Realisierung eines SDL Simulationstools

Daniel Toggweiler

Institut für Informatik
Universität Bern
Länggassstr. 51
CH-3012 Bern

toggweil@iam.unibe.ch

IAM-94-001

Februar 1994

Zusammenfassung

Konformitätstests sind die Basis für eine harmonische Zusammenarbeit von Kommunikationsprodukten. Mit Hilfe von Testfällen werden Implementierungen gegen Protokollspezifikationen in wichtigen Aspekten auf Konformität getestet. Die Generierung von Testfällen ist ein grosses Problem. Im Rahmen des Forschungsprojekts *Conformance Testing - Ein Tool zur Generierung von Testfällen*[1] wurde an der Universität Bern das Tool SAMSTAG zur automatischen Testfallgenerierung entwickelt. SAMSTAG ist die Abkürzung für ***S**DL **a**nd **M**SC based **T**est C**a**se **G**enerator.* SAMSTAG basiert auf der Simulation einer SDL Spezifikation und eines MSCs. Es werden Systemabläufe generiert, die bestimmmte Bedingungen erfüllen.

Dieser Bericht beschreibt den Bau eines Simulationstools für SDL Spezifikationen. Das Simulationstool wurde im Rahmen des obigen Projekts entwickelt.

CR Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General; C.2.2 [Computer-Communication Networks]: Network Protocols; D.2.5 [Software Engineering]: Testing and Debugging

[1] Vertragsnummer 233/257, finanziert durch die Generaldirektion der schweizerischen PTT

Inhaltsverzeichnis

1 Einleitung **1**

2 Funktionalität des Simulationstools **2**
2.1 Schnittstelle zum SDL Frontend 2
2.2 Schnittstelle zum Testfallgenerator 2

3 Realisierung des Simulationstools **4**
3.1 Architekturvarianten 4
3.2 Wahl einer Architekturvariante 5

4 Funktionalität des Simulators **6**
4.1 Die Schnittstelle zum SDL Transformator 6

5 Realisierung des Simulators **8**
5.1 Zustände und Ereignisse des SDL Systems 8
5.2 Funktionen des Simulators 9
5.3 Generische Klassen 9
5.4 Klassen des Simulators 10
5.5 Abhängigkeiten der Datenstrukturen 13
5.6 Testprogramme, lauffähige Teilprogramme 13

6 Funktionalität des Transformators **14**

7 Realisierung des Transformators **15**
7.1 Transformationsphasen 15
7.2 Klassen des Transformators 16
7.3 Abhängigkeiten der Datenstrukturen 22
7.4 Kommunikations-Struktur-Algorithmus 22

8 Abschliessende Bemerkungen **24**

Abbildungsverzeichnis **25**

Literatur **26**

1 Einleitung

Das Simulationstool, das hier vorgestellt wird, wurde im Rahmen eines F & E Projektes[2] entwickelt. Es ist Teil des Werkzeugs SAMSTAG.

SAMSTAG generiert einen einzelnen Testfall für eine Systemspezifikation und einen Testzweck. Die Systemspezifikation wird durch eine SDL Beschreibung [CCI92b] gegeben und der Testzweck durch einen MSC [CCI92a]. Der Testfall wird in TTCN/MP [ISO91] ausgegeben.

Ein Testfall enthält das beobachtbare Verhalten von Systemabläufen. SAMSTAG berechnet diese Systemabläufe und deren beobachtbare Verhalten. Zu diesem Zweck simuliert SAMSTAG eine SDL Spezifikation und einen MSC parallel. Nach der Berechnung wird der Testfall ausgegeben.

SAMSTAG besteht aus vier Komponenten (vgl. Abb. 1): Das *SDL Simulationstool* und das *MSC Simulationstool* sind für die Simulation der SDL Spezifikation, resp. des MSCs verantwortlich. Der *Testfall Generator* steuert die Simulationstools so, dass die beobachtbaren Verhalten mit möglichst geringem Aufwand berechnet werden und speichert sie in einer internen Datenstruktur ab. Dem *TTCN Verwalter* obliegt die Ausgabe des Testfalls.

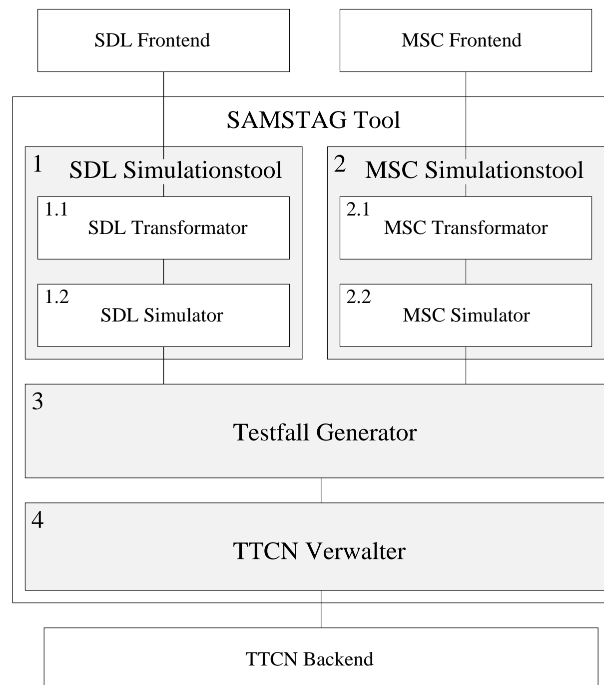

Abbildung 1: Architektur des SAMSTAG Werkzeugs

Die SAMSTAG zugrundeliegende Methode ist in [GHN93] beschrieben. Das Tool wurde komplett in C++ implementiert. Wie SAMSTAG angewendet wird, kann [TN93] entnommen werden.

[2] *Conformance Testing - Ein Tool zur Generierung von Testfällen*, Vertragsnummer 233/257, finanziert durch die Generaldirektion der schweizerischen PTT

Das kommerzielle SDT [Tel93a, Tel93b] dient als SDL Frontend und wird benutzt, um die SDL Spezifikation zu erstellen. Es können auch andere SDL Editoren, die eine SDL/PR Schnittstelle besitzen, verwendet werden. Als Frontend für das MSC Simulationstool wurde ein MSC Editor gebraucht, welcher an der Universität Bern entwickelt wurde [Tog92]. Als TTCN Backend dient das ITEX Tool [Swe92].

Welche Anforderungen das SDL Simulationstool erfüllen muss ist in Kapitel 2 beschrieben. Die Simulation der SDL Spezifikation spielt in SAMSTAG eine wichtige und zeitkritische Rolle. Deshalb ist es auch wichtig, die verschiedenen möglichen Architekturen für ein Simulationstool zu untersuchen. Kapitel 3 beschreibt die verschiedenen Architekturen und zeigt die Wahl der geeigneten Architekturvariante. Das Simulationstool ist in zwei Module zerlegt: Der *Transformator* liest eine SDL Spezifikation ein und speichert sie in geeigneter Form. Welche Funktionalität der Transformator erfüllt ist in Kapitel 6 zu finden. Wie der Transformator realisiert wurde, beschreibt Kapitel 7. Der *Simulator* interpretiert die vom Transformator zur Verfügung gestellten Daten. Die Anforderungen und die Realisierung des Simulators sind in den Kapiteln 4 und 5 zu finden.

2 Funktionalität des Simulationstools

Das SDL Simulationstool (siehe auch Abb. 1, 1) muss über zwei Schnittstellen kommunizieren können:

- Schnittstelle zum SDL Frontend
- Schnittstelle zum Testfallgenerator

2.1 Schnittstelle zum SDL Frontend

Das SDL Simulationstool muss eine Systemspezifikation in SDL/PR Form vom Frontend einlesen können. Die Spezifikation kann in drei verschiedenen Formen von SDL/PR vorliegen:

1. SDL/PR nach CCITT Recommendation Z.100 (vgl. [CCI92b]).
2. SDL/PR nach Z.100 mit in Kommentaren befindlichen C-Anweisungen (dies wird von SDT [Tel93b] angeboten).
3. SDL/PR mit C-Datenbeschreibungen und C-Anweisungen.

Da es sich beim SAMSTAG Werkzeug um einen Prototypen handelt, unterstützt das Simulationstool nur folgende SDL Sprachkonstrukte: `start`, `stop`, `state`, `input`, `output`, `task`, `decision`, `save`, `continuous-signal`, `timer`, `comment`, `join`, `newtype`, `dcl`, `syntype` und `synonym`. Diese Schnittstelle ist im SDL Transformator realisiert.

2.2 Schnittstelle zum Testfallgenerator

Die Schnittstelle zum Testfallgenerator ist im SDL Simulator realisiert. Sie basiert auf Zuständen und Ereignissen. Zustände charakterisieren globale Zustände des SDL Systems.

