
  if ( !inFile )
    {
      return -1;
    }
  while (inFile.getline(tmpchars,MaxChars))
    {
      outfile1<<tmpchars<<"\n";

      tmpstr1=tmpchars;
      istringstream sbuf2(tmpstr1);
      sbuf2>>tmpstr2>>tmpstr3>>tmpstr4;

      if((upper_string(tmpstr2)=="ENTITY")&&(upper_string(tmpstr4)=="IS"))
        {
          tmpEntityName=tmpstr3;
          tmpstr5=tmpEntityName;
          tmpstr5+=";";
          tmpflag1=2;
        }

      if(((upper_string(tmpstr2)=="END")&&(upper_string(tmpstr3)==upper_string(
    tmpEntityName)))||((upper_string(tmpstr2)=="END")&&(upper_string(tmpstr3)==upper_string
    (tmpstr5))))
          {
            tmpflag1=1;
          }
      if((tmpflag1==0)&&(tmpstr1.size()>0))
        {
          tmpstr[tmpdat1++]=tmpchars;
        }
      if(tmpflag1==2){tmpflag1=0;}

    }
  }

outfile1<<"
    ----------------------------------------------------------------------------------\
    \n";
outfile1<<"library ieee;\nuse IEEE.STD_LOGIC_1164.ALL;\nuse IEEE.
    STD_LOGIC_ARITH.ALL;\nuse IEEE.STD_LOGIC_UNSIGNED.ALL;\n";
outfile1<<"library UNISIM;\nuse UNISIM.Vcomponents.ALL;\n";
outfile1<<"entity SDatReg is\n";
outfile1<<"port( clk : in std_logic;\n";
outfile1<<"      clk2  : in std_logic; \n";
outfile1<<"      en  : in std_logic; \n";
outfile1<<"      en2  : in std_logic; \n";
outfile1<<"      we  : in std_logic; \n";
outfile1<<"      we2  : in std_logic; \n";
outfile1<<"      Addr: in std_logic_vector("<<SignalWidth<<" downto 0);\n";
outfile1<<"      Addr2: in std_logic_vector("<<SignalWidth<<" downto 0);\n";
outfile1<<"      di  : in std_logic_vector("<<DataWidth-1<<" downto 0);\n";
```

```
    outfile1<<"       di2   : in std_logic_vector("<<DataWidth-1<<" downto 0);\n";
    outfile1<<"       do    : out std_logic_vector("<<DataWidth-1<<" downto 0);\n";
    outfile1<<"       do2   : out std_logic_vector("<<DataWidth-1<<" downto 0));\
        n";
    outfile1<<"end SDatReg; \n";
    outfile1<<"architecture Behavioral of SDatReg is \n";
    outfile1<<"component RAMB16_S36_S36\n";
    outfile1<<"port(  ADDRA : in    std_logic_vector (8 downto 0); \n";
    outfile1<<"       ADDRB : in    std_logic_vector (8 downto 0); \n";
    outfile1<<"       CLKA  : in    std_logic; \n";
    outfile1<<"       CLKB  : in    std_logic; \n";
    outfile1<<"       DIA   : in    std_logic_vector (31 downto 0); \n";
    outfile1<<"       DIB   : in    std_logic_vector (31 downto 0); \n";
    outfile1<<"       DIPA  : in    std_logic_vector (3 downto 0); \n";
    outfile1<<"       DIPB  : in    std_logic_vector (3 downto 0); \n";
    outfile1<<"       ENA   : in    std_logic; \n";
    outfile1<<"       ENB   : in    std_logic; \n";
    outfile1<<"       SSRA  : in    std_logic; \n";
    outfile1<<"       SSRB  : in    std_logic; \n";
    outfile1<<"       WEA   : in    std_logic; \n";
    outfile1<<"       WEB   : in    std_logic; \n";
    outfile1<<"       DOA   : out   std_logic_vector (31 downto 0); \n";
    outfile1<<"       DOB   : out   std_logic_vector (31 downto 0); \n";
    outfile1<<"       DOPA  : out   std_logic_vector (3 downto 0); \n";
    outfile1<<"       DOPB  : out   std_logic_vector (3 downto 0)); \n";
    outfile1<<"end component; \n";
    outfile1<<"signal addra : std_logic_vector (8 downto 0); \n";
    outfile1<<"signal addrb : std_logic_vector (8 downto 0); \n";
    outfile1<<"signal DOPA : std_logic_vector (3 downto 0); \n";
    outfile1<<"signal DOPB : std_logic_vector (3 downto 0); \n";


