Diplomarbeit

Entwurf und Implementierung eines konfigurierbaren SDL Transpilers für eine C++ Laufzeitumgebung

Christian Weber

September 2005

Erklärung

Ich erkläre hiermit, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen verwendet habe.

Kaiserslautern, den 12. September 2005

Christian Weber

Danksagung

Diese Diplomarbeit entstand von April bis September 2005 in der AG Vernetzte Systeme an der Technischen Universität Kaiserslautern. Ich möchte mich hiermit bei den Personen bedanken, die mich in dieser Zeit unterstützt haben.

Mein besonderer Dank gilt Herrn Prof. Dr. Reinhard Gotzhein für die Beratung während dieser Arbeit und meines Studiums.

Bedanken möchte ich mich auch bei Herrn Dipl. Technoinf. Ingmar Fliege der mich während dieser Arbeit betreute und mir bei der Lösung von Problemen behilflich war. Er programmierte außerdem parallel die notwendige Laufzeitumgebung, ohne die keine ausführbaren Programme möglich wären.

Mein Dank gilt auch Herrn Dipl. Inf. Alexander Geraldy für die Bereitstellung des von ihm entwickelten ADA Protokolls, das in dieser Arbeit als Beispielsystem dient, sowie allen Mitarbeitern der AG für die Unterstützung während der Arbeit.

Zum Schluss möchte ich mich bei meinen Eltern bedanken, die mir meine Ausbildung und mein Studium ermöglichten sowie bei meiner Schwester Petra und ihrem Mann Christian, die mich während meines Studiums hilfreich unterstützten.

Inhaltsverzeichnis

1 Einleitung

Ziel dieser Arbeit ist die Entwicklung eines SDL-Compilers, der die Spezifikations- und Designsprache SDL in die Programmiersprache C/C++ umsetzt. Dabei soll der Code speziell auch auf in Leistung, Speicherkapazität und Energieversorgung beschränkten Systemen genutzt werden können. Ausgangspunkt der Umsetzung von SDL nach C++ ist eine textuelle Beschreibung (**P**hrase **R**epresentation Datei) eines Systems in SDL. Die PR Datei kann dabei mit Hilfe geeigneter Werkzeuge, wie in Kapitel 2.1 beschrieben, erstellt werden und dient als Eingabe für den Compiler, der daraus objektorientierten C++ Code erzeugt. Dieser kann wiederum mit gängigen C++ Compilern in Maschinencode transformiert und letztendlich auf verschiedenen Plattformen ausgeführt werden.

1.1 Motivation

Der Einsatz von Eingebetteten Systemen[1] erfordert eine Abstimmung und Anpassung der Betriebssysteme und Anwendungen an die stark eingeschränkten Randbedingungen der eingesetzten Hardware. Häufig werden diese Systeme nur von einer Batterie gespeist, was unter anderem Energiesparmassnahmen in Form von geringem Speicher und verminderter Rechenleistung erfordert. Trotzdem sollen diese Systeme in der Lage sein, Daten zu erfassen, zu versenden und zu empfangen. Dazu sind spezielle Protokolle erforderlich, die neben dem reinen Datenverkehr auch den Verbindungsauf- und abbau sowie die Datenweiterleitung (Routing) unterstützen. Solche Protokolle können häufig in SDL spezifiziert werden.

Durch die Entwicklung eines eigenen SDL Compilers und die gleichzeitige Entwicklung einer Laufzeitumgebung, beschrieben in Kapitel 4.5, kann eine individuelle Anpassung an die Anforderungen von Eingebetteten Systemen gewährleistet werden. So ist es z.B. möglich, aufwändige Sprachkonstrukte wie etwa die Kreierung neuer Prozesse oder Prozeduraufrufe von SDL bei der Codegenerierung gezielt nicht zu unterstützen.

Ein weiterer Vorteil dieses Compilers ist die Erzeugung von strukturiertem, objekt-orientiertem Code, der eine leicht nachvollziehbare Abbildung von SDL nach C++ ermöglicht. Dies erleichtert die Suche nach möglichen Designfehlern durch Debugging zur Laufzeit sowie eine einfache und schnelle Fehlerbehebung. Zusätzlich können einzelne Module (`Packages`) unabhängig von einander kompiliert und zusammengesetzt werden. Dadurch ist es möglich, Mikroprotokolle, die in SDL spezifiziert wurden, auch in C++ wieder zu verwenden und in andere Systeme einfließen zu lassen.

[1] Bei Eingebetteten Systemen handelt es sich um Computersysteme die (weitestgehend unsichtbar) ihren Dienst in einer Vielzahl von Anwendungsbereichen versehen, wie z.B. Flugzeugen, Autos, Unterhaltungselektronik oder Sensornetzen

1.2 Überblick

Im folgenden Kapitel sollen zu Beginn kurz die notwendigen Grundlagen zur Spezifikations- und Designsprache SDL vermittelt werden. Anschließend wird in Kapitel 2.2 ein Überblick über die verschiedenen Technologien und die Arbeitsweisen eines Compilers gegeben und in Kapitel 2.3 die verwendeten Werkzeuge Telelogic Tau, Flex und Bison näher beschrieben. Kapitel 3 beschäftigt sich eingehend mit dem Aufbau des entwickelten Compilers und des grundlegenden Syntaxbaums, dessen notwendigen Transformationen und schließlich dem Mechanismus der Codeerzeugung. Einen Überblick über den erzeugten Code mit entsprechenden Beispielen und Erläuterungen findet sich in den Kapiteln 4.1, 4.2 und 4.3. Anschließend soll der Vorteil der strukturierten Codeerzeugung durch einen Vergleich des generierten Quellcodes mit dem erzeugten Code des Telelogic Tau Compilers demonstriert werden. In den Kapiteln 4.5 und 4.6 wird näher auf die parallel entwickelte Laufzeitumgebung eingegangen und der im Rahmen dieser Arbeit entwickelte ConTraST Compiler mit vier z.T. kommerziellen Compilern verglichen. An diese Kapitel schließen sich noch eine kurze Zusammenfassung sowie ein Ausblick auf die geplanten, zukünftigen Arbeiten am Compiler an.

2 Grundlagen

Im folgenden Kapitel wird kurz auf die notwendigen Grundlagen zu SDL, der Arbeitsweise eines Compilers, sowie der zur Entwicklung verwendeten Programmen eingegangen.

2.1 SDL

SDL [5][6] (**S**pecification and **D**escription **L**anguage) ist eine formale Standardsprache zur Beschreibung und Spezifikation von Systemen. Sie dient speziell zur Beschreibung von Verhalten, Struktur und Daten. Ausgangspunkt für das Verhalten sind durch Nachrichten asynchron kommunizierende erweiterte endliche Automaten. Daten werden durch algebraische Datentypen und die Struktur durch hierarchische Dekomposition und Typhierarchien gebildet. SDL findet in der Industrie breite Anwendung, speziell im Bereich der Protokollentwicklung bei der Kommunikation von Systemen. Die Sprache stellt eine formale abstrakte Grammatik (*abstract grammar*) in *Backus-Naur-Form* (BNF) sowie eine syntaktische Auswahl zur Repräsentation von Systemen zur Verfügung. Dabei handelt es sich einerseits um die *Graphical Representation* (SDL/GR) sowie die textuelle *Phrase Representation* (SDL/PR). Eine Untermenge von SDL/PR und SDL/GR ist gleich. Sie wird auch als *common textual grammar* bezeichnet. Den Zusammenhang von SDL/PR und SDL/GR verdeutlicht noch einmal Abbildung 2.2.1-1.

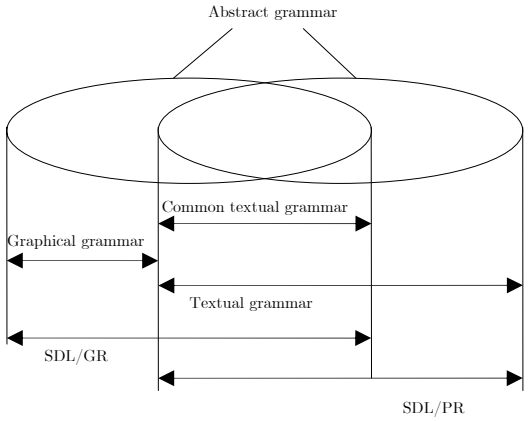


Abbildung 2.2.1-1:
SDL Grammatik

2.2 Compiler

„Ein Compiler (…) ist ein Computerprogramm, das ein in einer Quellsprache geschriebenes Programm in ein semantisch äquivalentes Programm einer Zielsprache umwandelt. Üblicherweise handelt es sich dabei um die Übersetzung eines von einem Programmierer in einer Programmiersprache geschriebenen Quelltextes nach Assemblersprache, Bytecode oder Maschinensprache (…) “.[2]

Im Fall des vorliegenden Compilers findet keine direkte Umsetzung der formalen Sprache SDL in ein ausführbares Programm statt, sondern eine Transformation der in SDL spezifizierten Systeme in die Programmiersprache C/C++. Man spricht hier daher auch von einem Transpiler oder Transcompiler.

Der Ablauf der Codeerzeugung von Compilern lässt sich in 2 Hauptphasen unterteilen: Die Analysephase und die Synthesephase.

[2] Aus http://de.wikipedia.org/wiki/Compiler

Die Analysephase kann wiederum in 3 Teile gegliedert werden:

- Lexikalische Analyse
- Syntaktische Analyse (Kapitel 2.2.2) und
- Semantische Analyse.

Bei der Lexikalischen Analyse wird mit Hilfe eines Scanners (oder Lexers) der Quelltext in spezielle Schlüsselwörter, Operatoren, Zahlen etc. (sog. Token) unterteilt. Die Funktion des hierbei eingesetzten Scanners ist in Kapitel 2.3.2 näher beschrieben. Die Token dienen der syntaktischen Analyse als Eingabe. Dabei wird der eingelesene Quellcode durch den Parser, vgl. Kapitel 2.3.3, auf syntaktische Korrektheit überprüft. In diesem Teil der Analyse wird untersucht, ob der Quellcode der der Quellsprache zugrunde liegenden Grammatik entspricht. Dabei erzeugt der Parser einen Syntaxbaum. An die syntaktische Analyse des Codes schließt sich eine semantische Analyse an. Hier prüft der Compiler z.B., dass Variablen vor ihrer Verwendung deklariert worden sind oder dass Zuweisungen mit korrekten Datentypen erfolgen. Er vermerkt die dafür notwendigen Informationen an den Knoten des Syntaxbaums. Der nach der semantischen Analyse zur Verfügung stehende Syntaxbaum wird auch als dekorierter Syntaxbaum bezeichnet.

Die 2. Hauptphase ist die Synthesephase. Auch diese Phase kann in 3 Unterphasen unterteilt werden:

- Zwischencodeerzeugung
- Codeoptimierung
- Codegenerierung

Die Zwischencodeerzeugung wird vor allem von Compilern verwendet, die Code für verschiedene Zielplattformen erzeugen sollen. Sie ist optional und kann auch, wie bei dem hier entwickelten Compiler, ganz entfallen. Bei der Codeoptimierung werden normalerweise Zuweisungen, Variablen- und Speicherzugriffe und vor allem auch Schleifendurchläufe optimiert. Die Optimierung findet in der Regel auf Zwischencode statt. Bei dem hier entwickelten Compiler wird keine Optimierung durchgeführt. Andere SDL zu C Compiler wie z.B. Cadvanced oder Cmicro von Telelogic Tau führen hingegen eine Optimierungen durch. Sie analysieren unter anderem die

Struktur des Systems. Sendet z.B. ein Prozess Signale an einen anderen Prozess, so wird der Pfad zwischen den Prozessen optimiert, d.h. das Signal wird direkt dem Zielprozess zugestellt, ohne die Strukturen dazwischen zu durchlaufen. Dies ist bei dem in dieser Arbeit entwickelten Compiler nicht erwünscht, da dadurch die 1:1 Abbildung zwischen SDL und dem generierten C++ Code verloren geht. Bei der Codeerzeugung wird aus dem (optimierten) dekorierten Syntaxbaum endgültig der Programmcode in der Zielsprache erzeugt.

2.2.1 Formale Sprachen und Grammatiken

Um den Begriff der formalen Sprache und der Grammatik zu erläutern, sind einige Definitionen (aus [1]) notwendig:

Ein *Alphabet* ist eine endliche, nicht leere Menge von Symbolen. Ist T ein Alphabet, so nennt man jede Folge $t_1 \dots t_n$ von Zeichen $t_i \in T$, $1 \le i \le n$, ein (nichtleeres) *Wort* über T.

Die *Zusammensetzung von Worten* $x = x_1 \dots x_n$ und $y = y_1 \dots y_m$, $x_i, y_j \in T$, ist festgelegt durch $xy = x_1 \dots x_n y_1 \dots y_m$

Es wird das *Leerwort* ε als neutrales Element bezüglich der Zusammensetzung von Worten definiert.

Die *Länge* eines Wortes x ist die Anzahl der Symbole in x und wird mit $|x|$ bezeichnet. Es gilt: $|\varepsilon| = 0$, $|t| = 1$ für alle $t \in T$ und $|xy| = |x| + |y|$ für alle Worte x, y über T.

Potenzen von Worten x bzw. Wortmengen X werden erklärt durch

$$x^0 = \varepsilon, x^1 = x \text{ und } x^i = x^{i-1}x \text{ für } i \ge 1$$

$$X^0 = \{\varepsilon\}, X^1 = X \text{ und } X^i = X^{i-1}X \text{ für } i \ge 1$$

Die *Sternhülle* und *Plushülle* einer Wortmenge X ist definiert durch

$$X^* = \bigcup_{i \ge 0} X^i, \qquad X^+ = \bigcup_{i \ge 1} X^i$$

Ist T ein Alphabet, dann ist T^* die Gesamtheit aller Worte über T einschließlich des Leerwortes; jede Teilmenge L von T^* heißt nun *formale Sprache*.

Eine *Chomsky Grammatik* G ist ein 4 Tupel $G = (N, T, P, S)$ mit

- N das Alphabet der *Nichtterminalsymbole*; sie werden i.a. mit Großbuchstaben wie A, B bezeichnet
- T das Alphabet der *Terminalsymbole*; sie werden i.a. mit Kleinbuchstaben wie a, b bezeichnet; es gilt immer $N \cap T = \phi$
- P eine endliche Menge von *Produktionen* der Gestalt $\alpha \to \beta$ (α "produziert" β) mit $\alpha \in (N \cup T)^* N (N \cup T)^*$ und $\beta \in (N \cup T)^*$
- S ein spezielles Nichtterminalsymbol, das sog. Startsymbol

$V = (N \cup T)$ wird als Gesamtalphabet bezeichnet.

Je nach der Gestalt der in P zugelassenen Produktionen definiert man vier Typen von Grammatiken. Eine Grammatik heißt

Typ 0 Grammatik, wenn die Gestalt der Produktionen nicht eingeschränkt ist.

Typ 1 Grammatik oder kontextsensitive Grammatik, wenn P nur Produktionen der Gestalt $\alpha \to \beta$ mit $|\alpha| \le |\beta|$ und eventuell die Produktion $S \to \varepsilon$ enthält, wobei $S \to \varepsilon$ nur dann zugelassen ist, wenn das Startsymbol S in keiner Produktion auf der rechten Seite auftritt.

Typ 2 Grammatik oder kontextfreie Grammatik, wenn P nur Produktionen der Gestalt $A \to \alpha$ mit $A \in N$ und $\alpha \in V^*$ enthält, d.h. die linke Seite jeder Produktion ist ein Nichtterminalsymbol.

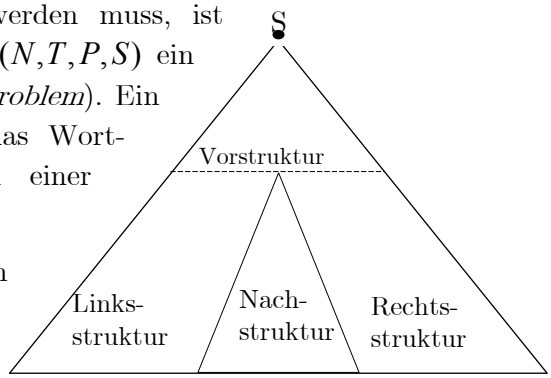
Typ 3 Grammatik oder reguläre Grammatik, wenn P entweder nur Produktionen der Gestalt $A \to \alpha$ mit $A \in N$ und $\alpha \in NT^* \cup T^*$ oder nur Produktionen $A \to \alpha$ mit $\alpha \in T^*N \cup T^*$ enthält.

Eine formale Sprache heißt vom Typ 0, 1, 2 oder 3, wenn sie von einer Grammatik des entsprechenden Typs erzeugt werden kann.

Kontextfreie Grammatiken wie die SDL Grammatik entsprechen der *Backus-Naur-Form (BNF).*

2.2.2 Syntaxanalyse von kontextfreien Grammatiken

Das Problem, das bei der Syntaxanalyse gelöst werden muss, ist festzustellen, ob mit einer gegebenen Grammatik $G = (N, T, P, S)$ ein Wort $w \in T^*$ erzeugt werden kann oder nicht (*Wortproblem*). Ein Algorithmus zur Syntaxanalyse löst immer auch das Wortproblem durch Feststellen, ob die Konstruktion einer syntaktischen Struktur gelingt oder nicht.



Strukturbaum 2.2.1-1

Algorithmen zur Analyse können in Tabellenstrategien (z.B. Early oder Cocke-Kasami-Younger) und ableitungsorientierte Strategien unterteilt werden, wobei hier die Tabellenstrategien nicht näher erläutert werden sollen. Detaillierte Informationen dazu sind in [1] zu finden.

Bei den ableitungsorientierten Analysestrategien können wiederum Klassifizierungen eingeführt werden. Man unterscheidet die Analyse von *Links nach Rechts* (Linksstruktur total erkannt) bzw *Rechts nach Links* (Rechtsstruktur total erkannt) sowie die *Top-Down* (Vorstruktur total erkannt, Nachstruktur unbekannt) und *Bottom-Up* (Nachstruktur total erkannt, Vorstruktur unbekannt) Analyse. Daraus ergeben sich 4 mögliche Kombinationen (Top-Down von Links nach Rechts, Top Down von Rechts nach Links, Bottom–Up von Links nach Rechts und Bottom –Up von Rechts nach Links), wobei in der Regel nur Ansätze von Links nach Rechts eine Rolle spielen.

2.2.2.1 LL(k)-Parser

LL(k) Parser gehören der Klasse der Top-Down-Parser an. Sie parsen die Eingabe von Links nach Rechts (**L**eft to right parse) und versuchen eine *Linksableitung* (**L**eftmost derivation) aus einer Menge von **k** vorausschauend gelesenen Token (*Lookahead*) zu erzeugen. Linksableitung bedeutet, dass versucht wird, immer das am weitesten links stehende *Nichtterminalsymbol* zu ersetzen. LL(k) Parser werden über eine Steuer- oder Parsertabelle gesteuert. Die Größe der Parsertabelle ist direkt von der Länge von k abhängig. Um die Tabelle möglichst klein zu halten, wird in der Praxis daher in der Regel k=1 verwendet. Die Erstellung einer eindeutigen Parsertabelle für einen LL(1) Parser ist nur dann möglich, wenn aus dem *Nichtterminalsymbol* (linke Seite der Produktionen) und dem *Lookahead* eine anzuwendende Produktion eindeutig zu bestimmen ist. Dazu ist es notwendig, die *FIRST* und *FOLLOW Mengen* jedes *Nichtterminalsymbols* der zugrunde liegenden LL(1)-Grammatik zu berechnen

Die FIRST Menge:

Für zwei Regeln der Form $A \to \alpha$ und $A \to \beta$, $\alpha, \beta \in V^*$ müssen Unterscheidungsmerkmale gefunden werden, um die richtige Regel bei der Expansion des Syntaxbaumes auszuwählen. $FIRST(\alpha)$ und $FIRST(\beta)$ liefern dabei alle k *Terminalsymbole*, mit denen α und β bei weiterer Ableitung beginnen können, bzw. ε,wenn sich α und β nach ε ableiten lassen.