Ereignisse entsprechen Aktionen der Prozesse wie `input`, `output` oder `task`. Der SDL Simulator arbeitet in der folgenden Weise:

1. Der Simulator muss zuerst im Startzustand initialisiert werden.

2. Um einen nächsten Zustand zu erreichen, müssen alle von der Kontrollstruktur möglichen Nachfolgeereignisse bestimmt werden. Davon werden diejenigen Ereignisse gestrichen, welche aufgrund des Daten- oder Signalzustandes nicht ausführbar sind. Ein *Datenzustand* ist die Belegung der Prozess Variablen. Ein *Signalzustand* ist durch den Inhalt der Signalwarteschlangen bestimmt. Von den übriggebliebenen Ereignissen muss eines ausgewählt werden.

3. Danach sind in dem Prozesszustand die Änderungen auszuführen, welche durch dieses Ereignis hervorgerufen werden. Der Simulator kann anschliessend wieder Schritt 2 ausführen.

Die Schritte 2 und 3 werden solange wiederholt, bis die Ausführung durch ein Terminierungskriterium abgebrochen wird. In unserem Fall wird dies vom Testfallgenerator bestimmt.

Da das SAMSTAG Tool in C++ implementiert ist, wird die Schnittstelle mittels der Klasse `SDL_State` realisiert. Sie speichert einen Zustand des Simulators. Zur Steuerung des Simulators stellt sie dem Testfallgenerator folgende drei Funktionen zur Verfügung:

- `startState()` überführt einen Zustand in den Startzustand
- `enabled()` liefert die ausführbaren Ereignisse
- `nextState()` vollzieht einen Zustandsübergang

Die Funktionsaufrufe entsprechen dem Input und die Veränderung der Zustände dem Output an der Schnittstelle zum Testfallgenerator. Die Auswahl des nächsten auszuführenden Ereignisses wird dem Testfallgenerator überlassen.

Der Testfallgenerator versucht, möglichst wenig Systemabläufe zu generieren, ohne dass Lösungen verloren gehen. Deshalb wurden verschiedene Heuristiken implementiert, welche diese Absicht unterstützen. Einige sollen hier genannt werden:

- Die *First Sent First Consumed* Heuristik versucht, ein gesendetes Signal so früh wie möglich zu konsumieren.
- Die Tester Prozesse können zu jeder Zeit jedes Signal senden und empfangen. Um dies zu beschränken wurde die *Strong Reasonable Environment* Heuristik eingeführt. Diese erlaubt den Testern nur ein Signal zu senden, wenn sich das System in einem stabilen Zustand befindet, d.h. wenn alle Signalwarteschlangen leer sind und sich alle Prozesse in SDL Zuständen befinden.
- Da während der Simulation keine Systemzeit mitsimuliert wird, kann ein gesetzter Timer zu jeder Zeit ablaufen. Durch die *Strong Reasonable Timer* Heuristik wird dem Timer nur erlaubt abzulaufen, wenn sich das System in einem stabilen Zustand befindet.

Eine komplette Beschreibung der verwendeten Heuristiken ist in [Nah94] zu finden.

Um diese Heuristiken implementieren zu können, braucht der Testfallgenerator zusätzliche Information. Deshalb stellt ihm die Klasse `MSC_State` noch einige Zusatzfunktionen zur Verfügung:

- `isFinalState()` gibt an, ob sich das SDL System wieder in dem Startzustand befindet.
- `isStableState()` gibt an, ob ein Zustand stabil ist. Ein Zustand ist stabil wenn sich alle Prozesse in SDL Zuständen befinden und alle Signalwarteschlangen leer sind.
- `consumesOldestSignal()` prüft, ob ein Ereignis das älteste Signal konsumiert.

3 Realisierung des Simulationstools

Eine wichtige Frage ist die Wahl einer Architektur. Deshalb sollen hier zuerst verschiedene Architekturen vorgestellt werden.

3.1 Architekturvarianten

Es existieren verschiedene Architekturen um Simulationswerkzeuge zu implementieren. Wir unterscheiden zwischen fixen und flexible Architekturen.

Die fixe Architektur. In der fixen Architektur (Abb. 2.a) wird eine Spezifikation (a) von dem Transformator (b) eingelesen und in Programmcode (c) umgesetzt. Anschliessend wird der erzeugte Programmcode (c) zusammen mit systeminvariantem Programmcode (d) von einem Compiler (e) in ein ausführbares Programm (f) übersetzt. Das ausführbare Programm ist der eigentliche Simulator (g) und liefert dann die Schnittstelle (h) zum Testfallgenerator.

Die flexible Architektur. In der flexiblen Architektur (Abb. 2.b) liest der Transformator (j) eine textuelle Spezifikation (i) ein und speichert sie in einer internen Datenstruktur (k) ab. Der Simulator (l) interpretiert diese Datenstruktur und stellt so die Schnittstelle (m) dem Testfallgenerator zur Verfügung.

Gemischte Architekturen. Zwischen den obengenannten Architekturen existieren mehrere Mischformen, bei denen nur die Daten oder die Ablaufstruktur dynamisch verwaltet werden.

Bewertung der Architekturen. Die beiden Architekturen weisen gewisse Vor- und Nachteil auf. Sie sind in Tabelle 1 dargestellt.

Der Geschwindigkeitsverlust, welcher gemischte Architekturen im Vergleich zur fixen Architektur aufweisen, orientiert sich am Anteil der Interpretation.

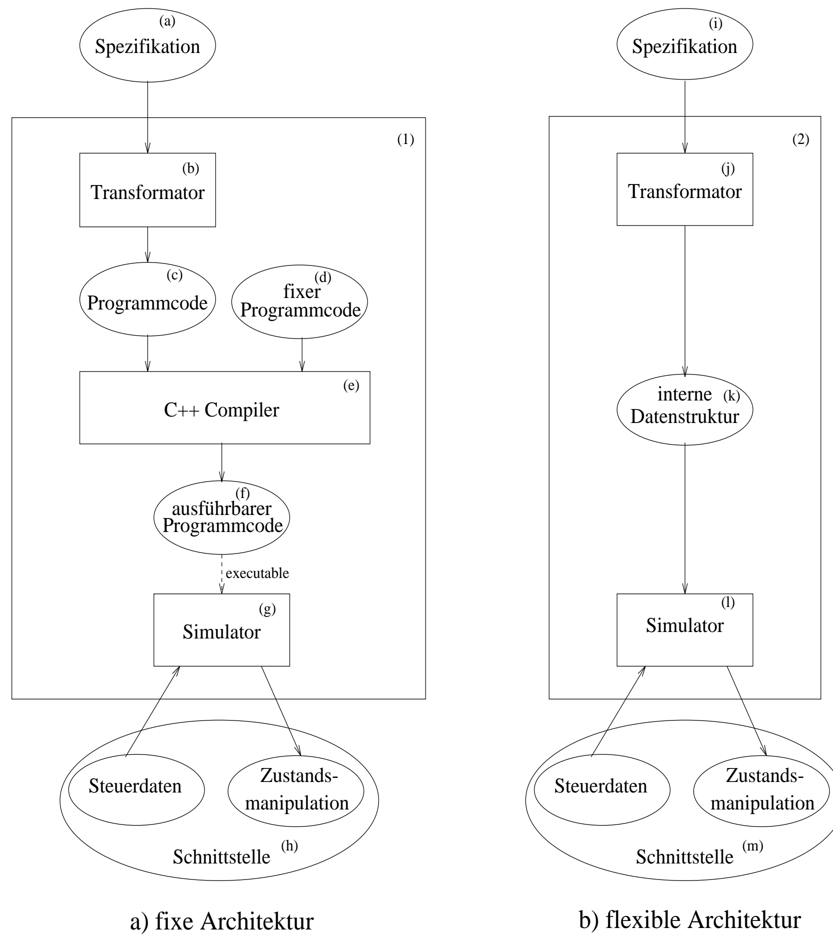

Abbildung 2: Architekturvarianten

3.2 Wahl einer Architekturvariante

Als erstes wurde geklärt, welche der in Abschnitt 3.1 dokumentierten Architekturvarianten implementiert werden soll. Um eine Variante auszuwählen, müssen die Vor- und Nachteile der einzelnen Architekturen gegeneinander abgewogen werden. Ein wichtiges Kriterium für den Bau eines Testfallgenerators ist die Schnelligkeit des Simulators. Die Kompilierungszeit ist irrelevant im Vergleich zur Ausführungszeit, da der Kompiliervorgang nur einmal für jede SDL Spezifikation durchgeführt werden muss. Da sich ein System gemäss unseren Anforderungen zur Laufzeit nicht verändert, d.h. keine Prozesse dynamisch initiiert werden, entfällt diese Restriktion für fixe Architekturen. Wir haben uns deshalb für eine fixe Architektur entschieden. Die genaue Bewertungstabelle ist in Tabelle 2 dargestellt.