    outfile1<<"begin \n";
    outfile1<<"addra<=conv_std_logic_vector(conv_integer(Addr),9); \n";
    outfile1<<"addrb<=conv_std_logic_vector(conv_integer(Addr2),9); \n";
    outfile1<<"SDatReg: RAMB16_S36_S36 port map (addra, addrb, \n";
    outfile1<<"                                  clk, clk2, di, di2, \"0000\
        ", \"0000\", en, en2, '0', '0', we, we2, do, do2, DOPA, DOPB); \n\n"
        ;

    outfile1<<"end Behavioral; \n\n";
    outfile1<<"
        ---------------------------------------------------------------------
        n\n";

    outfile1<<"
        ---------------------------------------------------------------------
        n";
    outfile1<<"library ieee; \nuse IEEE.STD_LOGIC_1164.ALL; \nuse IEEE.
        STD_LOGIC_ARITH.ALL; \nuse IEEE.STD_LOGIC_UNSIGNED.ALL; \n";
    outfile1<<"library UNISIM; \nuse UNISIM.Vcomponents.ALL; \n";
    outfile1<<"entity blkinterface is \n";
    outfile1<<"  port(  cclk : in std_logic; \n";
    if(SimDebugMethod==1)
      {
        outfile1<<"       blkinterfaceDebug : out std_logic_vector(31 downto
            0); \n";
      }
    outfile1<<"       reset : in std_logic; \n";
    outfile1<<"       tick : in std_logic; \n";
    outfile1<<"       wremit : in std_logic; \n";
    outfile1<<"       wrpreemption : in std_logic; \n";
    outfile1<<"       wrpreemptiontype : in std_logic_vector(1 downto 0); \n
        ";
    outfile1<<"       Sinout : inout std_logic_vector("<<SignalNum-1<<"
        downto 0); \n";
    outfile1<<"       SDir : in std_logic;     --- '0': Sinout, SDatReg,
        etc., Control by environment\n";
    outfile1<<"                                --- '1': Sinout, SDatReg,
        etc., Control by KEP\n";
    outfile1<<"       SDatClk : in std_logic; \n";
    outfile1<<"       SDat : inout std_logic_vector("<<DataWidth-1<<" downto
        0); \n";
    outfile1<<"       SDatID : in std_logic_vector("<<SignalWidth<<" downto
        0); \n";
    outfile1<<"       SDatWR : in std_logic; \n";
    outfile1<<"       rdpresent : out std_logic; \n";
    outfile1<<"       regDatin : out std_logic_vector("<<DataWidth-1<<"
        downto 0); \n";
    outfile1<<"       regDatout : in std_logic_vector("<<DataWidth-1<<"
        downto 0); \n";
    outfile1<<"       innerop : in std_logic_vector(4 downto 0); \n";
    outfile1<<"       inneraddr : in std_logic_vector(8 downto 0); \n";
    outfile1<<"       innermark : in std_logic_vector(5 downto 0); \n";
    outfile1<<"       innerdata : in std_logic_vector(15 downto 0); \n";
    outfile1<<"       instrclk : in std_logic; \n";
    outfile1<<"       CurAddr : in std_logic_vector("<<AddressWidth-1<<" downto
        0); \n";
    outfile1<<"       WatcherSignal : out std_logic_vector("<<WatcherMaxNum-1<<
        " downto 0); \n";
    outfile1<<"       LWatcherSignalCode : in std_logic_vector("<<SignalWidth
        -1<<" downto 0);   --- no PRE\n";
    outfile1<<"       LWatcherSignal : out std_logic; \n";
    outfile1<<"       TWatcherSignalCode : in std_logic_vector("<<SignalWidth
        <<" downto 0);     --- PRE \n";
    outfile1<<"       TWatcherSignal : out std_logic; \n";
    outfile1<<"       BlocksCE : in std_logic); \n";
    outfile1<<"end blkinterface; \n";
    outfile1<<"    architecture Behavioral of blkinterface is \n";
    outfile1<<"    attribute clock_signal : string; \n";
    outfile1<<"    attribute clock_signal of cclk : signal is \"yes\"; \n";
    outfile1<<"    constant EMITR : std_logic_vector(4 downto 0):=\"10010\";   
        --- EMITR & SUSTAINR\n";
    outfile1<<"    constant EMIT : std_logic_vector(4 downto 0):=\"01000\";    
        --- EMIT & SUSTAIN & CANCEL\n";
    outfile1<<"component SDatReg is \n";
    outfile1<<"port( clk : in std_logic; \n";
    outfile1<<"      clk2 : in std_logic; \n";
    outfile1<<"      en : in std_logic; \n";
    outfile1<<"      en2 : in std_logic; \n";
    outfile1<<"      we : in std_logic; \n";
```

