Projektgruppe 481

Test and Testing Control Platform

Endbericht

31.10.2006

Veranstalter
Lehrstuhl 5, Universität Dortmund

Betreuer
Prof. Dr. Bernhard Steffen
Dr. Oliver Rüthing
Dipl. Inform. Harald Raffelt

Begleiter (Fa. NOKIA)
Dr. Thomas Deiß

Das Team
Adem Altinata
Jorge Carrillo de Albornoz
Marguerite Djomkouo Simo
Edward Fondis
Alberto Garcia
Christian Holle
Alexander Kout
German Martinez
Maik Merten
Aboubakr Mkhdramine
Dominik Opolony
Murat Zabun

VORWORT

Die Projektgruppe an der Universität Dortmund ist ein wichtiger Bestandteil der Informatik. Das Ziel der Projektgruppe ist es, den Studenten zu ermöglichen, innerhalb von 2 Semestern, ein umfangreiches Projekt, in Zusammenarbeit mit anderen Studenten (8 bis 12 Teilnehmer), zu bewältigen. Die Absicht dieses großen Projektes ist, die Einübung in Teamarbeit, Projektleitung und Beschreibung komplexer Aufgaben in einem längeren Zeitrahmen.

Inspiriert wurde dieses Projekt durch die Firma Nokia und den Informatik Lehrstuhl 5 an der Universität Dortmund. Nokia stand als Kooperationspartner für die Projektgruppe zur Seite und besaß die Kundenrolle.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Beispielverzeichnis

1. Einleitung

1.1. Einführung

Die Elektronik entwickelt sich zu einem immer stärker integrierten Teil unseres alltäglichen Lebens. Sichtbar und unsichtbar verrichten große und kleine Helfer ihre Dienste; viele Bereiche, wie zum Beispiel Forschung, (moderne) Medizin, Verkehrswesen sowie Unterhaltung und private Kommunikation, kommen ohne hoch entwickelte elektronische Geräte nicht mehr aus.

Mit dem Grad, wie diese in unserem Alltag integriert werden, steigt auch die Komplexität dieser Geräte. Gleichzeitig ist eine Verschmelzung kleinerer Geräte zu einem größeren *verteilten System* zu beobachten. Der Gedanke der Wiederverwendung der Komponenten und einheitlicher Schnittstellen und Kommunikationsstandards wie z.. Ethernet, FireWire oder Bluetooth ermöglichen und beschleunigen diese Entwicklung. Spekulativ sei an dieser Stelle auch das „Mooresche Gesetz“ zu erwähnen, das eine exponentielle Steigerung der Integrationsdichte voraussagt, eine Beobachtung die seit deren Entdeckung vor über vierzig Jahren nicht an Gültigkeit verloren hat [22]. Gleichsam steigen die Kundenanforderungen an die Integrität, Verfügbarkeit, Zuverlässigkeit, Vertraulichkeit und Verlässlichkeit dieser Systeme.

Der Einsatz der Mikrosystemtechnik steigert zum einen die Hardwarekomplexität und koppelt andererseits ihre Entwicklung an die der Softwaretechnologie: Die Funktionalität und Komplexität von Softwaresysteme steigt trotz Bereitstellung von standardisierten Schnittstellentechnologien wie COM, SOAP oder CORBA. Diese lassen sich zu einem unüberblickbaren Ganzen zusammenschließen. Auch bestehen moderne Betriebssysteme aus unzähligen Komponenten, die miteinander interagieren, sich beeinflussen und – fehlerhaft entwickelt – sich sogar untereinander stören können. Dadurch, dass diese Systeme immer neue alltägliche Anwendungen (wie z.. Mobilfunkgeräte, DVD-Recorder, Internet fürs Wohnzimmer und Küche) oder sicherheitsrelevante Geräte (wie z.. Verkehrsampeln, Bahnschranken) ansteuern oder Dienste den anderen Softwaresysteme anbieten, unterliegen auch Softwaresysteme den gleichen Anforderungen wie Hardwaresysteme.

Dabei müssen sowohl einzelne Komponenten diese Anforderungen für sich allein korrekt (d.. ihrer Spezifikation entsprechend) erfüllen als auch verbunden zu einem Gesamtsystem. Diese Korrektheit lässt sich durch verschiedene Methoden feststellen, beispielsweise durch einen mathematischen Beweis, Verifikation der Schaltung oder Quellcode durch Dritte oder auch durch Testverfahren. Steigt die Komplexität der Komponente oder des Gesamtsystem an, wird die Beweis- oder Verifikationsmethode schwer oder im Rahmen der gegebenen Ressourcen nicht mehr realisierbar. Oftmals werden einzelne Komponenten auch von verschiedenen Herstellern geliefert und die Einsicht in den inneren Aufbau oder Quellcode – technisch oder rechtlich bedingt – kaum mehr möglich. Es verbleibt in diesem Fall nur noch die Testmöglichkeit, wobei ohne Aufbaueinsicht von einem *Blackbox-Test* gesprochen wird.

Während Beweis- und Verifikationsmethoden tatsächlich die Korrektheit von Komponenten und des Gesamtsystems nachzuweisen versuchen, erlaubt es nur der Test die festgelegten Aufwandsgrenzen einzuhalten, wobei der Vollständigkeitsfaktor entfällt und insbesondere beim Blackboxtest durch das Wesen der Sache beschränkt wird: Ein Test besteht darin, dass das Testobjekt mit ausgewählten Eingaben konfrontiert wird (mit oder ohne die Kontrolle der inneren

Abläufe) und die tatsächliche Ausgaben bzw. sein Verhalten mit den erwarteten Ausgaben verglichen werden. Der Test ist abgeschlossen, wenn sich das Testobjekt auf der ausgewählten Eingabemenge korrekt verhält.

Mit der steigenden Zustandsmenge eines Systems steigt die Menge der möglichen Eingaben für das System exponentiell an. Bereits kleinere Systeme können erhebliche Zustandszahlen aufweisen und deren Eingabelänge ausufern lassen. Dies impliziert, dass die vollständige Eingabemenge bei testrelevanten Objekten kaum getestet werden kann. Ein Test ist daher immer ein Stichprobenverfahren: Ein Test kann nicht nachweisen, dass das getestete Objekt keine Fehler mehr enthält, sondern nur, dass bestimmte Fehler nicht auftreten: "Testing can only reveal the presence of errors, never their absence."(Dijkstra, [11])

Die Hauptschwierigkeit beim Testen liegt deshalb darin, mögliche Fehler anhand des Aufbaus bzw. der Spezifikation des Testobjektes zu ermitteln, möglichst effiziente Testfälle zu erstellen, die auf möglichst alle diese Fehler testen, für die Testfälle erforderliche Ressourcen einzuschätzen und geeignete Werkzeuge auszuwählen. Ist das Testobjekt zudem als Blackbox vorhanden (keine Aufbaueinsicht möglich) und somit seine Zustandsmenge verborgen, so kann sich die Testfallauswahl nur noch nach der Spezifikation des Testobjektes richten.

Ein Test läuft in mehreren Phasen ab [10]: *Testplanung*, *Testspezifikation*, *Testdurchführung*, *Testprotokollierung*, *Testauswertung* und *Test-Ende*. Bei der Testplanung wird das Testproblem analysiert, Ressourcen eingeschätzt und Werkzeuge ausgewählt. Bei der Testspezifikation werden Testfälle spezifiziert. Ein Testfall definiert Testdaten, Soll-Verhalten des Testobjektes und Vorbedingungen, die eingehalten werden müssen, damit das Sollverhalten wahr ist. Bei der Testdurchführung wird die Testumgebung aufgebaut, die Testprozedur erstellt und mit hergestellten Vorbedingungen ausgeführt. Bei der Testauswertung wird das Ist-Verhalten mit dem Soll-Verhalten verglichen. Das Testprotokoll ist erforderlich, damit der Testvorgang z.. durch den Kunden nachvollziehbar ist. Hierfür soll es diesen vollständig festhalten. Testendkriterien definieren Metriken, die den Testfortschritt zu messen erlauben.

Diese Teilaufgaben durch menschliche Hand auszuführen ist äußerst ineffizient, da es meist monotone Tätigkeiten sind, die mit der menschlichen Geschwindigkeit zu langsam sind und bei denen selbst Fehler unterlaufen können. Jedoch ist eine vollständige Übertragung dieser Aufgaben auf Software-Tools nicht möglich. Es existieren mehrere Tools, welche Teilaufgaben übernehmen können: Es existieren Assistenten für die Testplanung und Spezifikation, Protokollierungswerkzeuge, sowie Werkzeuge zum Kontrollieren der Testendkriterien. Zwei solcher Tools werden intensiv von der Projektgruppe benutzt (JUnit, Abschnitt 3.4) und eingesetzt (jABC, Abschnitt 3.6).

JUnit ist ein Software-Testwerkzeug, welches einem Programmierer in den Phasen Testspezifikation, Testprotokollierung und Testauswertung zur Hand gehen kann und für Regressionstest besonders geeignet ist. Mit JUnit lassen sich ein oder mehrere Testfälle einer Klasse programmieren (spezifizieren). Nach der Ausführung der Testfälle werden Testergebnisse mit Sollwerten verglichen und ein Testurteil gebildet. Nach jeder Weiterentwicklung der Klasse lässt sich praktisch per Knopfdruck ermitteln, ob sie noch ihrer durch Testfälle beachteten Spezifikation genügt oder ein Programmierfehler unterlaufen ist. Die JUnit-Testfälle werden in einer üblichen Programmiersprache erstellt (kein Lernaufwand erforderlich,) sie lassen sich mit der programmierten Klasse weitergeben und schildern sehr gut, was und wie getestet wurde.

JUnit erlaubt es dem Programmierer selbst, die Testfälle zu schreiben und sich dabei auf die Klassenspezifikation zu konzentrieren. Dies entschärft teilweise das Problem, dass Programmierer, die ihre Klassen testen müssen, keine Testerfahrung besitzen und nicht ausreichend getestete Arbeit liefern [10, s. 2]. Insbesondere für Eclipse existiert eine JUnit-Unterstützung, die das Erstellen automatischer Testsuiten einfach ermöglicht, das Testurteil graphisch darstellt

und Testfälle kennzeichnet, deren Testergebnisse von spezifizierten Sollergebnissen abweichen.

jABC [4] ist eine Java-Portierung und Weiterentwicklung des am Lehrstuhl 5 entwickelten ABC Frameworks. ABC steht dabei für **A**pplication **B**uilding **C**enters und erlaubt es, ein Projekt grafisch mit einem Baukastenprinzip zu modellieren. Dies hat mehrere Vorteile: Das Modell kann von einem Kunden erstellt werden, es reduziert Missverständnisse, die einige Entwicklungszeit und -kosten (wenn der Kunde seine Anforderungen nach Einsicht des Zwischenergebnis korrigiert). Dieses virtuelle Modell kann bereits auf Design- oder Spezifikationsfehler getestet werden, was wiederum Entwicklungszeit einspart, da das Modell leichter korrigiert werden kann. Die Umsetzung des Modells kann weiterhin vereinfacht werden, da viele Bauteile als fertige Funktionen oder Klassen in einer Bibliothek zur Verfügung stehen können und lediglich gelinkt werden.

Es gibt auch ein weiteres Werkzeug, das die Hauptrolle spielen wird: *TTCN-3*.

TTCN ist eine Notationssprache (*Testing and Test Control Notation*, siehe 2.1), die gleichzeitig als ein Werkzeug fungiert und auf der Testdurchführungsphase aufsetzt. Weiterhin dient TTCN als eine Trennschicht zwischen dem zu testenden System und dem Testobjekt. Inzwischen ist TTCN in der dritten Version verfügbar, die als TTCN-3 bezeichnet ist. Fortan wird nur diese behandelt.

TTCN-3 ist für Blackboxtests konzipiert. Das setzt ein funktionierendes System mit einer bekannten Spezifikation voraus, welches zu testen ist (*System unter Test*) und hierbei ein beliebiges Objekt sein kann – von einfachen Softwarekomponenten bis hin zu großen Hardwareanlagen – und über ein Interface (*Systeminterface*) von einem System, das die Tests ausführt (*Testsystem*), angesprochen werden kann. Dabei wird das System unter Test als gegeben betrachtet und in den TTCN-3 Testfällen nicht spezifiziert (nur die Schnittstelle des Systeminterface, Abschnitt siehe 2.1). Das Systeminterface trennt somit das testende und das System unter Test vollständig voneinander. Das Testsystem baut selbständig Verbindungen zum Systeminterface auf und ab und zwar vor, nach sowie dynamisch während des Tests, übergibt Testdaten an das Systeminterface, empfängt Antworten, wertet sie aus und entscheidet über den weiteren Testablauf. Dieser kann protokolliert werden, da eine Testlogging-Komponente ein Bestandteil von TTCN-3 ist. Sowohl Testsystem als auch das System unter Test können verteilt realisiert werden.

Betrachtet man TTCN-3 nun als ein Tool, so unterstützt es automatische Tests in fünf Phasen: Testspezifikation, Testdurchführung, Testprotokollierung, Testauswertung und Testende. TTCN-3 liegt jedoch lediglich als Spezifikation, die in einer gedruckten oder elektronischen Form verfügbar ist, vor [3], [6]. TTCN-3 ist also eine Notation. Tatsächlich hat sich TTCN-3 funktionell zu einer Programmiersprache entwickelt, in der der gesamte Testablauf spezifiziert bzw. programmiert wird.

Die Programmiersprache TTCN-3 zeigt objektorientierte Ansätze. So wird der Test durch Testkomponenten durchgeführt, die Instanzen von in TTCN-3 spezifizierten Klassen sind, und die Kommunikationskanäle (Verbindungen über definierte Ports) öffnen, um darüber Nachrichten zu versenden. Zum Aufbau, Versenden, Empfang, Speichern und Auswerten dieser Nachrichten stellt TTCN-3 mächtige Sprachkonstrukte bereit. Dabei wird der Test durch eine ausgezeichnete Komponente gesteuert, alle andere Komponenten sind gleichberechtigt. Dadurch, dass andere Komponenten das Systeminterface wie eine normale Komponente „sehen“ [1], bleibt das Nachrichtennetz der spezifizierten Tests zu jeder Zeit konsistent. Jede Komponente bildet eine abgeschlossene Einheit. Der Zugriff oder Datenaustausch zwischen Komponenten ist nur über Ports und Nachrichten möglich. Damit der Test auch bei fehlerhaften oder nicht zustande gekommenen Verbindungen noch terminieren kann, sorgen Timer für das Einhalten der

[1] In Wirklichkeit darf das Systeminterface keine aktive TTCN-3 Sprachelemente besitzen.

Timeouts und Testabbrüche bei Nichterhalt erwarteter Nachrichten. Die Verarbeitung von Timerereignissen ist nahtlos ins Nachrichtenverarbeitungskonzept integriert.

Aufgrund seiner Mächtigkeit und der strikten Trennung von Testsystem und des Systems unter Test, eignet sich TTCN-3 hervorragend zum Überprüfen, ob das zu testende System seine gestellten Spezifikationen einhält (das ursprüngliche Entwicklungsziel von TTCN). Wie bereits angesprochen ist TTCN-3 jedoch kein Tool an sich, sondern eine Programmiersprache, die von einem ausführenden System erkannt werden muss. Dieses übernimmt die in TTCN-3 spezifizierten (programmierten) Testfälle und führt den Code aus. Dabei sorgt dieses System für die korrekte Ausführung des TTCN-3 Codes, inklusive der Verbindungen zum Systeminterface, das auch eine Hardwareschnittstelle sein kann (Bluetooth beispielsweise) und für den Datenaustausch zwischen den in Software realisierten Testkomponenten untereinander und mit dem Systeminterface. Durch den dynamischen Verbindungsaufbau ist TTCN-3 sehr flexibel und ist in der Lage, umfangreiche Testfälle zu realisieren.

Obwohl TTCN-3 einen großen praktischen Nutzen hat, existieren kaum mehr offene TTCN-3 Projekte auf dem Markt. Die Sprache selbst ist frei verfügbar und wird aktiv weiterentwickelt, jedoch benötigt sie, wie bereits angesprochen, ein Softwaresystem, das diese Sprache erkennt, das Testsystem realisiert und in Verbindung mit dem System unter Test dieses ausführt. Zwar setzen verschiedene Hersteller TTCN-3 für Tests ihrer Produkte ein, doch existieren nur wenige Anbieter von TTCN3-Tools. Eine Firma, die ein TTCN-3-Tool (TT-Suite) vertreibt, ist die Firma TestingTech, die für die Lehre eine Halbjahreslizenz für ein TTCN-3-Plugin für Eclipse bereitstellt [7]. Kommerziell entwickelt jeder Hersteller sein Tool eigenständig, was eine standardisierte Entwicklung der Sprache nicht gerade fördert. Ein öffentlich entwickeltes TTCN-3 Tool könnte sowohl auf korrekte Umsetzung der Sprache hin überprüft werden, als auch die korrekte Sprachumsetzung der hauseigener Herstellerentwicklungen verifizieren. Ferner lässt sich auch die TTCN-3 Sprache einfacher erlernen, wenn ein solches Tool frei vorliegt.

Eine sinnvolle praktische Anwendung dafür ist Bluetooth (3.7). Ein TTCN3-Tool müsste es erlauben, Testfälle schnell zu entwickeln, wenn notwendig schnell zu ändern und effizient auszuführen. Die Testfälle müssen übersichtlich und nicht nur für den Produktentwickler nachvollziehbar sein, sondern auch für den Ingenieur, der die Weiterentwicklung übernommen hat, sowie für den Endkunden, der wissen muss, wie und worauf das Produkt getestet wird.

Die Entwicklung eines solchen Tools und ein Nachweis seiner Praxistauglichkeit anhand von Bluetoothgeräten sind die Ziele der Projektgruppe „**T**est and **T**esting **C**ontrol **P**lattform“, kurz **TTCP**.

1.2. Projektgruppe TTCP

TTCN-3 ist eine umfangreiche Sprache zur Spezifikation und Ausführung von Testfällen, was sowohl Vorteile als auch Nachteile mit sich bringt. Der wesentliche Vorteil besteht darin, dass der Programmierer die von ihm spezifizierten Tests direkt zur Ausführung bringen und hierbei von TTCN3-Tools unterstützt werden kann. Zu den Nachteilen der Testsprache TTCN-3 gehört die Mächtigkeit dieser Sprache. Sie ist schwer zu erlernen, sowohl für den Testingenieur, als auch für den Endkunden, der den Testvorgang nachvollziehen will.
Das Ziel dieser Projektgruppe ist es, eine Testplattform (Test and Testing Control Platform (TTCP)) zu erstellen, die es dem Ingenieur erlaubt, auf eine komfortable Art und Weise, Tests seiner Entwicklung zu spezifizieren und durchzuführen. Hierbei soll die Testsprache TTCN-3 in eine graphische Entwicklungs-, Koordinations- und Ausführungs-Plattform integriert werden. Dieses System soll zur systematischen, industrierelevanten Testfall-Erstellung und deren Vali-

dierung und Evaluierung eingesetzt werden können, mit Anwendungsschwerpunkt im Bereich der Mobilfunktechnologie, insbesondere Bluetooth.

Dafür soll die Projektgruppe auf das vom Lehrstuhl 5 entwickelten Framework jABC (**J**ava **A**pplication **B**uilding **C**enter) zurückgreifen. Mit Hilfe von jABC stellt der Ingenieur einen Testfall graphisch in einer Diagrammform zusammen. Anschließend wird die Darstellung, bei Zufriedenheit des Ingenieurs, in einem entsprechenden TTCN-3 Code übersetzt und dieser durch eine TTCN-3 Ausführungsplattform ausgeführt. Zusätzlich können die aus der graphischen Darstellung oder bereits verfügbare, im Texteditor erstellten Testfälle, ausgeführt werden.

Mit jABC modellierte Testfälle können noch in der Modellform verifiziert werden, aber auch die Testausführung kann auf die graphische Darstellung zurückgreifen und den Testverlauf 'live' wiedergeben. Ist die Darstellung zudem selbsterklärend, kann sie auch der Endkunde lesen und den Test nachvollziehen.

1.3. Motivation

Die Komplexität moderner informations-technologischer Systeme nimmt immer mehr zu. Um den ökonomischen Anforderungen nach kurzen Test- und Entwicklungszeiten, sowie den hohen Qualitätsansprüchen der Anwender gerecht zu werden, setzt die Industrie auf breiter Front interoperable Testspezifikations-Sprachen, wie beispielsweise im Telekommunikations-Bereich TTCN-3 (Test and Testing Control Notation 3) ein [3].

Bei dieser Vorgehensweise setzt der Testingenieur seine subjektive Vorstellung von dem geforderten Verhalten des Systems in konkrete Steuerungsanweisungen und Testfälle für das System um. Aktuelle, in der Wissenschaft etablierte, formale Methoden zur Testerzeugung und Validierung, wie z.. modellbasiertes Testen [19] oder Modell Checking [17], werden im produktiven Umfeld nur selten eingesetzt.

Die Ergebnisse der Projektgruppe sollen anhand einer Bluetooth Anwendung evaluiert werden. Das Projekt findet in enger Zusammenarbeit mit der Firma NOKIA statt. Es ist vorgesehen, dass Mitarbeiter von NOKIA die Kundenrolle einnehmen, wobei Kundenanforderungen laufend mit dem Projektstatus abgeglichen werden. Insbesondere soll am Ende des Projektes die Endabnahme des Produktes stehen.

1.4. Ziel

Die Projektgruppe erstellt eine integrierte Umgebung, mit der sich Testspezifikationen in TTCN-3 modellieren, validieren und ausführen lassen. Im Einzelnen sind folgende Teilaufgaben vorgesehen:

1. Erstellung einer *Entwicklungsumgebung*: Es wird eine graphische Entwicklungsumgebung erstellt, mit der sich einzelne Testfälle in Form von Flussgraphen darstellen lassen. Einzelne TTCN-3-Komponenten und insbesondere bereits mit TTCN-3 entwickelte System-Under-Test Adapter werden dem Test-Ingenieur in Form eines Baukastensystems zur Verfügung gestellt. Zur Erstellung der Entwicklungsumgebung bietet sich die am Lehrstuhl entwickelte jABC Plattform (Java Application Building Center) [20, 1] an. Diese Plattform stellt bereits einen Großteil der für visuelle Software-Entwicklung nach dem Baukastenprinzip notwendige Funktionalität bereit, wie beispielsweise eine graphische, Java-basierte Benutzeroberfläche.

2. Erzeugung von TTCN-3-*Testfällen*: Mit Hilfe der Entwicklungsumgebung lassen sich die modellierten Test-Graphen in TTCN-3 Code übersetzen, der mit herkömmlichen TTCN-3 Werkzeugen weiterverarbeitet werden kann.

3. Synthese von *Test-Fluss-Graphen*: Mit Hilfe der Entwicklungsumgebung lassen sich bestehenden TTCN-3 Spezifikationen analysieren und in den graphischen Editor importieren. Dieses Teilziel umfasst die Entwicklung eines Parsers für TTCN-3 Spezifikationen, der zur Realisierung des Teilziels 5 benötigt wird.

4. *Verifikation* von Testfällen: Mit Hilfe von Modell-Checking lassen sich temporal-logische Konsistenzbedingungen von Testfällen überprüfen. Auch für dieses Teilziel bietet sich die Verwendung des jABC an, da bereits ein umfangreicher Modell-Checker integriert ist. Die Entwicklungsumgebung hat aber zusätzlich dafür Sorge zu tragen, dass die entsprechenden Konsistenzbedingungen geeignet erfasst und verwaltet werden können.

5. *Interpreter*: Für eine geeignet auszuwählende Teilsprache von TTCN-3 soll eine Umgebung für die interaktive Ausführung von Testfällen entwickelt werden. Technisch geschieht dieses, indem eine virtuelle Maschine realisiert wird, auf der TTCN-3 Anweisungen interpretiert werden. Für die Anbindung an das System-Under-Test kann dabei das standardisierte TTCN-3 Interface (TRI) verwendet werden. Der Interpreter soll den Benutzer zusätzlich leistungsfähige Hilfsmittel, z.. im Stile konventioneller Tracer oder Debugger, zur Verfügung stellen.

6. *Reale Anwendung (Bluetooth)*: Es ist geplant, dass das entwickelte Werkzeug letztendlich in der Lage ist, bereits existierende TTCN-3 Testfälle für das Serial Port Profile von Bluetooth auszuführen. Gegenstand dieser Tests ist das Aufsetzen von seriellen Verbindungen über Bluetooth und der Austausch von Daten über die eingerichtete Verbindung. Diese Testfälle kommen aus der industriellen Praxis. Eine Anbindung von TTCN-3 basierten Testsystemen an übliche Bluetooth USB Dongles unter Linux existiert bereits.

1.5. Organisation und zeitlicher Ablauf

1.5.1. Ablauf im 1. Semester

1.5.1.1. Seminare

Anfangs bekommt jeder Teilnehmer der Projektgruppe eine Seminararbeit. Diese besteht aus relevanten Themen für das Projekt. Jeder Teilnehmer präsentiert nach einer 15 bis 20 seitigen Ausarbeitung diese auf einem Seminar. Dadurch bekommt jeder Teilnehmer der Gruppe einen allgemeinen Überblick zu den Themen TTCN-3, Testen und Testmethoden, Bluetooth und über die Tools, die während des Projekts benutzt werden, wie z.. jABC und JUnit.