Architektur	Vorteile	Nachteile
Flexible Architektur	- Es können mehrere Spezifikationen simuliert werden ohne dass jeweils ein neuer Simulator generiert werden muss	- Langsam, da jeder Zustandsübergang interpretiert wird
Fixe Architektur	- Sehr schnell, da alle Funktionen fest implementiert sind und keine interne Datenstruktur interpretiert wird	- Für jede Spezifikation muss ein neuer Simulator generiert werden - Die Spezifikationen dürfen keine Systeme beschreiben, die sich zur Laufzeit beliebig verändern
Gemischte Architekturen	- Schneller als flexible Architektur - Einschränkungen bei fixer Architektur können unter Umständen vermieden werden	- Für jede Spezifikation muss ein neuer Simulator erzeugt werden

Tabelle 1: Bewertung der Architekturen

Kriterium	fixe Architektur	flexible Architektur
Rekompilierung	0	+
Schnelligkeit	++	– –
dynamische Prozessinitiierung	0	0
Summe	++	–

Tabelle 2: Bewertung der Vor- und Nachteile der Architekturen

4 Funktionalität des Simulators

Wie aus der Architektur (Abb. 1) des SAMSTAG Werkzeugs zu ersehen ist, besitzt der SDL Simulator zwei Schnittstellen:

- Die Schnittstelle zum SDL Transformator (Abschnitt 4.1) und
- die Schnittstelle zum Testfallgenerator (Abschnitt 2.2)

4.1 Die Schnittstelle zum SDL Transformator

Diese Schnittstelle besteht aus dem Programmcode der unten aufgelisteten Funktionen, die vom Transformator erzeugt werden. Sie haben eine zentrale Bedeutung für die Arbeitsweise des Simulators. Der restliche Programmcode ist für jede SDL Spezifikation fest vorgegeben und wird nicht vom Transformator erzeugt.

- Die `InitFunction()` initialisiert den Simulator, d.h. sie führt das System in den Startzustand.
- Die `NextEventFunction()` gibt für jeden Zustand diejenigen Ereignisse an, die anhand des *Kontrollzustands* des Systems ausführbar sind. Der Kontrollzustand eines Systems setzt sich aus den Kontrollzuständen der einzelnen Prozesse zusammen. Der Kontrollzustand eines Prozesses wird durch das zuletzt ausgeführte Ereignis repräsentiert.
- Die `TestFunction()` überprüft, ob ein bestimmtes Ereignis in einem Zustand ausführbar ist. Sie berücksichtigt dabei nur den Daten- und den Signalzustand.
- Die `FillEventFunction()` berechnet die aktuellen Werte der Signalparameter.
- Die `TransferFunction()` führt einen Zustand in den ausgewählten Nachfolgezustand über.

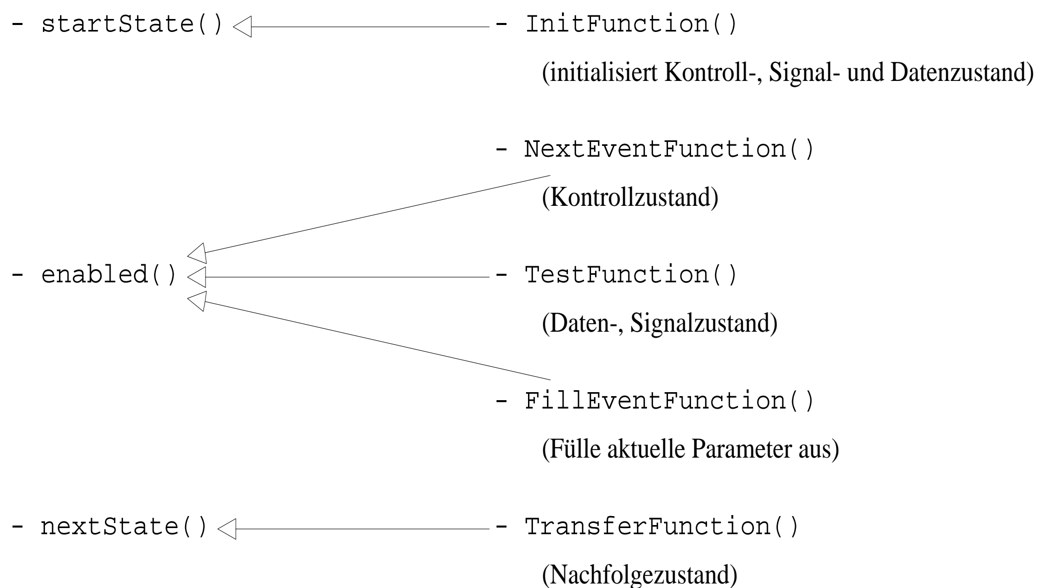

Abbildung 3: Zusammenhang zwischen den Schnittstellenfunktionen

Die `InitFunction()` ist äquivalent zur Funktion `startState()` (siehe Abbildung 3). Die `NextEventFunction()` und die `TestFunction()` bestimmen gemeinsam die ausführbaren Ereignisse unter Berücksichtigung des ganzen SDL Zustandes (Kontroll-, Daten- und Signalzustand). Zusätzlich gibt die `FillEventFunction()` die aktuelle Belegung der Signalparameter an. Die drei Funktionen `NextEventFunction()`, `TestFunction()` und `FillEventFunction()` realisieren zusammen die Funktion der Schnittstelle zum Testfallgenerator `enabled()`. Die Funktion `nextState()` wird von der Funktion `TransferFunction()` realisiert. Die `TransferFunction()` ist eine statische Funktion und

vollzieht bei einem Zustandsübergang nur die Änderungen, die schon bei der Transformation des SDL Systems bekannt sind (z.B. Ereignisnummern, Ereignistyp, Signalnamen, etc.). Die dynamischen Änderungen, die erst zur Laufzeit ermittelt werden können, wie die aktuelle Belegung der Signalparameter, werden von Hilfsfunktionen ausgeführt, die von der `TransferFunction()` aufgerufen werden.

5 Realisierung des Simulators

In diesem Abschnitt wird beschrieben, wie die geforderte Funktionalität im SAMSTAG Werkzeug realisiert ist. Die benutzten Beispiele sind dem Inres Protokoll entnommen [Hog89].

5.1 Zustände und Ereignisse des SDL Systems

Zustände. Ein globaler Zustand eines SDL Systems besteht aus dem Kontroll- (zuletzt ausgeführte Ereignisse), dem Daten- (Belegung der Prozess Variablen) und dem Signalzustand (Inhalt der Signalwarteschlangen). Die Speicherung von Zuständen oder von ganzen Folgen von Zuständen wird vom Testfallgenerator durchgeführt. Die Klasse `SDL_State` wird jedoch im Simulator realisiert.

- **Kontrollzustand.** Für eine effiziente Darstellung des Kontrollzustands, wird für jeden Prozess die Nummer des zuletzt ausgeführten Ereignisses gespeichert. Die Speicherung wird mittels einer Array-Struktur realisiert.

- **Datenzustand.** Um Prozess Variablen mit identischen Namen eindeutig referenzieren zu können, wird für jedes statische SDL Objekt (System, Block oder Prozess) eine eigene Klasse generiert (Objektklasse). Die Klasse enthält dann alle Prozess Variablen als Datenelemente. Die Namen der Klassen werden nach folgender Regel erzeugt:

  $$< ObjektTyp >< ObjektName > Data$$

  z.B.

  ```
  BlockMediumData
  ProcessInitiatorData
  ```

  Im `SDL_State` erhält jedes Objekt ein Datenmember seines Typs mit dem Namen des Objektes, z.B.

  ```
  ProcessInitiatorData Initiator;
  ```

- **Signalzustand.** Um den Signalzustand speichern zu können, enthält jede Prozessklasse eine Signalwarteschlange, d.h. ein Datenmember der Klasse `SignalQueue` (Abschnitt 5.4)

Funktionen. Die Funktionen dieser Klasse sind die Funktionen der Schnittstelle zum Testfallgenerator. Sie sind in Abschnitt 2.2 beschrieben.

Ereignisse. Die Klasse `Event` dient zur Speicherung von Ereignissen. Ereignisse werden in verschiedenem Zusammenhang benutzt: Zur Darstellung von Ereignissen in MSCs oder SDL Spezifikationen und zur Darstellung von Ereignissen in einem Ablauf oder Testfall. Dementsprechend umfassend ist die Klasse `Event` definiert.