```
outfile1<<"      we2 : in std_logic; \n";
outfile1<<"      Addr: in std_logic_vector("<<SignalWidth<<" downto 0);\n";
outfile1<<"      Addr2: in std_logic_vector("<<SignalWidth<<" downto 0);\n";
outfile1<<"      di  : in std_logic_vector("<<DataWidth-1<<" downto 0);\n";
outfile1<<"      di2 : in std_logic_vector("<<DataWidth-1<<" downto 0);\n";
outfile1<<"      do  : out std_logic_vector("<<DataWidth-1<<" downto 0);\n";
outfile1<<"      do2 : out std_logic_vector("<<DataWidth-1<<" downto 0));\n
    ";
outfile1<<"end component; \n";

// Declares the logic block as component of the interface block
if(CoDesignFlag==true)
  {

    outfile1<<"component "<<tmpEntityName<<" is \n";
    tmpdat1=0;
    while(tmpstr[tmpdat1].size()>0)
      {
        outfile1<<"    "<<tmpstr[tmpdat1++]<<"\n";
      }
    outfile1<<"end component;\n";
  }

outfile1<<"    constant ABORT: std_logic_vector(4 downto 0):=\"00010\";
    --ABORT\n";
outfile1<<"    constant ABORTI: std_logic_vector(4 downto 0):=\"00011\";
    --ABORTI\n";
outfile1<<"    constant WABORT: std_logic_vector(4 downto 0):=\"00100\";
    --WABORT\n";
outfile1<<"    constant WABORTI: std_logic_vector(4 downto 0):=\"00101\";
    --WABORTI\n";
outfile1<<"    constant SUSPEND: std_logic_vector(4 downto 0):=\"00110\";
    --SUSPEND\n";
outfile1<<"    constant SUSPENDI: std_logic_vector(4 downto 0):=\"00111\";
    --SUSPENDI\n";

outfile1<<"    signal tickflag : std_logic;\n";
outfile1<<"    signal instrclkflag : std_logic;\n";
outfile1<<"    signal wremitflag : std_logic;\n";
outfile1<<"    --signal wrpreemptionflag : std_logic;\n";
outfile1<<"    signal SinoutBuff: std_logic_vector("<<SignalNum-1<<" downto
    0); \n";
outfile1<<"    signal Sinoutreg : std_logic_vector("<<SignalNum<<" downto 0)
    ; \n";
outfile1<<"    signal preSinoutreg : std_logic_vector("<<SignalNum<<" downto
    0);\n";
outfile1<<"    signal initSinout : std_logic_vector("<<SignalNum<<" downto
    0); \n";
outfile1<<"    signal innerSinoutFlag : std_logic_vector("<<SignalNum<<"
    downto 0);\n";
outfile1<<"    type WatcherSignalCode_type is array ("<<WatcherMaxNum-1<<"
    downto 0) of std_logic_vector("<<SignalWidth<<" downto 0);\n";
outfile1<<"    signal WatcherSignalCode: WatcherSignalCode_type;\n";

outfile1<<"    signal SDatRegClkMux: std_logic;\n";

outfile1<<"    signal SDatRegClkKEP: std_logic;\n";
outfile1<<"    signal SDatRegWRKEP: std_logic;\n";
outfile1<<"    signal SDatRegIDKEP: std_logic_vector("<<SignalWidth<<"
    downto 0);\n";
outfile1<<"    signal SDatRegDIKEP: std_logic_vector("<<DataWidth-1<<"
    downto 0);\n";
outfile1<<"    signal SDatRegDOKEP: std_logic_vector("<<DataWidth-1<<"
    downto 0);\n";