1.5.1.2. Infrastruktur

Damit das Projekt ein Erfolg wird, wird zunächst eine Infrastruktur gebildet. Diese Infrastruktur soll jedem Mitglied der Projektgruppe ermöglichen, auf dem Aktuellen Stand zu bleiben, daher wird eine Wiki (PmWiki) auf einem Server aufgesetzt. Auf dieser Wiki wird die Gruppe vorgestellt und die gebildeten Arbeitsgruppen. Mit Hilfe dieser Seite hat der Projekt-Manager, welcher sich jeden Monat ändert, die Möglichkeit zu wissen, wie der aktuelle Stand der Arbeitsgruppen ist. Dazu werden die Wochenberichte jeder Gruppe in das Wiki gestellt, sowie

die Protokolle der Wochensitzungen. Die Gruppe trifft sich mehrmals in der Woche, um über bestimmte Entwicklungsphasen zu diskutieren. Der Kontakt mit den PG-Betreuern wird immer gut gepflegt.

1.5.1.3. Teambildung

Da das Projekt im wesentlichen darin besteht, einen Compiler zu bauen, wird die Projektgruppe in 3 Gruppen mit jeweils 3-5 Personen pro Gruppe geteilt. Die erste Gruppe soll einen Scanner und Parser entwickeln, der den Quellcode auf syntaktische Fehler kontrolliert und eine Abstraktion des Quellcodes liefert, damit dieser zur weiteren Verarbeitung geeignet ist. Die zweite Gruppe soll den Semantikcheck durchführen. Zum Aufgabenspektrum der dritten Gruppe zählt, einen Compiler zu implementieren, der den syntaktisch und semantisch korrekten Code in die gewünschte Sprache (hier Java) umsetzt. Die Herausforderung ist das Laufzeitsystem, das den eigentlichen Test ausführen, kontrollieren und protokollieren soll.

1.5.1.4. Erste Implementierungsphase

Implementiert wird in jeder Gruppe. Die Parser Gruppe ist damit beschäftigt, neben der Implementierung des Parsers Testfälle zu generieren. Mit diesen sucht sie nach Fehlern im Parser. Zum Ende des 1. Semesters steht der Parser, ohne bekannte Fehler, für die Compiler Gruppe bereit.
Die Gruppe testet mittels Testfällen den Code Stück für Stück. Zum Ende des 1. Semesters ist der Semantikchecker schon 50% fertig. Die Compiler Gruppe muss sich zunächst mit TTCN-3 auseinander setzen. Ihre Aufgabe ist die Core Language und die einzelnen Komponenten, wie z.. Port, Timer und ALT-Statement zu verstehen. Anschließend ist sie auch mit der Implementierung beschäftigt, so dass am Ende des 1. Semesters die kleine Core Language fertig entwickelt ist.

1.5.1.5. Zwischenbericht

Der Zwischenbericht besteht aus 6 Kapiteln und dokumentiert den Stand des Projekts nach dem 1. Semester. Am Anfang wird die Projektgruppe vorgestellt, die Themen, die für das Projekt relevant sind. Schlussendlich wird die Entwicklungsphase des Projektes dargestellt.

1.5.1.6. Zwischenpräsentation

Die Präsentation wird im Lehrstuhl 5 vorgetragen. Für die Präsentation wird Herr Thomas Deiß der Firma Nokia von dem Lehrstuhl 5 eingeladen. Jede Gruppe trägt ihren Teil vor und diese wird von einem Moderator koordiniert. Die Präsentation endet mit einer Diskussion.

1.5.2. Ablauf im 2. Semester

1.5.2.1. Teambildung

Zum Anfang des 2. Semesters wird die Projektgruppe zum Teil neu strukturiert, da die Parser Gruppe ihren Aufgabenteil beendet hat. Zum einen kommt die Testfallgenarator Gruppe hinzu, die für die automatische Generierung der Testfälle zuständig ist. Diese Testfälle sollen anschließend dazu dienen, Fehler vom Compiler zu entdecken. Des weiteren kommt noch die Endbericht Gruppe hinzu, die für die Erstellung des Endberichtes zuständig ist.

1.5.2.2. Endpräsentation

Die Abschlusspräsentation findet auf dem PG-Tag des Lehrstuhl 5 am 6. Oktober 2006 statt.

1.6. Team

Veranstalter

- Lehrstuhl 5, Universität Dortmund [8]
- Firma NOKIA -Research-Center, Bochum [9]

Betreuer

- Prof. Dr. Bernhard Steffen
- Dr. Oliver Rüthing
- Dipl. Inform. Harald Raffelt

Mitglieder

- Adem Altinata
- Jorge Carrillo de Albornoz
- Marguerite Djomkouo Simo
- Edward Fondis
- Alberto Garcia
- Christian Holle
- Alexander Kout
- German Martinez
- Maik Merten
- Aboubakr Mkhdramine
- Dominik Opolony
- Murat Zabun

Begleiter (Vertretung der Firma **NOKIA**)

- Dr. Thomas Deiß

1.7. Überblick der Kapitel

Dieser Abschnitt gibt einen kurzen Überblick über die einzelnen Kapitel des Endberichtes. Im ersten Kapitel *Kapitel 1* wird eine kurze Einleitung in die Problemstellung des Projekts dargestellt. Das *Kapitel 2* bietet einen Blick auf die *Testsprache TTCN-3*, mit der sich das Projekt hauptsächlich beschäftigt. Weiterhin wird in diesem Kapitel etwas Wissenswertes über den Kunden *Nokia* dargestellt.
Im *Kapitel 3* werden Software und Werkzeuge beschrieben, die während der Entwicklung des Projektes eingesetzt worden sind.
Die Entwicklungsphase, insbesondere die Unterteilung in Teilgruppen, wird im *Kapitel 4* dargestellt.
Das darauf folgende *5 Kapitel* gibt zunächst Informationen über die Systemarchitektur und beschreibt anschließend deren Schnittstellen und die Implementierung des Gesamtsystems. In diesem Kapitel wird des Weiteren die automatische Testfallgenerierung erläutert.
Das *6 Kapitel* fasst das Ergebnis des Projektes zusammen und gibt die Ziele an, die die Projektgruppe erreichen soll. Darüber hinaus wird hier auf die Weiterentwicklung des entwickelten Tools TTCP eingegangen.

2. Grundlagen

2.1. TTCN-3

2.1.1. Entstehungsgeschichte

TTCN ist eine Sprache, die weltweit in der Testtechnologie eingesetzt wird. Die Sprache wurde 1992 zum ersten Mal veröffentlicht. TTCN steht zu dieser Zeit für Tree and Tabular Combined Notation und wurde damals, wie auch jetzt, für Tests von Systemen benutzt. Dieses Testsystem wird für die Spezifikationen unterschiedlicher Technologien, wie GSM (Global System for Mobile Communication, DECT (Digital Enhanced Cordless Technologie), INAP (Inteligent Network Application Protocol) oder N-ISDN, -ISDN (Integrated Service Digital Network) eingesetzt.

1996 wurde eine neue Version TTCN-2 herausgebracht, welche von ISO und ITU standardisiert wurde. Zeitgleich wurde TTCN-2++ von ETSI (European Telecommunications Standards Institute) standardisiert, mit den zusätzlichen Funktionen, wie die Adressierung von Testmodulen, parallelen Testkonfigurationen und die Nutzung von ASN-1.

1998 wird von den Mitgliedern der ETSI erwartet, eine neue Version von TTCN zu entwickeln. Die neue Version soll den komplexeren Einsatz in der Telekommunikation ermöglichen. Im Jahr 2000 wird TTCN-3 als neue Version von TTCN-2 vorgestellt, welche auch in den Bereichen der Mobiltechnologien einsetzbar ist, Abbildung 2.1.

2.1.2. Entwicklung

TTCN-3 steht für Testing and Test Control Notation und ist eine Testsprache zur Beschreibung von ausführbaren Prüfvorschriften (Spezifikationen). Die Beschreibung von Testverhalten und Testverfahren für das allgemeine Verständnis, und die Entwicklung einer anerkannten Spezifikations- und Implemetierungssyntax ist für die Testtechnologie zwingend notwendig. Mit der Entwicklung von TTCN-3, einer einheitlichen Testnotation, wird dieses erreicht. Diese Entwicklung wird durch die Industrie und Wissenschaft gefördert.
Die Neuerungen in TTCN-3 ermöglichen den Gebrauch einheitlicher Methoden und Verfahren, die dann zu einer einfachen Wartung der Testsuiten und Testprodukte führt.
Die Testsuiten können die Tester, mit Hilfe von TTCN-3, auf abstraktem Niveau spezifizieren und dadurch mehr Augenmerk auf den eigentlichen Testzweck, die Testlogik, werfen. Außerdem erreicht man mit einer standardisierten Sprache, dass sich der Tester bei der Implementierung keine Gedanken machen muss, wie er seine Testsuite an das System anpasst. Die Tester brauchen also keine andere Testsprache zu erlernen. Eine standardisierte Sprache hat einheitliche Testsuites. Es ergibt sich dadurch eine Menge von Unterlagen, Beispielen und fertigen Testsuites, die bei der Ausbildung und dem Training nützlich sein können. Die Wartung und die Weiterentwicklung der Sprache TTCN-3 ist durch den ständigen Gebrauch gewährleistet.
Zu den Möglichkeiten der TTCN-3 Sprache ist folgendes nennenswert:
Mit der TTCN-3 Sprache ist es möglich Testeingaben und -ausgaben, einfache und komplexe Testverhalten von Sequenzen, Alternativen und Schleifen, zu beschreiben. Anwendungsbe-

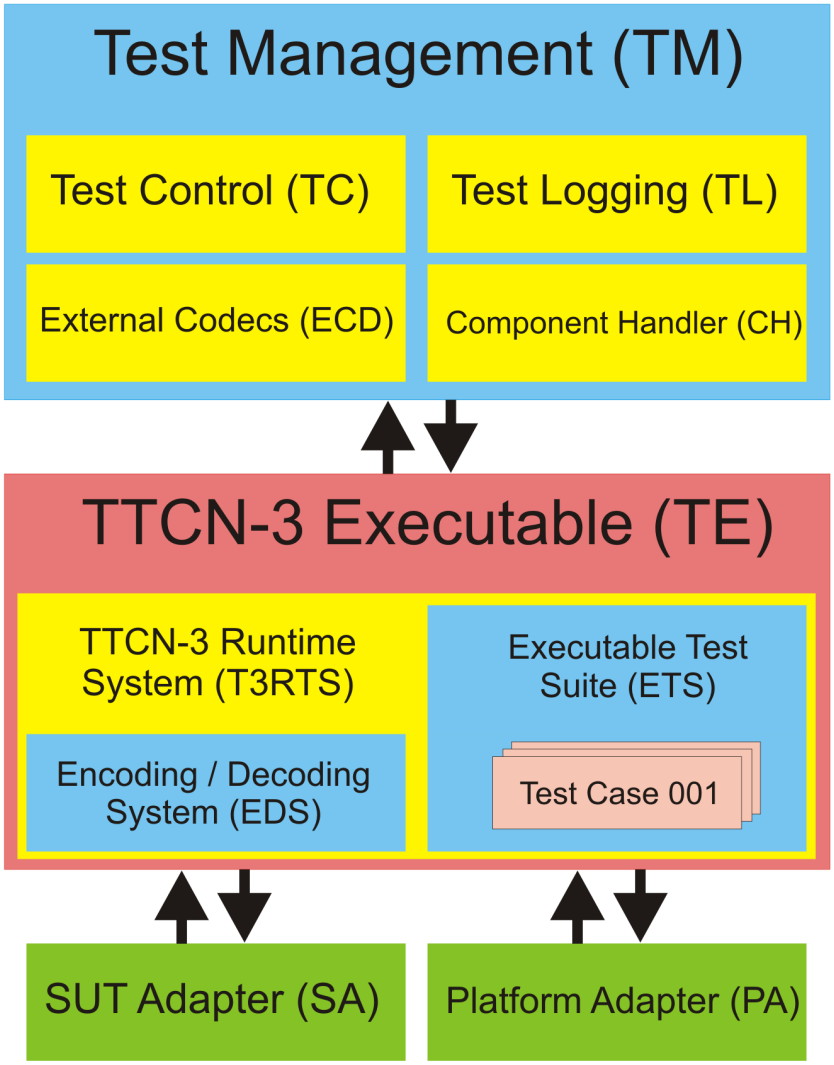

Abbildung 2.1.: Struktur des TTCN-3 Test Systems (Quelle: ETSI)

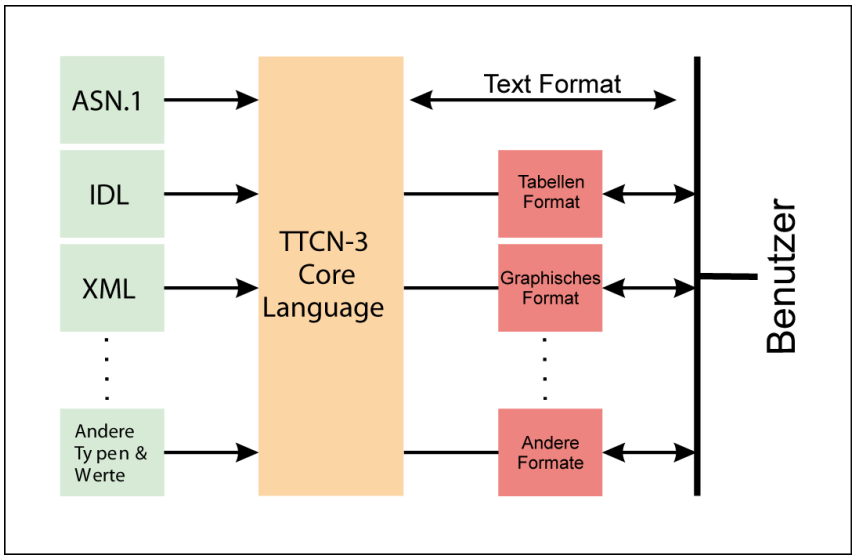

Abbildung 2.2.: TTCN-3 Sprache

reiche der TTCN-3 Sprache sind Protokolle, APIs und Software-Module. Mit TTCN-3 lassen sich verschiedene Arten von Tests durchführen, wie z.. Robustheits-, Leistungs-, Performanz-, System- und Integrationstests.
Zur Spezifikation von Black-Box-Test für reaktive, lokale und verteilte Systeme kann die TTCN-3 Sprache eingesetzt werden.
Dank der wohldefinierten Syntax werden die Ausführungen der Tests präzise definiert. Dadurch ist es möglich, parallele Testprozeduren durchzuführen. Statische oder dynamische und lokale oder verteilte Konfigurationen sind mit TTCN-3 einfach darzustellen. Dieses liegt an der universellen, flexiblen und benutzerfreundlichen Sprache, die auch einfach zu erlernen, zu verwenden und einzuführen ist.

Die Kommunikation zwischen dem Testsystem und dem zu testenden System (SUT: System Under Test), kann sowohl synchron, als auch asynchron erfolgen. Zur Versendung der Daten bietet TTCN-3 Datenschablonen (die Templates) an. Um diese Daten zu validieren, werden Vergleichsmechanismen der Templates verwendet. Die Testdatentypen und -werte können in TTCN-3 beschrieben oder aus anderen Sprachen importiert werden.

Die Core-Language der Sprache TTCN-3 ist die textuelle Kernsprache, mit der Tests definiert werden. Auch die graphische und tabellarische Definition von Tests ist in TTCN-3 möglich, d.. in TTCN-3 gibt es 3 verschiedene Darstellungsformate. Schnittstellen zur Nutzung von Datentypen und -werten anderer Sprachen, wie z.. ASN.1, IDL, XML und andere Datenbeschreibungssprachen sind in TTCN-3 möglich. Die Beschreibungssprache Abstract Syntax Notation One (ASN.1) ist fr die Definition der Datenstrukturen zustï¿½dig. Auch fr die Festlegung und Umsetzung von diesen Datenstrukturen sowie Elementen in ein netzeinheitliches Format spielt ASN.1 ein wichtige Rolle.
Die Abbildung 2.2 veranschaulicht die verschiedenen Definitionsmöglichkeiten von Tests und die Verwendung der Datentypen und -werte, die in ASN-1, IDL oder XML spezifiziert sind, importieren und genutzt werden können.

2.1.3. Aktueller Stand

Mit der neuen Version der TTCN-3 Sprache wird ein großer Bereich der Test-Möglichkeiten abgedeckt. Neue und weitere Anwendungsgebiete werden dabei erst möglich. Damit kann man

nicht nur das Testen der Konformität und Interoperabilität von Kommunikationsprotokollen testen, sondern die Sprache kann auch in Tests der Interaktion von Sensoren und Steuereinheiten, die über ein Bussystem angeschlossen sind, eingesetzt werden. Daher wird TTCN-3 mittlerweile nicht nur in der Mobilfunkbranche eingesetzt, sondern auch noch in der Automobil- und Eisenbahntechnik, sowie in Luftfahrt- und Sicherheitssystemen.

2.1.4. ETSI - Standard

Das Europäische Institut für Telekommunikationsnormen (ETSI: European Telecommunications Standards Institute) wurde 1988 auf Initiative der Europäischen Kommission gegründet. ETSI gehört zusammen mit CENELEC (Europäisches Komitee für elektrotechnische Normung) und CEN (Europäisches Komitee für Normung) zu den drei größten Normungsorganisationen der Welt, mit Sitz in Europa. Wichtige Standards, die von ETSI geschaffen wurden, beziehungsweise an denen sie im ITU-Rahmen mitgearbeitet hat, sind zum Beispiel DSS1, GSM, UMTS, Digital Enhanced Cordless Telecommunications (DECT) und TETRA (Terrestrial Trunked Radio) und vor allem die Testsprache TTCN-3.

Der ETSI Standard für TTCN-3 enthält zur Zeit sieben Teile, die in der Standardreihe TTCN-3 Specifications Edition 3 zusammengefasst sind.

Die Kernsprache von TTCN-3, die die Konzepte und textuelle Syntax von TTCN-3 detailliert darstellt.

Das tabellarische Darstellungsformat, welche im Aussehen und in der Funktionalität früherer Versionen von TTCN ähnelt. TTCN-3 ermöglicht einige Darstellungsformate, die für den Benutzer konzipiert werden, die die Art der Darstellung von TTCN-2 Testsuiten bevorzugen, indem die TTCN-3 Module im tabellarischen Format als Ansammlung von Tabellen dargestellt wird.

Die graphischen Darstellungsformate, die die Darstellung von Interaktionen zwischen SUT (System Under Test) und Testsystem ermöglicht. Dieses ist das zweite Darstellungsformat und basiert auf Message Sequence Chart (MSC).

Die Operationale Semantik, welche die Semantik der TTCN-3 Sprachkonstrukte definiert. Außerdem stellt sie die zustandsorientierte Sicht auf die Ausführung eines TTCN-3 Moduls zur Verfügung.

Die Laufzeitschnittstellen (TRI) von TTCN-3, welche die Spezifikation einer allgemeinen API Schnittstelle zur Anpassung von TTCN-3 Testsystemen an eine SUT definiert, damit die Implementierung der Testsysteme eine plattformspezifische Anpassungsschicht haben.

Die Kontrollschnittstellen (TCI) von TTCN-3, welche die Spezifikation der Schnittstellen, die eine TTCN-3-Ausführungsumgebung für das Kodieren und Dekodieren von Testdaten enthält. Auch das Testmanagement, die Verwaltung von Testkomponenten, die Nutzung von externen Daten und die Protokollierung von Testabläufen wird hier auch vorgestellt.

Die Nutzung von ASN-1 mit TTCN-3. Hier werden die Richtlinien und Abbildungsregeln für die gemeinsame Nutzung von ASN-1 mit TTCN-3 erklärt.

2.1.5. Das Testsystem unter TTCN-3

Bevor die verschiedenen Einheiten der Sprache etwas genauer dargestellt werden, wird hier erst einmal der Aufbau des Testsystems unter TTCN-3 aufgezeigt. Zu finden ist die genaue Spezifikation unter [13].

Wenn man den Aufbau des Systems grob betrachtet, so besteht diese aus drei großen Einheiten, dem Test Management (TM), dem TTCN-3 Executable (TE) und der Adaptereinheit. Diese teilt sich aber noch einmal in die Adaptereinheit zur SUT (SA) und die Adaptereinheit zur Testplattform (PA).

Die Abbildung 2.1 zeigt den Gesamtaufbau der Einheiten und die Zusammenhänge eines TTCN-3 Testsystems auf, die in den nachfolgenden Kapiteln näher erläutern werden.

2.1.5.1. Testmanagement (TM)

Das Testmanagement besitzt zwei wesentliche Aufgaben. Zum einen die Tests auszuführen und zu kontrollieren, zum anderen im Erstellen und Schreiben der Testlogs. Um diesen Aufgaben gerecht zu werden, besteht die Testmanagement Einheit aus vier weiteren untergeordneten Einheiten. Der Test Control (TC), der Testlogging (TL), der External CoDecs (ECD) und der Component Handler (CH).

2.1.5.1.1 Test Control (TC)

Test Control ist die Einheit, die für die Ablaufkontrolle des gesamten Tests verantwortlich ist. Nachdem das Testsystem initialisiert ist, startet die TC den Testablauf. Diese Einheit ist verantwortlich dafür, dass die nötigen und richtigen TTCN-3 Module aufgerufen werden und ihnen die richtigen Parameter übergeben werden. Diese Einheit implementiert auch ein User Interface, so dass über diese Einheit eine Kommunikation mit der TTCN-3 Executable Einheit möglich ist.

2.1.5.1.2 Test Logging (TL)

Wie der Name es bereits verdeutlicht, ist diese Einheit für die komplette Steuerung des Testlogs verantwortlich. Diese Einheit besitzt ein bidirektionales Kommunikationsinterface, so dass jede Einheit der TE seine Loggingentitäten in das Log schreiben kann. Darüber hinaus kann natürlich auch jede Einheit der TM seine Testeinträge übergeben.

2.1.5.1.3 External CoDecs (ECD)

Die External CoDecs Einheit hat die Aufgabe Daten zu codieren oder zu decodieren, so dass externe Datenkommunikation mit unterschiedlichsten Komponenten stattfinden kann. Dieses Interface ist standardisiert, so dass die Möglichkeit besteht, die verschiedensten Codecs zu implementieren und so eine Kommunikation zwischen TTCN-3 und anderen Tools zu gewährleisten.

2.1.5.1.4 Component Handler (CH)

Der Component Handler eröffnet die Möglichkeit, Testkomponenten parallel zu bedienen. Diese Einheit ermöglicht es der TM, mehrere Testkomponenten zu starten und zu kontrollieren.

2.1.5.2. TTCN-3 Executable (TE)

Die TE ist verantwortlich für die Interpretation und Ausführung der einzelnen Tests. Sie besteht aus der TTCN-3 Runtime System Einheit, die wiederum aus den Einheiten Encoding und De-

coding System (EDS) und der Executable Test Suite (ETS) bestehen. Alles in allem ist sie für die Handhabung und die Kontrolle der einzelnen Timer zuständig.

2.1.5.2.1 TTCN-3 Runtime System (T3RTS)

Das T3RTS interagiert mit dem gesamten Einheitenkomplex. Das System ist die Schnittstelle zwischen allen Großeinheiten, wie TM, SA und PA. Diese Einheit ist für den endgültigen Start der Tests zuständig. Zum einen erstellt und löscht das System sämtliche TTCN-3 Testkomponenten, und zum anderen ist es für die gesamte Kommunikation während der Tests zuständig. Die Daten werden nicht nur zwischen den einzelnen Einheiten hin und her geschickt, sondern auch mit Hilfe der EDS in jede nötige Form übersetzt.

2.1.5.2.2 Encoding Decoding System (EDS)

Die EDS Einheit ist verantwortlich für das Übersetzen der Testdaten. Sie ist nicht nur für das Übersetzen der Kommunikationsdaten zuständig, sondern auch für Daten, die in den TTCN-3 Modulen ausgeführt werden.

2.1.5.2.3 Executable Test Suite (ETS)

Der ETS überwacht die Ausführung und die Interpretation der einzelnen Testfälle. Er kontrolliert das Einhalten der Reihenfolgen und die Abstimmung der einzelnen Testfälle, so wie in den einzelnen TTCN-3 Modulen definiert. Er interagiert mit dem T3RTS, empfängt die Aufgaben und sendet wiederum die Ergebnisse. Des weiteren hat die Einheit die Aufgabe TTCN-3 Testkomponenten zu erzeugen und zu löschen. Diese Einheit steuert auch externe Funktionen und Timer, doch kommuniziert sie mit diesen nur über die übergeordnete T3RTS Einheit.

2.1.5.2.4 Timer (TID)

Die Timer sind die kleinsten Einheiten. Sie steuern den Abbruch eines Testes und garantieren so, dass eine Testkomponente terminiert, auch wenn sie nicht ordnungsgemäße Ergebnisse liefert und z.. in einer Endlosschleife hängen bleibt. Die Timer werden in der TE klassifiziert. Timer, die von der TE gestartet wurden, um Operationen zu überwachen, die der TE bekannt sind, heißen implizite Timer. Timer, die zwar der TE durch ein Timer ID (TID) bekannt sind, aber in einem Plattform Adapter (PA) laufen, nennt man explizite Timer.