5.2 Funktionen des Simulators

In diesem Abschnitt werden einige Angaben zur Realisierung der Funktionen gemacht, die vom Transformator generiert werden (Abschnitt 2.2).

Um einen Zustand in den Initialzustand zu überführen, schreibt die `InitFunction()` die Nummern der START Ereignisse der Prozesse in die für den Kontrollzustand vorgesehene Array-Struktur, leert die Signalwarteschlangen und setzt die Prozess Variablen auf die Anfangswerte.

Die `NextEventFunction()` und die `TransferFunction()` sind als grosse `case` Schleifen implementiert. Bei einmaligem Aufruf der `NextEventFunction()` liefert sie ein Nachfolgeereignis des als Parameter übergebenen Ereignisses. Um alle möglichen Nachfolgeereignisse zu erhalten, wird der Funktion eine Zählervariable mitgegeben. Sie wird von der Funktion automatisch erhöht, solange noch Alternativen vorhanden sind. Die `NextEventFunction()` wird solange aufgerufen, bis der mitgegebene Zähler nicht mehr inkrementiert wird. Für jedes Ereignis wird die `TestFunction()` aufgerufen. Falls diese den Wert *true* liefert, übergibt man dieses Ereignis der `FillEventFunction()` um den Parameterstring auszufüllen. Anschliessend wird das Ereignis an eine Liste angefügt. So erhält man die Liste der ausführbaren Ereignisse, wie sie von der Funktion `enabled()` gefordert wird.

5.3 Generische Klassen

In diesem Abschnitt werden Klassen behandelt, die sowohl vom Simulator als auch von Transformator verwendet werden. *Generische Klassen* sind Klassen, die ein Datenmember eines nicht festgelegten Typs enthalten. Zum Beispiel eine Liste von Objekten, wobei die Objektklasse nicht festgelegt ist. Aus der generischen Klasse können verschiedene Klassen erzeugt werden, indem jeweils die Objektklasse festgelegt wird, z.B. Signalwarteschlangen. In C++ gibt es dafür das `template` Konstrukt.

DLLE<T>. Die Klasse `DLLE<T>` ist eine generische Klasse, die verwendet wird, um lineare Listen aufzubauen. Ein Objekt der Klasse `DLLE<T>` ist ein Element einer doppelt verketteten Liste (DLLE: *Double Linked List Element*). Die linearen Listen enthalten als Memberdaten Zeiger auf Objekte der Klasse `DLLE<T>`.

List<T>. Die Template-Klasse `List<T>` wird verwendet um lineare Listen mit Elementen beliebiger Obiekttypen zu implementieren. `List<T>` definiert eine homogene Liste, d.h. alle Elemente einer Liste sind vom gleichen Typ. Die Liste verhält sich wie ein Stack.

ListOfList<T>. Die Template-Klasse `ListOfList<T>` speichert Objekte von einer beliebigen Klasse in einer Liste von Listen. `ListOfList<T>` ist realisiert mit einem Datenmember der Instanz-Klasse `List<List<T>>`.

Queue<T>. Die Template-Klasse `Queue<T>` speichert Objekte einer beliebigen Klasse in einer Liste. Im Gegensatz zur Klasse `List<T>` verhält sich die Klasse `Queue<T>` nach dem *First In First Out* Prinzip.

5.4 Klassen des Simulators

In diesem Abschnitt sind nicht nur Klassen beschrieben, die vom SDL Simulator benutzt werden, sondern auch solche, welche das SAMSTAG Werkzeug verwendet, die aber in direktem Zusammenhang mit dem SDL Simulator stehen.

Signale. Da die Signale eine unterschiedliche Anzahl von Parametern besitzen können, wird für jeden Signaltyp eine eigene Klasse erzeugt, welche die geeigneten Datenmembers besitzt, um die Signalparameter speichern zu können. Die allen Signalen gemeinsamen Elemente sind in der Klasse `SIGNAL` zusammengefasst. Die speziellen Members werden in den Klassen mit Namen `SIG<SignalName>` (z.B. `SIGICONreq`) definiert.

SIGNAL. Die Klasse `SIGNAL` ist Basisklasse aller vom Transformator generierten `SIG<SignalName>` Klassen. Sie speichert den Signalnamen, enthält aber keine Felder für Parameterwerte. Diese Klasse ist eine Basisklasse und wird nur im Zusammenhang mit der Vererbung benutzt. Ihre Funktionen zur Konsumierung oder zur Erzeugung einer Kopie eines Signals sind als virtuelle Funktionen deklariert, um den `SIG<SignalName>` Klassen die Möglichkeit zu geben, eigene Funktionen unter diesen Namen zu definieren.

SIG<SignalName>. Die `SIG<SignalName>` Klassen sind abgeleitet von der Klasse `SIGNAL`. Sie enthalten die Datenmembers für die Signalparameter. Eine virtuelle Funktion speichert bei der Konsumierung die Signalparameter in den angegebenen Prozessvariablen. Ein Funktion, welche das Signal mit allen Parametern kopiert, ist ebenfalls vorhanden.

SignalQueue. Die Klasse `SignalQueue` ist eine heterogene Queue, welche Zeiger auf Ableitungen der Klasse `SIGNAL` enthält, und implementiert eine Signalwarteschlange gemäss der Semantik von SDL. Neben den normalen Funktionen einer Queue verfügt `SignalQueue` über die Möglichkeit, ein SAVE gemäss der SDL Semantik auszuführen.

FirstSentFirstConsumedQueue. Die *FirstSentFirstConsumedQueue* implementiert die Warteschlange, welche zur Realisierung der Heuristic *FirstSentFirstConsumed* von SAMSTAG benötigt wird. Diese Heuristik versucht, die zuerst gesendeten Signal auch zuerst zu konsumieren. Sie speichert dazu von jedem gesendete Signal dessen Namen und den Empfänger-Prozess ab. Diese Datenstruktur besteht aus den Klassen FirstSentFirstConsumedQueue (`FSFCQ`) und FirstSentFirstConsumedQueueElement (`FSFCE`). `FSFCE`

definiert die Elemente der Queue, welche den Signalnamen und den Namen desjenigen Prozesses speichern, der das Signal empfängt. `FSFCQ` definiert die Queue mit den zugehörigen Funktionen.

Timer. Die Klasse `Timer` implementiert eine Stoppuhr, die bei jedem Ablesen der Zeit wieder zurückgesetzt wird. Diese Klasse wird für Geschwindigkeitsmessungen und die Kontrolle der während der Simulation verstrichenen Zeit verwendet.

Heuristic. Der Aufzähldatentyp `HEURISTIC` definiert einen Typen `HEURISTIC`, der die Werte

- `NO_HEURISTIC`,
- `FIRST_SENT_FIRST_CONSUMED`,
- `STRONG_REASONABLE_TIMER`,
- `STRONG_REASONABLE_ENVIRONMENT`,
- `NO_DOUBLE_FINAL_STATE`,
- `MSC_EVENT` und
- `NO_NULL_CONSUMPTION`

annehmen kann. Die Anordnung, so dass jeder Wert ein anderes Bit belegt, ermöglicht die Bildung beliebiger Kombinationen von Heuristiken. Es wird ebenfalls ein Array von Heuristik-Kombinationen definiert, welches spezifiziert, unter welchen Heuristik-Kombinationen der SDL Simulator auf Geschwindigkeit und Verzweigungsfaktor getestet werden soll. Zusätzlich wurde ein Ausgabeoperator definiert, welcher die Namen aller sich in einer Kombination befindlichen Heuristiken ausgibt.

CounterArray. Die Klasse `CounterArray` definiert, unter welchen Heuristik-Kombinationen der SDL Simulator getestet werden soll und zählt für jede Heuristik-Kombination die Anzahl expandierter Alternativen. Verschiedene Funktionen ermöglichen die einfache Auswertung der Resultate und das Erstellen einer Statistik.

TimeTester. Die Klasse `TimeTester` kontrolliert die Zeit während der normalen Ausführung des SAMsTAG Werkzeugs. Sie überwacht die Anzahl der expandierten Zustände, die Zeit, die gebraucht wird, um eine gewisse Anzahl (normal zehntausend) Zustände zu expandieren und den aktuellen Verzweigungsfaktor. Ebenfalls speichert sie die bisher abgelaufene Zeit, die Anzahl Alternativen, die obere Schranke für die Suchzeit und die maximale bisher erreichte Tiefe im Suchbaum. Wenn die Suchzeit abgelaufen ist, wird die öffentlich zugängliche Variable `TimeLimitExeeds` auf *true* gesetzt.