outfile1<<"    signal SDatRegClkKEP2: std_logic;\n";
outfile1<<"    signal SDatRegWRKEP2: std_logic;\n";
outfile1<<"    signal SDatRegIDKEP2: std_logic_vector("<<SignalWidth<<"
    downto 0);\n";
outfile1<<"    signal SDatRegDIKEP2: std_logic_vector("<<DataWidth-1<<"
    downto 0);\n";
outfile1<<"    signal SDatRegDOKEP2: std_logic_vector("<<DataWidth-1<<"
    downto 0);\n";


outfile1<<"    signal SDatRegClk: std_logic;\n";
outfile1<<"    signal SDatRegWR: std_logic;\n";
outfile1<<"    signal SDatRegEN: std_logic;\n";
outfile1<<"    signal SDatRegID: std_logic_vector("<<SignalWidth<<" downto
    0);\n";
outfile1<<"    signal SDatRegDI: std_logic_vector("<<DataWidth-1<<" downto
    0);\n";
outfile1<<"    signal SDatRegDO: std_logic_vector("<<DataWidth-1<<" downto
    0);\n";

outfile1<<"    signal SDatRegClk2: std_logic;\n";
outfile1<<"    signal SDatRegEN2: std_logic;\n";
outfile1<<"    signal SDatRegWR2: std_logic;\n";
outfile1<<"    signal SDatRegID2: std_logic_vector("<<SignalWidth<<" downto
    0);\n";
outfile1<<"    signal SDatRegDI2: std_logic_vector("<<DataWidth-1<<" downto
    0);\n";
outfile1<<"    signal SDatRegDO2: std_logic_vector("<<DataWidth-1<<" downto
    0);\n";

if(CoDesignFlag==true)
  {
    outfile1<<"    signal SDatBoolean: std_logic_vector("<<SignalNum<<"
    downto 0);\n";
    outfile1<<"    signal SDatBooleanLogic: std_logic_vector("<<SignalNum<<"
    downto 0);\n";
  }

// (modified by sga)
if(CoDesignFlag==true)
  {
    tmpdat1=0;
    tmpdat2=0;
    while(tmpstr[tmpdat1].size()>0)
      {
        replace(tmpstr[tmpdat1].begin(), tmpstr[tmpdat1].end(), '(', ' ' );
        replace(tmpstr[tmpdat1].begin(), tmpstr[tmpdat1].end(), ')', ' ' );
```

```
      replace(tmpstr[tmpdat1].begin(), tmpstr[tmpdat1].end(), ':', ' ' );
      tmpstr[tmpdat1]=upper_string(tmpstr[tmpdat1]);
      tmpdat1++;
    }

  tmpdat1=0;

  // store the names of all interface signals of the logic block and their
  // direction in 'tmpSignalName'. An additionally 'V' indicates that the signal
  // is boolean
  while(tmpstr[tmpdat1].size()>0)
    {
      istringstream sbuf2(tmpstr[tmpdat1]);
      tmpstr1="␣";
      while(tmpstr1.size()>0)
        {
          tmpstr2=tmpstr1;
          sbuf2>>tmpstr1;
          if((tmpstr1=="IN")||(tmpstr1=="OUT"))
            {
              string type;
              sbuf2 >> type;
              tmpSignalName[tmpdat2][0]=tmpstr2;
              tmpSignalName[tmpdat2][1]=tmpstr1;
              if((type != "STD_LOGIC") && (type != "STD_LOGIC;")) tmpSignalName[
 tmpdat2][1] += "V";
              tmpdat2++;
            }
        }
      tmpdat1++;

    }

  tmpdat1=0;
  // declare local signals with the name and type of the interface signals of the logic
   block
  while(tmpSignalName[tmpdat1][0].size()>0)
    {
      outfile1 << "␣␣␣␣signal␣" << tmpSignalName[tmpdat1][0];
      // test if signal is boolean
      if((tmpSignalName[tmpdat1][1]=="IN") || (tmpSignalName[tmpdat1][1]=="OUT"))
        outfile1 << ":␣std_logic;␣␣\n";
      else
        outfile1 << ":␣std_logic_vector(1␣downto␣0);␣␣\n";