Die FOLLOW Menge:

Für ein *Nichtterminalsymbol* A liefert $FOLLOW(A)$ alle k *Terminalsymbole*, die direkt hinter A stehen können.

Ann: Zwei Regeln der Form $A \to \alpha$ und $A \to \beta$ mit $\alpha \to \varepsilon$, d.h. $FIRST(\alpha) = \{\varepsilon\}$ und $FIRST(\beta) = \{t\}$.

Wenn nun $t \in FOLLOW(A)$ wäre, so könnte der Parser keine eindeutige Entscheidung fällen, sofern t das aktuelle Eingabesymbol ist, da sowohl die Ableitung $A \to \alpha \to \varepsilon$ als auch $A \to \beta$ in Frage kommen.

Die Berechnung der *FIRST* und *FOLLOW Mengen* ist algorithmisch recht einfach lösbar und kann z.B. in [1] gefunden werden.

Die Berechnung der Parsertabelle:

Die Tabelle enthält alle in der Grammatik vorkommenden *Nichtterminalsymbole* und alle in Frage kommenden Lookahead-Token als Index. Eine Produktion $A \to \alpha$ wird in (A, t) aufgenommen für alle $t \in FIRST(\alpha)$. Ist α nach ε ableitbar, so wird $A \to \alpha$ auch in (A, t') mit $t' \in FOLLOW(A)$ geschrieben.

Enthält die Tabelle Doppeleinträge, so ist die zugrunde liegende LL(k)-Grammatik nicht eindeutig. Mit Hilfe geeigneter Umformungen kann versucht werden, eine mehrdeutige Grammatik in eine eindeutige Grammatik umzuwandeln. Aus einer eindeutigen Steuertabelle kann direkt ein einfacher LL(k) Parser implementiert werden.

2.2.2.2 LR(k)-Parser

LR(k) Parser (**L**eft to right parse, **R**ightmost derivation, **k** Token Lookahead) wie der in dieser Arbeit verwendete Bison-Parser sind Bottom-Up-Parser, d.h. sie warten mit der Entscheidung der Reduktion der rechten Seite einer Produktion, bis alle Token der rechten Seite plus k weitere Token erkannt sind. Dazu ist es notwendig, dass bereits gelesene Token (und die zu dem entsprechenden *Nichtterminalsymbol* reduzierten rechten Seiten einer Produktion) im Speicher auf dem Stack abgelegt werden. Abhängig von Zustand des Stacks und dem *Lookahead* wird entweder eine *SHIFT* oder *REDUCE* Aktion ausgeführt.

Bei *SHIFT* wird das nächste Token der Eingabe auf den Stack kopiert. Kann ein *REDUCE* für eine Regel $A \rightarrow \alpha_1\alpha_2 \ldots \alpha_k$ ausgeführt werden, so werden $\alpha_k\alpha_{k-1} \ldots \alpha_1$ vom Stack genommen und anschließend A abgelegt. Nach der Reduktion muss der Automat wieder in dem Zustand seine Arbeit fortsetzen, der vorlag, bevor die Token der Produktion (hier: $\alpha_1\alpha_2 \ldots \alpha_k$) geshiftet wurden. Daher wird zu jedem Symbol auf dem Stack auch der aktuelle Zustand abgelegt. Auf die Berechnung der LR(k) Parsertabelle soll hier nicht weiter eingegangen werden. Auch hier gilt wie schon bei der Klasse der LL(k) Parser, dass die Tabelle eindeutig sein muss. Die Größe einer LR(k) Tabelle wächst exponentiell mit k, daher wird auch hier häufig k=1 als *Lookahead* verwendet. Eine weitere Reduktion der Tabelle ist durch Zusammenfassen verträglicher Zustände möglich. Dabei werden Einträge der Parsertabelle zusammengefasst, die sich nur in den *Lookahead* Mengen unterscheiden. In der Praxis kommen häufig spezielle LR(k) Parser, nämlich LALR (*Lookahead LR*) oder SLR (*Simple LR*) zum Einsatz. Diese Parser zeichnen sich durch eine weitere Reduktion der Steuertabellen aus.

2.3 Verwendete Werkzeuge

2.3.1 Telelogic Tau

Telelogic Tau [9][10] ist eine Entwicklungsumgebung zur Erstellung von verteilten Systemen mit SDL (SDL96) und ASN.1 [8] (**A**bstract **S**yntax **N**otation One), einer Plattform– und sprachunabhängigen Notation zur Spezifikation von Datenstrukturen auf hoher Ebene. Im Rahmen dieser Diplomarbeit kam die Version 4.5 zur Erzeugung von SDL Systemen zum Einsatz. Tau besitzt einen grafischen Editor zur einfachen und schnellen Erzeugung der Systeme und ist in der Lage, die für den hier entwickelten Compiler benötigten PR Dateien zu generieren. Außerdem ist es möglich, im Editor, durch spezielle Direktiven, C-Quellcode direkt in das SDL System einzubinden. Zusätzlich beinhaltet Tau einen Analyzer, der die mit Hilfe des Editors erstellten Systeme bereits auf ihre syntaktische und semantische Korrektheit überprüft. Tau bringt auch zwei eigene SDL nach C Compiler (Cadvanced und Cmicro (speziell für eingebettete Systeme)) mit, durch die direkt C-Quelldateien und damit auch ausführbare Programme der SDL Systeme erzeugt werden können. Cadvanced und Cmicro werden in Kapitel 4.6 mit dem hier entwickelten Compiler verglichen. Mit Hilfe des Targeting Expert, einem Konfigurationswerkzeug, kann direkt eine Applikation aus einem SDL System mittels Cadvanced oder Cmicro Compiler erzeugt werden.

2.3.2 Flex

Flex [2][3] ist ein Scannergenerator zur Unterteilung einer vorgegebenen Eingabe in Token. Die Eingabe kann entweder durch eine Datei oder durch die Standardeingabe erfolgen. Der Nutzer definiert Regeln in Form von regulären Ausdrücken, mit deren Hilfe die Eingabetoken erkannt werden. Zu jeder Regel kann eine Aktion in C-Code angegeben werden, die entsprechend ausgeführt wird. Flex stellt eine Reihe von vorgefertigten Ausdrücken (*Patterns*) zur Verfügung, mit denen alle gewünschten regulären Ausdrücke zusammengesetzt werden können. Außerdem können Startzustände sowie weitere Zustände definiert werden, um z.B. Kommentare o.ä. zu analysieren. Wechsel zwischen den Zuständen sind einfach durch einen entsprechenden Befehl im C-Code der Aktionen möglich.

Flex kann als eigenständiges Programm ausgeführt werden, wird aber hauptsächlich zusammen mit *yacc*, einem Parsergenerator verwendet. Dieser erwartet eine spezielle Routine („*yylex()*“) als Schnittstelle zum Auffinden des nächsten Eingabetoken. Siehe dazu das folgende Kapitel 2.3.3. Flex liefert beim Aufruf dieser Routine den Typ des nächsten erkannten Tokens. Zusätzlich kann in einer globalen Variable ein beliebiger Rückgabewert angegeben werden. *Yacc* erzeugt eine (Header) Datei mit den Definitionen aller Token der *yacc* Eingabe. Diese Headerdatei wird dann im Flex Scanner eingebunden. Neben der von *yacc* benötigten *yylex()* Routine können vom Benutzer eigene Routinen zur Analyse der Eingabe in C-Code geschrieben werden.

Letztendlich generiert Flex aus der vom Nutzer erzeugten Quelle eine C-Datei, die wie bereits erwähnt in weitere Projekte (*yacc* oder *Bison*) integriert oder mittels eines C-Compilers in ein ausführbares Programm überführt werden kann.

2.3.3 Bison

Bison [2][4] ist ein Parsergenerator. Das Programm ist aufwärts kompatibel zu *yacc*, d.h. Bison soll auch mit allen für *yacc* erzeugten Grammatiken ohne Veränderung arbeiten. Bison erzeugt für eine Beschreibung einer *kontextfreien* LALR(1) Grammatik (vgl. dazu auch Kapitel 2.2.2.2) ein C-Programm, um diese Grammatik zu analysieren. Die Beschreibung der Grammatik erfolgt dabei in *BNF*. LALR(1) Parser sind deterministisch, da die zugrunde liegende Grammatik immer eindeutig ist. Bison stellt zusätzlich noch Funktionalität zum parsen von indeterministischen kontextfreien Grammatiken, das sog. GLR (Generalisierte LR) Parsen zur Verfügung. Das Merkmal indeterministischer kontextfreier Grammatiken ist, dass trotz *Lookahead* die anzuwendende Regel nicht eindeutig bestimmt ist, also mehrere Aktionen zur Auswahl stehen. Bisons GLR Parser kann alle kontextfreien Grammatiken verarbeiten, deren Anzahl an Aktionen eines Ausdrucks endlich ist. Mögliche Fälle wären dabei z.B. ein *SHIFT* / *REDUCE* Konflikt (ein Ausdruck kann sowohl mit einer oder mehreren Produktionen reduziert werden oder das nächste Eingabetoken kann auf den Stack kopiert werden) oder ein *REDUCE* / *REDUCE* Konflikt (es stehen mehrere Produktionen zur Reduktion zur Verfügung). Erreicht Bison beim GLR Parsen eine Stelle in einem Ausdruck, bei der die Anwendung der nächsten Regel nicht feststeht, so wird der Stack entsprechend der Anzahl der Möglichkeiten kopiert und auf allen Stacks jeweils eine Aktion (*SHIFT* bzw. *REDUCE*) ausgeführt. Anschließend wird das Parsen in gewohnter Weise auf jedem der Stacks fortgesetzt. Ein Stack wird gelöscht, sobald feststeht, dass keine gültige Aktion mehr angewendet werden kann. Werden durch eine Reduktion mehrere Stacks identisch, so werden sie zu einem einzigen zusammengefasst.

Zur syntaktischen Analyse der Grammatik ist eine Beschreibung in *BNF* erforderlich. Zusätzlich kann zu jeder Produktion der Grammatik wie bei Flex eine Aktion in C-Code angegeben werden. Die Aktionen werden ausgeführt, sobald die Produktion reduziert wird. Die Werte der Token und Rückgebewerte der Nichtterminalsymbole der Produktionen können in den Aktionen mit `$x` angesprochen werden. Dabei bezeichnet `$1` das erste Token/Nichtterminalsymbol, `$2` das zweite usw. Rückgabewerte von Produktionen nach ihrer Reduktion können jeweils in `$$` der zugehörigen Produktion abgelegt werden, so dass sie weiteren Produktionen zur Verfügung stehen.

Um die semantische Analyse durchführen zu können, wird jedem Nichtterminalsymbol der Grammatik ein Datentyp zugeordnet. Dabei dürfen auch selbstdefinierte Datentypen zum Einsatz kommen.

Die Zusammenarbeit zwischen dem von Bison generierten Parser und dem durch Flex generierten Scanner erfolgt wie bereits in 2.3.2 beschrieben. Bison erzeugt wie Flex aus den zur Verfügung stehenden Informationen eine C-Datei sowie bei Bedarf die Headerdatei welche zur Zusammenarbeit mit Flex benötigt wird.

3 Der ConTraST Compiler

In dieser Arbeit wurde ein SDL zu C++ Compiler in der Programmiersprache C/C++ implementiert. Der Compiler selbst wurde nach objekt-orientierten Gesichtspunkten entwickelt, wie auch im folgenden Kapitel 3.1 beschrieben. Der generierte Code ist ebenfalls objekt-orientiert ausgerichtet (siehe Kapitel 4). Parallel zur Entwicklung des Compilers selbst fand die Programmierung von SDL/RE, einer Laufzeitumgebung (**R**untime **E**nvironment; Kapitel 4.5) streng nach ITU Z.100.F3 [7] durch Dipl. Technoinf. Ingmar Fliege statt. Da es sich bei der ITU Z.100.F3 um eine Spezifikation durch abstrakte Zustandsautomaten (**A**bstract **S**tate **M**achines) für die neueste Version von SDL (SDL2000) handelt, die dem Parser zugrunde liegende Grammatik jedoch dem älteren SDL96 entspricht, konnten nicht alle in SDL96 vorhandenen Spracheigenschaften verwendet werden oder mussten entsprechend abgewandelt und angepasst werden. Dies ist unter anderem in Kapitel 3.2 beschrieben. Hauptgrund für die Entscheidung, beim Parser die Spezifikation von SDL96 zu verwenden, ist die fehlende Unterstützung von SDL2000 durch geeignete Entwicklungsumgebungen. Außerdem ist es möglich mittels einer Konfigurationsdatei den gewünschten vom Compiler zu unterstützenden SDL-Sprachumfang festzulegen. Die Konfigurierbarkeit ist ein zentraler Bestandteil und schlägt sich daher auch in der der Namensgebung nieder: **Con**figurable **Tran**spiler for **S**DL to C++ **T**ranslation (ConTraST). Dies ist genauer in Kapitel 3.2 beschrieben.

3.1 Objektorientierter Aufbau

3.1.1 Scanner und Parser des ConTraST Compilers

ConTraST verwendet einen in Kapitel 2.3.3 beschriebenen, von *Bison* erzeugten Parser. *Bison* benutzt den in Kapitel 2.3.2 vorgestellten *Flex*-Scannergenerator zum analysieren der Eingabedateien. Leider unterstützt *Bison* derzeit nicht die Generierung eines C++ Parsers. Auch der Versuch, durch *Flex* einen C++ Scanner erzeugen zu lassen, scheiterte aufgrund einiger Fehler im vom Scannergenerator erzeugten Code. Die Fehler konnten wegen der knapp bemessenen Entwicklungszeit nicht behoben werden. Daher sind Scanner und Parser im Kern des Compilers reine C-Komponenten. Die Tokenliste des Scanners ergibt sich direkt aus der SDL96-Grammatik nach ITU Z.100 [6]. Zusätzlich sind im Scanner Zustände zum Analysieren von Kommentaren, C-Quellcode Abschnitten sowie `include` Anweisungen zur Einbindung weiterer PR Dateien programmiert. Der Scanner wird im weiteren Verlauf der Codeerzeugung auch dazu verwendet, bereits geparste und in den Syntaxbaum aufgenommene SDL-Ausdrücke (*Expressions*) zu parsen. Gründe und Erläuterungen zu diesem Umstand sind in Kapitel 3.1.2 zu finden.

Dem Parser liegt die nach in [6] spezifizierte Grammatik zu Grunde. Bei der Übernahme der Grammatik von der Spezifikation in die Quelldatei des Parsers wurde versucht, die auftretenden Mehrdeutigkeiten der SDL Grammatik soweit als möglich zu beseitigen. Dies gelang weitestgehend durch Anwendung geeigneter Transformationsregeln:

Umwandlung linksrekursiver Regeln in rechtsrekursive Regeln:

Beispiel:	wird transformiert in
`A ::= A t1`	`A ::= u1 A``
`A ::= A t2`	`A ::= u2 A``
`A ::= u1`	`A`::= t1 A``
`A ::= u2`	`A`::= t2 A``
	`A`::= ε`

Dabei entstehen neue Epsilon Regeln.

Elimination gemeinsamer Anfangsstücke in Regeln für ein Nichtterminalsymbol durch Faktorisierung:

Beispiel:	wird transformiert in
`A ::= B t1`	`A ::= B A``
`A ::= B t2`	`A`::= t1`
	`A`::= t2`

Die *FIRST* Mengen sind für beide Regeln gleich

Leider stellte sich bei der Hinzunahme der Unterstützung von parametrisierten Prozeduren zu Ende der Entwicklung des ConTraST Compilers heraus, dass die dabei auftretenden Mehrdeutigkeiten nicht ohne erheblichen Aufwand beseitigt werden können. Daher wurde entschieden, die Mehrdeutigkeiten nicht zu beseitigen, sondern beim Parser lediglich die GLR Funktion (siehe dazu Kapitel 2.3.3) zu aktivieren. Außerdem wurde die Grammatik dahingehend erweitert, dass auch von Telelogic Tau generierte PR Dateien ohne Abwandlung analysiert werden können. Die dazu erforderlichen Erweiterungen sind allerdings minimal: So wurde z.B. in `task` Produktionen der Grammatik die Unterstützung für den in Tau erlaubten C-Quellcode hinzugefügt. Außerdem sind bei Tau generierten PR Dateien einige `task` Produktionen in `{` und `}`geklammert. Bedingt durch die ASN1 Unterstützung mussten zudem einige `CHOICE` Produktionen bei der Datenstrukturdefinition eingefügt werden. Nicht unterstützt werden allerdings Makrodefinitionen und –aufrufe. Die entsprechenden Regeln sind auch als einzige der SDL Spezifikation nicht im Parser enthalten. Bis auf Makro-Regeln wurde also die komplette Spezifikation umgesetzt, auch wenn (noch) nicht für alle Teile Code generiert oder von der Laufzeitumgebung unterstützt wird (siehe auch Kapitel 4.6).

3.1.2 Aufbau des Syntaxbaums und dessen semantische Korrektheit

Um die semantische Korrektheit der syntaktischen Regeln zu gewährleisten, werden entsprechende objekt-orientierte Programmiervorgaben geschaffen. Dazu wurde mit `SDLContainer` eine Oberklasse programmiert, die primär dem Aufbau des erforderlichen Syntaxbaums dient. Sie enthält einen Vektor, in den die Söhne eines Containers wiederum als `SDLContainer`–Objekt eingefügt werden können. Des Weiteren beinhaltet ein `SDLContainer` Variablen und Methoden, um Typ und Name des Containers zu speichern und zu verändern sowie `SDLContainer`–Objekte (bzw. von `SDLContainer` vererbte Objekte) und komplette `SDLContainer`–Vektoren hinzuzufügen. Außerdem sind hier die allgemeinen Methoden `print()` und `generateCode()` implementiert. Sie dienen dazu, den Syntaxbaum zu traversieren und die entsprechenden `print()` und `generateCode()` Methoden der im Vektor gespeicherten Söhne aufzurufen. Diese beiden Methoden sind als virtuell deklariert, da sie in allen von `SDLContainer` abgeleiteten Klassen an deren spezielle Eigenschaften anzupassen sind. `Print()` wird benutzt, um den Syntaxbaum vor und nach erforderlichen Transformationen, welche genauer in

Kapitel 3.1.3 erläutert sind, in textuell lesbarer Form in eine Datei zu schreiben. Dies dient in erster Linie dem Auffinden von Parse- und Transformationsfehlern. Auf die Funktionalität, die die `generateCode()` Methoden zur Verfügung stellen, wird genauer in Kapitel 4 eingegangen.

Mit 42 von `SDLContainer` erbenden Klassen können dann die wichtigsten Informationen des SDL Systems in Form von Objekten gekapselt werden. Bei der Entscheidung, für welche SDL-Strukturen eigene Klassen anzulegen sind, war sowohl die SDL-eigene Strukturierung als auch die objekt-orientierten Anforderungen des SDL/RE ausschlaggebend. Die wichtigsten Klassen sind dabei diejenigen, die Struktur repräsentieren wie `GlobalSystem` (bildet die Wurzel des Syntaxbaums, ist aber keine SDL-spezifische Klasse), `Package`, `System`, `Block`, Kanäle(`Channels`) bzw. `Signalrouten`, `Gates` usw., aber auch solche, die dazu dienen, das Verhalten des SDL Systems zu beschreiben: Zustände (`States`) und `Transitionen`. Zudem existieren Klassen, die dazu dienen, die von anderen Klassen benötigten Informationen zur Verfügung zu stellen. Als Beispiele wären hier Pfade (`Path`) für `Gates` oder `Channel` und `Variable` zu nennen.