SimulatorTester. Die Klasse `SimulatorTester` testet einen erzeugten SDL Simulator unter verscheidenen Heuristik-Kombinationen auf Geschwindigkeit und Verzweigungsraten. Dies ermöglicht eine Kalkulation des Zeitaufwands und der in einer bestimmten Zeit

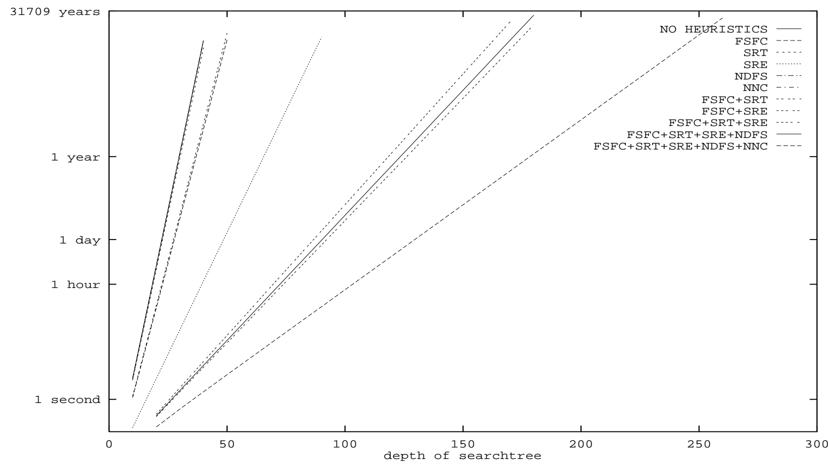

Abbildung 4: Zeit für die Suche in Abhängigkeit der Suchtiefe und der Heuristik

erreichbaren Tiefe des Suchbaums in Abhängigkeit der eingeschalteten Heuristiken. Für den Test gibt es zwei Möglichkeiten:

1. Bei der Expansion eines Knotens im Suchbaum werden für alle Heuristik-Kombinationen die Anzahl Alternativen gezählt. Diese parallele Testart ist schnell, aber sehr ungenau, da nach dem Zählen die Alternativen nach einer bestimmten Heuristik-Kombination ausgewählt werden müssen. Für alle andern Heuristik-Kombination wird das Resultat verfälscht.

2. Alle Heuristik-Kombinationen werden seriell (nacheinander) getestet. Dies bringt genauere Resultate, erfordert aber viel Zeit.

Für beide Methoden existiert eine eigene `event`-Funktion. Diese Funktion evaluiert die ausführbaren Ereignisse unter Berücksichtigung der eingeschalteten Heuristiken. Die `forward`-Funktion, die den Simulator einen Schritt vorwärts treibt, musste ebenfalls abgeändert werden. Ansonsten verwendet der `SimulatorTester` analog zum SAMSTAG

1 Sek	1 Min	1 Std	1 Tag	1 Jahr	unter Heuristik-Kombination
8	13	18	22	29	NO HEURISTICS
9	16	22	27	36	FSFC
8	13	18	22	30	SRT
15	27	39	48	65	SRE
8	13	18	22	29	NDFS
8	13	18	22	29	NNC
9	16	22	27	36	FSFC+SRT
26	50	73	92	126	FSFC+SRE
25	47	69	86	118	FSFC+SRT+SRE
26	49	72	90	123	FSFC+SRT+SRE+NDFS
35	69	103	129	178	FSFC+SRT+SRE+NDFS+NNC

Tabelle 3: Erreichbare Tiefe im Suchbaum in Abhängigkeit der Heuristiken und der Zeit

Werkzeug eine Tiefensuche. Es existieren Funktionen, welche die gewonnenen Daten so aufbereiten, dass sie mit GNU-Plot in eine Grafik weiterverarbeitet werden können (Abb. 4).

Wie tief der Suchbaum in einer gewissen Zeit ausgewertet werden kann, ist aus der LaTeX-Tabelle zu ersehen, welche durch eine andere Funktion erstellt wird. Ein Beispiel ist in Tabelle 3 zu sehen.

5.5 Abhängigkeiten der Datenstrukturen

Wie die Klassen logisch voneinander abhängen, ist aus Abbildung 5 zu ersehen.

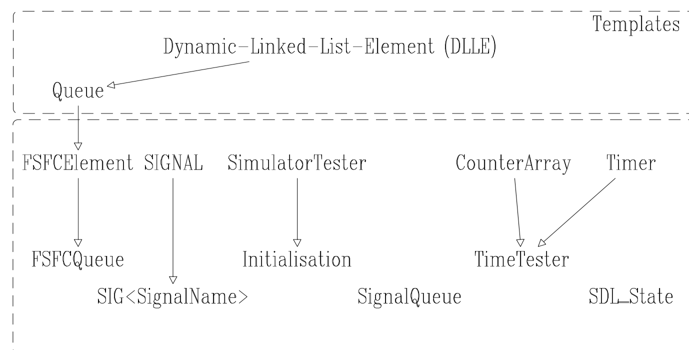

Abbildung 5: Abhängigkeiten der Datenstrukturen des Simulators

5.6 Testprogramme, lauffähige Teilprogramme

Einige Funktionen wurden zusätzlich geschrieben um den Simulator allein betreiben zu können und um gewisse Messungen durchführen zu können.

Eigenständig lauffähige Version. Um den SDL Simulator als eigenes Programm ausführen zu können, mussten Funktionen ähnlich denen des Testfallgenerators geschrieben werden. Folgende Funktionen wurden definiert:

- `main()` ist das Hauptprogramm. Diese Funktion initialisiert den Simulator und steuert die Suche so, dass der gesamte Suchbaum traversiert wird.
- `forward()` führt einen Vorwärtsschritt aus.
- `backward()` führt einen Rückwärtsschritt aus.
- `event()` wählt alle ausführbaren Ereignisse den eingeschalteten Heuristiken entsprechend aus.

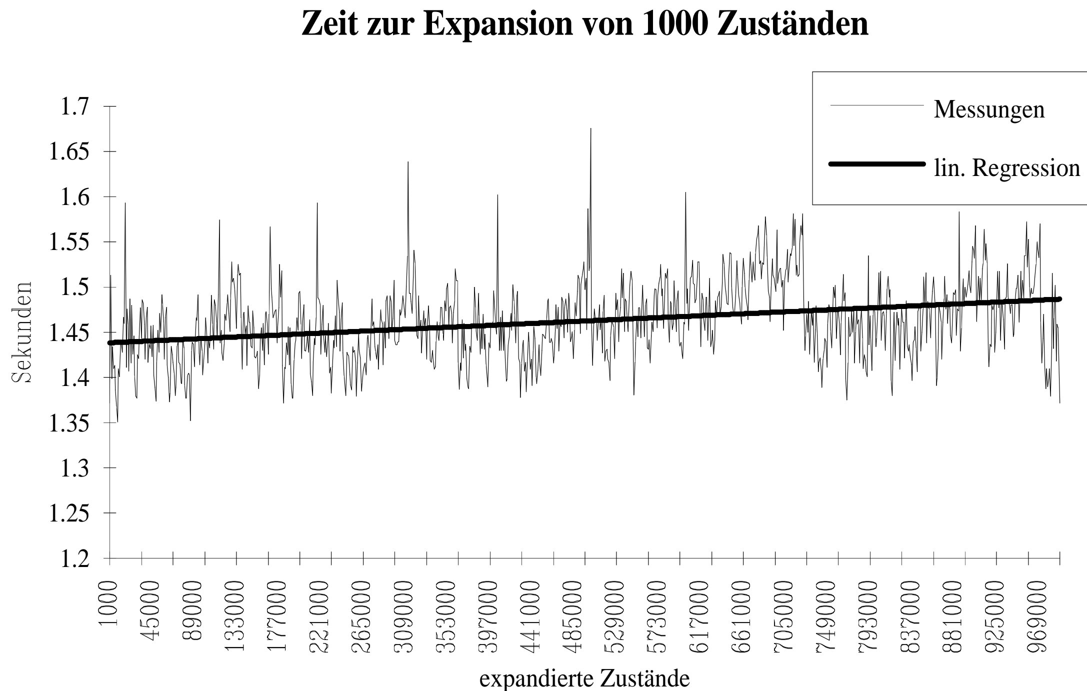

Abbildung 6: Performance Messungen

Performance-Messungen. Das selbständige SDL Simulator Programm ist mit einer Funktion ausgestattet, welche Zeitmessungen in eine Datei ausgibt, so dass diese mit dem Programm Excel von Microsoft weiterverarbeitet werden kann. Eine solche Messung ist in Abbildung 6 zu sehen.