      //outfile1<<"\n";
      tmpdat1++;
    }

}

outfile1<<"begin\n\n";

outfile1<<"SDat_Reg:␣SDatReg␣port␣map␣(SDatRegClk,␣SDatRegClk2,␣
    SDatRegEN,␣SDatRegEN2,␣SDatRegWR,␣SDatRegWR,␣SDatRegID,␣SDatRegID2
    ,␣\n";

outfile1<<"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣SDatRegDI,␣SDatRegDI2,␣SDatRegDO
    ,␣SDatRegDO2);␣\n";


// instantiate the logic block
if(CoDesignFlag==true)
  {
    outfile1<<tmpEntityName<<"_instant:␣"<<tmpEntityName<<"␣port␣map␣(␣\n";
    tmpdat1=0;
    while(tmpSignalName[tmpdat1][0].size()>0)
      {
        if(tmpdat1>0)
          {
            outfile1<<",\n";
          }
        outfile1<<"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"<<tmpSignalName[tmpdat1][0];
        tmpdat1++;
      }

    outfile1<<");\n";
  }
outfile1<<"process(cclk,reset)\n";
outfile1<<"begin\n";
outfile1<<"␣if␣falling_edge(cclk)␣then\n";
outfile1<<"---␣if␣BlocksCE='1'␣then\n";
outfile1<<"␣␣␣␣␣if␣reset='1'␣then\n";
outfile1<<"␣␣␣␣␣␣␣␣␣Sinoutreg<=(others=>'0');\n";
outfile1<<"␣␣␣␣␣␣␣␣␣preSinoutreg<=(others=>'0');\n";
outfile1<<"␣␣␣␣␣␣␣␣␣initSinout<=(others=>'1');\n";
outfile1<<"␣␣␣␣␣␣␣␣␣tickflag<='0';\n";
outfile1<<"␣␣␣␣␣␣␣␣␣instrclkflag<='0';\n";
outfile1<<"␣␣␣␣␣␣␣␣␣wremitflag<='0';\n";
outfile1<<"␣␣␣␣␣␣␣␣␣--wrpreemptionflag<='0';\n";
outfile1<<"␣␣␣␣␣␣␣␣␣SDatRegWRKEP<='0';\n";
outfile1<<"␣␣␣␣␣␣␣␣␣for␣i␣in␣0␣to␣"<<WatcherMaxNum-1<<"␣loop\n";
outfile1<<"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣WatcherSignalCode(i)<=(others=>'0');\n";
outfile1<<"␣␣␣␣␣␣␣␣␣end␣loop;\n";
outfile1<<"␣␣␣␣␣else\n";

// Produces VHDL code to wire the Logic Block with the interface block
// 'tmpSignalName' contains the interface signals of the logic block
// 'SignalTable' maps the signal name used in the Esterel program to
// the address in the SDatReg
// (code modified by sga)
if(CoDesignFlag==true)
  {
    tmpdat1=0;
    while(tmpSignalName[tmpdat1][0].size()>0)
      {
        tmpdat2=0;
        while(SignalTable[tmpdat2][0].size()>0)
          {
            if(SignalTable[tmpdat2][0]==tmpSignalName[tmpdat1][0])
              {
                istringstream sbuf2(SignalTable[tmpdat2][2]);
```

```
            sbuf2>>tmpdat3;                // write address of the signal in the SDatReg
    to tmpdat3
                                           // note that SDatReg has twice the size of
    SinoutReg
                                           // or SDatBoolean to store the pre values
            if((tmpSignalName[tmpdat1][1]=="IN") || (tmpSignalName[tmpdat1][1]=="INV")
    )
              {
                outfile1<<"        "<<tmpSignalName[tmpdat1][0];
                if(tmpSignalName[tmpdat1][1]=="IN") {
                  outfile1<<"<=innerSinoutFlag("<<tmpdat3/2<<");\n";
                }
                else {
                  outfile1<<"(0)<=innerSinoutFlag("<<tmpdat3/2<<");\n";
                  outfile1<<"        "<<tmpSignalName[tmpdat1][0]<<"(1)<=
    SDatBoolean("<<tmpdat3/2<<");\n";
                }
              }
            else
              {
                outfile1<<"        SinoutReg("<<tmpdat3/2<<")<=";
                if(tmpSignalName[tmpdat1][1]=="OUT") outfile1 << tmpSignalName[tmpdat1
    ][0]<<";\n";
                else {
                  outfile1 << tmpSignalName[tmpdat1][0]<<"(0);\n";
                  outfile1 << "        SDatBoolean("<<tmpdat3/2<<")<="<<
    tmpSignalName[tmpdat1][0]<<"(1);\n";
                }
              }
          }
        tmpdat2++;
      }
    tmpdat1++;
  }
}