2.1.5.3. SUT Adapter (SA)

Der SUT Adapter (SA) ist eine wichtige Einheit, die die komplette Kommunikation zwischen dem zu testenden System und der Systemtesteinheit übernimmt. Ohne diese Einheit wäre der eigentliche Systemtest gar nicht möglich. Diese Einheit schickt die Anfragen und Testoperationen der TE an das SUT, empfängt im Gegenzug die Ergebnisse der SUT und schickt diese wiederum an die TE.

2.1.5.4. Plattform Adapter (PA)

Der Plattform Adapter (PA) implementiert externe Funktionen in das TTCN-3 Testsystem. In dieser Einheit werden diese externen Funktionen mit allen Timern bereitgestellt. Diese Timer haben eine eindeutige TID, so dass sie mit den Timer Instanzen in der TE eindeutig gesteuert

```
testcase MyTestcase()
  runs on MyTestComponent
  system SystemComponent {
        ...
        MyPort.call(myMethod:{MY_ID}, 3.0);
        alt {
          [] MyPort.getreply(myMethod:{MY_ID}  value "Hello World!") {
             verdict.set(pass);
          }
          [] MyPort.getreply {
             verdict.set(fail);
          }
          [] MyPort.catch(timeout) {
             verdict.set(fail);
          }
        }
        ...
  }
```

Abbildung 2.3.: Text Format ([24])

und verwaltet werden können. Die PA Einheit informiert die TE über abgelaufene Timer und ermöglicht es der TE, die Timer durch ihre eindeutige ID ständig starten, lesen und stoppen zu können.

2.1.6. TTCN-3 die Sprache

In TTCN-3 gibt es drei Definitionsformate, um Testfälle zu erstellen. Zu einem kann ein Testfall per Text, graphisch oder tabellarisch definiert werden.

Das **Textformat** (siehe Abb. 2.3) nennt man auch *Core Notation* und ist die Grundlage für die Kompilierung eines Testfalls. Mittels Editoren können die Tests, unter Benutzung der TTCN-3 Notationen, erstellt und bearbeitet werden. Da die Notationen mit gängigen Programmiersprachen ähnlich sind, ist dieser Art des Formats für Programmierer am ehesten geeignet.

Das **graphische Format** (siehe Abb. 2.4) stellt die Testabläufe im Form von Sequenzdiagramm dar. Diese Art des Formats dient hauptsächlich zur Analyse von Testresultaten, aber auch zur Dokumentation und der Konzeption eines Testfalls.

Beim **Tabellen Format** (siehe Abb. 2.5) werden anhand einer Tabelle die Parameter und Funktionswerte in den verschiedenen Stellen des Testfalls aufgerufen. Anschließend kann man diese ermittelten Werte vergleichen und fehlerhafte Werte schnell erkennen, ohne sich den Source-Code detailliert ansehen zu müssen.

Nachdem bisher die Einheiten des Testsystems betrachtet sind, werden jetzt die Einheiten der Sprache genauer dargestellt.

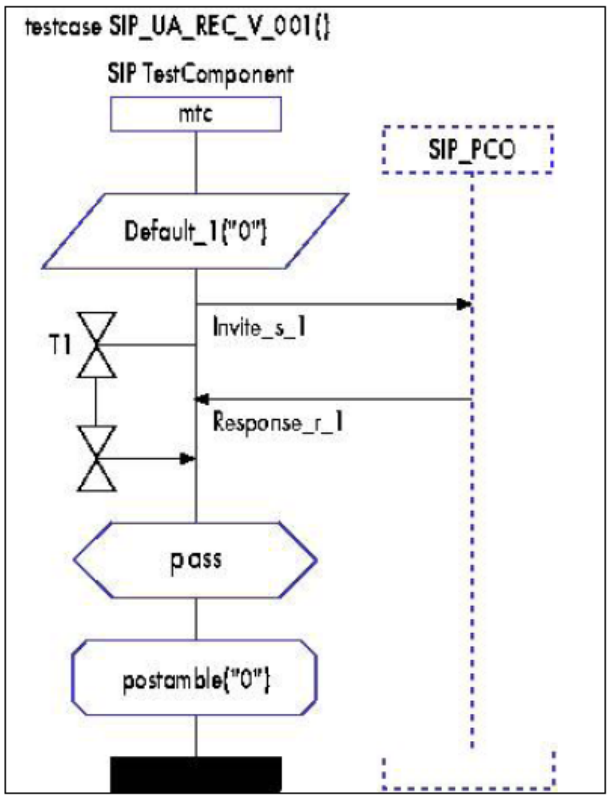

Abbildung 2.4.: Graphisches Format ([24])

Test Case Definition

Name	***:***	*MyTestcase*
Group	***:***	
Purpose	***:***	*Example Testcase*
System I/f	***:***	
MTC Type	***:***	*MyComponentType*
Comments	***:***	

Name	Type	Initial Value	Comments
MyVar	INTEGER	0	

Behaviour Definition	Comments
alt { [ ] MyPort.receive(Msg); [ ] : }	

DetailedComments:

Abbildung 2.5.: Tabellen Format ([24])

2.1.6.1. Module

Die oberste Einheit in der Strukturierung von TTCN-3 ist das Modul. Ein standardmäßiges Modul hat zum einen den Definitionsteil und zum anderen einen optimalen Kontrollteil. Der Definitionsteil besitzt alle für die Testreihe notwendigen Daten, wie Datentypen, Ports und Testkomponenten, sowie alle notwendigen Definitionen. Wie in Programmiersprachen üblich, können zusätzliche Definitionen auch aus anderen und mit anderen Modulen importiert werden.

Der Kontrollteil ist zu vergleichen mit dem Hauptteil eines Programms. Durch diesen Teil wird der Ablauf des Moduls kontrolliert. Darüber hinaus werden hier auch die Bedingungen, die im Definitionsteil erstellt und importiert sind, überprüft.

2.1.6.2. Templates

Die Templates, in deutscher Sprache Vorlagen genannt, werden von TTCN-3 zur Verfügung gestellt, um die Beschreibung von Testdaten zu vereinfachen. In dieser Vorlage können Daten nicht nur mit konkreten Werten, sondern auch mit speziellen Matching-Operatoren zur Verfügung gestellt werden. Mit Hilfe dieser Matching-Operatoren können z.. Wertebereiche definiert werden. Darüber hinaus lassen sich mit ihnen Mengen von Werten, Komplemente von Wertemengen oder auch Zeichenketten und bestimmte Felder definieren. Wenn dann Testausführungen stattfinden, wird mit Hilfe der Templates geprüft, ob die empfangenen Daten im Bereich dieser Vorlage liegen und diese konform sind. Die Templates dienen der Überprüfung der Daten. Sie können darüber hinaus entsprechende Ergebnisse auf eingegebene Daten zurückgeben.

2.1.6.3. Komponenten und Ports

In TTCN-3 werden die Testfälle von verschiedenen parallel arbeitenden Komponenten ausgeführt. Diese Testkomponenten interagieren dann untereinander und kommunizieren ihre Reaktionen und Interaktionen dann über so genannte Ports. Diese Ports sind in TTCN-3 sehr genau definierte Schnittstellen. Jeder Port ist nach Definition mit einer unendlich langen FIFO-Warteschlange **F**irst **I**n **F**irst **O**ut ausgerüstet. Die Kommunikationsmechanismen, die zur Verfügung gestellt werden, sind zum einen die Nachrichten basierte und zum anderen die prozedurale Kommunikation.

Bei der prozeduralen Kommunikationen werden ganze Prozeduren bei einem anderen Kommunikationspartner ausgeführt, der dann wiederum seine Ergebnisse an seinen Kommunikationspartner zurücksendet.

Die Kommunikation besteht in TTCN-3 aus Nachrichten, die ihrerseits wieder aus Datentypen bestehen. Diese definierten Datentypen werden dann wiederum über die definierten Ports gesendet. Somit werden sie als Werte zwischen den Testkomponenten übertragen.

Die Testkomponenten bestehen also nicht nur aus lokalen Ports und Timern, sondern können auch zusätzlich noch aus Variablen und Konstanten bestehen.

TTCN-3 besitzt zur Startzeit immer nur eine Testkomponente, die so genannte Hauptkomponente. Doch ist das System alles in allem dynamisch, denn jede Testkomponente hat die Möglichkeit, wieder beliebig viele Testkomponenten und Verbindungen aufzurufen. Jede Komponente kann sich darüber hinaus noch selbst stoppen oder wird mit der übergeordneten Komponente automatisch terminiert.

2.1.6.4. Testfall

Ein Testfall ist ein Programm. Dieses beschreibt wie Folgen von Testereignissen zusammenhängen und versucht sogar diese Folgen zu beurteilen. In TTCN-3 wird dieser Testfall durch eine Funktion beschrieben. Diese Funktion endet immer mit einen Testurteil. Durch diese Testurteile wird dann das weitere Verhalten der Test Management Unit beeinflusst.

2.1.6.5. Funktionen

Das Strukturieren des Testverhaltens und Berechnen von Werten in TTCN-3 wird durch die so genannten Funktionen realisiert. Jede Funktion spezifiziert ein Testverhalten und kann zusätzlich eine **runs on**-Anweisung besitzen, die auf den Typ der zugehörigen Testkomponente referenziert, auf der diese ausgeführt wird. Die Funktionen arbeiten dann auf den vorher definierten Variablen, Konstanten, Timern und Ports.

2.1.6.6. Testschritte

Das nächste wichtige Element von TTCN-3 sind die Testschritte. Diese Schritte ermöglichen es, auf unerwartete Abweichungen im Test zu reagieren und das weitere Agieren zu ermöglichen. In diesen Testschritten werden Fehlerbehandlungen definiert und mit Hilfe von Timern das Weiterarbeiten der Testumgebung bestimmt. Die Timer, senden z.. eine Antwort, falls die erwartete aufgrund einer Endlosschleife ausbleibt. Durch Timer wird nach einer festgelegten Zeit terminiert und die dann definierte Ersatzanweisung ausgeführt.

2.1.6.7. Testurteile

Um die durchgeführten Tests zu beurteilen und zu bewerten, besitzt TTCN-3 den Datentyp **verdicttype**. Er kann vier verschiedene Status annehmen.

Der Status **pass** beschreibt den oft gewollten Erfolg des Durchlaufes. Das Testurteil **fail** beschreibt den Misserfolg. Dann besitzt TTCN-3 noch zwei zusätzliche feinere Werte. Der Wert **inconc** kommt von inconclusive und besagt, dass ein Testzweck zwar erfolglos, der Ablauf aber nicht fehlerhaft war. Als vierten Status gibt es den Wert **error**, der besagt, dass der Fehler in einer Hardwarekomponente aufgetreten ist.

Zusätzlich zu den Status des beschriebenen Datentyps, besitzt TTCN-3 bei jeder Testkomponente auch ein lokales Urteil. Diese können mit **get** abgefragt und mit **set** sowohl vordefiniert als auch gesetzt werden. Wird ein Testfall vorzeitig beendet, so kann mit Hilfe der lokalen Urteile festgestellt werden, in welcher Testkomponente der Abbruch stattgefunden hat.

Für das Setzen von Testurteilen arbeitet TTCN-3 aber nach ganz speziellen Regeln, um so das nachträgliche Manipulieren von Tests zu verhindern.

2.2. Beschreibung der Architektur

Zunächst hier ein kleiner Exkurs, der grundlegendes Wissen über die allgemeine Software-Architektur vermittelt. Der Begriff Softwarearchitektur stellt die grundlegenden Elemente und die Struktur eines Softwaresystems dar. Dabei ist die Softwarearchitektur der erste Schritt bei der Software- oder Systementwicklung. Dieser erster Schritt ist der kritischste und wichtigste Entwicklungsprozess bei einer Softwareentwicklung. Die Wartungsarbeiten und die Sicherheit, als auch die Performance der Software hängt von der Softwarearchitektur ab. Je besser

Abbildung 2.6.: Einfache Kommunikation zwischen Komponenten [3]

der Entwurf, desto besser die Weiterentwicklung, da eine Änderung in der Softwarearchitektur einen großen Aufwand beansprucht. Die Architektur beschreibt hierbei das Zusammenspiel der Komponenten einer Software, welches durch Schnittstellen realisiert wird. Denn Softwareschnittstellen sind für den Datenaustausch zwischen verschiedenen Prozessen und Komponenten von großer Bedeutung. In den folgenden Unterabschnitten wird das Laufzeitsystem dargestellt. Dabei wird nicht die Implementierung, sondern nur die dahinter stehende Idee, erklärt. Die Implementierung dieser Schnittstellen wird im anschießenden Kapitel beschrieben.

2.2.1. TTCN-3 Konstrukte

Nach der Entscheidung, einen TTCN-3-Java-Compiler zu programmieren, wurden im nächsten Abschnitt des Arbeitsvorgangs der Compiler-Gruppe einige spezielle Konstrukte von TTCN3 untersucht und erläutert. Es wurde dabei verdeutlicht, welche Eigenschaften diese Konstrukte haben und wie sie entsprechend in Java übersetzt werden können.

In diesem Abschnitt liegt die Konzentration auf drei wesentlichen Konstrukten:

- die Ports
- die Timer
- das Alt-Statement

Im nachfolgenden werden die Aufgaben und die Definitionen dieser Konstrukte dargestellt.

2.2.1.1. Das Port-Konstrukt

Die Ports bei TTCN-3 dienen der Kommunikation. Alle Nachrichten werden über Ports zwischen den Komponenten hin und her geschickt (Abb. 2.6). Diese findet ohne Verzögerung statt. Das bedeutet, dass die Verbindung ständig offen ist. Die Nachrichten werden in porteigenen Message Queues gespeichert. Diese sind von der Spezifikation her nicht limitiert.

Im wesentlichen wird ein Port durch 2 Eigenschaften definiert. Zum einen wird festgelegt, ob der Port empfängt, sendet oder bidirektional ist. Zu anderen muss definiert werden, welche Art von Nachrichten dieser Port akzeptieren soll. Problematisch bei den Ports ist, dass sie über das Statement „all“ sehr lose definiert werden können, so dass sie praktisch alles annehmen, doch sollte dieses vermieden werden, da es ein effektives Type-Checking verhindert. Genauer wird dieses Konstrukt beschrieben in [23].

Die wichtigen Eigenschaften der Ports nochmal in Kürze.

1. Sie dienen zur Kommunikation,
2. sind in, out oder bidirektional Ports,
3. stehende Verbindung, senden ohne Verzögerung,

4. jeder Port besitzt seine eigene unlimitierte Message Queue,
5. der Typ, der zu akzeptierenden Nachrichten ist festgelegt,
6. oder akzeptiert alle Nachrichten (zu vermeiden).

In Abbildung 2.7 wird dargestellt, welche Kommunikations-Konstrukte erlaubt sind.
Was bei der Portkommunikation nicht erlaubt ist, zeigt Abbildung 2.8 und ist wie folgt definiert.

1. Ein Port einer Komponente darf nicht mit zwei oder mehr Ports von dieser verbunden sein.
2. Ein Port einer Komponente A sollte auch nicht mit mehr als 2 Ports der Komponente verbunden sein.
3. Ein Port von Komponente A kann nur eine 1-1- Verbindung mit dem SUT Interface besitzen, siehe auch 2.8 .) und d.).
4. Direkte Kommunikation des Test Interfaces mit sich selbst 2.8 f.).
5. Ein verbundener Port sollte nicht umgeleitet werden und ein umgeleiteter Port sollte nicht angebunden werden, wie in 2.8 g.) dargestellt.

Idee der Implementierung eines Ports in Java
Fr das Laufzeitsystem wird folgendes benötigt. Wenn wir davon ausgehen, dass unsere Komponenten in Threads abgebildet werden, dann bekommt jeder Port in einem Thread mindestens eine FIFO-Queue, dieser ist dann entweder mit eingehenden oder ausgehenden Nachrichten zu füllen. Ein bidirektionaler Port besteht somit aus zwei FIFO-Queues. Des weiteren müssen natürlich die oben genannten Einschränkungen beachtet und überprüft werden.

2.2.1.2. Das Timer-Konstrukt

Die Aufgabe der Timer ist es, die Eigenschaften der Zeit eines Testablaufes zu beschreiben und zu kontrollieren. Dazu stehen bei den Timern einige Funktionen zur Verfügung, wie `start`, `stop`, `read` und `timeout`. Die Hauptaufgabe der Timer ist, neben der Kontrolle von Zeitabschnitten, die Verhinderung von nicht terminierenden Programmzuständen, so dass ein Test einer SUT nicht beendet wird, weil diese in einer Schleife verharrt. Die Zeitabschnitte werden beim Timer in Sekunden gemessen und als Typ `float` angegeben.
Ein Timer hat folgende Funktionen(Methoden):

1. Er kann erstmal nur deklariert werden.
2. Er kann mit `start` gestartet werden.
3. Er kann mit `stop` angehalten werden
4. Er kann mit `restart` wieder gestartet werden.
5. Es kann mit `read` die abgelaufenen Zeit erfragt werden.
6. Er kann mit `timeout` beendet werden.

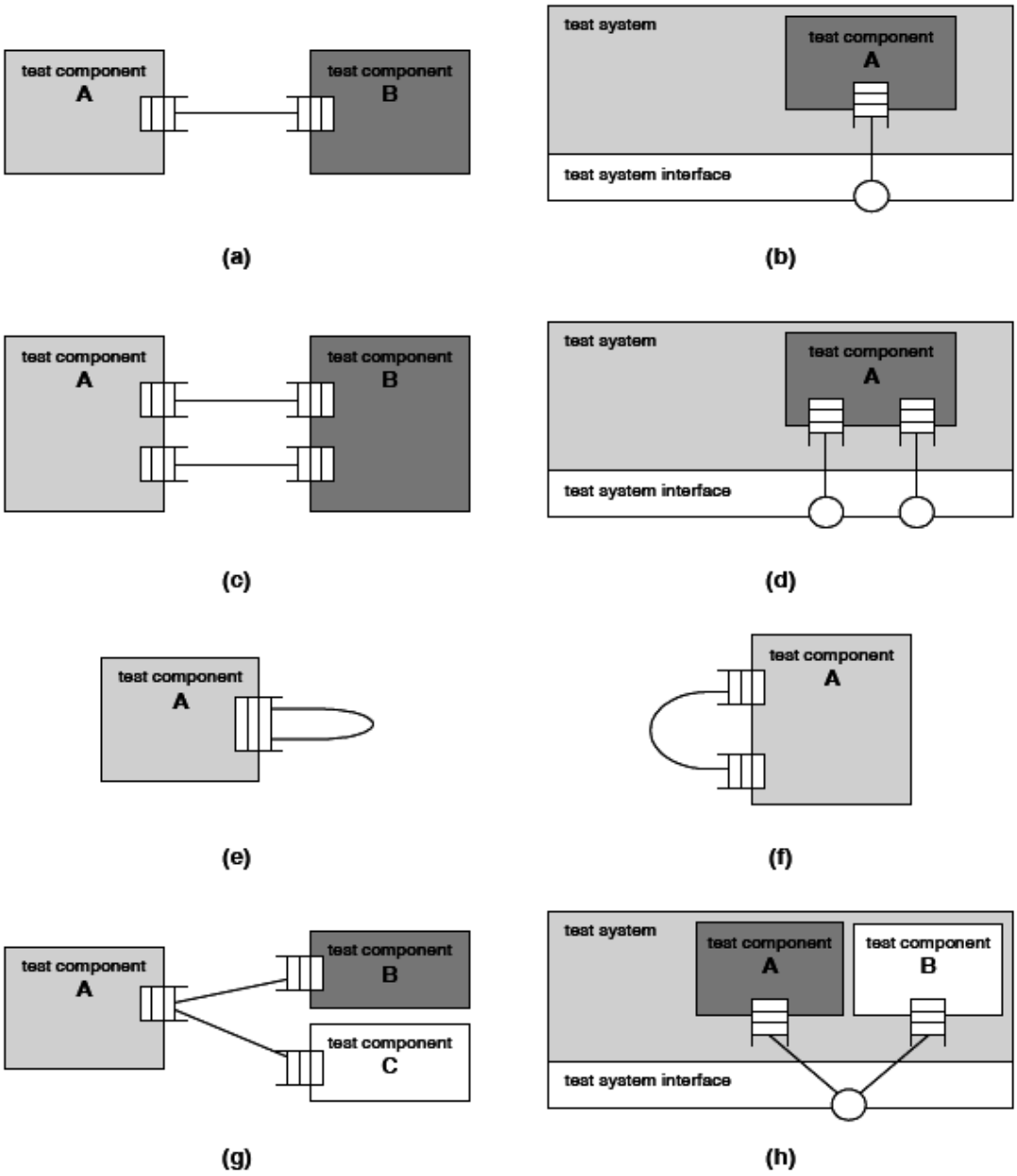

Abbildung 2.7.: Erlaubte Kommunikations-Konstrukte [3]

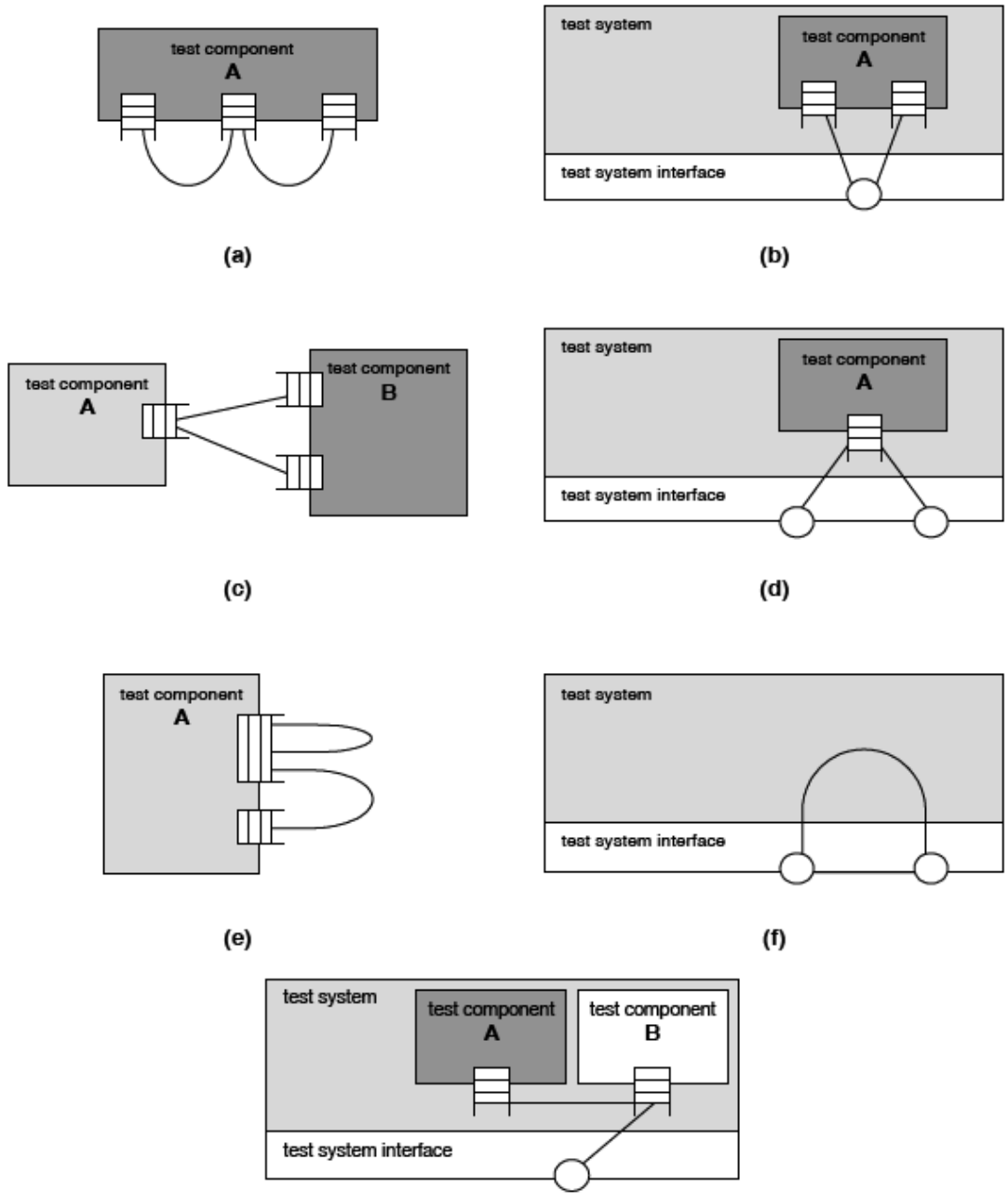

Abbildung 2.8.: Nicht erlaubte Kommunikations-Konstrukte [3]

7. Es kann mit `runnning` kontrolliert werden, ob ein Timer läuft.

Die Eigenschaften, die hier für das Laufzeitverhalten zu betrachten wären, ergeben sich neben dem parallelen Ablauf erst mit dem Alt-Statement, welches im nächsten Abschnitt näher betrachtet wird.

2.2.1.3. Das Alt-Statement-Konstrukt

Bei der Verwendung von „timeout“ und „recieve“ Operationen als einzelne eigenständige Anweisung ist es nicht möglich die Operation weiter fortzusetzen, bevor eine passende Nachricht ankommt oder der Timer abgelaufen ist.

Der „stand-alone timeout“ und die „recieve“ Operationen erfordern, dass die Nachrichten in der angegebenen speziellen Reihenfolge ankommen und nicht umgekehrt. Die unerwarteten Nachrichten können ansonsten den Testfall blockieren.