6 Funktionalität des Transformators

Der Transformator liest eine SDL Spezifikation ein und erzeugt mit Hilfe der gewonnenen Daten verschiedene kompilierbare C++ Funktionen. Der Transformator besitzt zwei Schnittstellen:

- Die Schnittstelle zum SDL Frontend (Teil (a) in Abb. 2.a)
- Die Schnittstelle zum SDL Simulator (b)

Die Schnittstelle zum Frontend. Die Schnittstelle zum Frontend besteht in dem Einlesen von SDL/PR Dateien. Für genauere Angaben siehe Abschnitt 2.1.

Die Schnittstelle zum SDL Simulator. Die Schnittstelle zum SDL Simulator besteht in der Generierung der zentralen Funktionen (Abschnitt 4.1).

7 Realisierung des Transformators

In diesem Abschnitt soll erklärt werden, wie der Transformator implementiert ist.

7.1 Transformationsphasen

Aufgrund der gewählten Implementierung lässt sich die Transformation in die fünf Phasen

- Transformationsanfang,
- Systemtransformation,
- Blocktransformation,
- Prozesstransformation und
- Transformationsende

einteilen. Sie sind in den nachfolgenden Abschnitten beschrieben.

Transformations-Anfang. Am Anfang der Transformation müssen die Köpfe der Funktionen der Schnittstelle zum Simulator geschrieben und die verwendeten Variablen deklariert werden. Der Einfachheit halber wird jede Funktionen in eine eigene Datei geschrieben.

System-Transformation. Der erste Teil der Spezifikation, der eingelesen wird, ist die Beschreibung des Systemdiagramms. Alle später gebrauchten Informationen, wie Signale, deren Parametertypen, deklarierte Blöcke, Kanäle, etc. werden in internen Datenstrukturen abgespeichert.

Aus der folgenden Tabelle ist zu ersehen, welche Daten in welchen Datenstrukturen gespeichert werden.

Information	Datenstruktur
deklarierte Signale	SignalList (S. 18)
deklarierte Kanäle	RouteTable (S. 17)
Name des System	NameList (S. 17)
deklarierte Variablen	VariableList (S. 19)
definierte Typen	TypeList (S. 19)

Zum komfortablen Einlesen der SDL/PR Datei wird die Klasse `FileReader` (S. 21) verwendet und zum Schreiben einer Datei die Klasse `FileWriter` (S. 21).

Block-Transformation. Danach werden die Beschreibungen der Blöcke eingelesen. Hier werden zu den Informationen der Systemtransformation noch folgende zusätzliche Informationen gespeichert:

Information	Datenstruktur
deklarierte Signalwege	RouteTable (S. 17)
Verknüpfungen von Signalwegen und Kanälen	ConnectionTable (S. 17)
Namen der Blöcke und deren Position in der hierarchischen Systemstruktur	NameList (S. 17)

Anschliessend an die Blocktransformation wird mittels des Kommunikations-Struktur-Algorithmus (vgl. Abschnitt 7.4) ermittelt, welches Signal von welchem Prozess wohin gesendet wird. Die Resultate dieses Algorithmus werden in der `MessageSenderProcessConsumerProcessTable` (S. 17) gesichert.

Prozess-Transformation. Danach werden die Beschreibungen der Prozesse eingelesen. Jedem Ereignis wird eine Nummer zugeteilt. Die Ereignisse werden nun sequentiell abgearbeitet. Diejenigen, welche unabhängig von andern Ereignissen behandelt werden können, bewirken direkt einen Eintrag in den erzeugten Dateien. Die andern werden in geeigneten Datenstrukturen zwischengespeichert und am Ende der Transformation weiterbehandelt. In der Prozesstransformation werden folgende Informationen in Datenstrukturen gespeichert:

Information	Datenstruktur
SDL Zustände und deren Nachfolgeereignisse [3]	StateFollowingEventnumberList (S. 20)
SDL Zustände und deren Vorgängerereignisse [4]	StateLastPerformedEventnumberList (S. 19)
Signale und Informationen zur Konsumierung	SignalConsumingEventnumberList (S. 18)
deklarierte Variablen	VariableList (S. 19)
Outkonnektoren und deren Vorgängerereignisse	PreviousEventsofOutConnector (S. 20)
Inkonnektoren und deren Nachfolgeereignisse	FollowingEventsofInConnector (S. 21)
Namen der Prozesse und deren Position in der hierarchischen Systemstruktur	NameList (S. 17)
Zwischenspeicherung von Textzeilen in Queue Form	LineSave (S. 21)
Zwischenspeicherung von Textzeilen in Stack Form	LineList (S. 21)

Transformations-Ende. Am Schluss werden die benötigten Datenstrukturen, Funktionen und Typendefinitionen in die Dateien geschrieben und diese wieder geschlossen.

7.2 Klassen des Transformators

An dieser Stelle soll ein Überblick über die verwendeten Datenstrukturen gegeben werden. Die Datenstrukturen werden in *Tabellen* und *Listen* aufgeteilt. Die Tabellen besitzen beliebig viele Einträge mit fester Länge. Sie sind also nur in einer Dimension variabel. Die Listen besitzen auch beliebig viele Elemente. Jedes Element einer Liste ist eine Tabelle.

[1]Ein Nachfolgeereignis eines Ereignisses x ist ein Ereignis, das unter Beachtung der Kontrollstruktur direkt dem Ereignis x folgt.

[2]Ein Vorgängerereignis eines Ereignisses x ist ein Ereignis, das unter Beachtung der Kontrollstruktur direkt dem Ereignis x vorangeht.

Im Folgenden soll der Ausdruck *Objekt* ein statisches SDL Struktur-Element (wie *System*, *Block* oder *Process*) darstellen.

Die Beispiele sind einem Transformationsprozess entnommen, der mit dem *Inres* Protokoll [Hog89] durchgeführt wurde. Die gross geschriebenen Worte entsprechen Feldern der Datenstrukturen, die klein geschriebenen sind zur Erläuterung.

RouteTable. Diese Datenstruktur wird zur Speicherung der deklarierten Kanäle und Signalwege verwendet. Die RouteTable speichert, welches Signal von welchem Objekt über welchen Kanal zu welchem Objekt gesendet wird und ob das sendende oder das empfangende Objekt ein SDL Prozess ist.

Beispiel:

ICONreq gesendet von env(Ini_Station) über ISAP_Ini zu Initiator
und ein Objekt ist ein Prozess

Die RouteTable setzt sich aus den Klassen `RouteTable` und `RouteTableElement` zusammen. Die Klasse `RouteTable` ist eine Ableitung der Klasse `Queue`, instanziiert mit der Klasse `RouteTableElement`. Die Klasse `RouteTableElement` definiert die Elemente der `Queue`.

ConnectionTable. Diese Datenstruktur speichert, welche Kanäle und Signalwege durch das SDL Konstrukt `connect` verbunden sind.

Beispiel:

Kanal oder Signalweg ISAP und Signalweg ISAP_Ini sind miteinander verbunden

Die ConnectionTable setzt sich aus zwei Klassen zusammen, der Klasse `ConnectionTable` und der Klasse `ConnectionTableElement`. Die Klasse `ConnectionTable` ist eine `Queue` von `ConnectionTableElementen`. Die Klasse `ConnectionTableElement` definiert die Elemente der ConnectionTable.

MessageSenderProcessConsumerProcessTable. Diese Tabelle wird nicht durch Informationen aus der SDL Spezifikation, sondern durch die Ausgaben des Algorithmus zur Ermittlung der Kommunikationsstruktur (vgl. 7.4) ausgefüllt. Sie gibt an, zu welcher Queue ein Prozess ein Signal sendet.

Beispiel:

Signal ICONreq wird von Prozess UpperTester zu Prozess Initiator gesendet

Diese Datenstruktur sich aus den Klassen `MessageSendProcConsProcTable` und `MessageSendProcConsProcTableElement` zusammen. Die Klasse `MessageSendProcConsProcTable` ist eine `Queue` von `MessageSendProcConsProcTableElementen`. Die Klasse `MessageSendProcConsProcTableElement` definiert die Elemente der Tabelle.

NameList. Die Klasse `NameList` speichert die Namen der statischen Objekte und zu jedem eine Nummer, welche die Position in der hierarchischen Struktur repräsentiert. Falls es sich um einen Prozess handelt, wird zusätzlich die Nummer des ersten und des letzten

Ereignisses gespeichert, das von dem Prozess behandelt wird. Ebenfalls ist es notwendig zu wissen, ob es sich um einen Tester-Prozess handelt.

Beispiel:

Initator ist ein Prozess, ist dem Objekt Ini_Station untergeordnet,
behandelt die Ereignisse von Nummer 16 bis Nummer 65 und ist Kein Tester

Die `NameList` ist von der Art der Programmierung her gesehen eine Tabelle. Logisch ist sie jedoch eine zweidimensionale Datenstruktur, da sie die SDL Objekte ihrer Hierarchie gemäss baumartig einordnet.