Wie bereits in Kapitel 2.3.3 erläutert ist es für die semantische Analyse notwendig, allen Produktionen Typen zuzuweisen. Auch hier kommen alle 42 Klassen zum Einsatz. Hat der Parser alle notwendigen Token einer Produktion auf dem Stack abgelegt, so wird beim Reduzieren der Produktion die zugehörige, in C++ Code implementierte Aktion ausgeführt. Ein Großteil dieser Aktionen beschreibt lediglich, wie die Objekte der rechten Seite der Produktion in das der linken Seite der Produktion zugeordnete (und in der Regel neu erzeugte) Objekt einzufügen ist. Das Einfügen kann meist durch die von der Oberklasse `SDLContainer` zur Verfügung gestellten und vererbten Methoden erfolgen. Eine Ausnahme hiervon bilden die Klassen `GlobalSystem`, `Package`, `System`, `Block`, `Prozess` und `Service`. Sie überschreiben die `SDLContainer` Methoden zum Einfügen von Söhnen. Diese Klassen besitzen neben dem Standardvektor weitere Vektoren, um gleiche bzw. verwandte Objekte einzusortieren. Dadurch entfällt das Suchen nach bestimmten Objekten im Standardvektor bei der Transformation des Syntaxbaums und der Codegenerierung für diese Objekte. Außerdem wird für jede dieser Klassen auch bei der Codeerzeugung eine entsprechende Klasse erzeugt. Näheres dazu erklärt Kapitel 3.1.4 .

Des Weiteren sind noch die Klassen *States* und *Transition* zur Kapselung des Verhaltens eines in SDL spezifizierten Zustandsautomaten zu nennen. Auch sie tauchen im generierten Code wieder auf. Die Codeerzeugung von *States* und *Transitionen* sind voneinander abhängig und am engsten von allen Klassen miteinander verzahnt. Zur Transitionsklasse ist noch festzuhalten, dass mit ihr alle auftretenden Arten von in SDL spezifizierbaren Transitionen beschrieben werden können. Dazu zählen insbesondere Starttransition, Task, C-Code (Tau spezifisch), Returnanweisung, Timerset und -reset, (Prioritäts-) Signalinput, Signaloutput, Verzweigung (Decision), Save, Spontane Transition, Konnektoren (Free_Action), Erzeugung von neuen Prozessen (Create), Marken (Label), Continous Signal und Prozeduraufrufe.

Alle Ausdrücke (*Expressions*), die in SDL Tasks angegeben sein können, werden geparst, um die syntaktische Korrektheit festzustellen. Anschließend werden die Ausdrücke in textueller Form (d.h. als C++ Strings) zusammengesetzt und in einem Expression-Objekt gespeichert. Bevor C++ Code für sie erzeugt werden kann, müssen die Strings erneut mit Hilfe des Scanners in einzelne Token zerlegt werden. Diese Verarbeitungsweise bot zum Zeitpunkt der Implementierung der Klassen und der Aktionen des Parsers die flexibelsten Möglichkeiten für die Codegenerierung. Die endgültige Entscheidung, wie die einzelnen Komponenten der Ausdrücke, insbesondere die darin enthaltenen Schlüsselwörter zur Datenmanipulation, später umgesetzt werden sollten, konnte erst gegen Ende der Fertigstellung der SDL/RE erfolgen. Eine effizientere

Reimplementierung der Expression-Klasse sowie aller damit in Verbindung stehenden Regeln (Aktionen) der SDL-Grammatik kam allein aus zeitlichen Gründen nicht mehr in Frage, soll aber Teil der zukünftigen Arbeit am Compiler sein.

3.1.3 Notwendige Transformationen

3.1.3.1 Transformationen des Syntaxbaums

Bevor aus dem Syntaxbaum Code erzeugt werden kann, sind gewisse Transformationen durchzuführen und Informationen zu sammeln. Dazu zählen im Wesentlichen:

- Transformation von System-, Block-, Prozess- und Service-Definitionen in System-, Block-, Prozess- und Service-Typdefinitionen
- Sammeln und Hinzufügen von Scope-Informationen bei Variablen-, Signal- und Timerdefinitionen

Wie bereits zu Anfang von Kapitel 3 erwähnt, basieren der ConTraST Compiler und die Laufzeitumgebung auf unterschiedlichen Versionen der Sprache SDL. ConTraST liegt die Gramma-

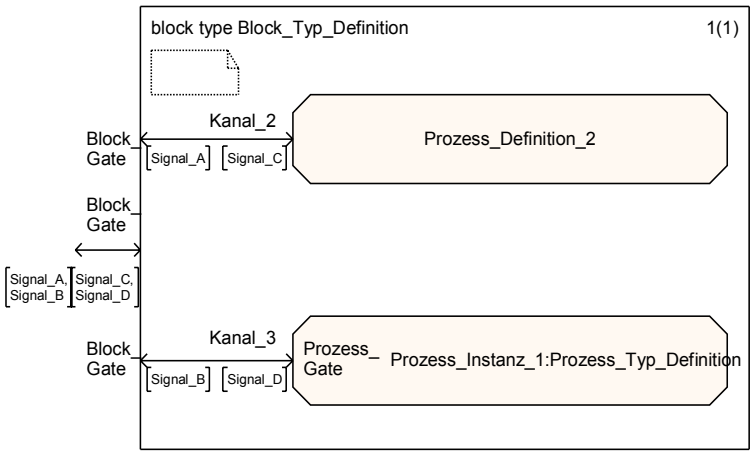


Abbildung 3.1.3-2

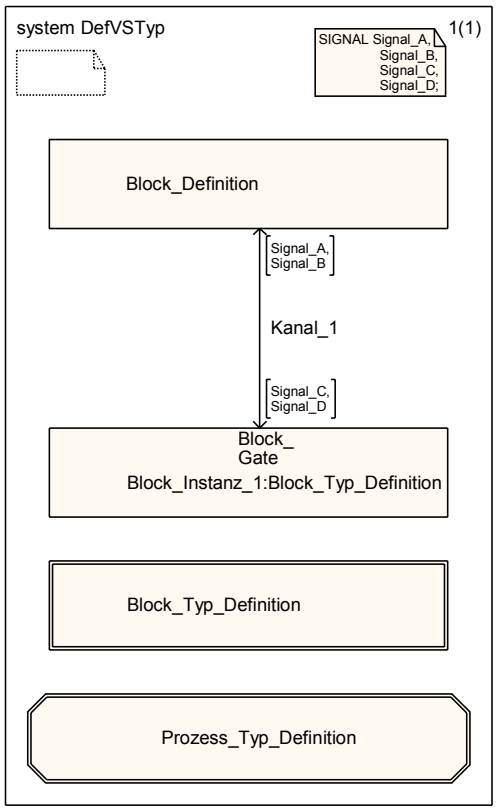


Abbildung 3.1.3-1

tik von SDL96 gemäß [6] zugrunde, wohingegegen die Laufzeitumgebung die *ASM* nach SDL2000 Standard implementiert. Die Laufzeitumgebung unterstützt lediglich zu instanziierende Strukturtypen wie z.B. `System-`, `Block-` oder `Prozess`-Typen. Im Gegensatz dazu sind bei der Erstellung von Systemen in Telelogic Tau auch Definitionen von einzelnen `Blöcken` oder `Prozessen` erlaubt. Dieser Umstand erfordert die Transformation dieser Strukturen in die entsprechenden Strukturtypen mit entsprechenden Instanziierungen. Bei der Umwandlung einer Strukturdefinition in eine Typdefinition müssen hauptsächlich neue Gates für die Strukturtypen erzeugt und an die entsprechenden Kanäle angefügt werden (siehe auch Abbildung 3.1.3-2 und Abbildung 3.1.3-1). Außerdem müssen evtl. Signalrouten in Kanäle umgewandelt werden, da Gates lediglich für Kanäle definiert sind. Kanäle und Signalrouten unterscheiden sich dadurch, dass für Kanäle eine Verzögerung (*delay*) für die transportierten Signale angegeben werden kann.

Da die Verzögerung von Signalen bei Kanälen nicht von der Laufzeitumgebung unterstützt wird, können hier Signalrouten und Kanäle als identisch angesehen werden. Die Transformation einer Signalroute in einen Kanal ist daher lediglich eine Typ-Umbenennung des SDLContainer-Objekts.

Im Detail läuft die Transformation der Strukturdefinitionen in 6 Schritten ab:

- Generierung einer Instanzdefinition mit dem ursprünglichen Namen der Strukturdefinition
- Generierung eines neuen Typ-Objekts aus dem ursprünglichen Definitions-Objekt (erfolgt im Wesentliche durch Generierung eines neuen Namens und Umsetzen der Typ-Variable; alle Söhne des Definitions-Objekts werden ohne Veränderung übernommen)
- Erzeugung eines neuen `Gate`-Objekts
- Entfernen aller Verbindungsreferenzen zur ursprünglichen Definition (d.h. dem jetzigen Typ-Objekt)
- Hinzufügen des neuen `Gate`-Objekts zu allen bestehenden Verbindungen (eingehende wie auch ausgehende Verbindungen)
- Hinzufügen von Typ- und Instanz-Objekten an den entsprechenden Stellen im Syntaxbaum

Die Erzeugung eines neuen `Gate`-Objekts beinhaltet das Auffinden aller ein- und ausgehenden Signale. Dieser Vorgang wird abhängig vom zugrunde liegenden Objekt auf zwei unterschiedliche Arten erledigt. Bei Service- und Prozess-Objekten werden alle Signale durch Durchsuchen der `Transitionen` gefunden, wohingegen bei System- und Block-Objekten die Informationen aus beinhalteten `Channel`- bzw. `Signalroute`-Objekten herangezogen werden. Beim Hinzufügen des neuen `Gates` zu bestehenden Verbindungen muss zusätzlich darauf geachtet werden, dass das neu erzeugte `Gate` auch zu allen Verbindungen zur nächst höheren Strukturebene zugefügt wird.

Ein weiterer zentraler Punkt bei der Codegenerierung ist das Sammeln von Scope-Informationen, d.h. der Informationen über Gültigkeitsbereiche. Scope-Informationen müssen sowohl für Signale als auch für alle im System vorhandenen Datentypen ermittelt werden, und zwar bevor die eigentliche Codeerzeugung stattfinden kann. Um diese Informationen zu sammeln, muss der Syntaxbaum traversiert werden und in den strukturbildenden `System`-, `Block`-, `Prozess`- und `Service`-Objekten nach Definitionen von Signalen und Datentypen gesucht werden. Anschließend werden die gefundenen Informationen in einer Daten bzw. Signal Map mit ihrem ermittelten Scope verknüpft. Bei der Codegenerierung kann die entsprechende Map nach dem ermittelten Scope abgefragt werden. Die Ermittlung der Scope-Informationen beim Parsen der Eingabe macht hier, wegen der möglicherweise erforderlichen Transformationen und den damit verbundenen Umformungen im Syntaxbaum, weniger Sinn.

3.1.3.2 Transformationen bei der Codeerzeugung

Neben den Transformationen die die Struktur des Syntaxbaums verändern, werden auch Transformationen direkt bei der Codeerzeugung durch ConTraST notwendig. Dazu zählen

- Umwandlung von `ANY` Zuständen
- Umwandlung von `ANY` Signalen beim speichern (*saven*) von Signalen
- Hinzufügen von `internal` *Channel* und den entsprechenden *Gates*
- Einfügen von Einsprungmarken in Form von *switch-case* Anweisungen in Transitionen

Eingabesignale eines mit `*` gekennzeichneten Zustands (`ANY` Zustand) können in allen vorhandenen Zuständen eines `Prozesses` bzw. `Services` empfangen werden. Die zum `ANY` Zustand gehörige Transition wird wie bei allen anderen Zuständen auch in einer eigenen Klasse abgehandelt. Normalerweise wird für alle explizit benannten Zustände eine eigene Zustandsklasse erzeugt, nicht jedoch für `ANY` Zustände. Vielmehr muss, wenn für alle anderen Zustände und Transitionen der Code erzeugt wurde, in jeden Konstruktor einer Zustandsklasse alle zur `ANY` Transition gehörigen Eingangssignale hinzugefügt werden. Gelöst wird dieses Problem durch Streams. Dabei wird pro Zustand ein neuer Stream erzeugt, in den alle Informationen des Zustandes einfließen. Eine Liste verwaltet alle Stream-Objekte der einzelnen Zustände. Treten `ANY` Zustände auf, so wird diese Liste durchlaufen und die `ANY` Transition mit den entsprechenden Eingabesignalen in jeden Stream eingefügt. Erst wenn der komplette Code für alle Zustände und Transitionen erzeugt wurde wird der Inhalt der Streams in die Datei geschrieben.

Die Codeerzeugung für `ANY` Signalen bei `Save` ist nicht weniger aufwändig. Alle Signale die ein *Prozess* / *Service* potenziell empfangen kann, sollen aufbewahrt werden. Ausgenommen sind dabei diejenigen Signale, die explizit in dem zu `Save` gehörigen Zustand als Eingabesignale auftreten. Auch hier müssen wieder entsprechende Listen geführt werden. Zum einen eine Liste aller Signale (inkl. Timersignale), die der `Prozess` bzw. `Service` empfangen kann, zum anderen müssen aus einer Kopie dieser Liste entsprechend die Signale entfernt werden, die im *Save*-Zustand explizit als Eingabesignale angegeben sind. Der verbleibende Rest an Signalen ist somit die Menge der zur `Save`-Anweisung hinzuzufügenden Signale.

Zusätzlich zu den explizit vorhandenen Gates und Kanälen wird bei `Prozessen` und `Services` ein `internal` *Channel* mit den entsprechenden *Gates* eingefügt. Über internal *Channels* können alle ausgehenden Signale auch wieder an den `Prozess` (bzw. `Service`) selbst geschickt werden.

Jede Anweisung einer Transition besitzt laut Spezifikation ein Continue-Label, das die weitere Ausführung nach Abarbeitung der Anweisung vorgibt. Die Continue-Label sind nicht explizit implementiert, sondern implizit durch das sequentielle Abarbeiten der Anweisungsfolgen einer Transition vorhanden. Die auf Anweisung A direkt folgende Anweisung B wird somit als die durch das Continue-Label von A beschriebene Folgeanweisung angesehen. Um dennoch die durch SDL definierten Sprungmarken im Code umsetzen zu können, ist jede Transition als *switch-case* Anweisung implementiert. Jedes Label wird auf einen *case* Einsprungpunkt abgebildet. Die Sprungmarken sind entweder explizit vom Nutzer durch Verwendung von Konnektoren in SDL generiert oder werden implizit bei der Erzeugung von PR Dateien durch Telelogic Tau erzeugt. Bei den implizit generierten Marken handelt es sich um Marken bei Verzweigungen (Decisions) oder im Zuge der Code-Wiederverwendung bei Schleifen in der SDL Spezifikation. Außerdem werden Sprungmarken bei Prozeduraufrufen notwendig, um die nach dem Aufruf folgenden Anweisungen der Transition weiter fortführen zu können. Durch die Umsetzung durch

switch–case ist es möglich, die Stelle des Einsprungs auch über Transitionsgrenzen hinweg einfach zu realisieren. Den Sprung zu einer neuen Marke `L` der Transition `T` vollzieht die von der Vaterklasse `Transition` geerbte Methode `Skip(T, L)`.

3.1.4 Codegenerierung

Bei der Codeerzeugung wird für jedes `Package`- und `System`-Objekt eine eigene Code- und Header Datei angelegt. In diesen Dateien wird der generierte Code der Objekte und ihrer Söhne festgehalten. Ein detaillierter Überblick über das Format und Aussehen des erzeugten Codes liefert Kapitel 4. Die Reihenfolge bei der Codegenerierung ist fest vorgegeben. Im Gegensatz zur Transformation des Syntaxbaums, die Bottom-Up erfolgt, durchläuft die Codegenerierung den Baum beginnend bei der Wurzel Top-Down. Für jedes struktur- und verhaltensrepräsentierende Objekt des Syntaxbaums wird analog eine C++ Klasse generiert. Die Klassen der Struktur-Objekte (`System`, `Block`, `Prozess`, `Service` und `Prozedur`) basieren auf den Informationen, die die zugehörigen Typ-Definitionen in der SDL Spezifikation zur Verfügung stellen. Sie erben dabei von entsprechenden Klassen die die Laufzeitumgebung definiert. Nacheinander wird in den Strukturklassen Code erzeugt für

- Datenstrukturen
- Variablen
- Timer (nur Prozesse und Services)
- Signale
- Gates (`CREATEGATES`)
- Kanäle (`CREATEALLCHANNELS`)
- Transitionen und Zustände
- Substrukturen (`CREATESUBSTRUCTURES`) sowie
- ein entsprechender Konstruktor zur Instanziierung der Klasse

Für in SDL neu definierte Datenstrukturen werden ebenfalls neue C++ Klassen erzeugt. Die in den SDL Strukturen (`STRUCT` oder `CHOICE`) enthaltenen Variablen werden innerhalb der C++ Strukturen in *struct* bzw. *enum* gekapselt. Die C++ Datenstrukturklassen implementieren zusätzlich Zugriffsoperatoren für die in *struct* / *enum* enthaltenen Variablen. Außerdem werden Zuweisungs- und Vergleichsoperatoren definiert und Methoden zum Kodieren (*encode*) und Dekodieren (*decode*) der Datenstrukturen in den elementaren Datentyp *Octet_string* zur Verfügung gestellt. Alle elementaren Datentypen und Typgeneratoren die Telelogic Tau zur Auswahl stellt sind von der Laufzeitumgebung in eigenen Klassen mit entsprechenden Operatoren und Funktionen (*encode* / *decode*) gekapselt. Sie werden auf diese Weise ebenfalls objekt-orientiert zugänglich gemacht. Dadurch werden sämtliche im System benutzten Variablen zu Objekten. Da diese Art der Interpretation sich von der Interpretationsweise von Telelogic Tau grundsätzlich unterscheidet, müssen bei der Codeerzeugung auch hier Umwandlungen der Variablenzugriffe und Funktionsaufrufe erfolgen. Während Telelogic Tau die Attribute von Variablen durch Funktionsaufrufe mit der Variablen als Parameter erhält, findet die Attributabfrage hier in einem eher C/C++ ähnlichen Syntax statt. Die unterschiedliche Art der Funktionsaufrufe soll an einem Beispiel verdeutlicht werden:

Um die Länge eines Arrays `A` in Telelogic Tau auszulesen, wird der Aufruf von `length(A)` verwendet.

Der ConTraST Compiler muss daraus hingegen einen Aufruf der Art `A.length()` generieren.

Diese auf den ersten Blick trivial erscheinende Umformung erfordert eine nicht unerhebliche Anstrengung, da wie bereits erwähnt, alle Ausdrücke als C++ Strings in `Expression`-Objekten vorliegen. Hier muss zuerst das entsprechende Schlüsselwort (`length`) erkannt und anschließend das zugehörige Objekt gefunden und korrekt extrahiert werden. Bei komplizierteren (verschachtelten) Ausdrücken ist das korrekte Erkennen und Ausschneiden des Objekts entsprechend aufwändig und fehleranfällig.

Der Code für Signale, Timer und Gates wird in eigenen Klassen innerhalb der generierten Strukturklassen dargestellt. Auch sie erben von Klassen der Laufzeitumgebung. Eventuell vorhandene Parameter werden in den Klassen als Variable mit entsprechendem Typ verwaltet. Als Besonderheit ist hier nur die größere Anzahl an Konstruktoren zu nennen. Neben dem (parametrisierten) Standardkonstruktor stellt jede Signalklasse einen Konstruktor für die zielgerichtete Signalversendung mit `to` und `via` (oder beidem) zur Verfügung.