Der Zweck des „alt-statements“ ist es, die Blockoperation zu kombinieren, indem die erste Blockoperation, die Fortsetzung der Ausführung des Problems, entscheidet.

Bei „alt statements“ kann man mehrere Alternativen definieren, um die Ausführung zu durchlaufen. Man kann z.. zwei Alternativen „pass“ und „fail“ verwenden.

Die Alternativen vom „alt-statement“ funktionieren nach dem top-down Prinzip. Man kann auch speziellen Fällen gegenüber, allgemeinen Fällen Prioritäten geben.
Bei der Definition von „alt-statements“ benutzt man zu Beginn einer neuen Alternative immer eine eckige Klammer. Diese Klammern können leer sein oder auch boolesche Werte beinhalten, siehe folgendes Beispiel 2.1.

```
alt {
[X > 2]  ptg. recieve ( a_msgl ) { setverdict ( pass ) };
[X < 0]  ptq. recieve ( a_msg2 ) { setverdict ( pass ) } ;
[else] { setverdict( fail }
}
```

Beispiel 2.1: Blockade des Alt-Statements durch busy-wait

2.3. Nokia

Nokia ist seit der Entwicklung der Mobilkommunikation ein weltweit führendes Unternehmen für diesen Bereich. Das Unternehmen leistet eine entscheidenden Beitrag zum Wachstum und zur Nachhaltigkeit der Mobilfunkindustrie. Die Stärke des Unternehmen, sind die leicht zu bedienenden und innovativen Produkte von Mobiltelefonen. Des weiteren hat Nokia gute Lösungen in den Bereichen Fotografie, Spiele und Multimedia. Nokia bietet auch Lösungen für die Netzbetreiber an. Mittels Verbesserung und Bereicherung der Technologiemöglichkeiten, versucht Nokia das Leben der Menschen zu verbessern.

2.3.1. Nokia Research Center

Das Nokia Research Center entwickelt richtungsweisende Technologien und erarbeitet Kompetenzen in technischen Bereichen. Dieses ist für den zukünftigen Erfolg von Nokia ein zentrales Entwicklungsfeld. Das Nokia Research Center unterstützt die oben aufgezählten vier Unternehmensbereiche (Mobile Phones, Multimedia, Enterprise Solutions und Networks), bei der

Entwicklung neuer Konzepte, Techniken und Anwendungen in diesen Feldern.
Die Zusammenarbeit mit Universitäten, wie z.. der Universität Dortmund, erweitern die Möglichkeiten für den Nokia Research Center den Rahmen der Technologie weiter zu stecken und zu entwickeln.

3. Verwendete Werkzeuge

Bei der Entwicklung des Tools **TTCP** werden verschiedene Werkzeuge und Bibliotheken eingesetzt. Dieses Kapitel beschreibt sowohl die benutzen diversen Werkzeuge als auch die Technologien, die unterstützend herangezogen worden sind.

3.1. PmWiki

Das Wissensmanagmentsystem PmWiki in der Version 2.1.10 basiert auf PHP und ist sehr einfach installier- und konfigurierbar. Sie ist ein ähnliches System wie die freie Enzyklopädie Wikipedia und dient der strukturierten Ablage von Wissen, das von verschiedenen Benutzern eingepflegt und erweitert werden kann. Sie dient in unserem Fall sehr zum Nutzen des Projektmanagements, da vor allem bei der Erstellung komplexer Systeme, wie **TTCP**, die Beschreibungen der Schnittstellen von großer Bedeutung sind. Der wesentliche Vorteil bei dem Einsatz von PmWiki liegt darin, dass „Links“ zu anderen internen Seiten und zu Webseiten, ohne großes Wissen über HTML, gesetzt werden können und somit die Flexibilität des Projektes erhöht wird.

3.2. UML

UML (Unified Modeling Language) ist eine von Objekt Management Group (OMG) standardisierte Modellierungssprache für die Softwaremodellierung. Fast alle Softwaresysteme werden heutzutage mit UML modelliert. Der wesentliche Vorteil der häufigen Benutzung beruht auf die Einheitlichkeit der Softwaremodelle .
Mittels UML werden Softwaresysteme und deren Architekturmodelle graphisch dargestellt. Somit wird die Kommunikation zwischen den Entwicklern und dem Auftraggeber erleichtert. Diese erstellten Modelle können anschließend in verschiedene Programmierfragmente umgesetzt werden.

3.3. Java

Die von Sun Microsystems entwickelte objektorientierte Programmiersprache **Java** ist im Rahmen eines Forschungsprojekts entstanden. Die Plattformunabhängigkeit von Java ist der wichtigste Vorteil. Die Java-Plattform ist für viele Betriebssysteme verfügbar. Programme, die für de Java-Plattform entwickelt werden, sind deshalb in vielen Umgebungen direkt ausführbar. Die Sprache ist stabil und leicht zu erlernen. Dies ist ein entscheidender Grund, das Projekt TTCP in Java zu implementieren. Stabilität bedeutet, dass fehlerarmes Programmieren unterstützt wird. Zudem wird Java aktiv weiterentwickelt. Das Tool TTCP wird mit der Java Version 1.5 entwickelt. Die Java-Plattform hat bei der Entwicklung des Tools TTCP Anpassungen in verschiedene Plattformen erspart.

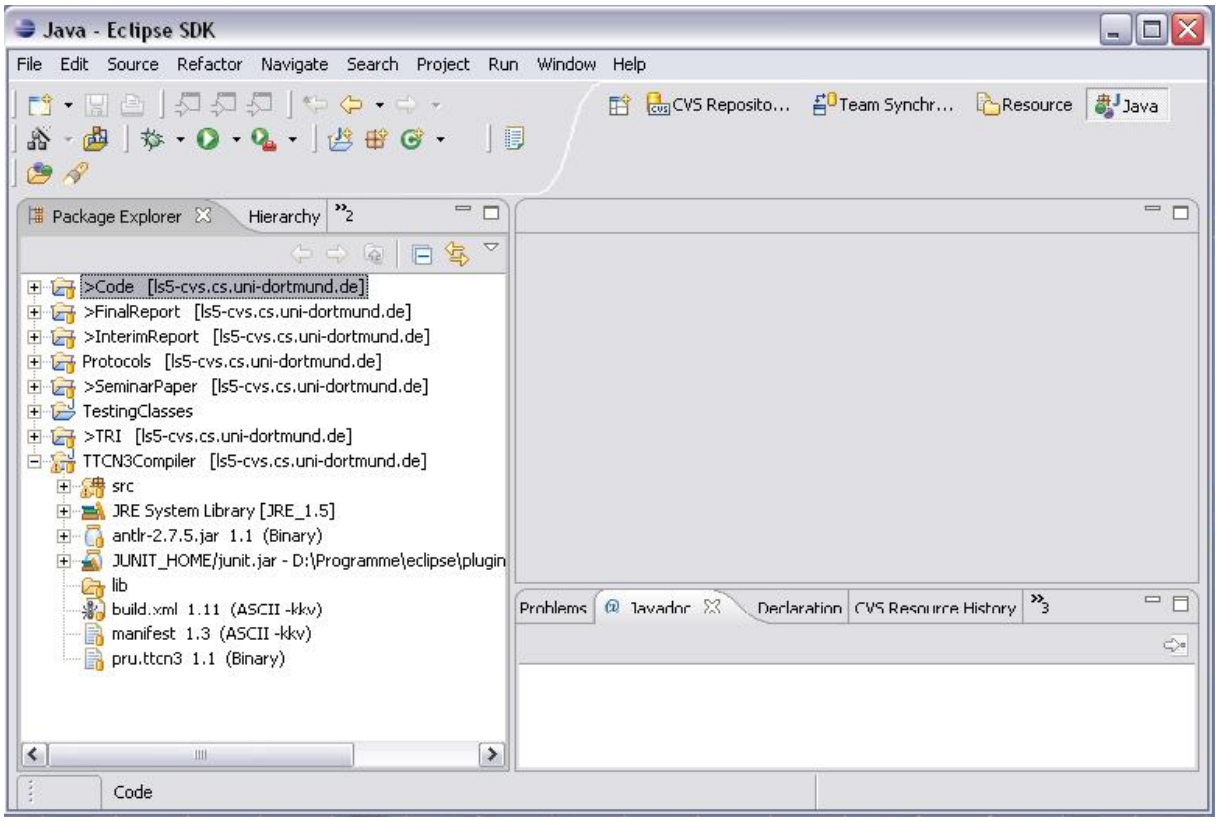

Abbildung 3.1.: Eclipse in der Version 3.1

3.4. Eclipse

Für die Entwicklung des Tools TTCP wird die Entwicklungsumgebung (IDE) **Eclipse** (Abbildung 3.1) in der Version 3.1 genutzt. Eclipse ist ein Open-Source-Projekt und stand unter der Leitung der Firma IBM. Nach der Freigabe der Quellcodes wurde die Verwaltung und Entwicklung von der Eclipse-Foundation übernommen. [1]

Eclipse ist ein Rahmenwerk zur Integration verschiedenster Anwendungen. Eine solche Anwendung ist z.. die mitgelieferte Java Entwicklungsumgebung **JDT (Java Development Toolkit)**. JDT ist im Grunde mit einer Gruppe von Plugins vergleichbar, die dem generischen Plattform-Ressourcen-Modell Java spezifisches Verhalten hinzufügt und den Workbench durch Java spezifische Sichten, Editoren und Aktionen ergänzt. Diese Anwendungen werden in Form von Plugins zur Verfügung gestellt und von der Eclipse-Plattform automatisch erkannt und integriert. Weitere Plugins, die benutzt wurden, sind SWT, Visual Editor, Omondo, JUnit und TTworkbench.

Der **Standard Widget Toolkit (SWT)** ist eine Bibliothek und wird für die graphischen Oberflächen mit Java verwendet. Der SWT stellt Gui-Komponenten (Widgets), wie zum Beispiel Buttons, äume oder Tabellen über native Komponenten des Betriebssystems zur Verfügung. Der größte Vorteil ist die Verfügbarkeit für mehrere Plattformen, auf denen ein SWT-Programm jeweils wie ein ‘normales’, plattformabhängiges Programm aussieht.
Eclipse bietet die Möglichkeit durch den Visual Editor-Plugin eine visuelle Graphische Oberfläche zu entwickeln. Der Eclipse Visual Editor (VE) kann mit verschiedenen Bibliotheken, wie unter anderem auch mit SWT, eingesetzt werden. Das Frontend für die Eingabe sowie Anzeige von Daten von TTCP (Abbildung 3.2) wurde mit dem **Visual Editor 1.1.0.1** und der **SWT**

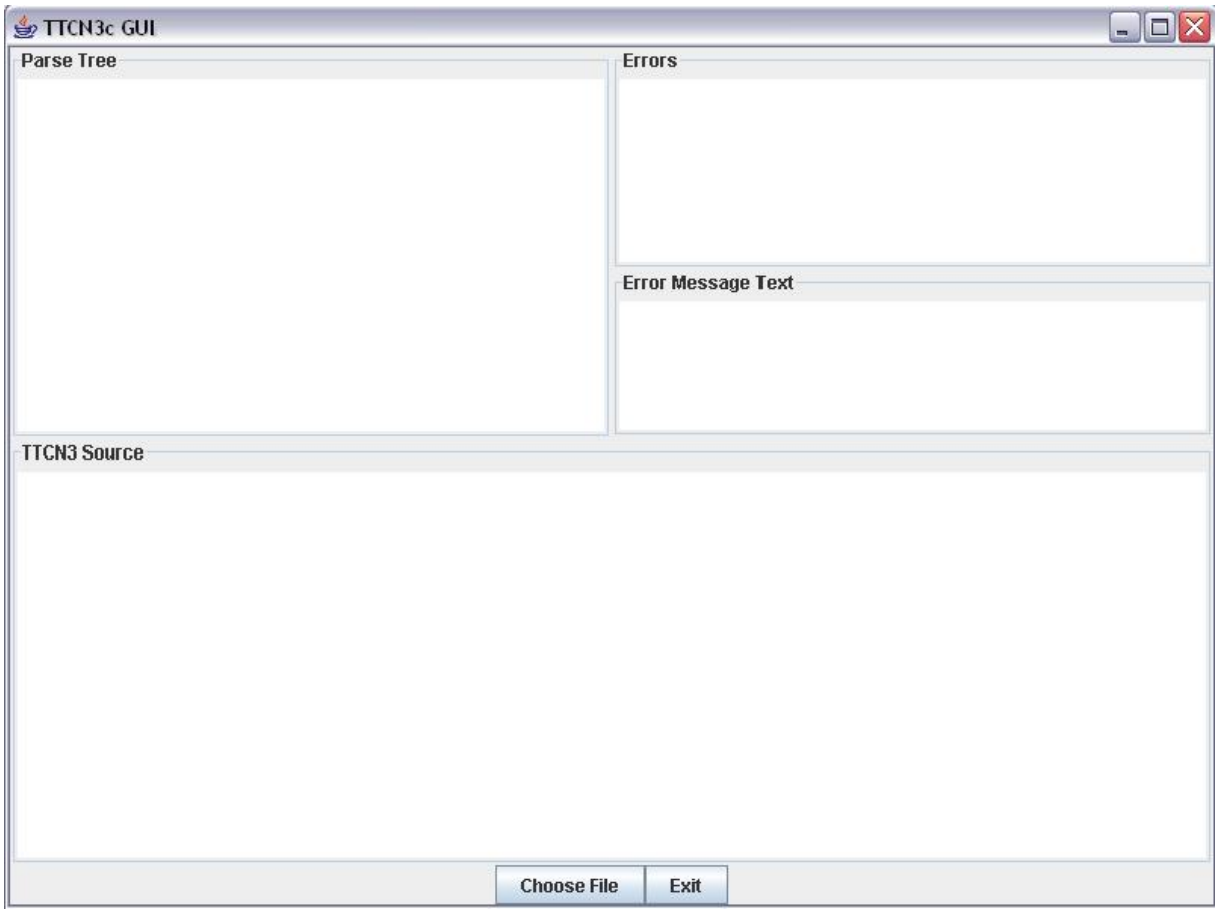

Abbildung 3.2.: GUI TTCP mit VE und SWT

Bibliothek entwickelt. [2]

Omondo's EclipseUML ist ein visuelles UML Modellierungs-Werkzeug, das in Eclipse als Plugin eingesetzt wird. Das UML-Tool besitzt alle möglichen Diagrammtypen, unter anderem Use-Cases, Klassendiagramme und Sequenzdiagramme. Diese UML-Diagramme kann Omondo direkt in Java Code umsetzen. Auch die andere Richtung ist möglich, also aus Java Code UML Diagramme zu erzeugen. Es gibt eine Professional-Version und eine freie Version. Die Projektgruppe hat die freie Version von **EclipseUML 2.1.0** benutzt, die kompatibel mit Eclipse 3.1 ist. [5]

JUnit ist ein kleines, mächtiges Java-Framework zum Schreiben und Ausführen automatischer Unit-Tests. JUnit ist in Eclipse integriert und wird somit mitgeliefert. Der Unit-Test ist ein Teil eines Softwareprozesses und dient zur Validierung der Korrektheit von Modulen einer Software, z.. von einzelnen Klassen oder Methoden. Da die Tests direkt in Java programmiert werden, ist das Testen mit JUnit so einfach wie das Kompilieren. Die Testfälle sind selbst überprüfend und damit wiederholbar. Die Größe der unabhängig getesteten Einheiten kann dabei von einzelnen Methoden über Klassen bis hin zu Komponenten reichen. Dabei wird bei jedem Test die zu testende Funktion oder Methode mit Testdaten (Parametern) konfrontiert und deren Reaktion auf diese Testdaten geprüft. Die zu erwartenden Ausgabewerte werden nun mit den von der jeweiligen Funktion oder Methode gelieferten Ergebnisdaten verglichen. Stimmt das erwartete Ergebnis mit dem gelieferten Ergebnis der Funktion oder Methode überein, so gilt der Test als bestanden. Liefern die Testszenarien erwartungsgemäß die korrekten Werte, so

kann der Entwickler davon ausgehen, dass seine Implementierung der Funktion oder Methode korrekt ist.

TTworkbench basiert auf Eclipse und ist eine grafische Testentwicklungs-und Ausführungsumgebung für TTCN-3. Diese Umgebung wurde bereits von der ETSI als TTCN-3 konform zertifiziert. Sie unterstützt die Testspezifikation, -ausführung und -analyse. Zusätzlich hat TTworkbench noch die Funktion Testfälle grafisch zu definieren und Testabläufe entsprechend grafisch zu visualisieren. Nach den Definitionen der Tests in TTCN-3 werden diese Tests in Testfälle compiliert. Diverse Arten von Testmethoden von TTworkbench leisten Tests zu administrieren, auszuführen und deren Ergebnisse zu analysieren.[7] TTworkbench wird benutzt, um TTCN-3 Testfälle zu verfassen.

3.5. CVS

Bei einer Gruppenarbeit ist es sehr wichtig, an einem gemeinsamen Code zu arbeiten. Damit die Arbeit reibungslos verlaufen kann, wird das **Concurrent Versions System** (kurz CVS) benutzt.

CVS ist ein Software-System, das die Änderungen der einzelnen Dokumente und Code überwacht. Es achtet dabei auf Konflikte zeitgleicher Änderungen und verlangt dann von dem Programmierer, dass dieser aufgelöst wird. D.. CVS verwaltet die Versionen der Daten, die in einem Projekt benutzt werden. CVS zeigt dem Programmierer genau die Stelle an, an der im Code Änderungen stattgefunden haben. Dem Programmierer wird dabei die Auswahl gewährleistet, die Änderungen manuell oder automatisch durchzuführen. Die Datei muss dabei als ein Software-Projekt an einer zentralen Stelle, also in einem sogenannten Repository, gespeichert sein. Erst dann ist die Verwaltung vom Code des Projektes vereinfacht. Hier kann nun der Programmierer die Dateien, die gerade bearbeitet werden, verändern, früheren Versionen der Dateien einsehen und gegebenenfalls wiederherstellen. Der Vergleich der verschiedenen Versionen ist ebenfalls möglich. CVS ist somit eine große Hilfe bei größeren Projekten, wenn verschiedene Programmierer an einem Codekomplex arbeiten. Dieses System erleichtert es, in so einem Fall, diesen Komplex von Daten zu koordinieren.

3.5.1. Arbeitsweise des CVS

Nun wird hier die Arbeitsweise vereinfacht dargestellt. Dies wird anhand der Abbildung 3.3 graphisch deutlich:

1. Zunächst holt ein Programmierer den aktuellen Stand aller Dateien eines Projektes aus dem Repository. Dies wird als checkout bezeichnet. Damit man die Versionen der zuletzt ausgecheckten Dateien erkennen kann, werden vom CVS Metadaten angelegt.

2. Nun kann der Programmierer Änderungen an einer oder mehreren Dateien vornehmen. Sobald die Änderungen beendet sind, werden diese Dateien geändert zurück ins Repository eingecheckt (checkin). Wenn mehrere Programmierer an den gleichen Dateien Änderungen vorgenommen haben, werden diese Konflikte manuell aufgeräumt.

Damit beim Abspeichern der Daten Platz gespart wird, verwendet CVS Delta-Kodierung und speichert nur die Differenzen zwischen den Dateiversionen anstatt der gesamten Version.

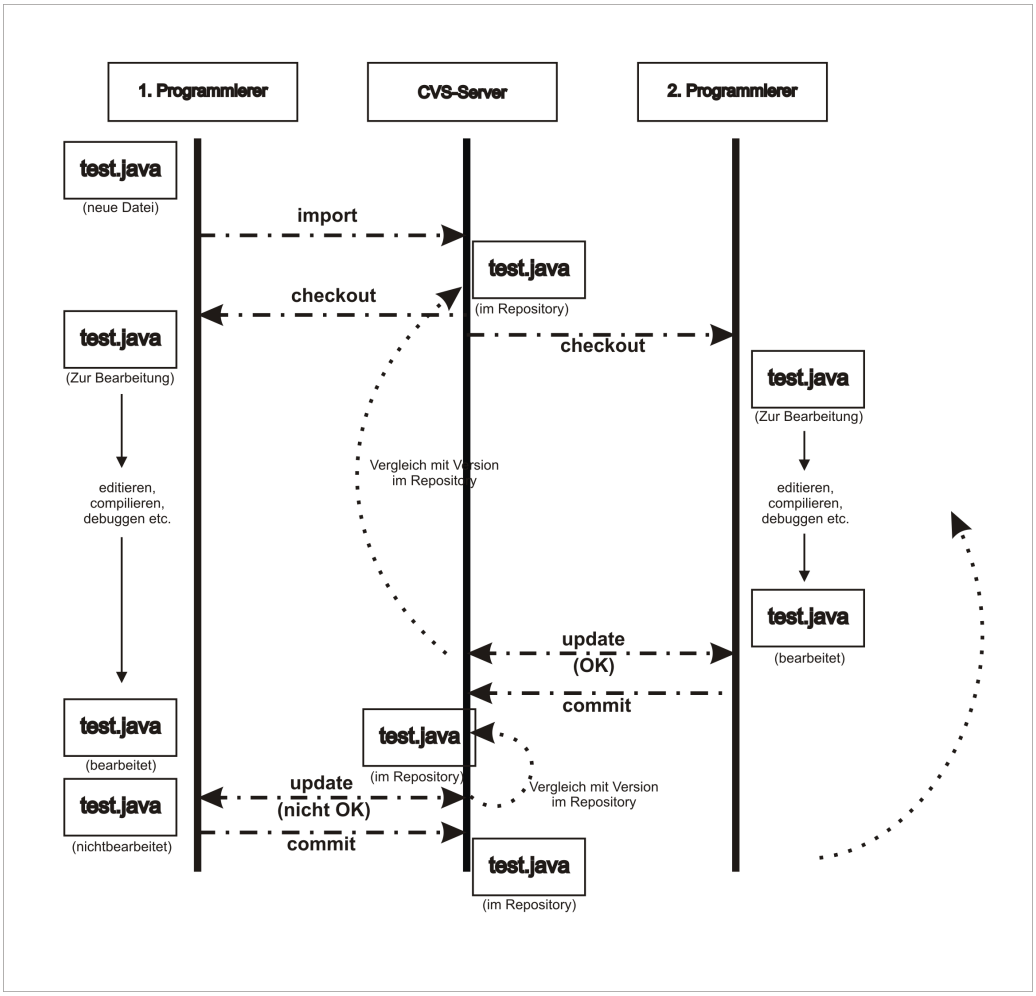

Abbildung 3.3.: CVS-Sitzung mit Versionskonflikt

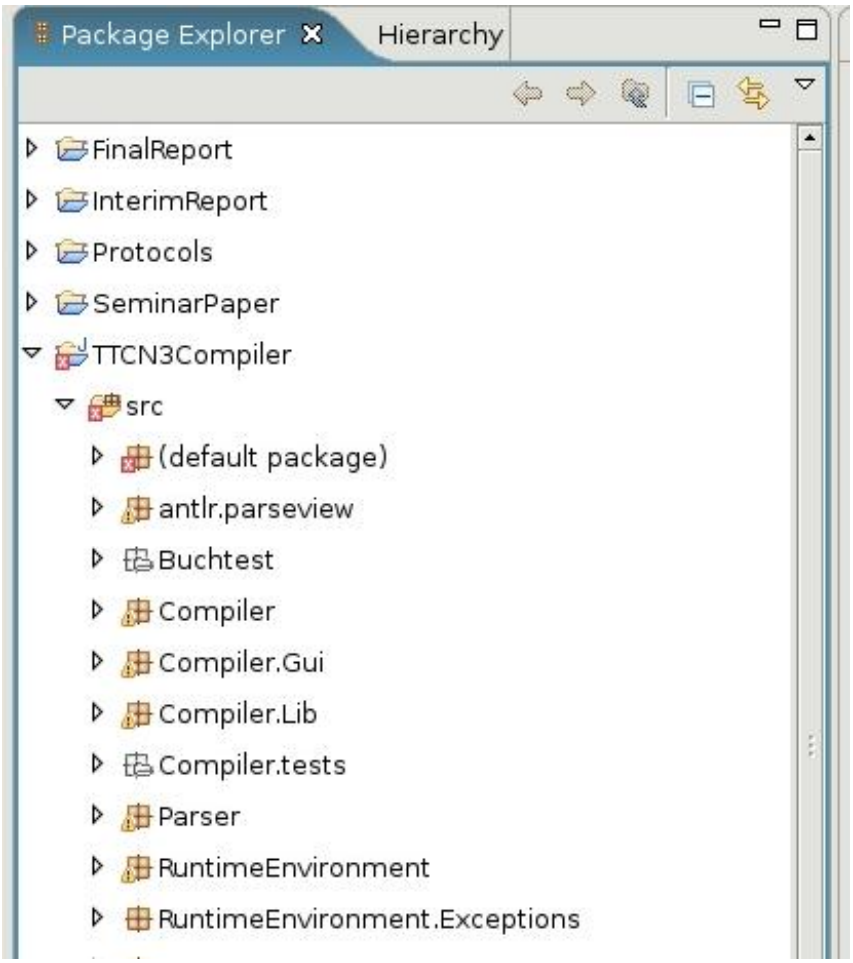

Abbildung 3.4.: Paketstruktur des Projektes. Ansicht unter Eclipse, Ausschnitt

3.5.2. Paketstruktur des CVS

Alle Dokumente, die für das Projekt benutzt werden, sind an einem zentralen und für alle Mitgliedern zugänglichen Ort des Lehrstuhls abgelegt. Der CVS-Server hat die folgende Struktur (Abbildung 3.4):

- FinalReport Hier befindet sich der Endbericht der Gruppe.
- InterimReport Hier befindet sich der Zwischenbericht der Gruppe.
- Protocols Hier sind alle Protokolle der Gruppensitzungen vorhanden, somit wird jedem Mitglied der Gruppe der aktuelle Stand des Projektes ermöglicht.
- SeminarPaper Hier befinden sich die Seminararbeiten der einzelnen Mitgliedern, die vor Beginn des Projektes präsentiert sind.
- TTCN3Compiler Hier befindet sich der programmierte Code.