Um sinnvoll speichern zu können, um welchen Block-Typ es sich beim zu speichernden Objekt handelt, war ein Aufzählungstyp zu definieren:

```
BLOCKTYPE := NONE | SYSTEM | BLOCK | PROCESS;
```

Die Klasse `NameList` ist als Instanziierung der Template-Klasse `Queue` mit der Klasse `NameListElement` realisiert. Die Klasse `NameListElement` definiert ein Element der Tabelle.

SignalList. Diese Datenstruktur speichert die deklarierten Signale und deren Parameter (Bsp. Abb. 7). In einer Signaldeklaration werden in SDL nur die Typen der Parameter bekanntgegeben. Deshalb erscheint im untenstehenden Beispiel kein Parametername.

Die Klasse `SignalList` ist eine Instanziierung der Template Klasse `ListOfList` mit Elementen der Klasse `SignalListElement`. Die Klasse `SignalListElement` definiert die Elemente der `SignalList` und entält Datenmembers zur Speicherung des Namens und des Typs von Signalen oder Parametern.

IDATreq	
SIGNAL	ISDUTyp

ICONreq
SIGNAL

MDATreq	
SIGNAL	MSDUTyp

Abbildung 7: SignalList

SignalConsumingEventnumberList. Die `SignalConsumingEventnumberList` enthält die Information, welches Signal von welchem Prozess in welchem Ereignis konsumiert wird (vgl. Abb. 8). Zusätzlich befinden sich hier Angaben über die Variablen, in welche die aktuellen Parameterwerte der Signale kopiert werden müssen. Die implementierte Datenstruktur ist analog zur `SignalList`.

Beipiel:

Das Signal DT wird von Prozess Codierer_Ini im Ereignis 3 konsumiert,
der Wert des ersten Parameters, der vom Typ Folgenummer ist, muss in die Variable Num kopiert werden

DT	Num	d
Codierer_Ini	Folgenummer	ISDUTyp
3		

ICONreq
Initiator
67

IDATreq	d
Initiator	ISDUTyp
38	

Abbildung 8: SignalConsumingEventnumberList

und der Wert des zweiten Parameters, der vom Typ ISDUTyp ist, in die Variable d.

VariableList. Die `VariableList` speichert die deklarierten Variablen pro Prozess ab. Im ersten Feld eines Eintrags befindet sich der Prozessname, in den weiteren Feldern je eine Variable mit dem zugehörigen Typ (Bsp. Abb. 9). Die implementierte Datenstruktur ist ebenfalls analog zur `SignalList`.

Initiator	Zaehler	d	Num	....	T
	int	ISDUTyp	Folgenummer		TIMER

Codierer_Ini	d	SDU	Num
	ISDUTyp	MSDUTyp	Folgenummer

Abbildung 9: VariableList

TypeList. In der `TypeList` werden die definierten Datentypen abgespeichert. Bei Typen wie `enum` oder `struct` werden die zugehörigen Elemente ebenfalls behalten (Bsp. Abb. 10). Die implementierte Datenstruktur ist analog zur `SignalList`.

MSDUTyp	id	Num	....	T
struct	IPDUTyp	Folgenum.		TIMER

ISDUTyp
int

IPDUTyp	CR	CC	DR	DT	AK
enum					

Abbildung 10: TypeList

StateLastPerformedEventnumberList. Diese Datenstruktur speichert zu jedem SDL Zustand die Vorgängerereignisse. Diese Information ist neben der Erstellung der

`NextEventFunktion()` dazu wichtig, um während eines Simulationslaufs feststellen zu können, ob man sich in einem SDL Zustand befindet. Ein Beispiel ist in Abb. 11 zu finden.

Die StateLastPerformedEventnumberList besteht aus den Klassen `SLPENTList` (StateLastPerformedEventnumberTableList) und `SLPENTEL` (StateLastPerformedEventnumberTableElement). Die Klasse `SLPENTList` instanziiert die Klasse `Queue` mit der Klasse `SLPENTEL`. Die Klasse `SLPENTEL` besitzt ein Element der Klasse `Queue` und definiert ein Element der StateLastPerformedEventnumberList.

Codierer_Ini/Bereit	1	5	7	10	12	14	15
Initiator/Unterbrochen	16	19	23	36	44	57	
Initiator/Warten	22	27	34				

Abbildung 11: StateLastPerformedEventnumberList

StateFollowingEventnumberList. Die StateFollowingEventnumberList speichert zu jedem SDL Zustand die Nummern derjenigen Ereignisse ab, die in diesem Zustand von der Struktur her ausführbar sind. Ein Beispiel ist in Abb. 12 zu finden.

Da die SDL Zustände nicht Teil einer Simulation sind, müssen die Vorgänger- und die Nachfolgeereignisse eines Zustands direkt miteinander verknüpft werden. Dies bedeutet, dass alle Ereignisse, die in der StateFollowingEventnumberList unter einem Zustand S (als dessen Nachfolger) verzeichnet sind, Nachfolger eines jeden Ereignisses sind, das in der StateLastPerformedEventnumberList unter dem Zustand S als dessen Vorgänger registriert ist.

Die StateFollowingEventnumberList besteht aus den zwei Klassen `SFENTList` (StateFollowingEventNumberTableList) und `SFENTEL` (StateFollowingEventNumberTableElement). Die Implementierung ist praktisch analog zur StateLastPerformedEventnumberList.

Codierer_Ini/Bereit	2	3	4	5
Initiator/Unterbrochen	17	18	19	
Initiator/Warten	24	25	26	27

Abbildung 12: StateFollowingEventnumberTableList

PreviousEventnumbersofOutConnectorList. Die PreviousEventnumbersofOutConnectorList (`PEOC`) ist das Analogon zur StateLastPerformedEventnumberList mit dem Unterschied, dass nicht die zuletzt ausgeführten Ereignisse vor einem SDL Zustand, sondern vor einem Outkonnektor abgespeichert werden. Die Implementation ist ebenfalls analog zur StateLastPerformedEventnumberList.

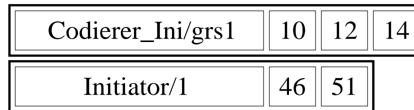

Abbildung 13: PreviousEventnumbersofOutConnectorList

FollowingEventnumbersofInConnectorList. Die Out- und die Inkonnektoren werden auf ähnliche Weise wie die SDL Zustände verknüpft. Die FollowingEventnumbersof-InConnectorList (FEIC) ist das Analogon zur StateFollowingEventnumberList mit dem Unterschied, dass nicht die unmittelbar nach einem SDL Zustand ausgeführten Ereignisse abgespeichert werden, sondern die unmittelbar nach einem Inkonnektor ausführbaren. Die Implementation ist ebenfalls analog zur StateLastPerformedEventnumberList.

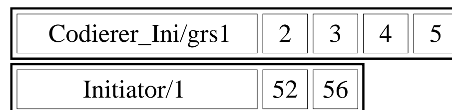

Abbildung 14: FollowingEventnumbersofInConnectorList

FileReader. Die Klasse `FileReader` ermöglicht das komfortable Abarbeiten der SDL/PR Dateien. Diese Klasse liest eine Textdatei zeilenweise in einen Buffer und stellt den Text zur Verfügung. Ebenfalls kann geprüft werden, ob sich in der aktuellen Zeile ein bestimmtes Wort befindet.

FileWriter. Die Klasse `FileWriter` hilft, die Ausgaben an eine Datei möglichst einfach zu gestalten. Sie stellt eine ungepufferte und eine gepufferte Ausgabe zur Verfügung. Somit kann eine direkte Ausgabe in eine Datei gemacht werden, es ist jedoch auch möglich, die auszugebende Zeile zuerst im Puffer aufzubereiten und sie erst danach in die Datei zu schreiben.

LineSave. Die Datenstruktur `LineSave` ermöglicht es, beliebig viele Textzeilen zwischenzuspeichern. Dies ist sehr praktisch, wenn die Transformation eines Teils der Spezifikation nicht in einem Durchlauf vollzogen werden kann. Es existiert dann die Möglichkeit, die benötigten Zeilen in die `LineSave` Datenstruktur einzulesen und diese mehrmals zu traversieren. Implementiert ist diese Struktur in den Klassen `LineSave` und `LineSaveElement`.

Die Klasse `LineSave` ist als `Queue` der Klasse `LineSaveElement` implementiert. Die Klasse `LineSaveElement` definiert ein Element der durch `LineSave` implementierten `Queue`.

LineList. Die `LineList` enthält dieselben Datenelemente wie die `LineSave` Klasse. Der einzige Unterschied besteht darin, dass sich die Datenstruktur nicht wie eine `Queue` sondern wie ein Stack verhält. Deshalb ist diese Klasse eine Instanziierung der Klasse `List`.