Ein Kanal besteht aus einem (unidirekt) oder zwei (bidirekt) Kanalpfaden und einem eindeutigen Namen. Die Kanalpfade verbinden zwei Gates in einer Richtung miteinander. Außerdem beinhalten sie eine Liste von Signalen, die jeweils über sie verschickt werden können. Die Erzeugung der notwendigen Kanal-Objekte erfolgt in `CREATEALLCHANNELS`, einer Methode die durch Vererbung jedem Strukturobjekt durch die Laufzeitumgebung zur Verfügung gestellt wird.

Bei der Codegenerierung der verhaltensrepräsentierenden Objekte `Transition` und `State` werden ebenfalls eigene Klassen erzeugt. Eine Transition endet immer mit einem Zustand, der wiederum (abhängig von Eingabesignalen) auf verschiedene Transitionen verweisen kann. Diese engen Verzahnungen und Abhängigkeiten machen hier die Codeerzeugung nicht einfach. Zusätzlich können `Transitions`-Objekte wie schon in Kapitel 3.1.2 erwähnt eine Vielzahl von SDL-Transitionen kapseln. Die Unterscheidung welche spezifische SDL-Transition ein `Transitions`-Objekt darstellt ist in einer eigenen Transitions-Typ-Variable festgehalten. Die Codeerzeugung erfolgt für Transitionen durch Abfrage dieser Variable innerhalb einer *switch* Anweisung in Verbindung mit der Ausführung des der Entscheidung zugehörigen Anweisungsteils. Jede `Transitions`-Klasse ist von einer Oberklasse Transition der Laufzeitumgebung abgeleitet und implementiert die Methode `fire()`. Innerhalb dieser Methode wird das Verhalten gemäß den SDL Vorgaben festgehalten und von der Laufzeitumgebung ausgeführt. Zusätzlich beinhaltet die Oberklasse Methoden zur Signalausgabe (`Output`), setzen und resetten von Timern (`Set` und `Reset`), erzeugen und terminieren von Prozessen (`Create` und `Stop`) sowie zur Angabe des Folgezustands (`NextState`). Die Zustandsklasse erbt von der Oberklasse SDLState. Diese stellt folgende Methoden bereit: `addTransition(T, S)` und `addPriorityTransition(T, S)` zur Angabe der nächsten Transition `T` bei (bevorzugter) Eingabe des Signals `S`, `addContinousSignal(T, B)` und `addSpontanousTransition(T, B)` zur Angabe der Folgetransition `T` bei Erfüllung des booleschen Ausdrucks `B` sowie `addSave(S)` zum Speichen von Eingabesignalen `S`. Diese Methoden werden bei Bedarf dem Konstruktor der entsprechenden Zustandsklasse hinzugefügt.

In den Konstruktoren der Strukturobjekte (`Prozess`, `Service` oder `Prozedur`) werden Variablen initialisiert und die entsprechenden Zustände miterzeugt. Zusätzlich wird durch die von der Laufzeitumgebung vererbten Funktion `addStartTransition(S)` die Starttransition des Strukturobjekts in den Konstruktor eingefügt. Durch diese Art der Implementierung konnte

auch im Bereich der Verhaltensrepräsentation eine gut lesbare (nahezu) 1:1 Abbildung zwischen dem in SDL spezifizierten System und dem erzeugten Code erreicht werden.

Für jede in einem Struktur-Objekt enthaltene Instanziierung weiterer Substrukturen wird eine entsprechende C++ Instanz der Substruktur innerhalb der von der Laufzeitumgebung vorgegebenen Methode `CREATESUBSTRUCTURES` erzeugt. Damit ist auch auf Strukturebene eine übersichtliche 1:1 Abbildung von SDL nach C++ gelungen.

Für Prozeduren erfolgt die reine Codeerzeugung analog der Erzeugung von `Prozess`- und `Service`-Code. Prozeduraufrufe können optional für zustandslose Prozeduren optimiert werden. Standardmäßig soll ein Prozeduraufruf mittels einer `CALL` Direktive realisiert werden. Bei der Optimierung wird direkt eine neue Instanz der entsprechenden `Prozedur`-Klasse erzeugt und durch deren `fire()` Methode die Ausführung erzwungen. Anschließend wird noch (falls nötig) der Rückgabewert im aufrufenden Objekt gespeichert, bevor mit der normalen Transitionsausführung fortgefahren wird. Wird der Prozeduraufruf mittels der `CALL` Anweisung realisiert, so muss der Rückgabewert in der Prozedur gespeichert und anschließend an den aufrufenden `Prozess` / `Service` übergeben werden. Dies bereitet in der Praxis noch Probleme, da die Spezifikation der SDL Laufzeitumgebung (ITU Z.100.F3) an dieser Stelle einige Ungenauigkeiten aufweist. Daher werden im Moment nur einfache (zustandslose) Prozeduren und optimierte Prozeduraufrufe unterstützt, obwohl für alle Prozeduren Code erzeugt werden kann.

3.2 Erweiterungen und Einschränkungen des ConTraST Compilers

Bei der Umsetzung der SDL96 Grammatik innerhalb des Compilers wurden einige zusätzliche Regeln eingeführt, um spezielle Erweiterungen von Telelogic Tau zu unterstützen:

- Unterstützung für die `#CODE` Direktive zur Eingabe von C/C++ Code innerhalb von Tasks im Scanner und Paserteil von ConTraST hinzugefügt
- Regeln für Tau spezifische `task` Darstellung mit `` `{` `` und `` `}` `` sowie Unterstützung für darin enthaltene Anweisungsfolgen im Parser hinzugefügt.
- Neben `STRUCT` auch `CHOICE` Definitionen in Datenstrukturen im Parser. Die `CHOICE` Unterstützung ist eine Tau Erweiterung zu ASN 1 [8]
- Unterstützung der leeren Initialisierung von Datentypen durch `` `(. .)` `` im Parser

Allerdings sind nicht alle Komponenten der SDL Grammatik implementiert:

Es werden z.B. keine Makros (Definitionen, Aufrufe usw.) unterstützt. Außerdem werden einige nicht SDL spezifische, aber von Tau unterstützte Eigenschaften nicht in die Grammatik eingebunden. Dazu zählen z.B. `if` und `for Statements` direkt in `tasks`. Auch Zeilenumbrüche innerhalb von Namen und Identifiern (realisiert durch `` `_` ``) in Telelogic Tau sind zurzeit nicht unterstützt.

Aber auch die Art der Codeerzeugung, speziell bei Ausdrücken in `tasks` bringt einige Einschnitte mit sich. So kann kein Code erzeugt werden, wenn innerhalb von Zuweisungen Prozeduraufrufe stattfinden und direkt mit den Rückgabewerten weitergerechnet werden soll wie z.B.:

```
X = (CALL GetRandID) MOD 100
```

Stattdessen muss der Prozeduraufruf explizit vorher erfolgen:

```
Result:= (CALL GetRandID); X = Result MOD 100
```

Weitere Einschränkungen betreffen nicht die SDL96 Grammatik an sich, sondern sind Einschränkungen, die durch fehlende Unterstützung durch die Laufzeitumgebung hervorgerufen werden. Da die *ASM* auf SDL2000 basiert, besteht z.B. keine Unterstützung für *Substrukturen* von Kanälen und Blöcken oder die *View* Funktionalität. Andererseits sind einige Funktionalitäten nicht erwünscht oder nicht in der Laufzeitumgebung implementiert. Dazu zählen u.a. *remote*, *extern* und *import* oder die Verzögerung (*delay*) von Kanälen. Die *ANY-Expression* ist nicht implementiert. Außerdem sind einige Funktionen auf Grund mangelnder Entwicklungszeit ausgespart worden. Dies sind z.B. Operator- und Generatordefinitionen. All diese Informationen sind aber dennoch im Syntaxbaum abgelegt, so dass eine spätere Unterstützung von Seiten des ConTraST Compilers durch Hinzufügen einer codeerzeugenden Methode erfolgen kann.

Eine zusätzliche Eigenschaft von ConTraST ist die automatische Makefile Erzeugung. Dabei wird ein Linux / Unix oder Windows Makefile erzeugt welches alle vom ConTraST erstellten Quellcode-Dateien beinhaltet. Standardmäßig werden die Quelldateien der zur Ausführung erforderlichen Laufzeitumgebung mit in das Makefile mit aufgenommen. Durch ausführen des `make` Befehls durch den Nutzer wird mit den im Makefile gespeicherten Informationen ein ausführbares Programm erzeugt. Der dazu verwendete Compiler ist unter Windows wie auch unter Linux/Unix `g++`.

ConTraST unterstützt zudem folgende Kommandozeilen Optionen:

- -c [Dateiname] Einlesen der unter [Dateiname] gespeicherten Konfigurationsdatei.
  In dieser Datei kann der vom Compiler zu unterstützende SDL-Sprachumfang festgelegt werden (siehe dazu insbesondere das folgende Kapitel 3.2.1). Erkennt der Compiler während des Parsens Sprachkonstrukte, die nicht durch die Konfiguration erlaubt sind, so wird eine Warnung ausgegeben. Nach Abschluss des Parsens kann der Nutzer entscheiden, ob trotzdem auch für die nicht unterstützten Konstrukte Code erzeugt werden soll, oder ob der Vorgang abgebrochen und der Compiler beendet werden soll.
- -i Parst die PR Datei Case-insensitive.
  Der Scanner wandelt dabei alle erkannten Token in Kleinbuchstaben um. Ist diese Option nicht gesetzt, sollte sichergestellt sein, dass die PR Datei durch Telelogic Tau Case-sensitive erzeugt wurde, da andernfalls Fehler bei Signal- /Daten- /Strukturobjekt-Namen durch unterschiedliche Schreibweisen auftreten können.
- -o [Dateiname] Schreibt alle fehlerfrei geparsten Zeilen der PR-Datei in [Dateiname].
  Diese Option ist vor allem beim schnellen Auffinden von Fehlern in der PR-Datei hilfreich. Alle Kommentare der PR Datei sind in [Dateiname] entfernt und der letzte Eintrag stellt das letzte fehlerfrei akzeptierte Token dar.
- -d [Dateiname] Schreibt Debuginformationen des Scanners und Parsers in [Dateiname].
  In [Dateiname] stehen alle durch den Scanner erkannten und den Parser weitergegebenen Token mit ihrem Namen / Wert. Zusätzlich wird der Aufbau des kompletten Stacks des Parsers abgelegt. Die Stackinformationen beinhalten die vom Parser in jedem Schritt durchgeführte Regel mit Angabe deren Nummer sowie den aktuellen Zustand des Parsers.
- -ds [Dateiname] Schreibt nur Debuginformationen des Scanners in [Dateiname].
- -dp [Dateiname] Schreibt nur Debuginformationen des Parsers in [Dateiname].

- -v | -vv | -vvv — Verbose Ausgabe Level 1-3.
  Gibt während des Parsen ausführlichere Informationen über die gerade ausgeführten Aktionen aus. Die Detailliertheit der Informationen steigt mit zunehmendem Level.
- -oproc — Optimiere Prozeduren soweit möglich.
  Durch diese Option werden die Prozeduraufrufe für zustandslose Prozeduren nicht durch die `CALL` Direktive ausgeführt sondern es wird an der Stelle des Prozeduraufrufs direkt eine Instanz der Prozedurtransition erzeugt und deren `fire()` Methode ausgeführt. Siehe dazu auch Kapitel 3.1.4
- -g[x] — Schreibt die –g[x] Option in den entsprechenden Stellen ins Makefile.
  Mit der –g[x] Option kann der g++ Compiler angewiesen werden debugbaren, Level [x] optimierten Code zu erzeugen.
- -p [Dateiname] — Schreibt den Parserbaum in [Dateiname].
  ConTraST traversiert den Parserbaum Top-Down und schreibt alle bekannten Informationen der einzelnen Container in textueller Form in [Dateiname]. Diese Option ist zusammen mit der –t Option vor allem zur Fehlersuche hilfreich.
- -t [Dateiname] — Schreibt den Parserbaum VOR der von ConTraST durchgeführten Baum-Transformation in [Dateiname].
  Zusammen mit –p können Transformationsfehler schnell aufgespürt werden.
- -n — Abschalten der Codeerzeugung.
  Es wird lediglich die PR-Datei gelesen, geparst und der erhaltene Parserbaum transformiert aber kein Code generiert.
- -include [Pfad] — Setzt den Pfad zum Verzeichnis mit den Dateien der Laufzeitumgebung.
- -dest [Pfad] — Setzt den Pfad zu einem Ausgabe-Verzeichnis.
  In das in [Pfad] abgegebene Verzeichnis werden ALLE von ConTraST erzeugten Dateien (Scanner/Parser Debugdateien, Parserbaum, generierte Quellcode-Dateien) geschrieben. Sollte das Verzeichnis nicht existieren, so wird es erzeugt.
- -linux — Erzeugt Code & Makefile die an Linux/Unix angepasst sind, wenn ConTraST unter einem Windows OS ausgeführt wird.

Die meisten Optionen dienen dem Erzeugen von Dateien die zum auffinden von Fehlern beim Parsen oder der Codeerzeugung des Compilers genutzt werden können. Dies ist vor allem bei der Erweiterung der Codeerzeugung für zurzeit nicht unterstützte Komponenten des SDL-Sprachumfangs hilfreich.

Weiterhin wurde eine direkte Einbindung des ConTraST Compilers in den Targeting Expert von Telelogic Tau ermöglicht, so dass für Windows direkt aus dem System ein ausführbares Programm erzeugt werden kann. Für Linux/Unix werden die erforderlichen Quellcode-Dateien sowie das Makefile für die Kompilation unter Linux/Unix erzeugt. Dadurch ist eine einfache und schnelle Anwendungserzeugung auch unter dem Betriebssystem Linux/Unix gewährleistet.

ConTraST selbst liegt ebenfalls in einer Windows und Linux/Unix Version vor und kann somit unter beiden Betriebssystemen eingesetzt werden.

3.2.1 Definition des unterstützten Sprachumfangs

Als weitere Besonderheit des ConTraST Compilers ist noch festzuhalten, dass es möglich ist, bestimmte Komponenten bereits beim Analysieren der Eingabe auszulassen bzw. eine Warnung oder Fehlermeldung anzuzeigen (siehe auch Anhang B Language Support) Der Nutzer kann dabei aus verschiedenen Subsets der Grammatik wählen oder gezielt eigene Subsets zusammenstellen. Die vordefinierten Subsets bestehen aus dem Kern (**C**ore), zwei statischen Erweiterungen und der dynamischen Erweiterung. Der Kern dient als minimale Grundlage aller folgenden Erweiterungen. Dazu zählen

- Alle strukturerzeugenden Komponenten außer `Prozeduren` und `Service` (`Package`, `System`, `Block`, `Prozess` sowie deren Typdefinitionen)
- Referenzen (diese werden bei der Transformation des Syntaxbaums entfernt)
- Instanzdefinitionen für eine einzige Instanz
- Kanäle, Signalrouten und Gates
- `Start-`, `Input` und `Output`-Transitionen
- Zustände und
- Signale

Diese Komponenten ermöglichen es bereits, sehr einfache Systeme zu spezifizieren. Sie enthalten keinerlei dynamische Komponenten, Variablen oder Parameter, die gespeichert werden müssten. Zusätzlich wurden zwei statische Erweiterungen (**S**tatic Extension **1** und **S**tatic Extension **2**) definiert. **S1** beinhaltet alle Core Komponenten und zusätzlich

- Instanzdefinitionen, bei denen die initiale und maximale Anzahl an Instanzen übereinstimmt
- `Timer` sowie `Set` und `Reset` Anweisungen
- `Decisions` (Verzweigungen)
- Anweisungen innerhalb von Tasks (`Statements`)
- `Save`
- Variablendefinitionen aller elementaren Datentypen außer `String`, `Array`, `Octet_string` und `Bag` / `Powerset` sowie Definitionen von `Newtype`, `Struct` / `Choice` und `Syntypes`

Die statische Erweiterung **S2** ergänzt **S1** durch

- `Services` und Service-Typdefinitionen
- Prioritätsinput, Continous Signal , Spontanous Transition und Enabling Condition
- `Synonym` Definitionen
- Vererbung (`Inheritance`)
- `Signalrefinement`

S1 und **S2** enthalten solche Komponenten, die zur Laufzeit keinen dynamischen Speicher benötigen werden. Dies ist wichtig im Hinblick auf die sehr begrenzten Ressourcen der eingebetteten Systeme, auf denen der Code ausgeführt werden soll. Der maximale Speicherbedarf kann somit bereits zur Compilezeit des Quellcodes und der Laufzeitumgebung ermittelt werden.

Natürlich ist es auch möglich, für die komplett unterstützte Palette an SDL Komponenten Code erzeugen zu lassen. Dies ermöglicht dann die dynamische Erweiterung (**D**ynamic Extension). Sie beinhaltet

- Prozedurdefinitionen und Prozeduraufrufe
- Instanzdefinitionen, bei denen die initiale Anzahl kleiner oder gleich der maximalen Anzahl an Instanzen ist
- `Create` und `Stop` Anweisungen für `Prozesse` / `Services`
- Variablendefinitionen von `String`, `Array`, `Octet_string` und `Bag` / `Powerset`
- Angabe von `Context-Parameter`

Über eine Konfigurationsdatei kann der Nutzer eigene Subsets mit den gewünschten SDL Komponenten als Erweiterung zum Kern zusammenstellen. Die Datei beinhaltet eine textuelle Beschreibung der Sprachkonstrukte:

[SDLCONFIG]

S1 0

INSTANCE 0
TIMER 0
SET 0
RESET 0
DECISION 0
TASK 0
SAVE 0
INTEGER 0
REAL 0
CHARSTRING 0
BOOLEAN 0
PID 0
NATURAL 0
CHARACTER 0
TIME 0
DURATION 0
NEWTYPE 0
STUCT/CHOICE 0
SYNTYPE 0

S2 1

SERVICE 0
PRIORITYINPUT 0
CONTINOUS 0
SPONTANEOUS 0
ENABELING 0
SYNONYM 0
INHERITANCE 0
SIGNAL_REFINEMENT 0

D 0

PROCEDURE 0
PROCEDURECALL 0
CREATE 0
STOP 0
STRING 0
ARRAY 1
OCTETSTRING 1
BAG/POWERSET 0
CONTEXT_PARAMETER 0

Durch eine `1` wird das Sprachkonstrukt unterstützt, eine `0` schaltet die Unterstützung ab. Die Aktivierung eines Subsets (**S1**, **S2** oder **D**) aktiviert alle zugehörigen Komponenten und Subsets automatisch. Sollen nicht alle Komponenten eines Subsets aktiviert werden, so deaktiviert man das Set und schaltet die erforderlichen Komponenten einzeln an. Im obigen Beispiel ist die statische Erweiterung **S2** (und automatisch auch **S1**) und zusätzlich noch die Unterstützung für `Array` und `Octet_strings` aktiviert.

4 Umsetzung von SDL in C++ Code

Das folgende Kapitel vermittelt einen Überblick über die schrittweise Umsetzung von SDL Code von der graphischen Erstellung in Telelogic Tau über die erstellte(n) PR Datei(en), bis hin zum generierten Code. Als Grundlage dieses Überblicks dient dabei ein Vorentwurf eines adaptiven Protokolls. Anschießend soll noch ein kurzer Vergleich zwischen dem von ConTraST erzeugten Quellcode mit dem von Cmicro, dem Telelogic Tau beiliegenden SDL zu C Compiler erfolgen.