3.6. jABC

Das **jABC Framework - kurz jABC -** (Abbildung 3.5) ist ein modulares, universelles und grafisches Modellierungswerkzeug für beliebige heterogene Softwaresysteme. Es ist eine auf Java

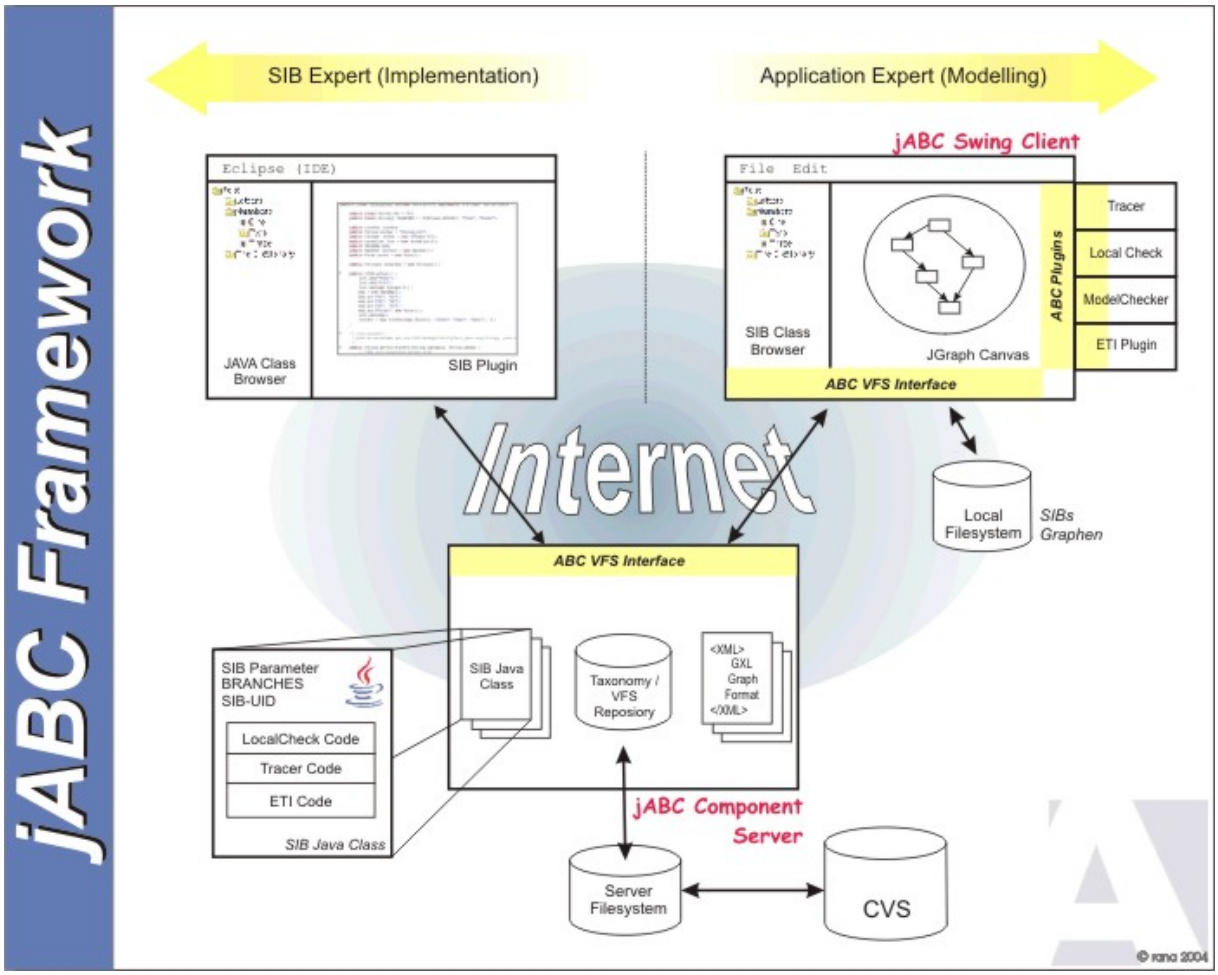

Abbildung 3.5.: jABC Framework

basierende Weiterentwicklung des Agent Building Centers, kurz ABC, welches 1993 am Lehrstuhl 5 für Programmiersysteme und Übersetzerbau der Universität Dortmund entwickelt wurde. Dabei greift es die wesentlichen Ideen des ABC auf und verbindet diese mit neuen Impulsen. Diese bestehen nicht nur aus der Weiterentwicklung von Bibliotheken und Programmiersprachen. Hierbei ist das jABC eine Symbiose aus einer studentischer Experementierplattform und einem kommerziellen Produkt.

Charakterisierend für das System ist die Verwendung einer grafischen Programmierschicht, auf der mittels vorher definierter Komponenten, den SIBs oder auch Service Independent Building Blocks, eine Anwendung in hierarchischer Graphen modelliert werden kann. Dieser Vorgang geht völlig ohne Programmierkenntnisse vonstatten und spricht somit sogenannte Anwendungsexperten an. Diesen Graphen werden mittels verschiedenster Plugins eine Semantik hinzugefügt, so dass eine Ausführungsschicht entsteht und die Ergebnisse somit direkt an dieser testbar sind.

Aktuelle Einsatzgebiete des jABC sind zum Beispiel:

- Entwicklung von WebServices
- Laufzeitumgebung für ausführbare Modelle

- Beschreibung und Ausführung von Workflows
- Modellanalysen mittels Temporallogiken

Das jABC ist gut zur Erstellung von Flussgraphen oder ähnlichem geeignet. Im Kontext unseres Projektes lässt es sich leicht modifizieren, so dass man ohne Programmierkenntnisse schnell kleinere Testfälle für TTCN-3 erstellen kann. Diese werden durch die Ausführungsschicht direkt ausgeführt. [4]

3.7. Bluetooth

Bluetooth ist ein Hersteller übergreifendes drahtloses Verbindungsprotokoll für Hardwaregeräte, das einen Verbindungsaufbau und Datenaustausch zwischen beliebigen Geräten erlaubt. Das Augenmerk liegt dabei auf der Störsicherheit der Verbindung und Verbindungsmöglichkeit einzelner Geräte unterschiedlichen Typs. Bluetooth ist aus der Idee entstanden, Verbindungskabel „zwischen Mobiltelefonen und Zusatzgeräten zu ersetzen“ [21]. In der heutigen Zeit hat Bluetooth eine enorme Beliebtheitssteigerung erfahren. So verbindet es inzwischen beispielsweise drahtlos Drucker und PCs, unterstützt ein Heimnetzwerk oder spricht ein Headset an.

Zwar ist das Bluetooth-Protokoll mit den ersten Geräten bereits vor über sechs Jahren vorgestellt worden. Doch dauerte es ein paar Jahre, bis der Markt das Potenzial diese Protokolls für sich entdeckte. Seitdem ist eine enorme Zahl von bluetoothfähigen Geräten auf dem Markt gekommen, viele Geräte erfüllen aber die Protokollspezifikation nicht vollständig. Ferner kann in Zeiten eines schnellen Technologiefortschritts eine Verletzung des Protokolls auftreten, einfach infolge der Weiterentwicklung des Geräts. Es erscheint also sinnvoll, ein effizientes Testtool zu verwenden, das das Einhalten des Protokolls, sowohl für andere Geräte prüft mit denen sich der eigens entwickelte Prototyp verbinden soll, als auch Spezifikationverletzungen des eigenen Geräts während seiner gesamten Entwicklung schnell feststellen kann.

4. Entwicklungsphase

In diesem Kapitel wird erläutert, wie die Projektgruppe das Problem angeht und welche Lösungsansätze gefunden werden müssen. Die Ergebnisse der einzelnen Arbeitsgruppen werden im Kapitel der Reihenfolge nach erläutert.

4.1. Unterteilung in Teilprobleme

Die Projektgruppe teilt das Problem in vier große Teilgebiete (siehe Abbildung 4.1). Diese werden dann in kleinen Gruppen untersucht und durch Lösungsansätze entsprechend gelöst.

Die vier Teilgebiete lauten wie folgt:

- Scanner und Parser
- Semantik-Checker
- Compiler vs. Interpreter
- Testgenerator

Diese Gruppen erhalten Teilziele, für die von Ihnen Lösungen erarbeitet werden. Auf den folgenden Seiten werden diese Ziele näher erläutert.

4.1.1. Scanner und Parser

4.1.1.1. Einleitung

Die Scanner Parser Gruppe hat die Aufgabe, einen Scanner und einen Parser für eine Untermenge von TTCN-3 zu entwickeln. Diese Untermenge von TTCN-3 wird vorgeschrieben. Als Ergebnis soll aus dieser Menge ein Syntaxbaum erzeugt werden. Der Syntaxbaum soll für die weitere Verarbeitung in den anderen Gruppen geeignet sein.

4.1.1.2. Scanner

Ein lexikalischer Scanner, auch Lexer genannt, ist ein Computerprogramm, das eine Eingabe in Folgen von logisch zusammengehörigen Einheiten zerlegt. Ein Scanner ist ein spezieller Teil eines Parsers, dieser leistet die Vorverarbeitung für den weiteren Parser. Dieser erkennt dabei innerhalb der Eingabe Schlüsselwörter, Bezeichner, Operatoren und Konstanten, so genannte Tokens. Diese werden mit ihrem jeweiligen Typ zurückgeliefert. Weil die Zerlegung in Tokens einer regulären Grammatik folgt, ist der Scanner als ein endlicher Automat zu sehen. Der nachfolgende Parser arbeitet dann auf den Token als atomare Symbole (auch Terminalsymbole genannt).
Die lexikalische Grundeinheit, die der Parser bearbeitet, ist ein Token (engl. “Tokens“). Man kann Tokens als Eingabesymbole (atomare Eingabezeichen) des Parsers verstehen. Beim Parsen wird jedes Token letztlich mit einem Terminalsymbol, einer Grammatik, verglichen. Nur

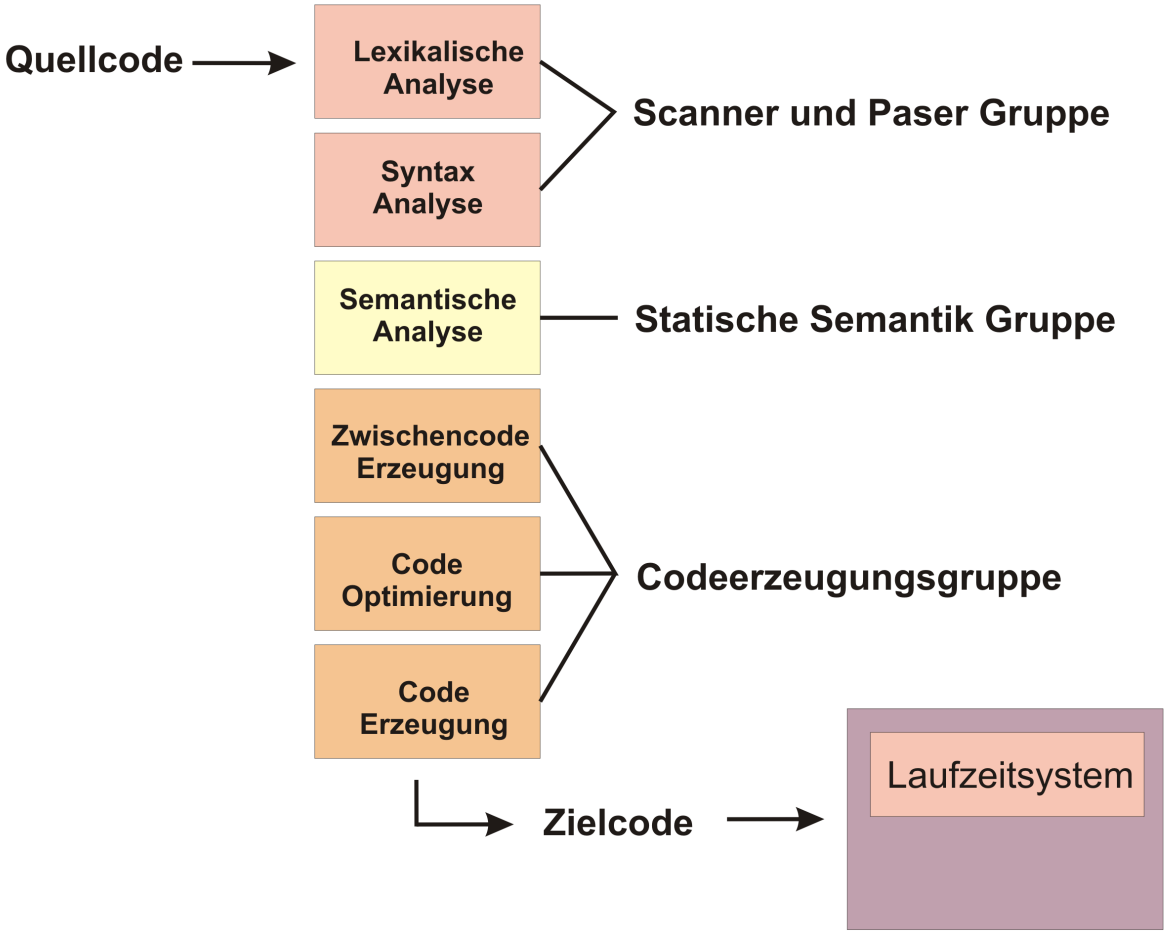

Abbildung 4.1.: Gesamtarchitektur

wenn das Terminalsymbol zu dem Token passt, kann die entsprechende Regel der Grammatik angewendet werden.

4.1.1.3. Parser

Ein Parser wird im Deutschen gelegentlich auch als “Zerteiler“ bezeichnet. Ein Parser ist ein Computerprogramm, das entscheidet, ob ein Eingabetext zur formalen Sprache einer bestimmten Grammatik gehört. Der Parser, als Implementierung eines abstrakten Automaten (meist realisiert als Kellerautomat), kümmert sich um die Grammatik der Eingabe und führt eine syntaktische Überprüfung der Eingangsdaten durch. Im Allgemeinen besteht seine Verwendung darin, einen Programmtext in eine neue Struktur zu übersetzen - z.. in einen Syntaxbaum. Die Hierarchie zwischen den Elementen wird durch diesen Syntaxbaum (auch Ableitungsbaum oder Parse-Baum bezeichnet) ausgedrückt. Dieser Baum wird anschließend zur Weiterverarbeitung der Daten verwendet; typische Anwendungen sind die semantische Analyse, Codegenerierung in einem Compiler oder Ausführung durch einen Interpreter.

4.1.1.4. Parser Generator

Anstatt einen eigenen Parser zu entwickeln, hat sich die Gruppe dazu entschlossen, verschiedene Parsergeneratoren zu betrachten. Mit dessen Hilfe soll dann ein Parser generiert werden.
Ein Parsergenerator ist ein Computerprogramm, das in der Lage ist, einen Parser zur grammatikalischen Analyse anderer Computerprogramme zu erzeugen. Es muss entschieden werden, welcher Parsergenerator eingesetzt wird. Wichtig ist, dass der erzeugte Parser als Zielsprache Java unterstützen soll. Er soll eine gute Dokumentation besitzen und einen möglichst gut lesbaren Code erzeugen.

Als Kandidaten sind folgende zu betrachten:

- YaCC (Yet Another Compiler Compiler) ist ein Parsergenerator, der für Unix Systeme entwickelt worden ist. Der YaCC braucht einen lexikalischen Scanner, der meist in Kombination mit diesem benutzt wird: Lex (Lexical Analyse Generator). Beide sind in C implementiert worden. Es wird eine Datei mit der Sprachsyntaxbeschreibung in einem der BNF ähnlichen Format verarbeitet. Als Ausgabe erhält man die erforderlichen Unterprogramme und Tabellen in einem C-Programm, welches dann von einem Compiler übersetzt wird. YaCC und Lex kann unter (siehe [15]) herunterladen werden.

- Cup ist ein LALR Parser Generator in Java. Cup (siehe [12]) wird meist in der Kombination mit JFlex (siehe [16]) eingesetzt. Hier ist der Einsatz auf Windows Systemen möglich.

- ANTLR (ANother Tool for Language Recognition) (siehe [18]) ist ein Parsergenerator, der aus der Grammatik einer Programmiersprache die Generierung eines kompletten Frameworks in Form von einem Parser, einem Lexer und darüber hinaus auch einem Syntaxbaumgenerator ermöglicht. ANTLR ist eine Weiterentwicklung von PCCTS und unterstützt als Ausgabesprachen Java, C++ und C#. Als Eingabe akzeptiert ANTLR EBNF-artige Grammatiken.

4.1.1.5. Entwicklung

Die Wahl der Gruppe für die Entwicklung fällt auf ANTLR. Der Grund für die Entscheidung, ist die gute Dokumentation und der übersichtliche und für den Menschen gut lesbare erzeugte Code. Ein weiteres starkes Argument für ANTLR besteht darin, dass er bereits erfolgreich zum Erzeugen eines Parsergenerators für TTCN-3 eingesetzt worden ist (siehe [25]). Die TTCN-3 BNF der ETSI Dokumentation (esfrm-e0187301v030101p) als Annex A (normative) (unter [3], auch [6]) ladbar) ist auf den Seiten 154 bis 171 zu finden. Auf den ersten beiden Seiten wird eine Liste für TTCN-3 spezielle Terminalsymbolen, eine Liste reservierter Wörter, Identifiers, Kommentare und die Konvention für die Syntax-Beschreibung definiert. Auf den folgenden 15 Seiten werden darüber hinaus 632 Regeln für die Produktionen aufgeführt.

Diese BNF wird nun, wie folgend beschrieben, für den Parsergenerator umgesetzt:

- Zu Anfang wird nur eine erste Core Language vereinbart und eine reduzierte Form der BNF erstellt. Hierfür wird ein Lexer und ein Parser mithilfe von ANTLR erzeugt. Nach der erfolgreichen Realisierung und dem Testen der ersten vereinbarten Core Language, wird die Implementierung des kompletten Sprachumfangs der BNF in Angriff genommen.
- Parallel dazu werden Testfälle geschrieben. Mit diesen soll jede mögliche Regel, mit möglichst geringen Bedingungen pro Testfall, überprüft werden, um die Fehlersuche zu erleichtern. Die Testfälle werden in drei Kategorien unterteilt:

  1. Negative(Fälle, die nicht laufen dürfen.),
  2. Positive(Fälle, die laufen müssen.),
  3. Valide(Fälle, die semantisch korrekt sind.)

- Um zu vermeiden, dass jeder Testfall manuell ausgeführt werden muss, wird ein JUnit Tester angelegt. Die auszuführenden Tests werden automatisch in allen Unterverzeichnissen eines angegebenen Verzeichnisses gesucht. Nach der Ausführung wird dann eine strukturierte Fehlerausgabe geloggt.

4.1.2. Semantik Checker

4.1.2.1. Einleitung

Kommen wir nun zu den Erkenntnissen der zweiten Gruppe. In der von uns festgelegten Ordnung hat diese Gruppe die Aufgabe, Dateien mit dem TTCN-3-Code semantisch zu analysieren und auf Korrektheit zu prüfen. Ihre Arbeit beginnt, sobald die Parsergruppe den Syntaxbaum vollständig generiert hat, so dass diese Gruppe nicht weiß, durch welche Dateien die Informationen entstanden sind; kann aber, wie in richtigen Compilern, direkt mit dem Syntaxbaum arbeiten, um die Korrektheit des Codes sicher zu stellen. Die Semantikprüfung wird durchgeführt, nicht nur um festzustellen, dass die Syntax des Programms korrekt ist, sondern auch dass verwendete Ausdrücke Sinn geben. Die Zahl, der an eine Funktion übergebenen Parameter, beispielsweise, muss der von der Funktion erwarteten entsprechen. Den größten Teil an der Semantikprüfung nimmt die Typüberprüfung ein: Den Typen der Ausdrücke bestimmen und Inkonsistenzen melden, wie z.. Vergleich eines booleschen Ausdruckes mit einem String oder Übergabe eines Arguments des falschen Typs an eine Funktion.

4.1.2.2. Semantische Prüfung

Die Semantische Prüfung geht von der Typüberprüfung bis zur Deklarationsdublizierung. Die wichtigste Prüfung ist die bereits oben beschriebene Typenprüfung, weil sie die Verifikation der deklarierten Typen in Zuweisungen und vielen anderen Prüfungen in einer Beziehung mit der Subtypisierung bringt. Jetzt schließt sie auch einige weitere Dinge ein, wie die Typenrückgabe der Funktionen oder Korrekturen der Variablen- oder Konstantendeklaration, sowie deren Verwendung im ganzen Programm mit ihren Operationen, die sie nutzen können.

4.1.2.3. Implementierung

Die Gruppe hat sich entschieden, die semantische Prüfung in zwei Schritte zu unterteilen. Idealerweise beginnt man mit dem Aufbau der Tabelle, wenn die Scanner und Parsergruppe den Syntaxbaum erstellt hat, wie es beinahe jeder andere Compiler implementiert hat, um Zeit zu sparen. Kommen wir nun zum Ablauf der Implementierung.

Im ersten Schritt wird die Deklaration von Variablen und Typen überprüft. Es wird kontrolliert, ob jede deklarierte Variable einen gültigen Typ hat oder ob keine Bezeichner mehrfach in Geltungsbereichen deklariert sind.

Im zweiten Schritt werden Zuweisungen, wie z.. Funktionsaufrufe, Port- und Komponentfunktionalität und andere überprüft. Beide Schritte können nicht zusammen erledigt werden, weil die Tabellenstruktur bekannt sein muss, um die Analyse fortzusetzen und mit korrekten Typen der deklarierten Funktionen, Konstanten und Altsteps zu arbeiten.

4.1.2.3.1 Deklaration in TTCN-3

Definitionen in TTCN-3 folgen ganz eigenen Regeln. Anders als in vielen anderen Programmiersprachen lassen sie sich nicht in einem Schritt behandeln. Die sehr freie Reihenfolge, in der Definitionen erlaubt sind, verhindert dies jedoch: Alles kann scheinbar ohne Ordnung in beliebiger Reihenfolge definiert und benutzt werden. Bei der Auslegung des Designs des Semantik-Checkers, ist diesem Umstand besonders Sorge zu tragen. Eine Variable kann z.. benutzt werden, bevor sie definiert ist. Weiterhin kann sie einen nicht definierten Typ haben. Es wird dazu der ganze Compiler gestoppt, wenn ein Fehler gefunden wird, da, wie es in C-Sprachen geschehen ist, der erste Fehler das Auftreten der restlichen verursacht (stellen Sie sich einen Typ vor der nicht definiert, aber benutzt ist, es kann den Sinn des gesamten Programm ändern).

4.1.2.3.2 Fehlerbehandlung

Kommen wir nun zur Struktur der Fehlermeldungen. Es wird dieselbe benutzt, die von der Scanner und Parsergruppe erstellt wurde, eventuell wird dort die ein oder andere Struktur ergänzt. Am Anfang ergaben sich Probleme bei der Struktur, da nicht nur der Dateiname benötigt wird, sondern auch die Zeilennummer und die Zeichenposition, an der der Fehler aufgetreten ist.

Die Problematik, um an diese Informationen zu kommen, besteht darin, dass man nicht nur die Blätter des Syntaxbaumes durchsuchen muss, um an diese zu kommen, sondern dass auch viele Fehler vorhanden sein können, die nicht beim durchsuchen des Baumes zu finden sind (z.. Deklarationsreihenfolge).

Damit dieses Problem sinnvoll gelöst werden kann, haben wir uns für das Zweischrittdesign entschieden, das in diesem Fall gegenüber dem Einschrittdesign wesentlich einfacher zu händeln ist.