outfile1<<"        tickflag<=tick;\n";
outfile1<<"        instrclkflag<=instrclk;\n";
outfile1<<"        wremitflag<=wremit;\n";
outfile1<<"        --wrpreemptionflag<=wrpreemption;\n";
outfile1<<"        SDatRegWRKEP<='0';\n";
outfile1<<"        SDatRegWRKEP2<='0';\n";
outfile1<<"        SDatRegIDKEP<=inneraddr("<<SignalWidth<<" downto 1) &
    '0';\n";
outfile1<<"        SDatRegIDKEP2<=inneraddr("<<SignalWidth<<" downto 1) &
    '1';\n";

outfile1<<"        if wremit='1' and wremitflag='0' then\n";
outfile1<<"            if innermark(5 downto 4)=\"11\" then    --CANCEL
    (SIGNAL IN)\n";
outfile1<<"                Sinoutreg(conv_integer(inneraddr("<<SignalWidth
    <<" downto 1)))<='0';\n";
if(presupport==true)
  {
    outfile1<<"                preSinoutreg(conv_integer(inneraddr("<<
    SignalWidth<<" downto 1)))<='0';\n";
  }
outfile1<<"            elsif innerop=EMIT then      --EMIT, SUSTAIN,
    SETV\n";
outfile1<<"                if innermark(5 downto 4)/=\"10\" then    --
    EMIT or SUSTAIN (not SETV) \n";
outfile1<<"                    Sinoutreg(conv_integer(inneraddr("<<
    SignalWidth<<" downto 1)))<='1';\n";
outfile1<<"                end if;\n";
outfile1<<"                if inneraddr("<<SignalWidth<<" downto 0)>"<<
    Gen_Init_Str(SignalWidth+1,1)<<" then\n";
outfile1<<"                    initSinout(conv_integer(inneraddr("<<
    SignalWidth<<" downto 1)))<='0';\n";
outfile1<<"                    SDatRegIDKEP<=inneraddr("<<SignalWidth<<"
    downto 1) & '0';\n";
outfile1<<"                    SDatRegDIKEP<="<<Gen_Init_Str(DataWidth-16,0)<<"
    & innerdata(15 downto 0);\n";
outfile1<<"                    SDatRegWRKEP<='1';\n";

if(presupport==true)
  {
    outfile1<<"                if innermark(5 downto 4)/=\"10\" then
    --EMIT or SUSTAIN (not SETV) \n";
    outfile1<<"                    if initSinout(conv_integer(
    inneraddr("<<SignalWidth<<" downto 1)))='1' then\n";
    outfile1<<"                        SDatRegIDKEP2<=inneraddr("<<
    SignalWidth<<" downto 1) & '1';\n";
    outfile1<<"                        SDatRegDIKEP2<=SDatRegDOKEP;\n
    ";
    outfile1<<"                    SDatRegWRKEP2<='1';\n";
    outfile1<<"                end if;\n";
    outfile1<<"            end if;\n";
  }
outfile1<<"        end if;\n";
outfile1<<"        else        ----EMITR, SUSTAINR\n
    ";
outfile1<<"            if innermark(5 downto 4)/=\"10\" then    --
    EMITR or SUSTAINR (not SETV) \n";
outfile1<<"                Sinoutreg(conv_integer(inneraddr("<<
    SignalWidth<<" downto 1)))<='1';\n";
outfile1<<"            end if;\n";
outfile1<<"            if inneraddr("<<SignalWidth<<" downto 0)>"<<
    Gen_Init_Str(SignalWidth+1,1)<<" then\n";
outfile1<<"                initSinout(conv_integer(inneraddr("<<
    SignalWidth<<" downto 1)))<='0';\n";
outfile1<<"                SDatRegIDKEP<=inneraddr("<<SignalWidth<<"
    downto 1) & '0';\n";
outfile1<<"                SDatRegDIKEP<=regDatout;\n";
outfile1<<"                SDatRegWRKEP<='1';\n";
if(presupport==true)
  {
    outfile1<<"                if innermark(5 downto 4)/=\"10\" then
    --EMITR or SUSTAINR (not SETVR) \n";
    outfile1<<"                    if initSinout(conv_integer(
    inneraddr("<<SignalWidth<<" downto 1)))='1' then\n";
    outfile1<<"                        SDatRegIDKEP2<=inneraddr("<<
    SignalWidth<<" downto 1) & '1';\n";
```