4.1 Struktur

Als Grundlage der graphischen Repräsentation wird das ADA Protokoll verwendet. Dabei handelt es sich um den Vorentwurf eines adaptiven Protokolls welches die Anzahl der Netzwerkknoten, den ungefähren Radius des Netzes auf Faktor 2 genau (d.h. den Radius bis maximal den Durchmesser in Hops) sowie künftig auch den Mobilitätsgrad des Netzes erfassen soll. Obwohl es sich hierbei um ein relativ kleines und einfaches Protokoll handelt, sollen hier nur Auszüge behandelt werden. Es wird jeweils ein Teil der Struktur-, Daten- und Verhaltendefinition betrachtet werden.

4.1.1 Graphische SDL Repräsentation in Telelogic Tau

Als Beispiel der Strukturdefinition soll das `Package` InfoCollector (Abbildung 4.1.1-1) betrachtet werden. Es besteht aus den 3 (zustandslosen) `Prozeduren` GetRandID, Time2Real und Real2Time, sowie einer `Block-Typ-Definition` InfoColl.

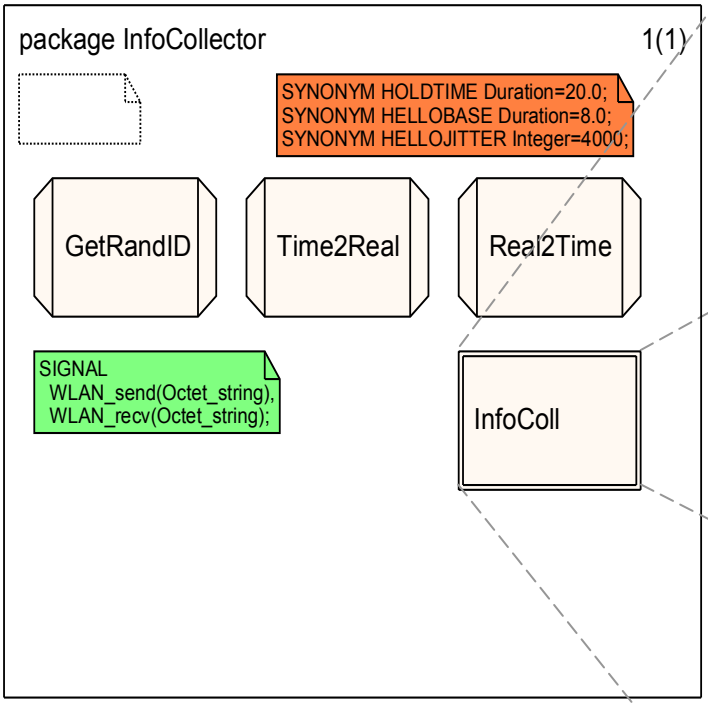


Abbildung 4.1.1-1

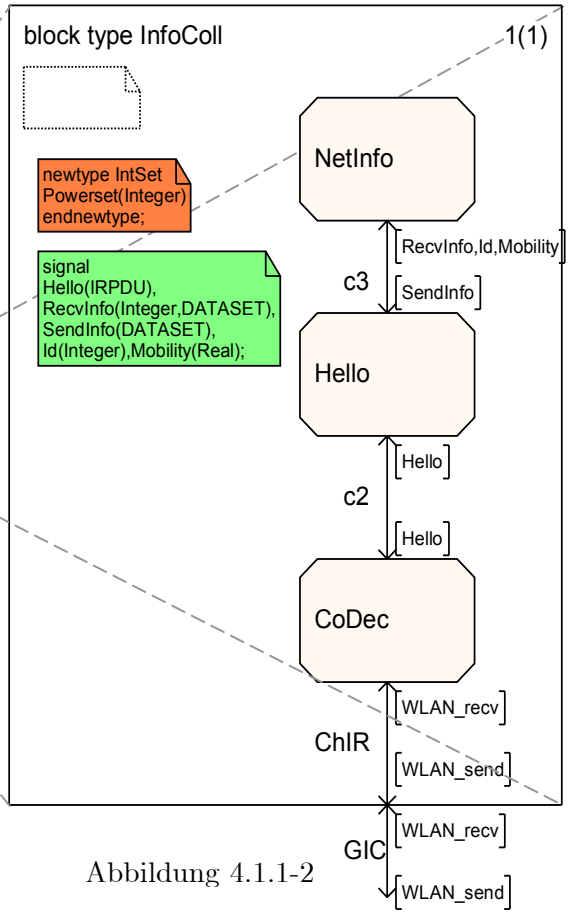


Abbildung 4.1.1-2

Die Substruktur des Block–Typs ist in Abbildung 4.1.1-2 abgebildet. Sie besteht aus den `Prozessen` NetInfo und Hello, die über die `Signalroute` c3 `Signale` austauschen, sowie dem `Prozess` CoDec, der über die `Signalrouten` c2 mit Hello und ChIR mit dem `Gate` GIC verbunden ist. Die orange markierten Synonym und newtype Deklarationen sind Datendefinitionen, die grünen markierten Abschnitte Signaldefinitionen. Auf die Signaldefinitionen wird im Folgenden noch genauer eingegangen.

4.1.2 Die generierte PR Datei

Wie bereits bei der Vorstellung von Telelogic Tau in Kapitel 2.3.1 angedeutet, ist das Tool in der Lage, direkt aus der graphischen Repräsentation des vorhergehenden Abschnitts eine textuelle PR Datei zu erzeugen.

Die hier dargestellten Abschnitte für das `Package` InfoCollector und den Block-Typ InfoColl wurden zur besseren Lesbarkeit bearbeitet. Dabei wurden in erster Linie die zahlreichen von Tau eingefügten Kommentare entfernt und die Ausgabe strukturiert sowie zusammengehörende Zeilen zusammengefasst. Die Reihenfolge der einzelnen Ausgaben blieb dabei allerdings unberührt.

Ausschnitt `Package` InfoCollector:

```
package InfoCollector;
SIGNAL
     WLAN_send(Octet_string),
     WLAN_recv(Octet_string);
SYNONYM HOLDTIME Duration=20.0;
SYNONYM HELLOBASE Duration=8.0;
SYNONYM HELLOJITTER Integer=4000;
block type InfoColl referenced;
procedure Time2Real referenced;
procedure Real2Time referenced;
procedure GetRandID referenced;
endpackage InfoCollector;
```

Ausschnitt `Block-Typ` InfoColl

```
block type InfoColl;
gate GIC
     out with WLAN_send;
     in with WLAN_recv;
signal
     Hello(IRPDU),
     RecvInfo(Integer,DATASET),
     SendInfo(DATASET),
     Id(Integer),Mobility(Real);
newtype IntSet Powerset(Integer)
endnewtype;
signalroute c3
     from NetInfo
     to Hello
     with SendInfo;
     from Hello
     to NetInfo
     with RecvInfo, Id, Mobility;
signalroute c2
     from Hello
     to CoDec
     with Hello;
     from CoDec
     to Hello
     with Hello;
signalroute ChIR
     from CoDec
     to env
     via GIC
     with WLAN_send;
     from env
     via GIC
     to CoDec
     with WLAN_recv;
process NetInfo referenced;
process Hello referenced;
process CoDec referenced;
endblock type InfoColl;
```

Man erkennt, dass im `Package` InfoColl als auch im Block-Typ InfoColl die enthaltenen Substrukturen nicht direkt abgebildet werden, sondern lediglich Referenzen auf sie verweisen. Bei der folgenden Codegenerierung müssen diese Referenzen aufgelöst werden. Dies geschieht bei der Transformation der referenzierten Strukturen.

4.1.3 Analyse der PR Datei und Transformation des Syntaxbaums

Da in der PR Datei die Substrukturen (`Block-Typ` InfoColl, `Procedure` Time2Real, `Real2Time`, GetRandID) als eigenständige Strukturen abgebildet und nur referenziert sind, werden sie beim Parsen der Datei direkt unter der Wurzel angeordnet. Ansonsten finden sich alle Informationen der PR Datei auch im Syntaxbaum wieder:
Syntaxbaum direkt nach dem Parsen, aber noch vor der Transformation:

```
##############Root_System SDL#########################

Root_System SDL consists of 21 containers:…
```

Lediglich die Elemente der in Kapitel 4.1.1 betrachteten Strukturen sind hier aufgelistet.

```
Subcontainer 3:Package InfoCollector …
Subcontainer 7:Block_Type InfoColl …
Subcontainer 9:Process CoDec
Subcontainer 10:Process Hello
Subcontainer 11:Process NetInfo …
Subcontainer 19:Procedure Time2Real
Subcontainer 20:Procedure Real2Time
Subcontainer 21:Procedure GetRandID

##############Package InfoCollector###################
```

Die Beschreibungen von Signal- und Datendefinitionen finden sich 1:1 in der textuellen Beschreibung wieder. Nur Referenzen werden in einem eigenen Abschnitt zusammengefasst.

```
Package InfoCollector consists of 4 containers:
Subcontainer 1:Signal_Definition
Subcontainer 2:Synonym
Subcontainer 3:Synonym
Subcontainer 4:Synonym

Package InfoCollector has 4 references:
Block_Type InfoColl
Procedure Time2Real
Procedure Real2Time
Procedure GetRandID

Signal_Definition:
```

In der Signal_Definition werden die Signalparameter als eigenständige Subcontainer der jeweiligen Signale verwaltet. Auf die Auflistung dieser Subcontainerstruktur wird hier aus Gründen der Übersichtlichkeit verzichtet.

```
Signal WLAN_send … Sort Octet_string
Signal WLAN_recv … Sort Octet_string

SYNONYM HOLDTIME Duration 20.0
SYNONYM HELLOBASE Duration 8.0
SYNONYM HELLOJITTER Integer 4000

##########End Package InfoCollector##################
…
```

```
###############Block_Type InfoColl#######################
```

Auch bei der `Block-Typ-Definition` ist die 1:1 Umsetzung der PR-Datei gut zu erkennen. Signaldefinitionen werden in einem eigenen Subcontainer zusammengefasst, Signalrouten und Gates hingegen einzeln als direkte Söhne des `Block-Typ` Knotens angelegt.

```
Block_Type InfoColl consists of 6 containers:
Subcontainer 1:Signal_Definition
Subcontainer 2:Signalroute c3
Subcontainer 3:Signalroute c2
Subcontainer 4:Signalroute ChIR
Subcontainer 5:Type_Definition IntSet
Subcontainer 6:Gate GIC

Block_Type InfoColl has 3 references:
Process NetInfo
Process Hello
Process CoDec

Signal_Definition
Signal Hello … Sort IRPDU
Signal RecvInfo … Sort Integer … Sort DATASET
Signal SendInfo … Sort DATASET
Signal Id … Sort Integer
Signal Mobility … Sort Real

Signalroute c3
 FROM NetInfo TO Hello WITH Signal SendInfo
 FROM Hello TO NetInfo WITH Signal RecvInfo,
                            Signal Id,
                            Signal Mobility

Signalroute c2
 FROM Hello TO CoDec WITH Signal Hello
 FROM CoDec TO Hello WITH Signal Hello

Signalroute ChIR
 FROM CoDec TO ENV VIA GIC WITH Signal WLAN_send
 FROM ENV VIA GIC TO CoDec WITH Signal WLAN_recv
```

Auch bei den Datendefinitionen wird wie bei Signaldefinitionen die Parameter in Subcontainern verwaltet.

```
Type_Definition NEWTYPE IntSet … Powerset … Integer

Gate GIC
 OUT WITH Signal WLAN_send
 IN WITH Signal WLAN_recv

###########End Block_Type InfoColl#######################
…
###########End Root_System SDL#############################
```

Nach der Transformation hat sich der Baum verändert (Rot markiert):

Die Wurzel enthält nun nur noch das `Package`. Die `Block-Typ-Definition` und die `Prozedurdefinitionen` sind in das `Package` verschoben worden, die Prozessdefinitionen sind Bestandteil der `Block-Typ-Definition`.

```
##############Root_System SDL#########################

Root_System SDL consists of 6 containers:…
Subcontainer 3:Package InfoCollector…

##############Package InfoCollector###################

Package InfoCollector consists of 8 containers:
Subcontainer 1:Block_Type InfoColl
Subcontainer 2:Procedure Time2Real
Subcontainer 3:Procedure Real2Time
Subcontainer 4:Procedure GetRandID
Subcontainer 5:Signal_Definition
Subcontainer 6:Synonym
Subcontainer 7:Synonym
Subcontainer 8:Synonym

##############Block_Type InfoColl#####################
```

Die `Block-Typ-Definition` beinhaltet jetzt die referenzierten `Prozessdefinitionen`. Diese wurden wiederum in jeweils eine Instanzdefinition (`Textual_Process_Type`) und eine `Prozess-Typ-Definition` transformiert

```
Block_Type InfoColl consists of 12 containers:
Subcontainer 1:Signal_Definition
Subcontainer 2:Textual_Process_Type NetInfo
Subcontainer 3:Process_Type NetInfo_Processtype
Subcontainer 4:Textual_Process_Type Hello
Subcontainer 5:Process_Type Hello_Processtype
Subcontainer 6:Textual_Process_Type CoDec
Subcontainer 7:Process_Type CoDec_Processtype
```

Aus den vorherigen Signalrouten sind nach der Transformation Channels geworden.

```
Subcontainer 8:Channel c3
Subcontainer 9:Channel c2
Subcontainer 10:Channel ChIR
Subcontainer 11:Type_Definition IntSet
Subcontainer 12:Gate GIC
```

Die Signaldefinitionen, das Gate sowie die Type_Definition werden an dieser Stelle weggelassen, da sich hier keine Veränderungen ergeben.

```
##############Textual_Process_Type NetInfo############

Textual_Process_Type NetInfo … NetInfo_Processtype

##########End Textual_Process_Type NetInfo############
##############Process_Type NetInfo_Processtype########
```

Das hier folgende Verhalten wird hier nicht weiter betrachtet.

```
##########End Process_Type NetInfo_Processtype########
##############Textual_Process_Type Hello##############

Textual_Process_Type Hello … Hello_Processtype

##########End Textual_Process_Type Hello##############
```

```
###############Process_Type Hello_Processtype###########
```

[Verhalten]

```
###########End Process_Type Hello_Processtype###########
###############Textual_Process_Type CoDec###############

Textual_Process_Type CoDec … CoDec_Processtype

###########End Textual_Process_Type CoDec###############
###############Process_Type CoDec_Processtype###########
```

[Verhalten]

```
###########End Process_Type CoDec_Processtype###########
```

Durch die Transformation der `Prozesse` in `Prozess-Typen` werden neue Gates eingeführt. Diese müssen in die Kanaldefinitionen mit aufgenommen werden.

```
Channel c3
 FROM NetInfo VIA NetInfo_Gate TO Hello VIA Hello_Gate
 WITH Signal SendInfo
 FROM Hello VIA Hello_Gate TO NetInfo VIA NetInfo_Gate
 WITH Signal RecvInfo, Signal Id, Signal Mobility

Channel c2
 FROM Hello VIA Hello_Gate TO CoDec VIA CoDec_Gate
 WITH Signal Hello
 FROM CoDec VIA CoDec_Gate TO Hello VIA Hello_Gate
 WITH Signal Hello

Channel ChIR
 FROM CoDec VIA CoDec_Gate TO ENV VIA GIC
 WITH Signal WLAN_send
 FROM ENV VIA GIC TO CoDec VIA CoDec_Gate
 WITH Signal WLAN_recv

###########End Block_Type InfoColl######################

###############Procedure Time2Real######################
```

[Verhalten]

```
###########End Procedure Time2Real######################
###############Procedure Real2Time######################
```

[Verhalten]

```
###########End Procedure Real2Time######################
###############Procedure GetRandID######################
```

[Verhalten]

```
###########End Procedure GetRandID######################
…
```

Die Signal- und Synonymdefinitionen wurden an dieser Stelle weggelassen, da sich hier keine Veränderungen ergeben

```
###########End Package InfoCollector###################

###########End Root_System SDL#########################
```

Auffällig nach der Transformation ist, dass die Referenzen der Prozesse alle durch die entsprechenden Instanzdeklarationen (`Textual_Process_Type`) ersetzt wurden. Gleichzeitig sind die zugehörigen Prozess-Typ-Definitionen in den Block-Typ verschoben worden. Auch die Prozedurreferenzen werden durch die entsprechenden Prozeduren ersetzt. Weiterhin ist zu beachten, dass bei der Umwandlung der Signalrouten in Kanäle jeweils die erforderlichen Gates der Prozess-Typen hinzugefügt wurden.

4.1.4 Erzeugter Code

Aus der Konvertierten PR Datei wird anschließend folgender C++ Code erzeugt:

Die erzeugte Header Datei:

```
/* Generated by ConTraST Version 1.0 */
#ifndef Package_InfoCollector_h
#define Package_InfoCollector_h

// Includes for SDL Runtime Environment
//-----------------------------------------------------------------
#include "/home/c_weber/Diplomarbeit/Compiler/…/sdl_asm.h"
#include "/home/c_weber/Diplomarbeit/Compiler/…/sdl_datatypes.h"
#include "/home/c_weber/Diplomarbeit/Compiler/…/includes/asm_typedefs.h"
//-----------------------------------------------------------------

#include "Package_DataStructures.h"
#include "Package_SenfConfig.h"
#include "Package_SenfLog.h"
```

Synonymdefinitionen

```
/* Synonym Definitions */
extern const Duration HOLDTIME;
extern const Duration HELLOBASE;
extern const Integer HELLOJITTER;

/* Package InfoCollector */
class InfoCollector {
     private:
     public:
```

Definition der in diesem Package verwendeten Signale

```
        /* Signal Definition of Signal WLAN_send */
        class WLAN_send : public PlainSignalInst {
          rtti(WLAN_send)
```

`#define` wird bei Anbindung eines Environment Frameworks zur Kommunikation über ein reales Medium benötigt. Jedes Signal besitzt mehrere Konstruktoren um das Versenden in SDL mit `` `TO` `` bzw. `` `VIA` `` zu ermöglichen.

```
          #define WLAN_send WLAN_send
          public:
            Octet_string Param1;
            WLAN_send(Octet_string P1){Param1 = P1;}
            WLAN_send(Octet_string P1, ToArg to){Param1 = P1; toArg = to;}
            WLAN_send(Octet_string P1, ViaArg via){Param1 = P1; viaArg = via;}
            WLAN_send(Octet_string P1, ToArg to, ViaArg via)
              {Param1 = P1; toArg = to; viaArg = via;}
        };
```

```
        /* Signal Definition of Signal WLAN_recv */
        class WLAN_recv : public PlainSignalInst {
          …
```

Analog zur Signaldefinition von WLAN_send

```
        };
};

/* Procedure Time2Real */

class Time2Real : public SDLProcedure {
```

Das hier folgende Verhalten wird hier nicht weiter betrachtet.

```
};

/* Procedure Real2Time */

class Real2Time : public SDLProcedure {
```

[Verhalten]

```
};

/* Procedure GetRandID */

class GetRandID : public SDLProcedure {
```

[Verhalten]

```
};
/* BlockType InfoColl */

class InfoColl : public SDLBlockType {
     private:
     public:
```

Variablentyp-Deklaration; Der Generator für Powerset ist in der Laufzeitumgebung implementiert und kann hier direkt ohne weitere Definition verwendet werden

```
        /* New type Generator */
        typedef Powerset< Integer> IntSet;
```

Definition der im Block-Typ verwendeten lokalen Signale. Alle Signaldefinitionen sind analog zu WLAN_send

```
        /* Signal Definition of Signal Hello */
        class Hello : public PlainSignalInst {
          …
        };

        /* Signal Definition of Signal RecvInfo */
        class RecvInfo : public PlainSignalInst {
          …
        };

        /* Signal Definition of Signal SendInfo */
        class SendInfo : public PlainSignalInst {
          …
        };

        /* Signal Definition of Signal Id */
        class Id : public PlainSignalInst {
          …
        };
```

```
/* Signal Definition of Signal Mobility */
class Mobility : public PlainSignalInst {
  …
};
```

Gates werden in eigenen Klassen innerhalb der Strukturen angelegt und halten alle relevanten Informationen über die ein- und ausgehenden Signale.

```
class gates : public SDLGates {
  public:
    gates(SdlAgentSet* ow){
    /* Gate GIC */
```

Angabe des benötigten Gates `GIC` mit Namen und ein- und ausgehenden Signalen

```
      CREATEGATE("GIC",ow,::InfoCollector::WLAN_recv::ID(),
        ::InfoCollector::WLAN_send::ID());
    }
};
```

`CREATEALLCHANNELS` und `CREATESUBSTRUCTURE` wird in der zugehörigen C++ Datei implementiert

```
virtual void CREATEALLCHANNELS(SdlAgent* ow);
virtual void CREATESUBSTRUCTURE(class SdlAgent* ow);
InfoColl(SdlAgentSet* ow, class SdlAgent *pa)
  : SDLBlockType("InfoColl",ow,0) {}
};

/* ProcessType NetInfo_Processtype */

class NetInfo_Processtype : public SDLProcessType {

[Verhalten]

};

/* ProcessType Hello_Processtype */

class Hello_Processtype : public SDLProcessType {

[Verhalten]

};

/* ProcessType CoDec_Processtype */

class CoDec_Processtype : public SDLProcessType {

[Verhalten]

};

#endif
```

In der Header Datei wird die beinahe 1:1 Abbildung zwischen der umgeformten PR Datei und dem generierten Code deutlich. Die Klassen der Prozess-Typ-Definitionen und Prozedurdefinitionen sind allerdings nicht in die Klasse der Block-Typ-Definition eingelagert. Eine entsprechende Einlagerung hätte speziell bei größeren Systemen eine Verschlechterung der Übersichtlichkeit und der Lesbarkeit zur Folge.