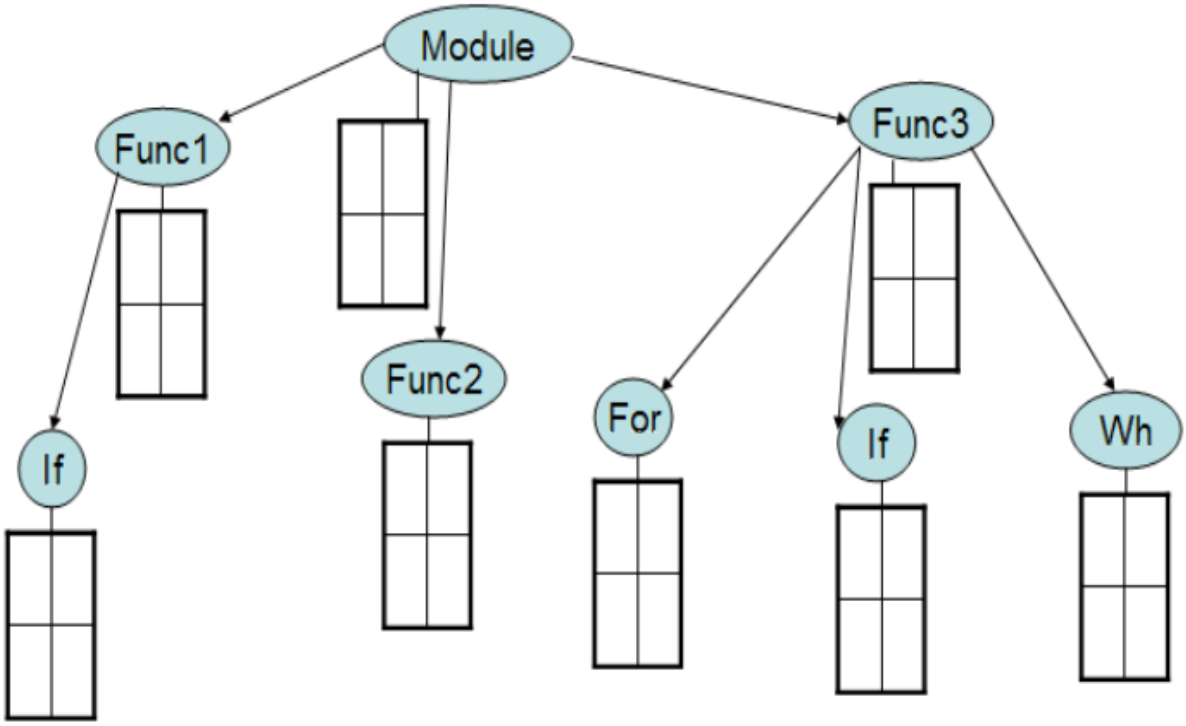

Abbildung 4.2.: Übersicht einer einfachen Tabellenstruktur in einem simplen Programm

Der Ablauf ist nun wie folgt:

- Im ersten Schritt wird der Syntaxbaum durchlaufen und die Symboltabelle erstellt und gefüllt.
- Im zweiten Schritt wird der Baum dann auf Löcher und andere Fehler getestet.

4.1.2.3.3 Umsetzung des Designs des Semantik Checkers

Nach Recherche der bereits existierenden Semantik-Checker wird beschlossen, den Semantik Checker von Grund auf neu zu gestalten, da es keine passenden Lösungen für das Ziel der statisch semantischen Überprüfung von TTCN-3 Code gibt. Dies versetzt uns in die Lage, volle Kontrolle über die Implementierungsdetails und die Funktionalität zu erlangen. Bereits existierende Tools erschienen uns nicht geeignet, unseren Anforderungskatalog zu erfüllen.

Zu Beginn der Entwicklung wurden die Produktionsregeln der Grammatik und einige Codebeispiele betrachtet. Hieraus erwuchs eine Arbeitsbasis mit einer Symboltabelle ohne Unterscheidung des Geltungsbereiches. Jeder Eintrag enthielt Informationen über den TTCN-3-Typ und eine Zahl, welche den Geltungsbereich in einer einfachen Weise darstellte.

Bei genauerer Prüfung stellte sich jedoch heraus, dass es unzureichend ist, die Geltungsbereiche der einzelnen Typen derart zu kodieren, da die einzelnen Knoten des Syntaxbaumes nicht genügend Informationen zur genauen Typzuordnung enthalten. Deshalb wurden Anpassungen dergestalt durchgeführt, dass die Geltungsbereiche nun einer Baumstruktur nach geordnet sind. Ähnliche Ansätze finden sich auch in anderen Compilerprojekten.

Beispielsweise würde ein einfaches Programm, wie im Beispiel 4.1, eine Struktur produzieren, wie in Abbildung 4.2.

```
module TestModule  {
   : //entrances of the first level of the table
   function func1(...)returns ...{
       : //entrances for the func1 level of the table
       if(...){
           : //entrances for the if level table in func1
       }
   }
   function func2(...)returns ...{
       : //entrances for the func2 level of the table
   }
   function func3(...)returns ...{
       : //entrances for the func3 level of the table
       for(...){
           : //entrances for the if level table
       }
       if(...){
           : //entrances for the if level table
       }
       while(...){
           : //entrances for the while level table
       }
   }
   : //entrances of the first level of the table
}
```

Beispiel 4.1: TTCN3 Quelltext

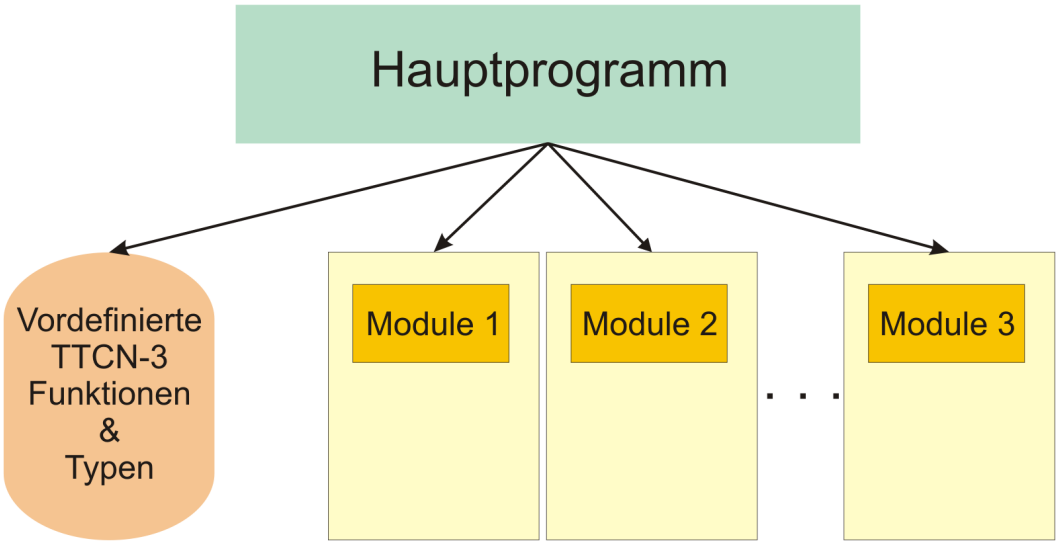

Abbildung 4.3.: Struktur einer einfachen Tabelle für ein einfaches Programm mit einem einzigen Modul, welches ein anders importiert

Wie aus der Abbildung ersichtlich ist, gibt es einen Wurzelknoten, der die Informationen des Hauptknotens enthält. Wir bezeichnen diesen als *Initialknoten*. Sobald eine Struktur deklariert ist, die den Geltungsbereich ändern kann (wie etwa eine Funktion), wird ein neuer Knoten in der Tabelle erstellt. Dieser enthält seine eigene Symboltabelle, die Elemente auf gleicher Ebene des Geltungsbereiches beinhaltet. Um zu vermeiden, dass inkorrekte Knoten hinzugefügt werden, müssen Änderungen am Geltungsbereich berücksichtigt sein. Eine wünschenswerte Eigenschaft der Struktur ist es, gute Kontrollmöglichkeiten über Geltungsbereiche für darauf bezogenen Elemente zu liefern.

Das Hinzufügen der Imports haben dazu beigetragen, dass sich die Struktur etwas geändert hat. Das Modul ist nun eine Verzweigungen, die abhängig vom Hauptprogramm ist. Diese Verzweigungen beinhalten alle vordefinierten TTCN-3 Funktionen, sowie alle Module, die in diesem Programm benutzt werden.

Wenn wir in das vorherige Beispiel diese Anweisung *import module2 all* in eine Zeile einfügen, sieht die Struktur folgendermaßen aus:

Jedes Modul in der Abbildung 4.3 würde eine Tabellenstruktur besitzen, wie in der Abbildung 4.2 zu sehen ist, mit all den Deklarationen und deren Geltungsbereichen. Die vordefinierten TTCN-3 Funktionen (wie *str2int*, *int2float*, *length*, etc.) und vordefinierten Typen (wie *integer*, *boolean*, *default*, etc.) sind in der Tabelle entsprechend des Hauptprogramms zu Beginn des Semantik-Checkers eingefügt, wenn die Struktur der Symboltabelle angefertigt wird. Dieses bringt auch den Vorteil der Platzersparnis.

Wie in der Abbildung 4.2 dargestellt, enthält jeder Knoten der Baumstruktur eine Hashtabelle. In jeder Hashtabelle sind Informationen über die Deklarationsnamen und den dazugehörigen Typen hinterlegt. Genauere Angaben hierzu geben wir im nächsten Kapitel 5.2. Im Initialknoten werden deklarierte Typen, Funktionen und Konstanten des Modul-Deklarationsteils vorgehal-

ten. Strukturen, welche Änderungen am Geltungsbereich bewirken, führen zu neuen Knoten in der Tabelle. Informationen über neue Variablen oder Konstanten des modifizierten Geltungsbereiches werden dort gespeichert. Diese Struktur erlaubt uns, duplizierte Variablen in einer einfachen Weise zu ermitteln und perfekte Kontrolle über die Bereiche zu erlangen. Hierzu reicht eine einfache Suche nach dem Namen oder Typ der deklarierten Elemente in der Hashtabelle des aktuellen Knotens und den übergeordneten Knoten bis zur Wurzel. Im Falle der obigen Beispieltabelle muss vor dem Einfügen einer neuen Variable unter **for** der **Func3** erst nach einer existierenden Deklaration mit demselben Typ in der Symboltabelle der Funktion selbst oder des Initialsknotens gesucht werden. Zudem wird kontrolliert, ob sie Bezug auf einen bereits deklarierten Typ nimmt. Ist dies der Fall und ist der Bezeichner noch nicht vergeben, so wird die Deklaration im entsprechendem Knoten vermerkt. Ansonsten wird ein Fehler gemeldet (Bezeichner bereits vorhanden) oder in einer Lochtabelle (Typ nicht vorhanden) eingetragen.

Die Konstruktion der Symboltabellenstruktur ergab sich aus der Orientierung der Symbole. Ein erstes Design für die Tabelle war sehr einfach und nutzte zur Konstruktion eine Menge von Klassen, die sich einander abhängig vom aktuellen Knoten im Syntaxbaum aufrufen. Hierbei analysiert jede Klasse einen Teilbaum entsprechend dem momentan in der Tabelle eingetragenen Typ. Es wurden Informationen wie Name, Typ und Bezeichner (Funktion, einfacher Typ oder Struktur etc.) für alle Typen vorgehalten. Dieser Entwurf kam durch Standardisierung über alle Typen mit einer einzigen Datenstruktur aus, was uns sehr ansprechend erschien. Beim Versuch, fortgeschrittene Typen, wie innere Records und Untertypen in diese Schema zu integrieren, traten jedoch Schwierigkeiten mit diesem Ansatz auf. Um derartige Strukturen im erweiterten Modell zu berücksichtigen, ist es notwendig, Informationen abzuspeichern, die an vielen Stellen nicht relevant sind. Dies hat uns dazu bewogen, eine Neukonzeption durchzuführen, die für jede TTCN-3 Deklaration eine unterschiedliche Struktur vorsieht. Ein Vorteil, der daraus entsteht, ist eine vereinfachte Wartbarkeit: Erweist sich, dass für einen Deklarationstyp Anpassungen notwendig sind, so können diese erfolgen, ohne Seiteneffekte auf andere Deklarationstypen befürchten zu müssen.

Die Deklarationsreihenfolge in TTCN-3 führt zur Notwendigkeit einer Lochtabelle, da in Deklarationen Typen benutzt werden können, die selbst noch nicht deklariert sind. Das Erstellen dieser Lochtabelle erwies sich als größte Herausforderung bei der Berücksichtigung von Geltungsbereichen. Anfangs wurde versucht, das Aufbauen einer Lochtabelle zu vermeiden, indem die Symboltabellen mit zusätzlichen Informationen versehen wurden. Dieser Ansatz erzwingt aber eine anschließende Suche in allen Tabellen, um fehlende Typen aufzuspüren. Ein anderer Versuch war das schrittweise Einfügen von Konstanten, Variablen und Deklarationen, die darauf aufbauen. Auch dies entpuppte sich als nicht realisierbar, weil hierfür mehrmals durch den Syntaxbaum gegangen werden muss. Des weiteren können einige Typen andere nicht deklarierte Typen aufrufen. Ein Beispiel ist 4.2.

```
type set of MyRecord1 MySet; type record
MyRecord1{
    const MyRecord2 something := {};
}
type record MyRecord2{
    :
}
```

Beispiel 4.2: Nichtdeklarierte Typen

Diese Struktur kann nicht durch eine vorangestellte Betrachtung der Typen analysiert werden,

da im Syntaxbaum als erstes *MySet* erscheint, das jedoch von einem nicht deklarierten Typ ist.

Diese nicht deklarierten Typen werden durch eine Struktur berücksichtigt, in der alle Variablen, Konstanten, Funktionen und Typdeklarationen gespeichert werden, deren Typ noch nicht bekannt war, als wir sie analysiert haben. Wenn der erste Schritt (Ausfüllen der Tabellen) abgeschlossen ist, wird überprüft, ob alle referenzierten Typen erstellt wurden. Verbleiben Typen in der Lochtabelle, so können wir sicher sein, dass ein Fehler vorliegt. Diese Prüfung ist die letzte, die das voranschreiten des Kompilierungsvorganges aufhalten kann. Alle nachfolgenden Diskrepanzen sind von nachrangiger Bedeutung.

Für den ersten und den zweiten Schritt wird jede Deklarationsart in einer eigenen Klasse gekapselt, so dass jede einzelne hier von selbständig arbeitet, ohne andere Klassen zu beeinflussen. Der Aufruf erfolgt von einer generischen Klasse, die den Fluss der Semantikprüfung kontrolliert, indem sie sich durch den Syntaxbaum bewegt und notwendige Analysen anstößt.

Nach der Erstellung des Designs wurde Schritt eins für alle TTCN-3 Deklarationen vollzogen. Jeder Deklaration erfuhr einen eigenen Implementationsschritt, da der Syntaxbaum für jede Deklaration eine eigene Struktur aufweist. Hierbei entstand für jede Deklaration eine eigene Methode, die gegebenenfalls Unterstrukturen an die jeweilig zuständigen Methoden zur Bearbeitung übergibt. Nach Durchführung des ersten Schritts sind alle betroffenen Strukturen ausgefüllt (Symboltabelle und Lochtabelle).

Im zweiten Schritt werden die Teile des Programms analysiert, die nicht in der Deklarationen sind - wie etwa Zuweisungen oder Operationen. Die grundsätzliche Prüfung hierbei ist, ob jede Variable oder Konstante entsprechend ihrem Typ verwendet wird. Dies bedeutet, dass beide Seiten einer Zuweisung vom verträglichen Typen sind. Ähnliches gilt für Operationen und Funktionen. Bei Ports und Komponenten muss geprüft werden, ob Funktionen, die sie benutzen, vorhanden und korrekt sind. Dies ist der aufwändigste Teil der Semantikprüfung.

Für den zweiten Schritt wird nochmals der Syntaxbaum durchlaufen:

1. Man geht den Syntaxbaum entlang der gefundenen kritischen Knoten.
2. Falls ein kritischer Knoten entdeckt wird, so wird er rekursiv analysiert, um auch einen anderen kritischen Knoten zu finden.
3. Es gibt einige spezielle Knoten innerhalb der Gruppen der kritischen Knoten. Diese speziellen Knoten müssen Funktionen aufrufen, die eine eigene Analyse durchführen. Einige Beispiele bezüglich der speziellen Knoten sind:
   - `VarInstance` Knoten (welches einen Aufruf im zweiten Schritt verursacht)
   - `ForStatement` Knoten
   - `IfConstruct` Knoten (die sowohl einen rekursiven Aufruf verursachen mit der inneren Analyse des Codes, als auch eine spezielle Kontrolle für die semantische Genauigkeit der Bedingung fortsetzen)
   - `Funktion` Aufruf
   - `Altstep` Aufruf (ruft im zweiten Schritt die Funktion des Altsteps auf)
4. In den speziellen Knoten können zwei Handlungen durchgeführt werden. Zuerst soll eine direkte Ausführung des zweiten Schritts der Deklaration durchgeführt werden. Als nächstes wird eine spezielle Funktion des Programms aufgerufen. Diese Funktionen werden für die Überprüfung spezieller Funktionsaufrufe, oder für Return-Anweisungen, Imports oder Guards, benötigt.

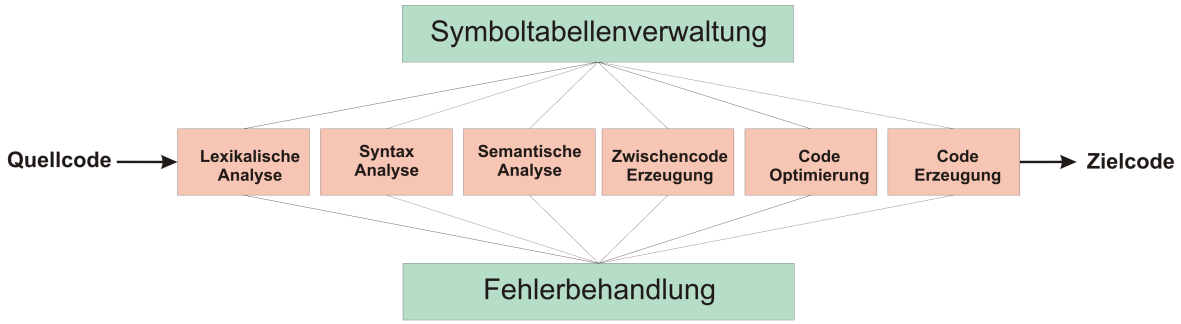

Abbildung 4.4.: Compileraufbau

4.1.3. Compiler

4.1.3.1. Einführung

Die Code-Erzeugungs-Gruppe hatte anfangs die Aufgabe, innerhalb von drei Wochen zu evaluieren, welcher der beiden Ansätze 'Compiler vs. Interpreter' von der PG umgesetzt werden soll. Die Gruppe hat eine Präsentation erstellt, die beide Ansätze darstellt und vergleicht. Für die Entscheidung wurde neben der Implementierungskomplexität auch die realisierbare Benutzerfreundlichkeit untersucht.

4.1.3.2. Der Compiler

4.1.3.2.1 Die Aufgabe des Compilers

Ein Compiler ist erstmal ein Übersetzer. Dieser ist in den meisten Fällen ein hoch komplexes Programm, das den Quellcode einer „höheren" Programmiersprache in (optimierten) Code einer Zielsprache übersetzt. Diese genannte Zielsprache ist oft eine Maschinen- oder Assemblersprache. Um dieses Ziel eines optimierten Zielcodes zu erreichen, besitzt ein Compiler mehrere Untereinheiten die im folgenden Abschnitt erklärt werden.

4.1.3.2.2 Aufbau des Compilers

Im allgemeinen besteht ein Compiler aus 8 Untereinheiten, Abbildung 4.4, die miteinander Verknüpft sind.

Bei der Erzeugung des Zielcodes durchläuft der Quellcode folgende 6 Stufen.

1. Die **lexikalische Analyse** oder auch 'Scanning' genannt, gruppiert den Quellcode in lexikalische Grundeinheiten und entfernt die Leerzeichen.

2. Die **syntaktische Analyse** auch 'Parsing' genannt, ist eine hierarchische Analyse des Quellcodes. Sie besitzt die Aufgabe, die Symbole des Quellcodes in grammatikalische Sätze zusammen zufassen. Diese Sätze werden üblicherweise durch einen sogenannten Parse-Baum dem Compiler zur Verfügung gestellt.

3. Die **semantische Analyse** überprüft den Quellcode auf grammatikalische Ausdrucksfehler. Sie benutzt den vorher erstellten Parsebaum zur Bestimmung und Trennung von Operatoren und Operanden von den Ausdrücken und Anweisungen, diese werden dann auf die Einhaltung der Spezifikationen der Quellsprache überprüft.

4. Die **Zwischencodeerzeugung** wird nicht bei jedem Compiler genutzt. Diese erzeugt nach der Analysen erst einmal eine Zwischendarstellung des Quellcodes. Dieser so genannte Zwischencode muss zwei Eigenschaften besitzen. Er sollte zum Einen leicht zu erzeugen sein und zum Anderen leicht in den Zielcode zu übersetzen sein.

5. Die **Code-Optimierung** hat eine einfach definierte Aufgabe, die mitunter schwierig zu lösen sein kann. Ihre Aufgabe ist es einfach nur, den Zwischencode vor der endgültigen Übersetzung in die Zielsprache zu optimieren.

6. Die **Code-Erzeugung** ist die letzte Phase. Sie erzeugt aus dem optimierten Zwischencode den Zielcode.

Parallel zu diese Einheiten gibt es noch folgende zwei Einheiten.

1. Die **Symboltabellenverwaltung** ist in erster Linie eine Datenstruktur, in der die benutzten Bezeichner und ihrer Attribute, wie z.. Speicherbedarf und Gültigkeitsbereich gespeichert werden.

2. Die **Fehlerbehandlung** dient dem Erkennen und Melden von Fehlern im Quellcode. Im Idealfall läuft sie parallel ab und stoppt nicht gleich bei dem kleinsten Fehler, sondern versucht kleinere Fehler schon selbst zu analysieren und zu korrigieren.

4.1.3.2.3 Vor- und Nachteile des Compilers

Vorteile Ein Vorteil ist die hohe Ausführungsgeschwindigkeit des erzeugten Programms. Ein weiterer Vorteil besteht darin, dass der Quellcode vor Manipulationen geschützt ist, da er zur Programmausführung nicht benötigt wird (besserer Urheberschutz). Des Weiteren stellt die Möglichkeit, der einfachen Verwendung vorgefertigter Teillösungen aus Programmbibliotheken, einen weiteren Vorteil dar (Modulare Programmierung), und letztendlich pro Durchlauf werden viele, im besten Fall alle Fehler erkennbar.

Nachteile Ein Nachteil ist die schwierige Fehlersuche, da bei der Compilierung die symbolischen Bezüge zum Quelltext verloren gehen. Außerdem ist die Programmentwicklung durch die zusätzlichen Arbeitsschritte, wie Compilierung ,Link und Load erschwert. Letztendlich besteht beim Compiler noch der Nachteil, dass er nicht dialogfähig ist.

4.1.3.3. Der Interpreter

4.1.3.3.1 Die Aufgabe des Interpreters

Ein Interpreter ist ein Programm, das den Quelltext eines Programms auf der Zielmaschine ausführt. Diese erfolgt Zeile für Zeile und wird in ein Maschinencode übersetzt, der dann anschließend vom Rechner abgearbeitet wird.

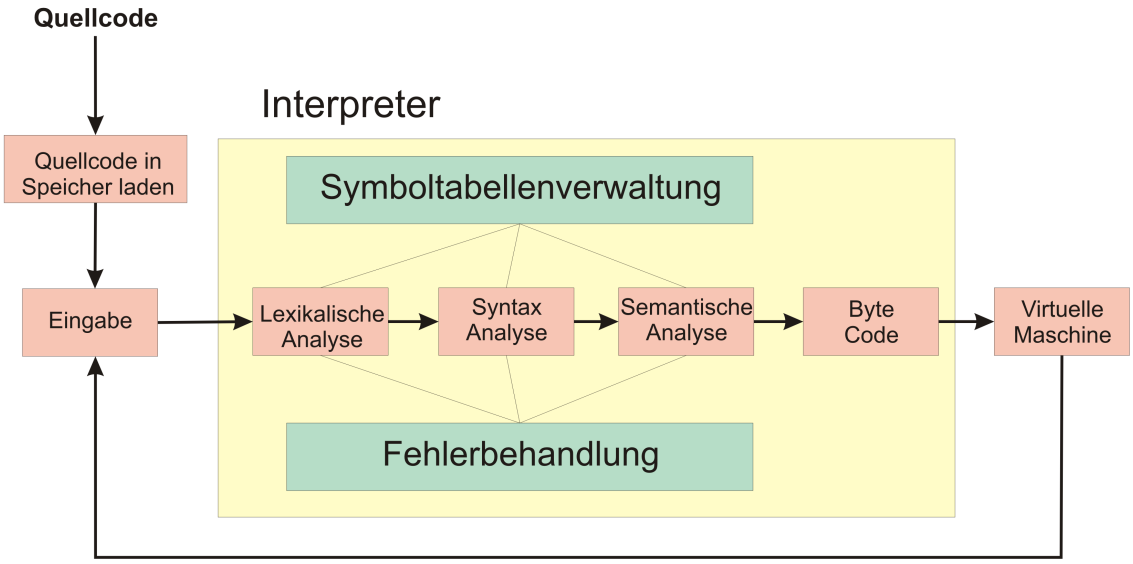

Abbildung 4.5.: Interpreteraufbau

4.1.3.3.2 Der Aufbau des Interpreters

Bei einem Interpreter, Abbildung 4.5, wird der Quellcode, im Gegensatz zu einem Compiler, befehlsweise übersetzt und ausgeführt. Dagegen muss bei einem Compiler der Quellcode, vor der Ausführung, in Maschinencode übersetzt werden. Im Bezug auf die Arbeitsweise eines Interpreters analysiert dieser den Quelltext. Er wird dabei nicht vollständig in eine Maschinensprache übertragen, sondern unmittelbar ausgeführt oder in ein Zwischencode übersetzt. Falls der Quelltext in ein Zwischencode übersetzt wird, ist hierfür ein *Bytecode-Interpreter* erforderlich, diesen nennt man *Virtuelle Maschine*. (vgl. Java Virtual Machine)
Im Hinblick auf die Arbeitsweise eines Interpreters, werden folgende Stufen abgearbeitet:

1. Der Quellcode wird zu Beginn in den Speicher geladen.

2. Dieser Quellcode wird nun vom Interpreter in 4 Stufen analysiert. Die ersten drei Stufen entsprechen dem Aufbau des Compilers (Abb. 4.5). Anschließend wird aus dieser Analyse der Bytecode generiert.