```
      outfile1<<"                                SDatRegDIKEP2<=SDatRegDOKEP;\n
          ";
      outfile1<<"                                SDatRegWRKEP2<='1';\n";
      outfile1<<"                                end if;\n";
      outfile1<<"                                end if;\n";
    }
  outfile1<<"                end if;\n";
  outfile1<<"              end if;\n";
  outfile1<<"            end if;\n";
  outfile1<<"            if (innerop=ABORT or innerop=ABORTI or innerop=
      WABORT or innerop=WABORTI or innerop=SUSPEND or innerop=SUSPENDI) and instrclk='1' and instrclkflag='0' then\n";
  outfile1<<"              WatcherSignalCode(conv_integer(innermark(4 downto 0)))<=inneraddr("<<SignalWidth<<" downto 0);\n";

  /*
    outfile1<<"            else --READ (LOAD, ADD, ...)\n";
    outfile1<<"              if inneraddr(0)='1' and initSinout(conv_integer
        (inneraddr("<<SignalWidth<<" downto 1)))='0' then\n";
    outfile1<<"                regDatIn<=SDatRegDOKEP2;\n";
    outfile1<<"              else\n";
    outfile1<<"                regDatIn<=SDatRegDOKEP;\n";
    outfile1<<"              end if;\n";
  */
  outfile1<<"            end if;\n";

  //outfile1<<"            if tickflag='1' and tick='0' then\n";
  outfile1<<"            if tickflag='0' and tick='0' then\n";
  if(presupport==true)
    {
      outfile1<<"              preSinoutreg<=(SinoutBuff & '1') or Sinoutreg
          ;\n";
    }
  outfile1<<"              SinoutReg<=(others=>'0');\n";
  outfile1<<"              initSinout<=(others=>'1');\n";
  outfile1<<"            end if;\n";
  outfile1<<"          end if;\n";
  outfile1<<"        --end if;\n";
  outfile1<<"      end if;\n";
  outfile1<<"      end process;\n";
  outfile1<<"
      ------------------------------------------------------------------------
      \n";
  outfile1<<"    --innerSinoutFlag<=(SinoutBuff & '1') or SinoutReg;\n";
  outfile1<<"    innerSinoutFlag<=(Sinout & '1') or SinoutReg;\n";
  outfile1<<"    --SinoutBuff<=Sinout when SDir='1' else (others=>'0');\n";
  outfile1<<"    Sinout<=SinoutReg("<<SignalNum<<" downto 1) when SDir='0' else (
      others=>'Z');\n";
  outfile1<<"
      ------------------------------------------------------------------------
      \n";
  outfile1<<"    WatcherSignalGen: \n";
  outfile1<<"    for i in 0 to "<<WatcherMaxNum-1<<" generate \n";
  if(presupport==true)
    {
      outfile1<<"      WatcherSignal(i)<=innerSinoutflag(conv_integer(
          WatcherSignalCode(i)("<<SignalWidth<<" downto 1))) when
          WatcherSignalCode(i)(0)='0' else\n";
      outfile1<<"                        preSinoutreg(conv_integer(
          WatcherSignalCode(i)("<<SignalWidth<<" downto 1)));\n";
    }
  else
    {
      outfile1<<"      WatcherSignal(i)<=innerSinoutflag(conv_integer(
          WatcherSignalCode(i)("<<SignalWidth<<" downto 1)));\n";
    }
  outfile1<<"    end generate; \n";
  outfile1<<"    ------------------------------------------------------------------------\n";
  outfile1<<"    LWatcherSignal<=innerSinoutflag(conv_integer(
      LWatcherSignalCode));\n";
  outfile1<<"    ------------------------------------------------------------------------\n";
  if(presupport==true)
    {
      outfile1<<"    TWatcherSignal<=innerSinoutflag(conv_integer(
          TWatcherSignalCode("<<SignalWidth<<" downto 1))) when
          TWatcherSignalCode(0)='0' else\n";
      outfile1<<"                      preSinoutreg(conv_integer(
          TWatcherSignalCode("<<SignalWidth<<" downto 1)));\n";
    }
  else
    {
      outfile1<<"    TWatcherSignal<=innerSinoutflag(conv_integer(
          TWatcherSignalCode("<<SignalWidth<<" downto 1)));\n";
    }