Die zugehörige C++ Datei hat folgende Struktur:

```
/* Generated by ConTraST Version 1.0 */
#include "Package_InfoCollector.h"
```

Wertzuweisung zu den definierten Konstanten der Header Datei

```
const Duration HOLDTIME( (Real)20 );
const Duration HELLOBASE( (Real)8 );
const Integer HELLOJITTER( (Integer)4000 );

/* Package InfoCollector */

/* Procedure Time2Real */
```

[Verhalten]

```
/* Procedure Real2Time */
```

[Verhalten]

```
/* Procedure GetRandID */
```

[Verhalten]

```
/* BlockType InfoColl */
```

Erstellung der benötigten Kanäle durch Angabe der Pfade mit jeweils IN und OUT Gate sowie einer Signalliste

```
void InfoColl :: CREATEALLCHANNELS(SdlAgent* ow) {

     CREATECHANNEL("c3",ow,
       Channel_path("NetInfo_Gate","Hello_Gate",::InfoColl::SendInfo::ID())&
       Channel_path("Hello_Gate","NetInfo_Gate",::InfoColl::RecvInfo::ID()&
         ::InfoColl::Id::ID()&::InfoColl::Mobility::ID()));

     CREATECHANNEL("c2",ow,
       Channel_path("Hello_Gate","CoDec_Gate",::InfoColl::Hello::ID())&
       Channel_path("CoDec_Gate","Hello_Gate",::InfoColl::Hello::ID()));

     CREATECHANNEL("ChIR",ow,
       Channel_path("CoDec_Gate","GIC",::InfoCollector::WLAN_send::ID())&
       Channel_path("GIC","CoDec_Gate",::InfoCollector::WLAN_recv::ID()));
}
```

Erzeugung der enthaltenen Substrukturinstanzen mit Angabe des Namens und der initialen und maximalen Anzahl an Instanziierungen

```
void InfoColl :: CREATESUBSTRUCTURE(class SdlAgent* ow) {

  new SDLProcess<NetInfo_Processtype>("NetInfo",(SdlAgent*)this,1,undefined);
  new SDLProcess<Hello_Processtype>("Hello",(SdlAgent*)this,1,undefined);
  new SDLProcess<CoDec_Processtype>("CoDec",(SdlAgent*)this,1,undefined);
}

/* ProcessType NetInfo_Processtype */
```

[Verhalten]

```
/* ProcessType Hello_Processtype */
```

[Verhalten]

```
/* ProcessType CoDec_Processtype */
```

[Verhalten]

Auch der Code für die Kanal- und Substrukturerzeugung (`CREATEALLCHANNELS` und `CREATESUBSTRUCTURE`) ist einfach lesbar. Jeder Kanal gibt seine vorhandenen Pfade (`Channel_path`) an. Diese bestehen wiederum aus dem eingehenden und ausgehenden Gate (Gatenamen werden als Strings in `" "` angegeben) sowie den zugehörigen Signal-IDs mit den entsprechenden Scope-Informationen. Bei der Erzeugung der Substrukturen werden jeweils neue Instanzen eines Prozess-Typs erstellt (`new`) mit dem ursprünglichen Namen sowie der initialen und maximal möglichen Anzahl. Die Menge der zu generierenden Instanzen kann bereits in der graphischen Darstellung auch für Prozesse angegeben werden. Ist keine Anzahl spezifiziert wird bei der Transformation immer initial `1` und eine unbestimmte (`undefined`) maximale Zahl an Initialisierungen angenommen.

4.2 Daten

4.2.1 Repräsentation in Telelogic Tau

Um den Verlauf der Codeerzeugung für Datenstrukturen zu demonstrieren dient eine Datenspezifikation von Ada I in *ASN 1*.

```
DataStructures
DEFINITIONS AUTOMATIC TAGS::= BEGIN

DATA ::= SEQUENCE
{
     origin INTEGER,
     origtime REAL,
     phase INTEGER,
     lasthop INTEGER,
     lastused REAL,
     nHops INTEGER,
     descts INTEGER,
     mmobility REAL
}

DATASET::= SET OF DATA

IRPDU ::= SEQUENCE
{
     sendtime REAL,
     thishop INTEGER,
     info DATASET
}

END
```

4.2.2 Die generierte PR Datei

Mit Hilfe des in Telelogic Tau integrierten *ASN 1* Encoders wird die Datenspezifikation in SDL Datenstrukturen umgewandelt:

```
package DataStructures;

newtype DATA struct
  origin  Integer;
  origtime  Real;
  phase  Integer;
  lasthop  Integer;
  lastused  Real;
  nHops  Integer;
  descts  Integer;
  mmobility  Real;
endnewtype;

newtype DATASET
   Bag(DATA)
endnewtype;

newtype IRPDU struct
  sendtime  Real;
  thishop  Integer;
  info  DATASET;
endnewtype;

endpackage DataStructures;
```

Auch hier wurden wieder die automatisch generierten Kommentare von Tau entfernt. Wie man sieht werden die *ASN 1* Strukturen in ein eigenes SDL Package verpackt, das dann später bei Bedarf in das System oder in andere Packages eingebunden werden kann.

4.2.3 Erzeugter Code

Die Informationen werden in eine Header Datei geschrieben. Diese Datei wird analog zu SDL an den entsprechenden Stellen im Code eingebunden.

```
/* Generated by ConTraST Version 1.0 */
#ifndef Package_DataStructures_h
#define Package_DataStructures_h

// Includes for SDL Runtime Environment
//-----------------------------------------------------------------------
#include "/home/c_weber/Diplomarbeit/Compiler/…/sdl_asm.h"
#include "/home/c_weber/Diplomarbeit/Compiler/…/sdl_datatypes.h"
#include "/home/c_weber/Diplomarbeit/Compiler/…/asm_typedefs.h"
//-----------------------------------------------------------------------

/* Package DataStructures */
```

Die Datenstrukturen werden 1:1 in C++ Code umgewandelt. Zusätzlich werden alle notwendigen Zugriffs- und Vergleichsoperatoren auf den Strukturen definiert. Außerdem sind Funktionen zur Kodierung der Datenstruktur in `Octet_Strings` implementiert.

```
class DataStructures {
     private:
     public:
       /*Structure Definition*/
       class DATA {
         protected:
           struct {
             Integer origin;
             Real origtime;
             Integer phase;
             Integer lasthop;
             Real lastused;
             Integer nHops;
             Integer descts;
             Real mmobility;
           } data;
```

Definition von Zugriffsoperatoren für die einzelnen Komponenten der Struktur

```
         public:
           Integer& origin(void) {return data.origin; }
           Real& origtime(void) {return data.origtime; }
           Integer& phase(void) {return data.phase; }
           Integer& lasthop(void) {return data.lasthop; }
           Real& lastused(void) {return data.lastused; }
           Integer& nHops(void) {return data.nHops; }
           Integer& descts(void) {return data.descts; }
           Real& mmobility(void) {return data.mmobility; }
```

Definition benötigter Vergleichsopertoren

```
           bool operator==(const DATA &Param) const {
             return ((data.origin == Param.data.origin)||
               (data.origtime == Param.data.origtime)||
               (data.phase == Param.data.phase)||
               (data.lasthop == Param.data.lasthop)||
               (data.lastused == Param.data.lastused)||
               (data.nHops == Param.data.nHops)||
               (data.descts == Param.data.descts)||
               (data.mmobility == Param.data.mmobility)}
```

```
    bool operator!=(const DATA &Param) const {
      return ((data.origin != Param.data.origin)&& …); }
    bool operator<(const DATA &Param) const {
      return ((data.origin < Param.data.origin)&& …); }
    bool operator>(const DATA &Param) const {
      return ((data.origin > Param.data.origin)&& …); }
```

Methode zur direkten Zuweisung von Werten in die Struktur

```
    void assign(Integer P1, Real P2, Integer P3, Integer P4,
                Real P5, Integer P6, Integer P7, Real P8){
      data.origin = P1;
      data.origtime = P2;
      … }
```

Funktionen zur En- und Dekodierung des Datentypen in `Octet_strings` Es fehlen noch `Header` und `Tail` da deren Format noch nicht endgültig feststeht

```
    int encode(Octet_string &os) {
      // ADD ENCODE HEADER HERE
      if (data.origin.encode(os) <= 0)
        return -1;
      if (data.origtime.encode(os) <= 0)
        return -1; …
      // ADD ENCODE TAIL HERE
      return 0;
    }
    int decode(Octet_string &os) {
      // ADD DECODE HEADER HERE
      if (data.origin.decode(os) <= 0)
        return -1; …
      // ADD ENCODE TAIL HERE
      return 0;
    }
};
```

Der Generator für Bag ist in der Laufzeitumgebung implementiert und kann hier direkt ohne weitere Definition verwendet werden

```
/* New type Generator */
typedef Bag< ::DataStructures::DATA> DATASET;

/*Structure Definition*/
class IRPDU {
  protected:
    struct {
      Real sendtime;
      Integer thishop;
      ::DataStructures::DATASET info;
    } data;
  public: …
```

Analog zu class DATA

```
    };
};

#endif
```

Zusätzlich zu den generierten Strukturen werden jeweils noch benötigte Zugriffsmethoden für die einzelnen Variablen, einige Standardoperatoren sowie `encode` und `decode` Funktionen implementiert. Auch hier steht die leichte Nachvollziehbarkeit der Abbildung von SDL (*ASN 1*) nach C++ im Vordergrund. Zu beachten ist, dass auch hier wieder bei allen nicht elementaren Datentypen bzw. Generatoren Scope-Informationen hinzugefügt werden müssen.

Alle Generatoren mit ihren zugehörigen Operatoren und Funktionen der in Telelogic Tau vordefinierten Datentypen [10] sind durch die Laufzeitumgebung definiert und bedürfen daher keiner expliziten Umsetzung von Seiten des Compilers.

Die zugehörige C++ Datei beinhaltet lediglich den Verweis auf den Header in einer entsprechenden `"#include"` Anweisung.

Bei lokalen Variablendefinitionen innerhalb von Strukturen werden die gleichen Mechanismen zur Codeerzeugung verwendet. Die erzeugten Klassen sind dann zusätzlicher Bestandteil der Strukturklassen genau wie z.B. Signale oder Gates.

4.3 Verhalten

4.3.1 Graphische Repräsentation in Telelogic Tau

Um die Umsetzung des Verhaltens zu beschreiben, dient ein Ausschnitt aus dem `Prozess` Net Info. Das Verhalten des Prozesses wird bei der Transformation nicht verändert und ist somit mit dem Verhalten des erstellten Prozess-Typs identisch. Zu Beachten ist, dass dieser Ausschnitt nachbearbeitet wurde. So fehlen aus Platzgründen z.B. zwei Entscheidungsbäume bei `Decisions` (Rot) und das Verhalten welches sich an den Konnektoren `processP1`, `processP2` und `processP3` anschließt. Auch die lokale Variablendefinition des Prozesses wurde entfernt. Sie spielt bei der Umsetzung des Verhaltens von SDL nach C++ eine untergeordnete Rolle und entspricht dem in Abschnitt 4.2.3 vorgestellten Schema.

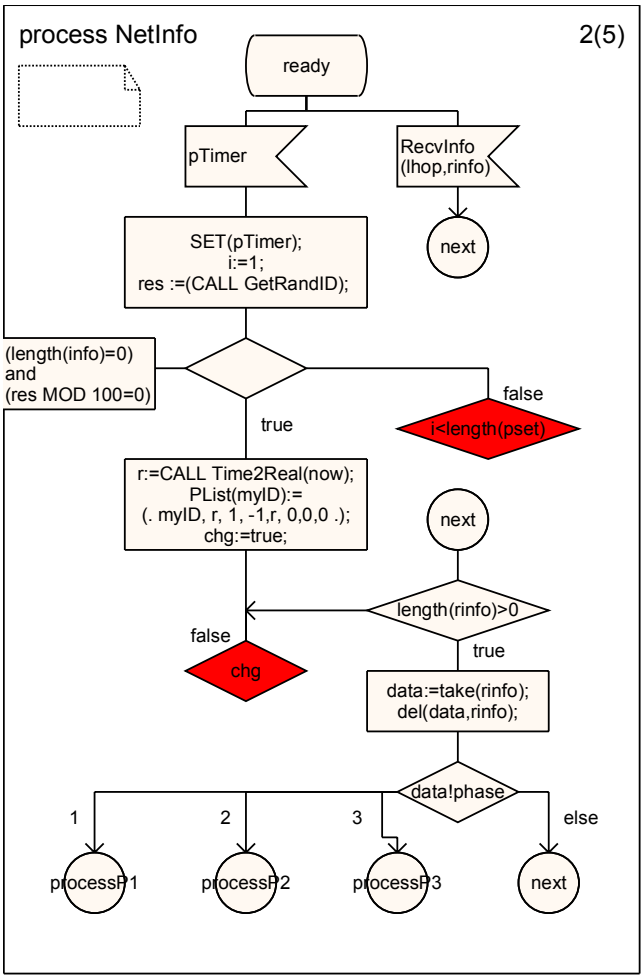


Abbildung 4.3.1-1

4.3.2 Die generierte PR-Datei

Aus der graphischen Repräsentation von Abbildung 4.3.1-1 ergibt sich die nachfolgende PR Datei. Natürlich geht bereits bei der Umwandlung der graphischen Beschreibung in die textuelle Form einiges an Übersichtlichkeit verloren. Daher wurde die PR Datei etwas formatiert und die zusammengehörenden Abschnitte eingerückt. Rot markiert sind die `Decisions`, deren Entscheidungsbäume weggelassen wurden.

```
process NetInfo;
…
state ready;
input RecvInfo(lhop, rinfo);
```

Explizite Deklaration der Sprungmarke durch den Konnektor `next`

```
join next;
input pTimer;
task {
   SET(pTimer);
   i:=1;
   res :=(CALL GetRandID);
   };
decision (length(info)=0) and (res MOD 100=0);
   (false):
```

Die mit `grst` markierten Sprungmarken wurden automatisch von Tau hinzugefügt.

```
     grst8:
     decision i<length(pset);…
        grst9:
        decision chg;…
   (true):
     task {
        r:=CALL Time2Real(now);
        PList(myID):=(. myID, r, 1, -1, r, 0,0,0 .);
        chg:=true;
     };
     join grst9;
enddecision;
endstate;

connection next:
decision length(rinfo)>0;
   (true):
     task {
        data:=take(rinfo);
        del(data, rinfo);
     };
     decision data!phase;
```

Deklaration der Marken durch die Konnektoren `processP1`, `processP2` und `processP3`

```
        (1): join processP1;
        (2): join processP2;
        (3): join processP3;
        else: join next;
     enddecision;
   (false):
     enddecision;
join grst9;
endconnection next;
…
endprocess NetInfo;
```

4.3.3 Erzeugter Code

Aus der generierten PR Datei wird folgende Header Datei erzeugt:

```
/* Generated by ConTraST Version 1.0 */
#ifndef Package_InfoCollector_h
#define Package_InfoCollector_h
…
/* ProcessType NetInfo_Processtype */
class NetInfo_Processtype : public SDLProcessType {
     private:
     public:
```

Die Variablen-, Daten- und Timerdefinition sind nicht in Abbildung 4.3.1-1 enthalten.

```
        /* New type Generator */
        typedef Array< Integer,::DataStructures::DATA> PhaseList;
        typedef Powerset< Integer> NodeSet;
        typedef Array< Integer,::NetInfo_Processtype::NodeSet> NodeSetList;

        /* Variables: */
        Integer i;
        Integer myID;
        ::DataStructures::DATASET info;
        Real mobility;
        Integer lhop;
        …
```

Definition eines Timers mit entsprechender Initialisierung von 10 Zeiteinheiten.

```
        /* Timer Definition of Timer pTimer */
        class pTimer : public TimerInst {
           rtti(pTimer)
           public:
              pTimer(){
                  duration = 10;
              }
        };

        class gates : public SDLGates {
           …
        };
        virtual void CREATEALLCHANNELS(SdlAgent* ow);
        virtual void CREATESUBSTRUCTURE(class SdlAgent* ow);
```

Transitionen werden als eigenständige Klassen definiert. Sie definieren die erforderliche `fire()` Methode, deren Implementierung zentrales Elemente der C++ Datei ist.

```
        /* Transition T16 */
        class T16 : public Transition {
           rtti(T16)
           private:
           public:
              T16(SDLProcessType* owner) : Transition(owner)
                 {}
              void fire(SDLInstance* owner, SignalInst* signal);
        };
```

Alle folgenden Transitionsklassen sind analog zu T16.

```
        /* Transition T17 */
        /* Transition T18 */
        /* Transition T19 */
        /* Transition T20 */
```

```
    /* Transition next */
    /* Transition processP1 */
    /* Transition processP2 */
    /* Transition processP3 */
```

Die Definition aller im Prozess enthaltenen Zustände erfolgt ebenfalls in Form von separaten Klassen. Die Implementierung der Konstruktoren findet ebenfalls in der C++ Datei statt.

```
    /* State awID */
    class awID : public SDLState {
      rtti(awID)
      private:
      public:
        awID(SDLProcessType* owner);
    };
    /* State ready */
```

Analog Klasse awID

Der Konstruktor des Prozesses enthält die Initialisierung von Variablen und das Hinzufügen der Starttransition, Zuständen und evtl. Konnektor-Transitionen

```
    NetInfo_Processtype(SdlAgentSet* ow, class SdlAgent *pa):
      SDLProcessType("NetInfo_Processtype",ow,0) {

      /* Initialize Variables */
      chg = false;
      /* Add Start Transition */
      addStartTransition(new T16(this));
      /*Add States*/
      addState(new awID(this));
      addState(new ready(this));
```