7.3 Abhängigkeiten der Datenstrukturen

Nachdem die verschiedenen Datenstrukturen vorgestellt sind, kann die logische Abhängigkeit dargestellt werden. Alle Klassen ausser den Klassen `FileReader`, `FileWriter` und `LineSave` sind von einer Template-Klasse abhängig. Die Klassen `StateLastPerformedEventnumberTableList` und `StateFollowingEventnumberTableList` sind einerseits von der Templateklasse `Queue` abhängig, aber auch von den `Queue` Elementen, welche selber die Klasse `Queue` instanziieren. Die genauen Abhängigkeiten sind in Abb. 15 dargestellt.

Abbildung 15: Abhängigkeiten der Datenstrukturen des Transformators

7.4 Kommunikations-Struktur-Algorithmus

Im SDL Simulator werden die Kanäle und Signalrouten nicht explizit mitsimuliert. Deshalb muss ein Prozess wissen, wenn er ein Signal aussenden muss, welchem Prozess er dieses an die Signalwarteschlange anzufügen hat. Dies ist aus der SDL Spezifikation a priori nicht zu ersehen, da sich die Definitionen der Kommunikationswege nur auf die umliegenden statischen Blöcke bezieht. Gefragt ist aber eine Tabelle mit dem Inhalt:

Signal *CR* wird von Prozess *Initiator* zu Prozess *Codierer_Ini* gesendet

Die Aufgabe ist es, aus den Informationen der `RouteTable` (S.17) und der `ConnectionTable` (S. 17) diese Tabelle zu generieren. Dies erfolgt über einen iterativen Algorithmus in fünf Schritten:

1. Suche den ersten Eintrag in der `RouteTable`, in der ein SDL Prozess als sendender Block auftritt.

2. Nun liegen Angaben über den benutzten Kanal und den empfangenden Block vor. Aus der `ConnectionTable` kann nun aus der Kanalinformation ein Nachfolge-Kanal ermittelt werden

3. Mit dem Nachfolge-Kanal aus dem vorhergehenden Schritt und der Blockinformation kann nun aus der `RouteTable` weiter ermittelt werden, zu welchem Block das Signal weitergeschickt wird.

4. Wiederhole die Schritte 2 und 3 bis der gefundenen empfangende Block ein Prozess ist und schreibe anschliessend den Eintrag in die zu erstellende Tabelle.

5. Suche den nächsten Eintrag in der `RouteTable`, in der ein SDL Prozess als sendender Block auftritt und wiederhole die Schritte 2 - 4 bis kein Eintrag mehr gefunden werden kann, der obengenanntes Kriterium erfüllt.

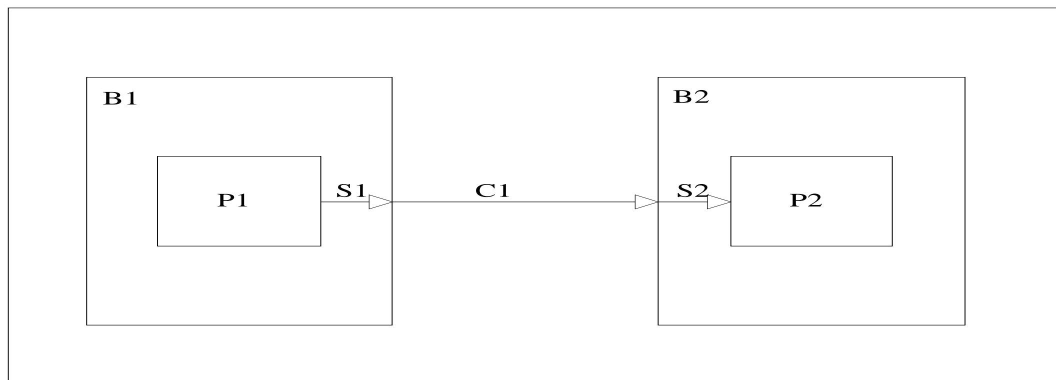

Abbildung 16: Beispiel zum Kommunikationsstruktur-Algorithmus

Als Beispiel betrachte Abbildung 16. In der `RouteTable` befinden sich folgende Angaben:

Signal	Channel	Send-Block	Rec-Block	Process
x	S1	P1	B1	true
x	C1	B1	B2	false
x	S2	B2	P2	true

Die `ConnectionTable` sieht folgendermassen aus:

Channel 1	Channel 2
C1	S1
C1	S2

Der Algorithmus sucht den Empfängerprozess, wie es in der untenstehenden Tabelle beschrieben ist.

Bekannte Angaben	aus Datenstruktur	neu ermittelte Angaben
Signal x gesendet von P1	`RouteTable`	über Kanal S1 zu B1
Kanal S1	`ConnectionTable`	verbunden mit Kanal C1
x gesendet von B1 über C1	`RouteTable`	zu B2
Kanal C1	`ConnectionTable`	verbunden mit Kanal S2
x gesendet von B2 über S2	`RouteTable`	zu P2
P2 ist Prozess		**nach P2**

8 Abschliessende Bemerkungen

Innerhalb kurzer Frist wurde ein Modul erstellt, das eine SDL Spezifikation simulieren kann und das von einem andern Modul aus steuerbar ist. Das SDL Simulationstool kann auch in einem andern Umfeld als Conformance Testing verwendet werden.

Das SDL Simulationstool erkennt ein reduziertes SDL'88. Es fehlen verschiedene SDL Konstrukte wie dynamische Prozesserzeugung oder Prozeduren. Da ein objektorientierter Ansatz gewählt wurde, ist eine Anpassung auf SDL'92 sicher in vernünftigem Rahmen realisierbar.

Die Ablaufgeschwindigkeit des Simulators könnte durch verschiedene Änderungen stark erhöht werden. Dazu müssten aber diverse Annahmen gemacht werden, z.B. eine maximale Anzahl Elemente in einer Signalwarteschlange.

Abbildungsverzeichnis

1 Architektur des SAMSTAG Werkzeugs 1
2 Architekturvarianten 5
3 Zusammenhang zwischen den Schnittstellenfunktionen 7
4 Zeit für die Suche in Abhängigkeit der Suchtiefe und der Heuristik 12
5 Abhängigkeiten der Datenstrukturen des Simulators 13
6 Performance Messungen 14
7 SignalList 18
8 SignalConsumingEventnumberList 19
9 VariableList 19
10 TypeList 19
11 StateLastPerformedEventnumberList 20
12 StateFollowingEventnumberTableList 20
13 PreviousEventnumbersofOutConnectorList 21
14 FollowingEventnumbersofInConnectorList 21
15 Abhängigkeiten der Datenstrukturen des Transformators 22
16 Beispiel zum Kommunikationsstruktur-Algorithmus 23

Literatur

[CCI92a] CCITT SG X. Message Sequence Chart (MSC). Recommendation Z.120, CCITT, 1992. Geneva.

[CCI92b] CCITT SG X. Specification and description language (SDL). Recommendation Z.100, CCITT, 1992. Geneva.

[GHN93] Jens Grabowski, Dieter Hogrefe, and Robert Nahm. A method for the generation of test cases based on SDL and MSCs. Technical Report IAM 93-010, University of Berne, Switzerland, April 1993.

[Hog89] Dieter Hogrefe. *Estelle, LOTOS und SDL - Standard Spezifikationssprachen für verteilte Systeme*. Springer Verlag, 1989.

[ISO91] ISO/IEC JTC 1/SC21. Information technology - Open Systems Interconnection - conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation. International Standard 9646-3, ISO, 1991.

[Nah94] Robert E.M. Nahm. *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts*. PhD thesis, University of Berne, Institute for Informatics and applied Mathematics, February 1994.

[Swe92] Swedish Telecom, S-123 86 Farsta. *ITEX-DE version 2.0*, 1992.

[Tel93a] Telelogic Malmö AB, S-203 12 Malmö. *SDT 2.3, Reference Manual*, 1993.

[Tel93b] Telelogic Malmö AB. SDT, the SDL design tool. In Ove Faergemand and Amardeo Sarma, editors, *6th SDL Forum*, SDL '93: Using Objects, pages 513–514. North-Holland, October 1993.

[TN93] Daniel Toggweiler and Robert Nahm. Automatic test case generation with SAMSTAG - a test suite for the Initiator process of the Inres protocol. Technical Report IAM-93-026, University of Berne, Länggassstr. 51, CH-3012 Bern, December 1993.

[Tog92] Daniel Toggweiler. TTCN-Testfallgenerierung für mit Sequence Charts spezifizierte verteilte Systeme. Diploma thesis, University of Berne, March 1992.