  outfile1<<"    ------------------------------------------------------------------------\n";
  if(presupport==true)
    {
      outfile1<<"    rdpresent<=innerSinoutflag((conv_integer(inneraddr("<<
          SignalWidth<<" downto 1)))) when inneraddr(0)='0' else\n";
      outfile1<<"               preSinoutreg((conv_integer(inneraddr("<<
          SignalWidth<<" downto 1))));\n";
    }
  else
    {
      outfile1<<"    rdpresent<=innerSinoutflag((conv_integer(inneraddr("<<
          SignalWidth<<" downto 1))));\n";
    }
  outfile1<<"
      ------------------------------------------------------------------------\
      n";
  outfile1<<"    SDatRegClk<=cclk when tick='1' else SDatClk;\n";
  outfile1<<"    SDatRegWR<=SDatRegWRKEP when tick='1' else SDatWR;\n";
  outfile1<<"    SDatRegID<=SDatRegIDKEP when tick='1' else SDatID;\n";
  outfile1<<"    SDat<=SDatRegDO when SDatWR='0' and tick='0' else (others=>'
      Z');\n";
  outfile1<<"    SDatRegDI<=SDat when SDatWR='1' and tick='0' else
      SDatRegDIKEP;\n";
  outfile1<<"    SDatRegEN<='1';\n";
  outfile1<<"    --SDatRegEN2<=BlocksCE;\n";
  outfile1<<"    SDatRegEN2<='1';\n";
```

```
// generates the SDatBoolean register that stores the values
// connected to the input with the falling edge of SDatRegClk
// (code modified by sga)
if(CoDesignFlag==true)
  {
    outfile1<<" process (SDatRegClk, reset)\n";
    outfile1<<" begin\n";
    outfile1<<"   if falling_edge(SDatRegClk) then\n";
    outfile1<<"     if reset='1' then\n";
    outfile1<<"       SDatBoolean(<=(others=> '0');\n";
    outfile1<<"     else\n";
    outfile1<<"       if SDatRegWR='1' then\n";
    outfile1<<"         SDatBoolean(conv_integer(SDatRegID)),<=
      SDatRegDI(0);\n";
    outfile1<<"       end if;\n";
    outfile1<<"     end if;\n";
    outfile1<<" end process;\n";
  }

outfile1<<"-------------------------------------------------------------------------------\n";
outfile1<<"SDatRegClk2<=cclk and tick;\n";
outfile1<<"SDatRegID2<=SDatRegIDKEP2;\n";
outfile1<<"SDatRegWR2<=SDatRegWRKEP2;\n";
outfile1<<"SDatRegDI2<=SDatRegDIKEP2;\n";
outfile1<<"SDatRegDOKEP2<=SDatRegDO2;\n";
outfile1<<"SDatRegDOKEP<=SDatRegDO;\n";
outfile1<<"-------------------------------------------------------------------------------\n";
if(presupport==true)
  {
    outfile1<<"regDatIn<=SDatRegDOKEP2 when inneraddr(0)='1' and
      initSinout(conv_integer(inneraddr(6 downto 1)))='0' else
      SDatRegDOKEP;\n";
  }
else
  {
    if(CoDesignFlag=true)
      {
        outfile1<<"regDatIn<=SDatRegDOKEP("<<DataWidth-1<<" downto 1) &
        SDatBoolean(conv_integer(SDatRegID));\n";
      }
    else
      {
        outfile1<<"regDatIn<=SDatRegDOKEP;\n";
      }
  }
outfile1<<"
    \n";
outfile1<<"end Behavioral;\n";
outfile1.close();
return 0;
}
```