Die Transitionen `next`, `processP1`, `processP2` und `processP3` sind Implementierungen der entsprechenden Konnektoren und werden direkt durch Aufruf ihrer `fire()` Methoden ausgeführt. Im Gegensatz zu allen anderen Transitionen ist dafür keine Bedingung oder Signal erforderlich. Die Transitionen müssen der Laufzeitumgebung bekannt sein und sind daher ebenfalls im Konstruktor enthalten.

```
      /*Add Transitions*/
      addTransition(new next(this));
      addTransition(new processP1(this));
      addTransition(new processP2(this));
      addTransition(new processP3(this));
    };
};

#endif
```

Die erzeugte C++ Datei beinhaltet letztlich das eigentliche Verhalten:

```
/* Generated by ConTraST Version 1.0 */
#include "Package_InfoCollector.h"
…
```

Der Inhalt von CREATEALLCHANNELS und CREATESUBSTRUCTURE wurde bereits als Teil der Struktur in Kapitel 4.1 behandelt.

```
/* ProcessType NetInfo_Processtype */
void NetInfo_Processtype :: CREATEALLCHANNELS(SdlAgent* ow) {…}

void NetInfo_Processtype :: CREATESUBSTRUCTURE(class SdlAgent* ow) {…}

…
```

Die Transition 18 ist Teil eines ANY Zustands und damit Teil aller im Prozess vorhandenen Zustände. Sie ist daher nicht in Abbildung 4.3.1-1 enthalten und soll hier auch nicht betrachtet werden.

```
/* Transition T18 */
void NetInfo_Processtype :: T18
     :: fire(SDLInstance* owner,SignalInst *signal){
       NetInfo_Processtype* TP = ((NetInfo_Processtype*)owner);
     …
}
/* Transition T19 */
void NetInfo_Processtype :: T19
     :: fire(SDLInstance* owner,SignalInst *signal){
```

TP definiert hier das aktuelle NetInfo-Objekt mit dessen Variablen gearbeitet wird.

```
       NetInfo_Processtype* TP = ((NetInfo_Processtype*)owner);

     switch(offset){
       case 0:
```

LeaveStateNode wird zur Verwaltung der Laufzeitumgebung benötigt. Nach Ausführung von LeaveStateNode wird an der angegebenen Einsprungstelle (1) fortgesetzt.

```
         LeaveStateNode(1);
         break;
       case 1:
```

Speichern der Signalparameter in den lokalen Variablen des Prozesses und freigeben des vom Signal belegten Speichers. TP repräsentiert die Prozessinstanz

```
         TP->lhop = reinterpret_cast<InfoColl::RecvInfo*>(signal)->Param1;
         TP->rinfo = reinterpret_cast<InfoColl::RecvInfo*>(signal)->Param2;
         delete reinterpret_cast<InfoColl::RecvInfo*>(signal);
```

Skip realisiert den Sprung zur Konnektor-Transition von next mit Angabe der Standardeinsprungstelle (0)

```
         Skip(::NetInfo_Processtype::next::ID(),0); // next
         break;
       default: break;
     } // end switch(offset)
}
```

```
/* Transition T20 */
void NetInfo_Processtype :: T20
:: fire(SDLInstance* owner,SignalInst *signal){
       NetInfo_Processtype* TP = ((NetInfo_Processtype*)owner);

     switch(offset){
       case 0:
         LeaveStateNode(1);
         break;
       case 1:
         delete reinterpret_cast<NetInfo_Processtype::pTimer*>(signal);
```

Elementare Datentypen sind in der Laufzeitumgebung in eigenen Klassen gekapselt. Sie stellen neben den erforderlichen Standardoperatoren und Funktionen wie alle neu definierten Datentypen eine `encode` und `decode` Funktion zur Verfügung. Vor der Verwendung ist ein expliziter Cast auf die entsprechende Klasse notwendig

```
TP->i = (Integer)1;
{
```

Prozeduraufruf; wie bereits in Kapitel 3.1.4 beschrieben kann zurzeit nur ein optimierter Prozeduraufruf für zustandslose Prozeduren durchgeführt werden. Dabei wird eine neue Prozedurinstanz erzeugt (`new`) und ausgeführt (`fire()`). Anschließend erfolgt die (evtl.) notwendige Zuweisung des Rückgabewertes zu einer lokalen Variable.

```
  //Optimization of procedure call
  GetRandID *PROC1 =
    new GetRandID(Transition::owner,(SdlAgent*)this);
  PROC1->fire(PROC1);
  TP->res = PROC1->RESULT;
}
```

Setzen des Timers

```
Set(new pTimer());
```

Objektorientierte Umsetzung von `length(info)` aus der graphischen Darstellung bzw. PR Datei. Das Objekt (`TP->info`) wurde extrahiert und vor den Methodenaufruf gesetzt.

```
if((((TP->info).length() == (Integer)0) &
  (TP->res % (Integer)100 == (Integer)0)) == false) {
```

Umsetzung eines Einsprungpunktes (hier von Tau implizit generiert) als `case` Anweisung

```
case 2: //grst8
       if(( TP->i < ( TP->pset ).length()) == true) {
         …
case 3: //grst9
            if(( TP->chg) == true) {
                 …
            }
       }
  }
  else if((((TP->info).length() == (Integer)0) &
         (TP->res % (Integer)100 == (Integer)0)) == true) {
       {
         //Optimization of procedure call
         Time2Real *PROC5 = new Time2Real(
                Transition::owner,(SdlAgent*)this,now());
         PROC5->fire(PROC5);
         TP->r = PROC5->RESULT;
       }
       TP->PList(TP->myID).assign(TP->myID, TP->r, (Integer)1,
         -(Integer)1, TP->r,(Integer)0, (Integer)0, (Integer)0);
       TP->chg = true;
```

```
                Skip(::NetInfo_Processtype::T20::ID(),3); // grst9
                break;
            }
        default: break;
      } // end switch(offset)
}
```

Die `next` Transition beschreibt das Verhalten welches sich direkt an den Konnektor `next` anschließt.

```
/* Transition next */
void NetInfo_Processtype :: next
     :: fire(SDLInstance* owner,SignalInst *signal){
       NetInfo_Processtype* TP = ((NetInfo_Processtype*)owner);

     switch(offset){
       case 0:
       case 1: //next
         if(( ( TP->rinfo ).length() > (Integer)0) == true) {
              TP->data = (TP->rinfo).take();
              (TP->rinfo).del(TP->data);
              if(( TP->data.phase()) == 1) {
                   Skip(::NetInfo_Processtype::processP1::ID(),0); //processP1
                   break;
              }
              else if(( TP->data.phase()) == 2) {
                   Skip(::NetInfo_Processtype::processP2::ID(),0); //processP2
                   break;
              }
              else if(( TP->data.phase()) == 3) {
                   Skip(::NetInfo_Processtype::processP3::ID(),0); //processP3
                   break;
              }
              else {
                   Skip(::NetInfo_Processtype::T20::ID(),1); // next
                   break;
              }
         }
         else if(( ( TP->rinfo ).length() > (Integer)0) == false) {
         }
         Skip(::NetInfo_Processtype::T20::ID(),3); // grst9
         break;
       default: break;
     } // end switch(offset)
}
```

Definition des Konstruktors der Zustandsklasse `ready`.

```
NetInfo_Processtype ::ready::ready(SDLProcessType* ow):SDLState("ready",ow){
```

Es werden alle zum Zustand gehörenden Transitionen mit ihren entsprechenden Eingabesignalen angegeben. Bei der Transition `T18` handelt es sich um eine `ANY` Transition die beim Empfang des Signals `Mobility` in jedem Zustand des Prozesses ausgeführt wird. Sie ist nicht Teil des in Abbildung 4.3.1-1 gezeigten Ausschnitts und soll daher auch nicht weiter betrachtet werden.

```
     /*addTransition(new T18(ow),::InfoColl::Mobility::ID());*/
     addTransition(new T19(ow),::InfoColl::RecvInfo::ID());
     addTransition(new T20(ow),::NetInfo_Processtype::pTimer::ID());
}
```

Insgesamt ist auch hier ein relativ übersichtlich gehaltener Code erzeugt worden. Eine direkte Umsetzung der Darstellung des Verhaltens durch die PR Datei und der C++ Datei ist offensichtlich.

4.4 Vergleich des generierten Quellcodes von ConTraST mit Cmicro

Zum Vergleich zwischen den beiden Compilern soll ein Codeauszug der SDL Spezifikation des Ada I Protokolls dienen. Dabei wird der im vorhergehenden Kapitel beschriebene Code der C++ Datei dem entsprechenden Code des Cmicro Compilers gegenübergestellt.

Cmicro generierter Code:

ConTraST generierter Code:

Definition des Prozesses NetInfo mit Variablen und Typdeklaration

```
/*++++++
* Function for process NetInfo
++++++*/
YPAD_FUNCTION(yPAD_z44_NetInfo){
  YPAD_YSVARP
  YPAD_YVARP(yVDef_z_InfoCollector_80_NetInfo)
  YPAD_TEMP_VARS
  SDL_Integer yDcn_SDL_Integer;
#ifdef XTRACE
  SDL_Boolean yDcn_SDL_Boolean;
#endif
  union {
    struct {
      z_InfoCollector_8B_IntSet Result1;
    } U5;
    struct {
      z_DataStructures_1_DATASET Result1;
    } U6;
    struct {
      SDL_Integer Result1;
    } U16;
    struct {
      z_DataStructures_0_DATA Result1;
      SDL_Real Result2;
      …
    } U25;
…
  BEGIN_PAD
  switch (XSYMBOLNUMBER) {
```

```
/* ProcessType NetInfo_Processtype */
class NetInfo_Processtype :
   public SDLProcessType {
   private:
   public:
     /* New type Generator */
     typedef Array< Integer,
       ::DataStructures::DATA> PhaseList;
     typedef Powerset< Integer> NodeSet;
     typedef Array< Integer,
       ::NetInfo_Processtype::NodeSet>
        NodeSetList;
…
```

Empfang des Signals `RecvInfo` mit Speicherung der Signalparameter in den entsprechenden Prozessvariablen `[input RecvInfo(lhop, rinfo);]`

```
/*-----
* INPUT RecvInfo
* ------*/
case 3:
  XDEBUG_LABEL(ready_RecvInfo)
  XAT_FIRST_SYMBOL(3, 30)
  XOS_TRACE_INPUT("RecvInfo")
  yAssF_SDL_Integer(yVarP->z_InfoCollector…,
    ((yPDef_z_InfoCollector_84_RecvInfo *)
     ySVarP)->Param1, XASS_AR_ASS_FR);
#ifdef XVALIDATOR_LIB
  yAssF_z_DataStructures_1_DATASET(yVarP->z_I…,
    &(((yPDef_z_InfoCollector_84_RecvInfo *)
     ySVarP)->Param2), XASS_MR_ASS_FR);
#else
  yAssF_z_DataStructures_1_DATASET(yVarP->z_I…,
    &(((yPDef_z_InfoCollector_84_RecvInfo *)
     ySVarP)->Param2), XASS_AR_ASS_FR);
#ifdef XMONITOR
 memset(&(((yPDef_z_InfoCollector_84_RecvInfo*)
   ySVarP)->Param2),0, ySDL_z_DataStructures_1_
    DATASET.SortSize);
#endif
#endif
```

```
…
void NetInfo_Processtype :: T19 :: fire(
   SDLInstance* owner,SignalInst *signal){
   NetInfo_Processtype* TP =
     ((NetInfo_Processtype*)owner);
   switch(offset){
     case 0:
       LeaveStateNode(1);
       break;
     case 1:
       TP->lhop = reinterpret_cast<
         InfoColl::RecvInfo*>(signal)->Param1;
       TP->rinfo = reinterpret_cast<
         InfoColl::RecvInfo*>(signal)->Param2;
       delete reinterpret_cast<
         InfoColl::RecvInfo*>(signal);
```

Umsetzung eines Sprungs zur Sprungmarke next `[join next;]`

```
/*-----
* JOIN next
* ------*/
  goto L_next;
```

```
    //next
    Skip(::NetInfo_Processtype::next::ID(),0);
      break;
    default: break;
    } // end switch(offset)
}
```

Empfang eines Timer Signals `[input pTimer;]`

```
/*-----
* INPUT pTimer
* -----*/
case 4:
  XDEBUG_LABEL(ready_pTimer)
  XAT_FIRST_SYMBOL(4, 31)
  XOS_TRACE_INPUT("pTimer")
  INPUT_TIMER_VAR(yTim_pTimer)
  XBETWEEN_SYMBOLS(11, 38, 952){
```

```
/* Transition T20 */
void NetInfo_Processtype :: T20 :: fire(
   SDLInstance* owner,SignalInst *signal){
   NetInfo_Processtype* TP =
     ((NetInfo_Processtype*)owner);
   switch(offset){
    case 0:
     LeaveStateNode(1);
     break;
    case 1:
     delete reinterpret_cast<
      NetInfo_Processtype::pTimer*>(signal);
```

Setzen des Timers `[SET(pTimer);]`

```
/*-----
* SET pTimer
* ------*/
  SDL_SET_DUR(xPlus_SDL_Time(SDL_NOW,
   SDL_DURATION_LIT(10.0, 10, 0)),
   SDL_DURATION_LIT(10.0, 10, 0), pTimer,
   z_InfoCollector_800S_pTimer,
   yTim_pTimer, "pTimer")
   XBETWEEN_STMTS(12, 39, 962)
```

```
Set(new pTimer());
```

Zuweisung eines Wertes zu einer Variablen eines elementaren Datentyps `[i:=1;]`

```
/*-----
* ASSIGNMENT i := ...
* ------*/
  yAssF_SDL_Integer(yVarP->z_InfoCollector_80…,
   SDL_INTEGER_LIT(1),XASS_MR_ASS_FR);
#ifdef XTRACE
  xTraceAssign("i := ");
#endif
  }
  XBETWEEN_SYMBOLS(13, 40, 974){
```

```
TP->i = (Integer)1;
```

Umsetzung eines Prozeduraufrufs mit Zuweisung des Rückgabewerts zu `res`

`[res :=(CALL GetRandID);]`

```
/*-----
* ASSIGNMENT res := ...
* ------*/
  yAssF_SDL_Integer(yVarP->z_InfoCollector_80…,
   z_InfoCollector_2_GetRandID(XGP_PARAM_CE),
   XASS_MR_ASS_FR);
#ifdef XTRACE
  xTraceAssign("res := ");
#endif
  }
  XBETWEEN_SYMBOLS(14, 41, 987)
```

```
{
 //Optimization of procedure call
 GetRandID *PROC1 = new GetRandID(
  Transition::owner,(SdlAgent*)this);
 PROC1->fire(PROC1);
 TP->res = PROC1->RESULT;
}
```

Verzweigung `decision (length(info)=0) and (res MOD 100=0);]`

```
/*-----
* DECISION
* ------*/
#ifdef XTRACE
  yAssF_SDL_Boolean(yDcn_SDL_Boolean,
   xAnd_SDL_Boolean(yEqF_SDL_Integer(
    yLength_z_DataStructures_1_DATASET(
    &(yVarP->z_InfoCollector_800V_info)),
    SDL_INTEGER_LIT(0)),
   yEqF_SDL_Integer(xMod_SDL_Integer(
    yVarP->z_InfoCollector_8015_rand,
```

```
if((((TP->info).length() == (Integer)0) &
 (TP->res % (Integer)100 == (Integer)0))
  == false) {
  …
```

```
    SDL_INTEGER_LIT(100)),SDL_INTEGER_LIT(0))),
   XASS_MR_ASS_FR);
  if (yDcn_SDL_Boolean) {
#else
  if (xAnd_SDL_Boolean(yEqF_SDL_Integer(
   yLength_z_DataStructures_1_DATASET(
    &(yVarP->z_InfoCollector_800V_info)),
     SDL_INTEGER_LIT(0)),yEqF_SDL_Integer(
      xMod_SDL_Integer(yVarP->z_InfoCollector…,
       SDL_INTEGER_LIT(100)),
    SDL_INTEGER_LIT(0)))) {
#endif
#ifdef XTRACE
  xTraceDecision("TRUE");
#endif
  XBETWEEN_SYMBOLS(41, 68, 1009){
```

```
else if((((TP->info).length() ==
 (Integer)0) & (TP->res % (Integer)100
  ==(Integer)0)) == true) {
  {
```

Umsetzung eines Prozeduraufrufs mit Zuweisung des Rückgabewerts zu `r`

`[r:=CALL Time2Real(now);]`

```
/*-----
* ASSIGNMENT r := ...
* ------*/
  yAssF_SDL_Real(yVarP->z_InfoCollector_8010_r,
   z_InfoCollector_0_Time2Real(XGP_PARAM_C
    SDL_NOW), XASS_MR_ASS_FR);
#ifdef XTRACE
  xTraceAssign("r := ");
#endif
  XBETWEEN_STMTS(42, 69, 1021)
```

```
  //Optimization of procedure call
  Time2Real *PROC5 = new Time2Real(
   Transition::owner,(SdlAgent*)this,
    now());
  PROC5->fire(PROC5);
  TP->r = PROC5->RESULT;
 }
```

Zuweisung von Werten zu einem komplexen, selbstdefinierten Datentyp

`[PList(myID):=(. myID, r, 1, -1, r, 0,0,0 .);]`

```
/*-----
* ASSIGNMENT PList... := ...
* ------*/
  yAssF_z_DataStructures_0_DATA((*yAddr_z_Inf…(
   &(yVarP->z_InfoCollector_8017_PList),
    &(yVarP->z_InfoCollector_800U_myID))),
    GenericMakeStruct(&yUVar.U42.Result1,
     (tSDLTypeInfo *)&ySDL_z_DataStructures_0…,
     &(yVarP->z_InfoCollector_800U_myID),
     &(yVarP->z_InfoCollector_8010_r),
     yMkAddr_SDL_Integer(SDL_INTEGER_LIT(1),
     &yUVar.U42.Result2),yMkAddr_SDL_Integer(
     xMonMinus_SDL_Integer(SDL_INTEGER_LIT(1)),
     &yUVar.U42.Result3), &(yVarP->z_InfoCol…),
     yMkAddr_SDL_Integer(SDL_INTEGER_LIT(0),
     &yUVar.U42.Result4),yMkAddr_SDL_Integer(
      SDL_INTEGER_LIT(0), &yUVar.U42.Result5),
      yMkAddr_SDL_Real(SDL_REAL_LIT(0.0, 0, 0),
       &yUVar.U42.Result6)),
  XASS_MR_ASS_FR);
#ifdef XTRACE
   xTraceAssign("PList... := ");
#endif
   XBETWEEN_STMTS(43, 70, 1042)
```

```
TP->PList(TP->myID).assign(TP->myID,
 TP->r, (Integer)1,-(Integer)1,
 TP->r, (Integer)0, (Integer)0,
 (Integer)0);
```

Zuweisung zu einem elementaren Datentyp `[chg:=true;]`

```
/*-----
* ASSIGNMENT chg := ...
* ------*/
  yAssF_SDL_Boolean(yVarP->z_InfoCollector_80…,
   SDL_True,XASS_MR_ASS_FR);
#ifdef XTRACE
  xTraceAssign("chg := ");
#endif
  }
…
```

```
TP->chg = true;
…
```

Bei Cadvanced und Cmicro kommen gleichermaßen sehr viele Makros zum Einsatz, was zu Lasten der Übersichtlichkeit, Lesbarkeit und Nachvollziehbarkeit geht. Außerdem sind die Funktionalitäten eines Prozesses innerhalb einer Code Datei verstreut und z.T. auch auf mehrere Dateien verteilt. Auch dadurch wird die Fehlersuche und Nachvollziehbarkeit erschwert. Im Gegensatz dazu steht der von ConTraST generierte Code. Der ConTraST-Code ist einfach zu lesen und kann leicht den entsprechenden Abschnitten der graphischen SDL Repräsentation zugeordnet werden.