3. Für die erste Prozedur eines Interpretier-Vorgangs ist eine Eingabe bereitgestellt.

4. Der entstandene Bytecode wird nun von der **Virtual Machine** eingelesen und ausgeführt.

Sobald die erste Prozedur ausgeführt wird, startet der nächste Vorgang im Ablauf des Interpreters. Es werden erneut Eingabedaten eingelesen. Dieser Prozedurvorgang wiederholt sich immer wieder, solange der Quellcode abgearbeitet wird.

Kurzgefasst analysiert und übersetzt der Compiler das Programm einmal **vor** der Ausführung. Der Interpreter hingegen bei jeder Anweisung **während** der Ausführung.

4.1.3.3.3 Vor- und Nachteile des Interpreters

Vorteile Einer der größten Vorteile des Interpreters ist, dass sich Programme ohne vorgehende Kompilation sofort ausführen lassen. Somit können eventuelle Änderungen an einem Programm, während der Ausführung des Programms, ohne großen Aufwand vorgenommen werden. Dieses hat zur Folge, dass Fehlerbehandlungen flexibel durchgeführt werden können. Es wird dadurch sogar eine inkrementelle Entwicklung ermöglicht, denn der Interpreter benötigt keinen schwer verständliches Maschinen-Code. Dieser ist dann bei dessen Ausführung dementsprechend leichter nachvollziehbar.
Zuletzt hat ein Interpreter noch den Vorteil. Er ist maschinenunabhängig implementierbar, wie die Java Virtual Machine.

Nachteile Ein Interpreter besitzt im Gegensatz zu seinen Vorteilen auch gravierende Nachteile.

Die Ausführung eines Programms dauert bei einem Interpreter länger, als die Ausführung compilierter Programme, weil z.. eine Ausführung einer Schleife, die 50 mal abgearbeitet wird, diese auch 50 mal durchlaufen werden muss. Dieses führt so zu einer erhöhten Laufzeit des Programms.
Des Weiteren können bei einem Interpreter die Syntaxfehler, erst zur Laufzeit des Programms, aufgespürt werden, und nicht wie beim Compiler bei der Übersetzung.

4.1.3.4. Fazit - Compiler vs. Interpreter

Bei der Betrachtung der vorherigen Kapitel ist es ersichtlich, dass in unserer Situation die Entscheidung nur auf den Compiler fallen kann.

Der einzige Vorteil des Interpreters liegt in der Möglichkeit des guten am Source-Code nahen Debugging (oder hochdeutsch, der Fehlersuche). Denn hier ist ganz klar der Nachteil für den Compiler, da man für ihn eine Kontrollstruktur der Breakpoints im Zielcode bedenken muss. Ansonsten ist der Interpreter eine extrem Ressourcen verschwendende Möglichkeit. Dieses könnte bei komplexen Programmen zu Konflikten führen, da zusätzlich auch noch die Java Machine als Interpreter laufen muss.

Deswegen fällt unsere Wahl auf den Compiler. In unserem Projekt wird die Arbeit durch den bestehenden Java Compiler erheblich erleichtert, denn es wird lediglich aus dem Parsebaum und der Variablentabelle einen Zwischencode erstellt. Die Optimierung des Zwischencodes und die Erstellung des Bytecodes wird dann vom Java Compiler übernommen.

Der Nachteil, der schlechten Fehlerbehandlung, muss dann dafür in Kauf genommen werden. Eventuell kann dann die gewonnene Zeit dafür genutzt werden, einen Debugger zu entwickeln. Es wurde also auf die Empfehlung der Gruppe und nach einer Diskussion in der Projektgruppe beschlossen, einen TTCN-3-Java-Compiler zu programmieren.

5. Entwurf und Implementierung

5.1. Scanner - Parser

5.1.1. Der Lexer und der Parser

5.1.1.1. Die Eingabedateien für den Parsergenerator ANTLR

Für den Parser bedient sich die Gruppe dem Parsergenerator ANTLR. Dieser Generator benötigt zur Generierung von Lexer und Parser je eine Eingabedatei, in unserem Falle `T3Lexer.g` und `T3Parser.g` benannt. In der Eingabedatei für den Lexer wird definiert, wie aus dem Zeichenstrom der Eingabe Tokens extrahiert werden. In der Eingabedatei für den Parser werden Regeln festgehalten, mit denen aus den Tokens der abstrakte Syntaxbaum (AST) erzeugt werden kann.

Wie man unter Zuhilfenahme der TTCN-3-Spezifikation erkennen kann, entsprechen die einzelnen Regeln der BNF der Sprache. Allerdings mussten die Regeln an die ANTLR-Syntax angepasst werden, was halbautomatisch erfolgen konnte. Weiterhin sind semantische Prädikate notwendig, die Mehrdeutigkeiten aufgrund des beschränkten Lookaheads beseitigen und das Parsen beschleunigen.

5.1.1.2. Aufruf von Lexer und Parser

Durch die Wahl von ANTLR als Parsergenerator, wurde implizit dessen API übernommen. ANTLR generiert eine Lexer- und eine Parserklasse - in unserem Fall die Klassen `T3Lexer` und `T3Parser`. Bei der Konstruktion des Lexers wird ein Objekt der Klasse `ExtentLexerSharedInputState` übergeben. Es dient als Quelle für den zu bearbeitenden Code. Das Lexerobjekt wird im Anschluss dem Konstruktor für das Parserobjekt übergeben. Es ist möglich, dem Lexer und Parser erweiterte Klassen zur Verwendung als Datenstrukturen zu übergeben. Diesen Vorteil nutzen wir anschließend z.. mit der Klasse `TokenAST`, um komfortable Methoden zur Verarbeitung des AST zur Verfügung zu haben.

Der folgende Code initialisiert sowohl den Lexer, als auch den Parser. Er übernimmt die Verknüpfung von Lexer und Parser und bindet darüber hinaus auch noch unsere erweiterten Datenstrukturen ein, Beispiel 5.1

```
lsis = new ExtentLexerSharedInputState(file);
lexer = new T3Lexer(lsis);
lexer.setTokenObjectClass("Parser.ValueExtentToken");
parser = new T3Parser(lexer);
parser.setASTNodeClass("Parser.TokenAST");
parser.setFilename(file);
parser.pr_TTCN3Module();
```

Beispiel 5.1: Initialisierung Lexer, Parser

Sobald der Parser entsprechend instantiiert wurde, reicht ein simpler Aufruf, um den abstrakten Syntaxbaum zu erhalten:

```
TokenAST root = (TokenAST) parser.getAST();
```

5.1.1.3. Die Fehlerbehandlung

Die Fehlerbehandlung sollte eine gewisse Intelligenz besitzen, um dem User nicht nur Fehler anzuzeigen, sondern ihn auch im gewissen Maße zu unterstützen. Um dieses zu gewährleisten, wäre es wünschenswert, dass falls während des Parsens Fehler auftreten, diese in eine separate Datenstruktur implementiert werden. Hierzu wird die Klasse `ParserError` erstellt, die diese Fehlermeldungen in Form einer verketteten Liste speichert. Der Parser ist mit der Methode `getErrors()` ausgestattet, um auf diese Liste zugreifen zu können.

```
ParserError Error = parser.getErrors();
```

Die Klasse `ParserError` kapselt u.A. die Zeilennummer, die Spalte und die Fehlermeldung der auftretenden Fehler. Sie stellt Methoden zur formatierten Ausgabe (z.. Anzeige der Fehlerstelle durch Unterstreichung) bereit.

```
moduledefpartbroken.ttcn3:1:1-10: expecting "module", found 'modulebla'
modulebla test1 {
^^^^^^^^^^
```

Der Parser setzt, nach dem Auftreten von Fehlern, den Parsevorgang (soweit möglich) fort. Somit können im Fehlerfalle unvollständige ASTs erzeugt werden. Vor der Weiterverarbeitung, z.. im Semantikchecker, ist es deshalb ratsam, in jedem Falle das Vorhandensein von Fehlern zu prüfen.

5.1.1.4. Ein Beispiel

Der folgende TTCN3-Code soll geparst und der AST grafisch dargestellt werden, Beispiel 5.2.

```
/***
 * Module def
 */

module testmodule {
  control {
    const boolean testboolean := true
    var integer testinteger := 1
    var integer testinteger2 := 2
  }
}
```

Beispiel 5.2: TTCN-3 Beispielcode

Nach der Ausführung des Parsers, lässt sich mit folgendem Code eine interaktive grafische Sicht erzeugen, Beispiel 5.3

```
ASTFrame frame = new ASTFrame("tree", parseTree);
frame.setVisible(true);
```

Beispiel 5.3: Erzeugung grafischer Sicht

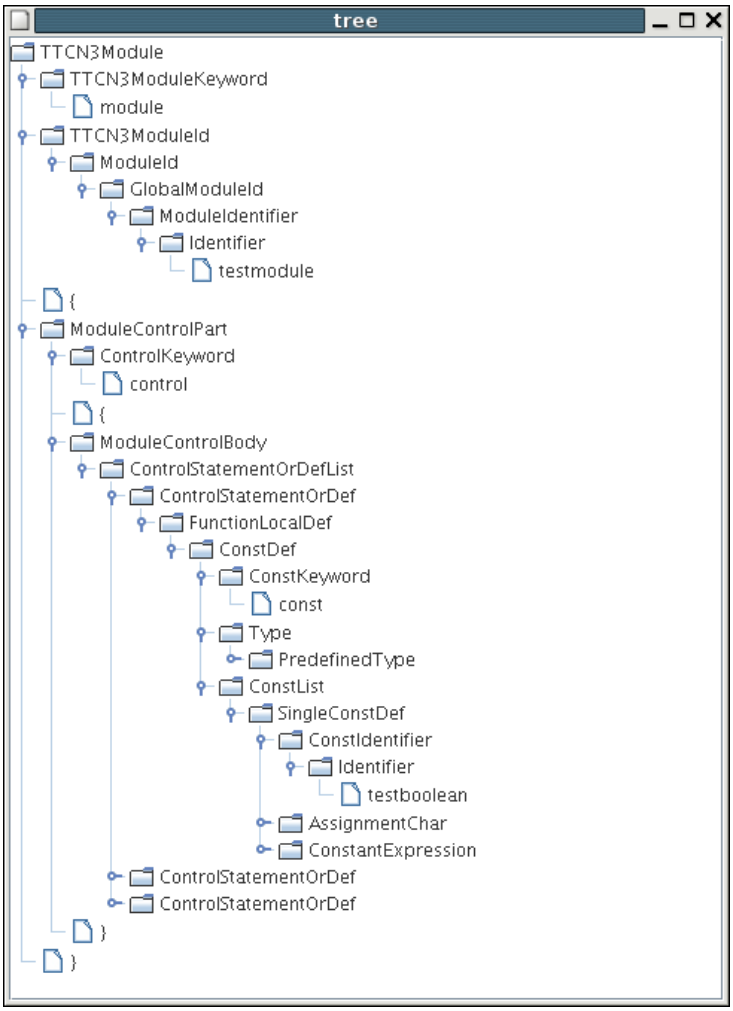

Abbildung 5.1.: Grafische Darstellung eines AST

5.2. Semantic Checker

Abseits aller theoretischen Prinzipien und des ausgewählten Designs wird versucht, auf eine praktische Weise zu erklären, wie alle Klassen aufgebaut sind und warum sie, wie vorliegend strukturiert sind. Ebenfalls werden einzelne Methoden der Klassen erläutert. Dieses Kapitel steht in einem engen Bezug zu den Semantik-Checker-Abschnitten des Kapitel 4.1.2.

Zunächst wird auf den Inhalt der Codepakete eingegangen. Diese drei Pakete sind `SemCheck`, `SemCheck.Definitions` und `SemCheck.Tree`. Das erste Paket traversiert den Baum und wenn passende Knoten identifiziert wurden, werden spezialisierte Methoden zur weiteren Analyse aufgerufen. Weiterhin gibt es in diesem Paket auch eine Fehler-Struktur-Klasse (wie im letzen Kapitel beschrieben), sowie eine Funktion, um den Baum auf zwei verschiedene Arten zu durchsuchen: Eine flache Suche und eine tiefe Suche.

In `SemCheck.Definitions` sind alle möglichen TTCN-3 Typen und Strukturen eingetragen. Jede Definition enthält allgemeine Attribute und Methoden, sowie spezifische Definitionsinformationen. Diese werden über eine abstrakte Klasse `Definitions` mit allgemeinen privaten Variablen und Funktionen definiert.

Für einige spezielle Definitionen (Variablen, Konstanten und Modul-Parameter) wird eine Zwischenklasse, die von „Definitions“ abgeleitet wird, herangezogen. Im `SemCheck.Tree` wird eine Baumstruktur definiert. Sie repräsentiert die Symboltabelle mitsamt aller benötigten Operationen (hinzufügen, suchen, ändern). In diesem Paket befindet sich des weiteren ein Interface, das die Symboltabelle beschreibt.

Schließlich gibt es noch einige Testfälle, die die Richtigkeit der Arbeit belegen sollen.

Im Folgenden wird jedes Paket, seine Funktion, die speziellen Methoden und die Gesamtfunktionalität beschrieben.

Zuerst wird geprüft, ob Fehler beim Parsen des TTCN3-Quellcodes aufgetreten sind. Wenn dies nicht der Fall ist, werden die Daten, die beim Parsevorgang gewonnen wurden, zum Semantikchecker weitergeben und bearbeitet. Es wird der Baum und zusätzlich die Datei, aus der der Baum generiert wurde, sowie eine neu initialisierte Liste von Fehlern, übergeben. Nach diesen Überprüfungen wird eine Symboltabelle mit einem Knoten, dem Wurzelknoten, erzeugt. In die Symboltabelle dieses Knotens werden die vordefinierten Funktionen und Typen eingefügt. Dadurch wird vermieden, jeden vordefinierten Typ in die entsprechenden Tabellen aller Module eintragen zu müssen. Mit diesen Daten wird die Methode `check` des Paketes `SemCheck` aufgerufen - somit beginnt die eigentliche Arbeit des Semantik-Checkers.

5.2.1. Paket `SemCheck`

Dieses Paket enthält vier Klassen, `CheckSemantic.java`, `CheckTypeDef.java`, `Searches.java` und `ErrorSemCheck.java`.

5.2.1.1. Klasse `CheckSemantic`

Die Klasse `CheckSemantic.java` ist der Treiber des Semantik-Checkers. Die Lochtabelle (für nicht deklarierte, aber bereits benutzte Typen) und RootAST sind private Attribute dieser Klasse. Die Methoden, die in der Klasse enthalten sind, sind oftmals Treibermethoden. Hiermit ist gemeint, dass sie den Baum durchsuchen und beim Antreffen eines relevanten Knotens die entsprechende Methode aufrufen, um die verschiedenen Teile des Baums zu verarbeiten. Zu Beginn der Methode wird die Lochtabelle erstellt, dann werden die adäquaten Teile des Baumes verarbeitet (jedes Modul hat einen eigenen Knoten, so dass dafür gesorgt werden muss, dass

immer dann ein neuer Knoten erstellt wird, wenn die Funktion aufgerufen wird). Nachdem sichergestellt ist, dass der korrekte Knoten vorliegt, wird die Symboltabelle gefüllt. Es gibt einen Spezialfall, in dem sich der Baum in zwei Äste verzweigt: Einen für den Definitionsteil, der andere für den Kontrollteil. Diese Tatsache muss beachtet werden, wobei einer dieser beiden Teile leer sein kann.

Um den Baum auszufüllen, wird die Funktion `fillUpTree` aufgerufen, mit der ersichtlich wird, ob in der Moduldefinition Knoten vorhanden sind. Diese können Typdefinitionen, eine Konstante oder eine Gruppe oder Anderes darstellen. Für jede Art muss ein anderer Weg gewählt werden. Ist es eine Konstante, so wird direkt eine Definition erstellt. Es wird der erste Schritt ausgeführt, der die Konstante aufruft und sie in die Symboltabelle einfügt. Ist es hingegen ein Typ, wird die Methode `check` der Klasse `CheckTypeDef` aufgerufen. Stellt sich nun heraus, dass es sich um eine neue Definition handelt, die neue Knoten in der Tabelle erstellt und zu einer Änderung des Sichtbarkeitsbereiches führt, muss zu einer speziellen Art der Analyse verzweigt werden.

Die Definition wird zuerst zur Tabelle hinzugefügt. Anschließend wird ein neuer Knoten des Baums im nächsten Level erstellt. Danach wird der Inhalt der Definition analysiert und der Knoten mit diesen Informationen ausgefüllt. Zurzeit wird der Inhaltsbaum rekursiv für folgende Funktionen analysiert: Altsteps, Gruppen und Funktionen. Für jede dieser Funktionen gibt es eine spezielle Methode.

In der Methode `recursiveAna` werden Funktionen analysiert; hier sind Ausdrücke, wie *if*, *else*, *while*, *do-while*, oder *for* zu beachten. Diese erstellen wiederum Knoten (Rekursion der Funktionen). In diesen Methoden können neue Deklarationen gefunden werden, so dass diese wiederum analysiert werden müssen.

In der Methode `checkHoles` wird kontrolliert, ob die Hashtabelle leer ist oder nicht. Enthält sie etwas, bedeutet dies, dass ein Fehler gefunden wurde, der lokalisiert und zur Fehlerliste hinzufügt werden muss.

Im zweiten Schritt wird die checkTree-Methode verwendet. Es wird nochmals durch den Baum gegangen und dabei die Analysen für den zweiten Schritt angestoßen. Hierfür werden die Knoten, die eine Aktion im Semantikchecker provozieren (Funktionsaufrufe, Zuweisungen etc.), lokalisiert. Die Methode kann den zweiten Schritt für einige Definitionen (wie Funktionen oder Variablen) veranlassen. Darüber hinaus verfügt sie über zugehörige Funktionen, die boolesche Bedingungen, Guards, return-Anweisungen und TTCN3-Spezialfunktionen (wie etwa Timer, Ports und Komponenten) prüfen. Der besondere Kniff bei der checkTree Methode hängt mit der Traversierung der Tabelle zusammen. Beim rekursiven Traversieren gibt es drei Sorten von Knoten: Eine, die prüft, ob der Abschnitt des Baumes korrekt ist und neue Aufrufe der checkTree-Methode durchführt, eine zweite für einfache Variablendeklarationen, Funktionsaufrufe und Konstanten und eine dritte für Strukturen und Änderungen beim Geltungsbereich. Teile der Symboltabelle müssen aktualisiert werden, wenn einer dieser Knoten gefunden wird. Hierzu wird in den entsprechenden Teil des Baumes verzweigt.

5.2.1.2. Klasse `CheckTypeDef.java`

In der Klasse `CheckTypeDef` wird kontrolliert, welche Art von Typ definiert wurde. Wird er durch Fundstellen im Baum direkt beschrieben, wird eine neue Definition erstellt und der erste Schritt direkt aufgerufen. Ist dieser Aufruf abgeschlossen, wird die Definition zur Symboltabelle direkt hinzugefügt.

5.2.1.3. Klasse `Searches.java`

Die Klasse `Searches.java` ist für das Suchen im Baum zuständig. Es gibt zwei Arten der Suche, *flat search* und *deep search* (*flache Suche* und *Tiefensuche*.) Es wird in dem Baum, der als zweiter Parameter übergeben wurde, nach Knoten gesucht, deren „Namen“ als erster Parameter übergeben wurde. Beide Sucharten, wenn aufgerufen, liefern eine Arrayliste mit allen Teilbäumen, dessen Wurzelknoten denselben Namen haben, wie der angeforderte String.

Flache Suche sucht nur unter den Kindknoten des Wurzelknoten des übergebenen Baumes. Tiefensuche sucht im gesamten Baum nach dem angeforderten Namen.

Die Klasse `ErrorSemCheck.java` kapselt analog zu der entsprechenden Fehlerklasse des Parsers Fehlermeldungen. Immer, wenn ein Fehler auftritt, wird eine Instanz dieser Klasse erzeugt.

5.2.2. Paket `SemCheck.Definitions`

Im zweiten Paket, `SemCheck.Definitions`, findet sich eine abstrakte Klasse `Definitions.java`.

5.2.2.1. Klasse `Definitions`

In dieser Klasse sind, als private Attribute, die allgemeinen Dinge abgelegt, die wir auch in allen Definitionen der TTCN-3-Sprache finden können. Es gibt nur zwei, *identifier* und *idTypen*. Wie bereits erwähnt, ist ein Typ eine Art von Definition, genau wie Record, Untertyp, Menge oder Port. „`row`“ und „`column`“ sind zwei neue Attribute, die zum Finden eines Fehlers vorhanden sind, wenn dieser produziert wurde. Sie bestehen nicht unbedingt zu Anfang, aber wenn die Fehlerstruktur zum Code hinzugefügt wird, erscheinen sie als notwendige Attribute für die Fehlermeldungsanzeige. Ein zusätzlich auftretendes Attribut ist der Dateiname. Da die ganze Zeit auf dem Parsebaum gearbeitet wird, muss der Dateiname für die Fehleranzeige, während des gesamten Ablaufes, zur Verfügung stehen. Andere Attribute der Definitionen können nicht verallgemeinert werden, weil jede Definition ihre eigene Attribute besitzt, die wiederum getrennt behandelt werden müssen. Dieser Umstand liefert auch die Begründung, warum für jede Deklaration eine eigene Klasse erstellt wird.

Das gleiche Problem tritt bei den Methoden auf. Es gibt allgemeine Methoden und private, die nur Schreib- und Lesemethoden für private Attribute sind, um Zugriffe von anderen Klassen für gewisse Attribute zu gewährleisten.

Allgemeine Methoden sind:

5.2.2.1.1 Methode `toString`

```
String toString(String visualization)
```

Dies ist eine sehr einfache Methode, die nur Attribute der Definition ausgibt. Implementiert wurde sie, um zu kontrollieren, ob die durchgeführte Arbeit korrekt ist. Der Stringparameter dient zur Ermittlung der Geltungsbereichsangabe. Jedes Mal, wenn der Geltungsbereich sich ändert, wächst der String `visualization`, so dass ein Stufeneffekt entsteht und der Geltungsbereich sehr einfach eingesehen werden kann (jeder Schritt ist ein neuer Bereichslevel.)

5.2.2.1.2 Methode `firstStep`

```
void firstStep(AST           tmpTree,
               TreeStructure tmpstructure,
               Hashtable     holesTable,
               String        file,
               ParserError   PError)
```

Diese Methode wird verwendet, um die Tabelle zu füllen. Sie sieht für viele Definitionen sehr ähnlich aus, weist aber bisweilen Unterschiede auf, die die Eigenheiten der verschiedenen Definitionen widerspiegelt. Die Methode erhält die Baumstruktur (`tmpstructure`, in der sich die Symboltabelle befindet) im richtigen Knoten, so dass überprüft werden kann, ob alles richtig definiert ist, wie etwa der Typ. Sie erhält die Lochtabelle (`holesTable`), da es Typen geben kann, die noch nicht definiert sind und in dieser eingetragen werden müssen. `file` und `PError` (Datei und Parser-Fehlerliste) wurden als Parameter für die Fehlerbehandlung aufgenommen. Schließlich erhält die Methode den AST Baum. Dieser Baum ist in der Klasse `CheckSemantic` oder auch `CheckTypeDef` zu finden und enthält Informationen, die benötigt werden, um einen Objekt der entsprechenden Klasse zu erstellen. In diesen Klassen finden wir den passenden Baum, normalerweise ist es der Name der Klasse, gefolgt vom Wort *Def* oder *Instance*. In den zwei Beispielen erhält die Methode als Parameter ASTs, die *AltstepDef* und *VarInstance* darstellen. Im Baum werden sinnvolle Informationen für behandelte Definitionen gesucht, wie z.. Bezeichner, Typ, idType und andere. Auffallend ist das Einfügen von nicht deklarierten Typen in die Lochtabelle. Dieses Einfügen wird in folgenden Codezeilen ausgeführt, Beispiel 5.4.

5.2.2.1.3 Methode `anaSubIdent`

```
Definitions anaSubIdent(String        identRec,
                        TreeStructure tmpStructure)
```

Die Methode `anaSubIdent (String identRec, TreeStructure tmpStructure)` wird für den zweiten Schritt der Typenanalyse für Typen mit Subidentifiers genutzt, in der die Korrektheit jedes einzelnen Parameters des Typs geprüft werden muss.

Die Methode `void secondStep(AST auxTree, TreeStructure tmpStructure, String file, Definitions defOrig, ErrorSemCheck error)` führt die Analysen für den zweiten Schritt durch. Beim ersten Schritt sind diese einander rekursiv. Wenn in einem Record ein Integer gefunden wird, wird die Analyse über die Basistypen durchgeführt. Wenn eine Variable einem der set-Typen entspringt, wird vom zweiten Schritt der Variablen-Klassen zum zweiten Schritt der set-Klasse gesprungen, in der die weitere Analyse stattfindet. Für Zuweisungen wird zur Prüfung der Typkompatabilität immer die originale Definition vorgehalten. In einer groben Sicht der Dinge wird eine Variable so lange analysiert, bis der schlussendliche Typ feststeht. Hierbei wird der Typ mit der ursprünglichen Definition verglichen. Wenn die Typen nicht zusammenpassen, wird ein Fehler erzeugt und in der entsprechenden Liste vermerkt.