4.5 Die Laufzeitumgebung SDL Runtime Environment

Um den vom ConTraST generierten Code ausführen zu können, benötigt man eine entsprechend angepasste Laufzeitumgebung. Sie wurde parallel entwickelt und mit dem ConTraST Compiler abgestimmt. Zur Implementierung der Laufzeitumgebung wurde die Spezifikation der *ASM* von SDL2000 durch die ITU Z.100.F3 herangezogen.

Abbildung 4.3.3-1 zeigt den strukturellen Aufbau der *ASM*. Sie kann in zwei Abschnitte geteilt werden: Dem oberen Abschnitt mit ASMRuntime, Agent, SdlAgent, SdlAgentSet und Link. Alle Elemente dieser Ebene entsprechen dabei der SDL2000 Spezifikation. Die Adaption von SDL2000 zu SDL96 erfüllen dann die Klassen des darunter liegenden Abschnitts.

Im Wesentlichen besteht die Laufzeitumgebung aus Agents. Diese Agents sind entweder SdlAgents, SdlAgentSets oder Links und implementieren dabei die virtuelle Methode Program() von Agent. Die ASMRuntime wählt Agenten aus und führt deren implementiertes Program aus. Von den SdlAgents leiten sich SDLProcessType, SDLBlockType und SDLSystemType ab. Bei der Generierung eines SDLProcesses als konkrete Instanz eines SDLProcessTypes fließt der SDLProcessType in Form eines Template-Parameters in SDLProcess ein. Der in Abbildung 4.3.3-1 beispielhaft aufgeführte SDLProcessType enthält SDLTransitionen und SDLStates. Alle Klassen dieser Ebene sind zugleich die Oberklassen für die bei der Codeerzeugung generierten C++ Klassen des umzusetzenden SDL Systems.

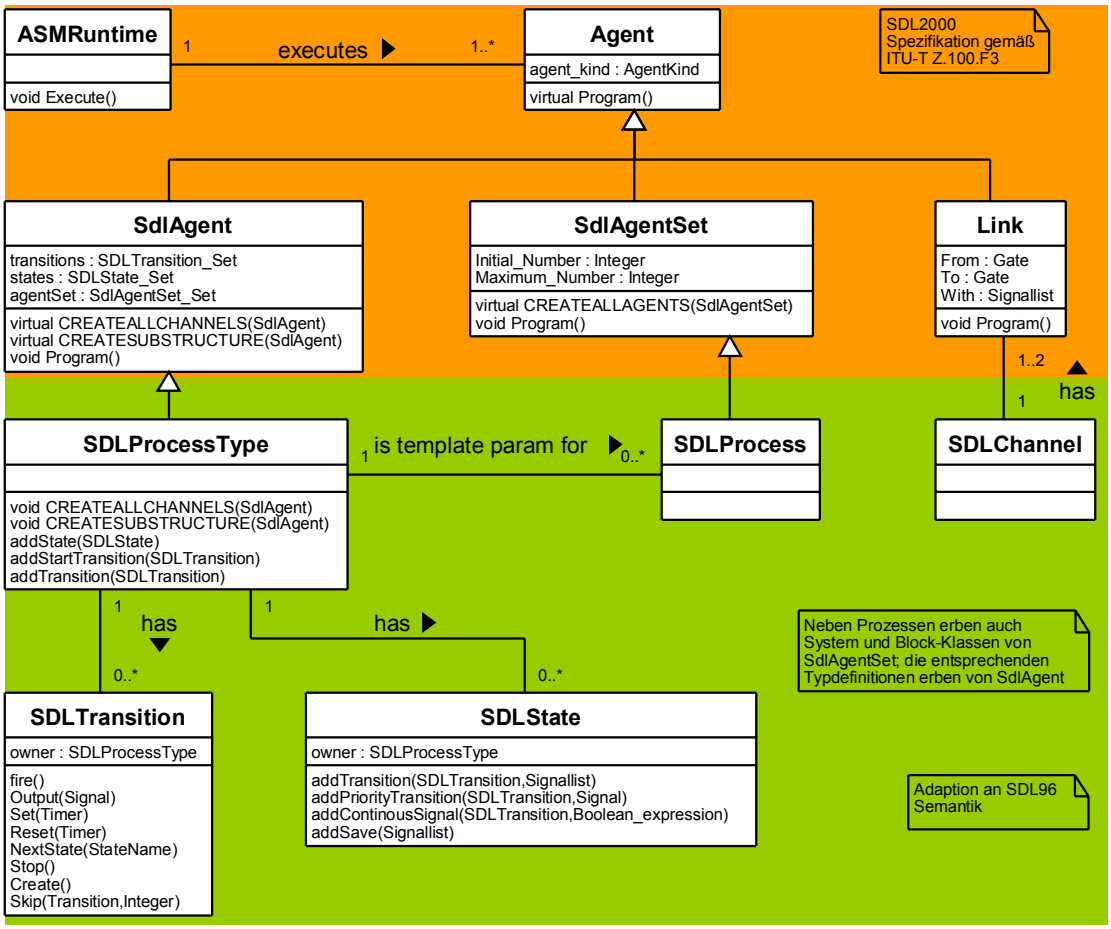


Abbildung 4.3.3-1

4.6 Vergleich mit bereits existierenden SDL Compilern

In Anhang B ist ein Vergleich des unterstützten Sprachumfangs bereits existierender SDL zu C Compiler mit dem hier entwickelten ConTraST Compiler gelistet. Als Referenzcompiler dienen dabei Cadvanced und Cmicro von Telelogic Tau, SDL2SPEETCL [11], ein SDL zu C++ Compiler der AixCom GmbH und SITE [12], eine Werkzeugsammlung entwickelt an der Humboldt Universität Berlin. Leider steht zu SITE keine explizite Dokumentation der im Einzelnen unterstützten SDL Komponenten zur Verfügung. Es wird lediglich angegeben, dass der enthaltene Compiler eine erweiterte SDL Syntax akzeptiert. Daher wurde bei SITE zum Vergleich der volle Sprachumfang von SDL als unterstützt angenommen. In der Dokumentation zum SDL2SPEETCL Compiler sind lediglich die unterstützten SDL Komponenten gelistet. Zusätzliche, wahrscheinlich unterstützte Komponenten wurden ohne Gewähr hinzugefügt und markiert.

Auf Grund der wenigen Informationen zu SITE und SDL2SPEETCL ist der Vergleich mit diesen Compilern nicht sehraussagekräftig. Die unterstützten SDL Komponenten von Cmicro und Cadvanced entsprechen bis auf wenige Abweichungen denen des hier entwickelten ConTraST.

Die wichtigsten Unterschiede zu Cmicro und Cadvanced sind die Unterstützung von *remote*, *imported* und *external* bei Prozeduren, die Unterstützung von *export* in Tasks, *Interfaces*, *Block-Substrukturen* und (eingeschränkt) *abstrakten Datentypen* durch Cmicro. Der Cadvanced Compiler unterstützt zusätzlich zu den bei Cmicro genannten Eigenschaften noch *Makros* und *import* Anweisungen. Dem gegenüber steht die Codeerzeugung für *Enabling Conditions* sowie für eingehende und ausgehende Signale bei Services durch den in dieser Arbeit entwickelten ConTraST Compiler, was bei Cmicro nicht möglich ist. Die fehlende Unterstützung durch den beschriebenen Compiler resultiert aber in fast allen Punkten entweder aus der bewussten Entscheidung gegen diese Komponenten (*remote*, *import*, *external, Makros*) oder ergibt sich aus Einschnitten im Sprachumfang von SDL2000, welcher der Laufzeitumgebung zugrunde liegt (*Block-Substrukturen*).

Als Ergebnis des Vergleichs kann festgehalten werden, dass der neu entwickelte ConTraST Compiler alle SDL Sprachkonstrukte in C++ Code abbilden kann, die zur Spezifikation eines SDL Systems benötigt werden. Die im Anhang B beschriebenen Einschränkungen sind größtenteils auf die fehlende Unterstützung durch die Laufzeitumgebung und die ihr zugrunde liegende SDL2000 Spezifikation zurückzuführen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde mit Hilfe verschiedener Werkzeuge ein SDL zu C++ Compiler entwickelt. Von *Telelogic Tau* generierte, textuelle Repräsentationen von SDL Systemen können damit durch *Flex* und *Bison* in einen Syntaxbaum eingelesen werden. Anschließend erfolgen notwendige Umwandlungen desselben, bevor aus den enthaltenen Informationen plattformunabhängiger, objekt-orientierter Code generiert wird. Die erzeugten Code Dateien sind durch den direkten Bezug zur SDL/PR Quelle leicht lesbar und verständlich. Durch Hinzunahme der parallel entwickelten Laufzeitumgebung wird ein ausführbares Programm erzeugt.

Mittels vor- oder eigens definierter SDL Subsets kann vor der Codegenerierung der Grad an zulässigen Sprachkonstrukten festgelegt werden. Dadurch ist es möglich, Code zu erzeugen, welcher an die vorgegebenen Einschränkungen der Zielhardware angepasst ist.

In Zukunft soll der Compiler und die zugehörige Laufzeitumgebung dahingehend verbessert und erweitert werden, dass zurzeit noch nicht oder nicht vollständig vorhandene Unterstützung von SDL Konstrukten hinzugefügt wird. Dies betrifft insbesondere Prozeduren mit Zuständen und deren Aufrufe. Auch die Unterstützung von Operatoren und Generatoren kommt als Teil der Weiterentwicklung des Compilers in Betracht.

Verbesserungen sind insbesondere im Bereich der Umsetzung von `Expressions` angestrebt. Dabei sollen die `Expressions` in `Tasks` nicht mehr komplett als C++ Strings gespeichert und zur Codeerzeugung erneut durch den Scanner in Token aufgesplittet werden. Statt dessen ist vorgesehen, die `Expression` bereits beim Parsen zu zerlegen. Damit ist eine typabhängige Speicherung der `Expression`fragmente in eigenen Klassen möglich. Dies erleichtert wiederum die Codeerzeugung für Funktionsaufrufe wie z.B. length() für die einzelnen Fragmente. Außerdem ist damit eine Überprüfung auf Typkonformität der einzelnen `Expressions` einfach möglich.

Auch die Codegenerierung bei `States` und `Transitionen` soll noch verbessert werden. So sollen alle `States` bereits vor der Codeerzeugung bekannt sein und nicht mehr bei der Codeerzeugung ermittelt werden müssen. Dadurch wird unter anderem eine Verbesserung bei der Generierung von `ANY`-States erreicht. Im Fall der `Transitionen` wird die Menge der Eingangssignale ebenfalls nicht mehr zur Compilezeit ermittelt werden. Da die Eingangssignale noch an weiteren Stellen bei der Codeerzeugung zur Verfügung stehen müssen, werden sie in Zukunft direkt im `Prozess` / `Service` / `Prozedur` gespeichert werden. Auch hier ist eine Vereinfachung bei der Codegenerierung für `ANY`-Signale z.B. bei Save-Anweisungen angestrebt.

Des Weiteren sollen auch noch die zurzeit ausgegebenen Warnungen und Fehlermeldungen verfeinert und erweitert werden.

6 Anhang

Anhang A Quellenverzeichnis

[1] „Syntaxanalyse“
Otto Mayer
3.Auflage Bibliographisches Institut, 1986

[2] „lex & yacc“
Helmut Herold
3.Auflage Addison-Wesley Verlag, 2003

[3] http://www.gnu.org/software/flex/

[4] http://www.gnu.org/software/bison/

[5] „SDL Formal Object-oriented Language for Communicating Systems“
Jan Ellsberger, Dieter Hogrefe, Amardeo Sarma
Prentice Hall Europe 1997

[6] „CCITT Specification and Description Language (SDL)“
ITU Z.100 (03/93) + (10/96) + (1998)
http://www.itu.int/ITU-T/

[7] „CCITT Specification and Description Language (SDL)“
ITU Z.100.F3 (11/00)
http://www.itu.int/ITU-T/

[8] „ASN.1 Complete“
John Larmouth
Morgan Kaufmann Academic Press

[9] http://www.telelogic.com/

[10] http://www.lkn.ei.tum.de/arbeiten/faq/man/tau42_help/online_contents.html

[11] http://aixcom.com/

[12] http://www.informatik.hu-berlin.de/SITE/

Anhang B Language Support

SDL96	SDL Set	Parser	ConTraST	Runtime Env	Cmicro*	Cadvanced*	SDL2SPEETCL**	SITE***
Package	C	X	X	X (->C++)	X	X	X	X
System	C	X	X	X	X	X	?	X
Systemtype	C	X	X	X	X	X	X	X
Block	C	X	X	X	X	X	?	X
Blocktype	C	X	X	X	X	X	X	X
Process	C	X	X	X	X	X	?	X
Processtype	C	X	X	X	X	X	X	X
Service	S2	X	X	O?	X	X		X
Servicetype	S2	X	X	O?	X	X		X
Procedure	D	X	X	without states only (will be fixed)	without states only	X	X	X
remote	-	X	-	-	X	X	X	X
imported	-	X	-	-	X	X		X
external	-	X	-	-	X	X		X
Reference	C	X	Replaced by Instan-ces	X	X	X	?	X
Inheritance	S2	X	O	->C++	No procedures	X		X
Operator	?	X	?	?	X	X		X

SDL96	SDL Set	Parser	ConTraST	Runtime Env	Cmicro*	Cadvanced*	SDL2SPEETCL**	SITE***
Generator	?	X	?	?	X	X		X
Instance	**C**: only one Instance; **S1**: initial number = max number; **D**: initial < max	X	X	X	No infinit number (x,) or (,)	X		X
Interface	?	X	?	?	X	X		X
Substructure	-	X	-	-	No channel substructures	X	X	X
View	-	X	-	-	-	X	X	X
Channel	C	X	X	X	X	X	?	X
delay	-	X	-	-	-	-		X
Signalroute	C	X	X	->Channel	X	X	?	X
Connection	C	X	X (->Gate)	->Gate	X	X	?	X
Gate	C	X	X	X	X	X	?	X
Transition								
Start	C	X	X	X	X	X	?	X
Input	C	X	X	X	Not in services; No enabeling condition	X	?	X
Priority Input	S2	X	X	X	X	X	?	X
Spontaneous	S2	X	X	O	No enabeling condition	X	?	X

SDL96	SDL Set	Parser	ConTraST	Runtime Env	Cmicro*	Cadvanced*	SDL2SPEETCL**	SITE***
Output	C	X	X	X	Not in services, not „via all“	X	?	X
To / Via	C	X	X	X	X	X	?	X
Create/Stop	D	X	X	O	Processes without FPARS only	X	?	X
Decision	S1	X	X	X (->C++)	Multiple paths from a decision for a certain decision expression value are not checked.	Multiple paths from a decision for a certain decision expression value are not checked.	?	X
Set	S1	X	X	X	X	X	?	X
Reset	S1	X	X	X	X	X	?	X
Export	-	X	-	-	X	X	?	X
Join / Free_Action		X	X	X(-> C++)	X	X	?	X
Procedure Call	D	X	Not allowed as Parame-ter for Procedures, Signals, Timers or Variables	no further calculation with return values after call in same expression	No nested procedure call data scope;A process that exports procedures cannot call (directly or indirectly) a global procedure contai-ning states.	A process that exports procedures cannot call (directly or indirectly) a global procedure contai-ning states.	?	X
remote	-	X	-	-	-	X	?	X
Task	S1	X	X	X (->C++)	X	X	?	X

SDL96	SDL Set	Parser	ConTraST	Runtime Env	Cmicro*	Cadvanced*	SDL2SPEETCL**	SITE***
Save	S1	X	X	X	X	X	?	X
Continous Signal	S2	X	X	X	-	X	?	X
Enabling Condition	S2	X	X	X	X	X	?	X
ANY Expression	-	X	-	-	-	-		X
Timer	S1	X	X	X	Duration cannot be real; not more than one parameter; no other parameter than Integer; no active expressions for Timers with parameters	no active expressions for Timers with parameters	?	X
State	C	X	X	X	X	X	?	X
Signallist	C	X	X	X	X	X	?	X
Signalset	C	X	X	X	X	X	?	X
Signal	C	X	X	X	X	X		X
refinement	S2	X	O (->C++)	->C++	-	-		X
Synonym	S2	X	X	X (->C++)	X	X	?	X
Variable	S1	X	X	O	X	X	?	X
remote	-	X	-	-	X	X		X
imported	-	X	-	-	X	X		X

SDL96	SDL Set	Parser	ConTraST	Runtime Env	Cmicro*	Cadvanced*	SDL2SPEETCL**	SITE***
Integer	S1	X	->Runtime Env	X	Restricted in precise & range	Restricted in precise & range	X	X
Real	S1	X	->Runtime Env	X	Restricted in precise & range	Restricted in precise & range	X	X
Charstring	S1	X	->Runtime Env	X	No NUL allowed	No NUL allowed	X	X
Boolean	S1	X	->Runtime Env	X	X	X	X	X
PID	S1	X	->Runtime Env	X	X	X	X	X
Natural	S1	X	->Runtime Env	X	X	X	X	X
Character	S1	X	->Runtime Env	X	X	X	X	X
String	D	X	->Runtime Env	X	X	X	?	X
Array	D	X	->Runtime Env	X	X	X	X	X
OctetString	D	X	->Runtime Env	X	X	X	?	X
Time	S1	X	->Runtime Env	X	X	X	X	X

SDL96	SDL Set	Parser	ConTraST	Runtime Env	Cmicro*	Cadvanced*	SDL2SPEETCL**	SITE***
Duration	S1	X	->Runtime Env	X	X	X	X	X
Bag / Powerset	D	X	->Runtime Env	X	X	X	?	X
Newtype	S1	X	X	X(->C++)	X	X	?	X
Struct	S1	X	X	X (->C++)	A component of a struct will have the same name as in SDL. Characters that are not letters, numerals or underscore characters will be removed. The remaining part of the name must be a valid C identifier to be accepted by the C compiler.	A component of a struct will have the same name as in SDL. Characters that are not letters, numerals or underscore characters will be removed. The remaining part of the name must be a valid C identifier to be accepted by the C compiler.	X	X
Syntype	S1	X	X	X (->C++)	any(Sort) where Sort is a syntype is only implemented if the syntype contains at most one range condition which is of the form a:b; If it is a syntype of a real type, e.g. Real or Time, with a range condition it is not implemented	any(Sort) where Sort is a syntype is only implemented if the syntype contains at most one range condition which is of the form a:b; If it is a syntype of a real type, e.g. Real or Time, with a range condition it is not implemented	?	X

SDL96	SDL Set	Parser	ConTraST	Runtime Env	Cmicro*	Cadvanced*	SDL2SPEETCL**	SITE***
Abstract Data Types	-	X	-	-	Axioms & literal mapping information not used; Name class literals & naming literals using charstrings cannot be handled	Axioms & literal mapping information not used; Name class literals & naming literals using charstrings cannot be handled	X	X
Macro	-	-	-	-	-	X	X	X
Export	-	X	-	-	-	X	X	X
Import	-	X	-	-	-	X	X	X
Names / Identifiers	C	X	X	X	Names of processes within different blocks must be different	X	?	X
Context Parameters	D	X	O	O	-	-		X
Sort	S2	X	O	O	Sort definition may not refer to itself (direct or indirect)	Sort definition may not refer to itself (direct or indirect)	?	X

* Restrictions due to SDL to C Compiler for Cmicro & Cadvanced (see Telelogic Tau Help : Known Limitations)

** Only explicitely mentioned features are reported here (see www.aixcom.com for details); ? = supposed to support

*** No explicite limitations found; www.informatik.hu-berlin.de/SITE mentions that the parser accepts a larger set of syntax than the specification allowes

Legend: X supported O to be supported ? unknown C minimal Core

S(x) static extension x D dynamic extension C++ realized by C++ features