Jede Klasse wird von `Definitions` abgeleitet und hat ihre eigenen Attribute und Funktionen.

Für den zweiten Schritt haben die meisten Klassen spezielle Funktionen, um ihre Besonderheiten zu analysieren. Jeder Typ kann referenziert werden, so dass in die Klassenmenge „eingetaucht“ werden muss, um den echten Typ zu suchen, so dass eine Korrektheitsanalyse möglich wird. Die Funktionen sind nicht von einem bestimmten Typ, haben aber einen getypten Rückgabewert, so dass dieser und die Parametertypen geprüft werden müssen. Records können in

```
//Ist der Typ deklariert??
if(treeStructure.searchEntryAllTree(this.Types) !=null)
{
    // Wenn existiert, zum referenzierten Typ ein Zeiger
    // darauf hinzuf"ugen:

    this.Types.add(treeStructure.searchEntryAllTree(this.Types));
}
else
{
    // Existiert nicht, in der Lochtabelle suchen...

    ArrayList sameType = (ArrayList)holesTable.get((String)this.Types);

    // Existiert dort der Name (als Schl"ussel des Hashing), das Objekt
    // in die Liste einf"ugen, die damit assoziiert ist, weil das
    // bedeutet, dass es einen Objekt in der Lochtabelle gibt,
    // dessen Typdeklaration bereits nicht korrekt ist und am Ende des
    // ersten Schrittes zu pr"ufen ist, ob der Typ deklariert wurde
    // oder ein Typfehler vorliegt.

    if(sameType != null)
    {
        sameType.add(this);
    }

    // Existiert das Objekt nicht in der Lochtabelle, f"ugen wir es
    // dort ein, weil es bedeutet dass es einen nicht deklarierten
    // Typ gibt, so dass wir eine neue Arrayliste erstellen m"ussen,
    // sie zum aktuellen Objekt hinzuf"ugen m"ussen und zu der Hash-
    // tabelle das neue 'Loch' in der Deklaration.

    else
    {
        sameType = new ArrayList();
        sameType.add(this);
        holesTable.put((String)this.returnTypes.get(0),sameType);
    }
}
```

Beispiel 5.4: Auffüllung der Lochtabelle

verschiedenen Arten vorliegen - als matching, als eigene Variable oder nur als Parameter innerhalb. Der Typ von Unions muss mit den beiden Typen verglichen werden, die angenommen werden können (Timer und Ports). Komponenten können ein spezielles Verhalten einnehmen. All diese Besonderheiten werden von speziellen Funktionen der entsprechenden Klassen geprüft.

Definitionen können neue Knoten erstellen und werden an zwei Stellen behandelt. Im Paket `SemCheck.Definitions` werden sie im ersten und zweiten Schritt analysiert. Die Erstellung von Knoten wird allerdings im Paket `SemCheck` in der Funktion `rekursiveWhatever.java` durchgeführt, wie oben bereits erklärt.

5.2.3. Paket `SemCheck.Tree`

Das dritte Paket enthält die Implementierung der Baumstruktur, sowie die Repräsentation der Symboltabelle, um sicher zu stellen, dass die Arbeit korrekt ist. Es wurde bereits erwähnt, wie der Baum aufgebaut ist. In diesem Paket werden die Methoden und Klassen definiert, um damit zu arbeiten. Java, in seiner Art, bringt den Baum in einer Hashtabelle mit. In diese Tabelle werden Definitionsobjekte hinzugefügt.

5.2.3.1. Klasse `TreeStructure.java`

Als Hauptklasse des Pakets, besitzt diese Klasse die Verantwortung, den ganzen Baum zu bearbeiten und seine Struktur in der Symboltabelle zu speichern. Die Struktur, wie bereits erwähnt, ist analog zum Baum, so dass private Attribute dieselben sind, wie in einem Baum.

Die Attribute sind:

5.2.3.1.1 Attribut `initial`

```
TreeStructure initial
```

Die Wurzel des Baums. Sie wird für die notwendigen Suchoperationen durch den ganzen Baum benötigt.

5.2.3.1.2 Attribut `node`

```
TreeNode node
```

Ein Baumknoten.

5.2.3.1.3 Attribut `firstChild`

```
TreeStructure firstChild
```

Wie in einem normalen Baum, gibt es einen ersten Kindknoten der Struktur. Diese stellt sich wie in einem Parsebaum dar.

5.2.3.1.4 Attribut `nextSibiling`

```
TreeStructure nextSibiling
```

Nächster Geschwisterknoten des aktuellen Knotens. Dieser wird ebenfalls zum Suchen benötigt.

5.2.3.1.5 Attribut `numChild`

```
int numChild
```

Anzahl von Kinderknoten des aktuellen Knotens.
Darüber hinaus gibt es Methoden zum Behandeln des Baums.
Diese sind:

5.2.3.1.6 Methode `getLastChildrenNode`

```
getLastChildrenNode()
```

Liefert den letzten Kindknoten des aktuellen Baumknotens. Wird benötigt, um neue Knoten an der adäquaten Position des Baumes hinzufügen zu können. Wenn ein Knoten hinzugefügt wird, dann an der letzten Position, das heiß, er wird dem `nextSibling` des aktuell letzten Kindknoten des aktuellen Knotens angehängt.

5.2.3.1.7 Methode `getActualChild`

```
getActualChild(int numChil)
```

Ermittelt den Kindknoten des aktuellen Knotens an der Position `numChil`. Rückgabe ist der Kindknoten.

5.2.3.1.8 Methode `searchEntryAllTree`

```
searchEntryAllTree(String name)
```

Sucht im aktuellen Knoten und allen Oberknoten bis zum Initialknoten. Wird eine Definition mit dem als Parameter angegebenen Namen gefunden, wird sie zurückgegeben.

5.2.3.1.9 Methode `addNode`

```
addNode(String      nam,
        String      file,
        ParserError PError)
```

Fügt einen Knoten zur Symboltabelle hinzu. Der Knotenname ist als Parameter angegeben. Ist die Baumstruktur leer, wird sie erstellt, indem der Initialknoten erstellt wird. Ist die Tabelle nicht leer, wird ein neuer Knoten erstellt und hinzugefügt. Hinzugefügt wird nach dem letzten Kindknoten des aktuellen Knoten.

5.2.3.1.10 Methode `checkName`

```
checkName(String name)
```

Diese Funktion ist analog zu `searchEntryAllTree`. Sie prüft allerdings, ob der Name bereits in der Tabelle des aktuellen Knoten existiert, sowie in allen oberen Knoten bis hin zum Initialknoten.

5.2.3.1.11 Methode `inicializationBasicTypes`

```
inicializationBasicTypes(String      file,
                         ParserError PError)
```

In dieser Funktion fügen wir zur Tabelle des Initialknotens einige spezielle Objekte mit Grundtypen, die in TTCN-3 definiert sind, hinzu.
`inicializationPredefinedFunctions(ErrorSemCheck error)`
In dieser Methode werden die Objekte mit vordefinierten Funktionen von TTCN3 in die Tabelle des Initialknotens gespeichert.

5.2.3.1.12 Methode `addDefinition`

```
addDefinition(Definitions def,
              String      file,
              ParserError PError)
```

Diese Methode fügt die als Parameter gegebene Definition zur Symboltabelle des aktuellen Knotens hinzu. Ihre Besonderheit besteht darin, dass diese Funktion die Fehlerdeklaration enthält. Der Fehler wird erzeugt, wenn bereits eine andere Definition existiert, deren Knoten deklariert wurde.

`searchActualFuntion_and_isFunction`

Suche die Funktion, in der man sich befindet, und überprüfe, ob der Name eine Funktion ist.

Weiterhin existieren Funktionen, die Imports überprüfen, d.. sie fügen zu der Tabelle `import-all` Strukturen, die Imports von Teilen von Modulen oder die Imports von Gruppen hinzu. Diese Methoden haben verwandte Funktionen zur Suche nach der zu importierenden Deklaration. Auch gibt es eine Funktion, um eine Gruppe der Tabelle hinzuzufügen.

5.2.3.2. Klasse `TreeNode.java`

Die Klasse `TreeNode.java` enthält die Struktur jedes Knoten in der Symboltabelle. Sie hat drei privaten Attribute, die *Hashtabelle*, die Symboltabelle des Knoten ist, *Name* des Knotens und einen TreeStructure-Attribut, namens *father*, der zur Suche im Baum verwendet wird, wenn etwas benötigt wird. Dieser Vaterknoten ist nützlich, denn nach jeder hinzugefügten Deklaration, die nicht Initialknoten ist, muss in allen Oberknoten gesucht werden. Er enthält auch einige boolesche Variablen, die zur Überprüfung von speziellen Strukturen, Modulen und Gruppen dienen. Diese werden gesondert behandelt, unter zu Hilfenahme von zwei Hashtabellen für die Imports (echte Deklarationen, welche aus anderen Modulen importiert worden sind) und notwendige Imports (notwendige Typen für die Import-Tabelle, wie etwa ein Typ für eine Variable der Imports Hash-Tabelle). Außer typischen Methoden, wie die Schreib- und Lesemethoden von privaten Attributen, hat die Klasse eine Methode zum Hinzufügen einer neuen Definition in die Hashtabelle namens `getObjectTable` und eine zum Auslesen von Definitionen aus der Hashtabelle. Diese Methode ist `Definitions getObjectTableFirst/SecondStep(String nam)`, die das Objekt zurückgibt. Der Unterschied zwischen `getObjectTableFirstStep` und `getObjectTableSecondStep` besteht darin, dass im Ersteren das Objekt nur in der Tabelle selbst sein kann (um auf duplizierte Identifikatoren zu prüfen) und dass im Letzteren die benötigte Deklaration auch einem Import entstammen kann.

5.2.3.3. Klasse `checkSemantic.java`

In diesem letzten Paket gibt es eine Klasse, die die Symboltabelle repräsentiert. In dieser Klasse haben wir diese Codezeilen:

```
TreeGUI treeGUI = new TreeGUI(structure);
treeGUI.createAndShowGUI();
```

Diese zwei Zeilen sorgen für die Repräsentation des Baums. Sie erstellen ein neues Objekt der Klasse `TreeGUI.java`, die zum Paket `SemCheck.tree` gehört. In dieser Klasse befinden sich Methoden zur Erstellung des graphischen Interface der Baumrepräsentation. Jeder deklarierte Knoten besitzt eine Information, die mit ihm assoziiert wird. Wenn der Name angeklickt wird, werden die gesamten Informationen in einem Fenster angezeigt. Änderungen des Gültigkeitsbereichs der inneren Typen werden tabellarisch angezeigt. Darüber hinaus werden Änderungen des Gültigkeitsbereiches in der Tabelle mit neuen Knoten angezeigt, die ihre eigene Tabellen besitzen.

Diese Klasse ist direkt aus dem Internet übernommen und angepasst, um den Baum korrekt anzeigen zu können. Die Methoden erstellen einfach ein gesplittetes BildAbbildung 5.2). Der obere Bereich zeigt die vollständige Struktur. Durch Anklicken, werden im unteren Bereich, die zugehörige Information angezeigt. Zur Verdeutlichung wird ein einfaches Beispiel dargestellt, um die Funktionen zu erläutern.

Die Klassen `Funtion.java` und `HashNode.java` werden nicht länger verwendet, sie gehören dem ersten Design, das entworfen wurde, an. `TreeViewer.java` ist eine vorherige Repräsentation der Tabelle, die nicht mehr genutzt wird.

5.2.4. Einige Beispiele

Zum Abschluss wird vorgeführt, wie dieser Teil des Programms arbeitet. Für dieses Codefragment in TTCN-3 sieht der erste Schritt wie im Beispiel 5.5 aus.

Der Semantik Checker produziert diesen Baum (Abbildung 5.2). Der Baum wird erstellt, wenn der erste Schritt abgeschlossen ist:

Der Baum enthält primitive Typen, wie in der Abbildung dargestellt. Im unteren Teilbild sehen wir die Information über einen der Knoten. In diesem Fall sehen wir die Information zur Funktion, mit allen analysierten Komponenten. Diese Analyse wird im ersten Schritt ausgeführt. Einige Fehler, die auftreten können, werden auf diese Weise im ersten Schrittes angezeigt. Es werden eine duplizierte Variable (die zweite Variable ist nicht in die Tabelle eingetragen) und einige undefinierte Typen angegeben. Die Idee ist, das Programm anzuhalten, wenn solche Arten von Fehlern auftreten, weil sie zu großen Problemen führen und der Rest der Fehler durch diese provoziert werden können. Dies war einer der positiven Testfälle. Nun prüfen wir zwei der negativen und werfen einen Blick auf die Darstellung der Fehler. Wenn ein Fehler im ersten Schritt auftritt, so wird kein Baum der semantischen Prüfung dargestellt, so dass wir keinen Fehler der Analyse des zweiten Schritts zuzuordnen haben. Hier haben wir ein Beispiel mit Fehlern im ersten sowie zweiten Schritt (Bsp. 5.6).

```
/***
 * valid: using dots in record
 */
module TestModule {

    type record MyRecordType
    {
        integer fieldInt,
        boolean fieldBool,
        MyNestedRecordType reg
    }

    type record MyNestedRecordType
    {
        charstring fieldChar,
        record {
            integer fieldInt
        } reg1
    }

    const integer size := 3;

    function MyFunction(in integer intParam1,
                        out boolean boolParam,
                        in integer intParam2) return MyRecordType
    {
        var MyRecordType MyReg1;
        var MyNestedRecordType MyReg2;
        var integer Int := 4;
        var integer IntNes := 7;
        MyReg1.fieldInt := Int;
        MyReg2.reg1.fieldInt := IntNes;
        MyReg1.reg.reg1.fieldInt := MyReg2.reg1.fieldInt;
        var boolean Bool := true;
        MyReg1.fieldBool := Bool;
        var charstring Char := "valid: using dots in record";
        MyReg1.reg.fieldChar := Char;
        return MyReg1;
    }

}

/* Testmodule finished */
```

Beispiel 5.5: Gültiger Ausgangscode, TTCN-3

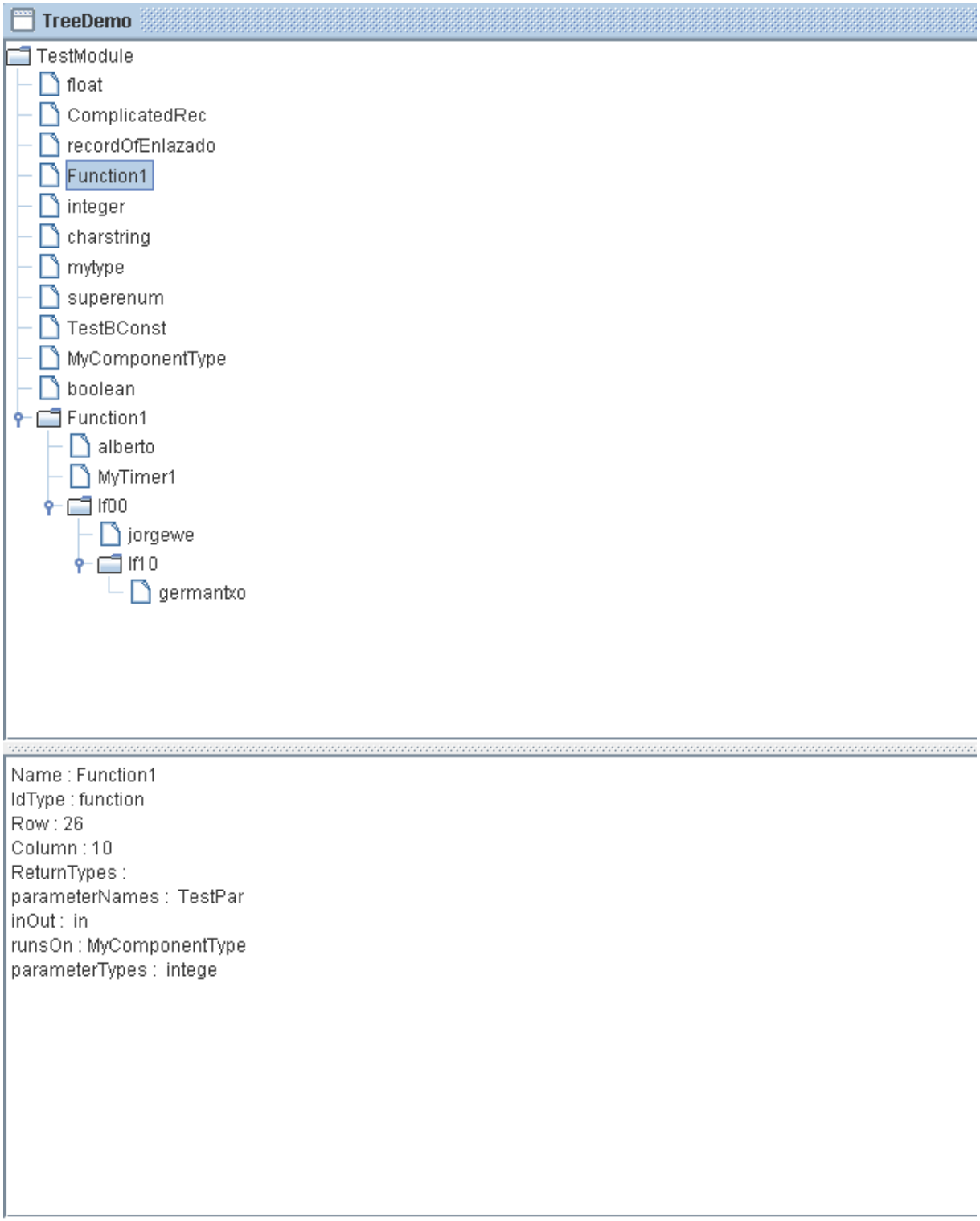

Abbildung 5.2.: Graphische Repräsentation der Symboltabelle

```
/***
 * invalid:  Checking simple returns returning a variable which
 * doesn' exist
 */
module TestModule {

    type record MyRecordType
    {
        integer fieldInt,
        boolean fieldBool optional,
        charstring fieldChar
    }

    function Function1(in integer intParam) return integer
    {
        var integer MyIntegerValue1 := -5;
        return MyIntegerValue2;
        /*
         * MyIntegerValue2 doesn' exist
         */
    }

    function Function2(in integer intParam) return integer
    {
        var integer MyIntegerValue1 := -5;
        var MyRecordType Record1 := {2,not true,"testing"};
        return Record1.fieldIntWRONG;
        /*
         * fieldIntWRONG doesn' exist
         */
    }

    function Function3(in integer intParam) return integer
    {
        var integer MyIntegerValue1 := -5;
        return MyIntegerValue1 and true;
        /*
         * operation not valid in a return
         */
    }
}
/* Testmodule finished */
```

Beispiel 5.6: Ungültiger Ausgangscode, TTCN-3

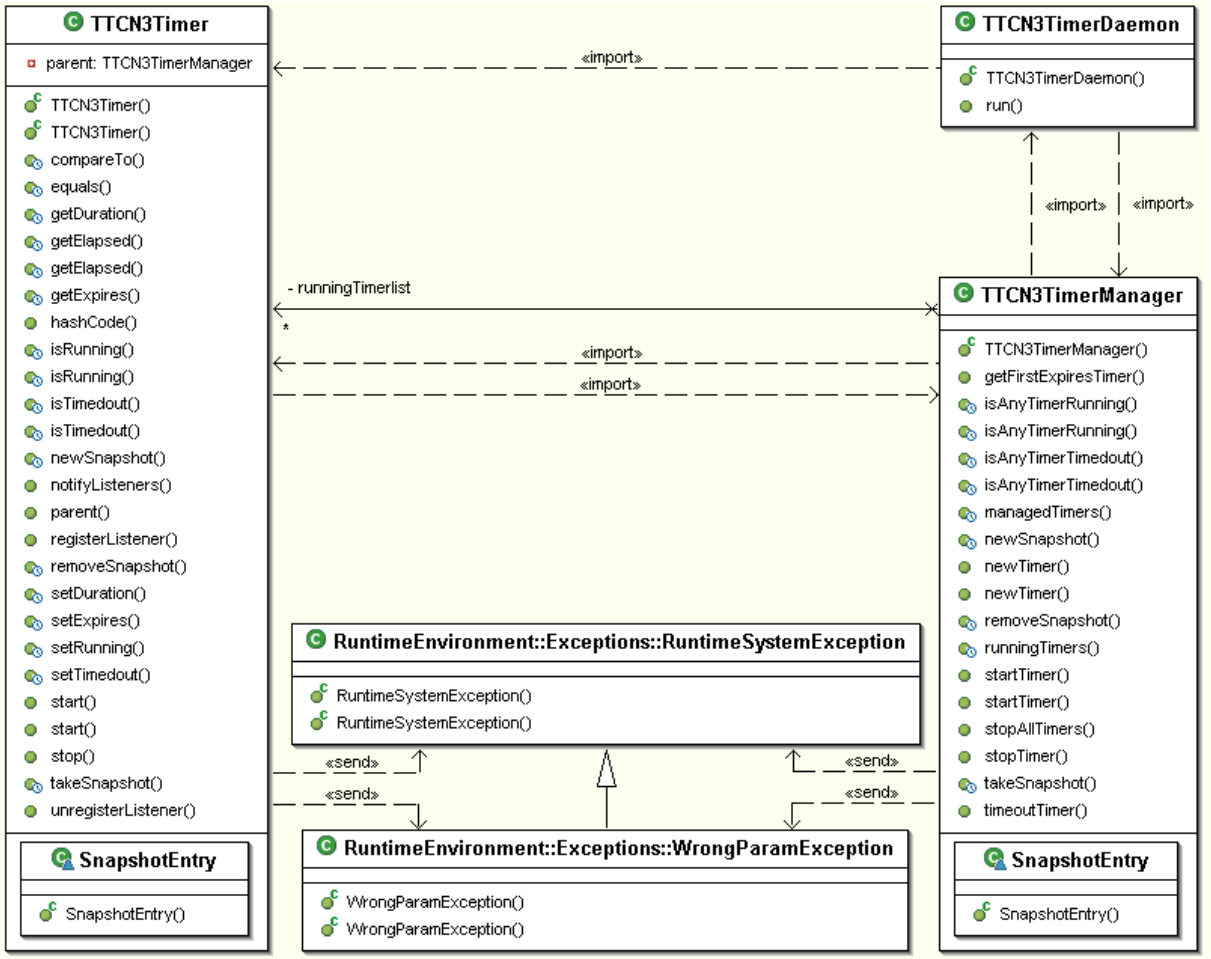

Abbildung 5.3.: Klassendiagramm: Timerverwaltung

5.3. Schnittstellen

5.3.1. Laufzeitsystem

Erst einmal einen kurzen Überblick darüber, was ein Laufzeitsystem ist und welche Aufgaben dieses lösen muss. Das Laufzeitsystem stellt alle möglichen Ressourcen bereit, die bei der Ausführung des Programms benötigt werden. Es steuert z.. das Lesen und Schreiben von Dateien. In diesem Fall steuert das Laufzeitsystem weitere Klassen, wie z.. Ports, Timer, Typen und Komponenten.

5.3.1.1. Timer

Die Schnittstelle der Timerverwaltung ist in der Abbildung 5.3 dargestellt.

5.3.1.1.1 Klasse TTCN3Timer

Timer sind Datenspeicher, die lediglich den Status, die Laufzeitdauer und den Ablaufszeitpunkt notieren und bis auf die Benachrichtigung vom *Listener* keine weitere Aktionen selbst ausführen.

EIGENSCHAFTEN

`TTCN3TimerManager parent`

speichert die Referenz auf den Timermanager, wendet sich dabei an seinen Manager um Timer spezifische Aufgaben erledigen zu lassen.

`Int duration`

speichert die Laufzeitdauer. Ist sie beim Start des Timers nicht belegt, wird der Startvorgang mit der Ausnahmemeldung vom Typ `WrongParamException` abgebrochen.

```
Boolean running,
boolean timedout
```

beschreiben den Status des Timers, dieses sieht wie folgt aus:

1. *running* = *true, timedout* = *false*: Der Timer ist im laufenden Zustand.
2. *running* = *false, timedout* = *true*: Der Timer ist unterbrochen worden.
3. *running* = *false, timedout* = *false*: Der Timer ist angehalten, aber nicht unterbrochen worden.
4. *running* = *true, timedout* = *true*: Dieser Fall kommt nicht vor.

`Long expires`

speichert die Ablaufszeit des Timers.

`LinkedList timeoutListeners`

ist die Liste der Listener, die beim Ablaufen oder Unterbrechen des Timers benachrichtigt werden sollen (siehe Abbildung 5.7).

`HashMap snapshotList`

ist die Liste der Snapshots. Jeder Snapshot hält die für TTCN-3 relevanten Zustände des Timers fest (siehe Abbildung 5.8).

KONSTRUKTOREN

Der Konstruktor

`TTCN3Timer(TTCN3TimerManager parent, int duration)`

initialisiert den Timer und setzt die Laufzeitdauer fest.

Die Initialisierung des Timers, ohne die Laufzeitdauer zu setzen, geschieht mit dem Konstruktor.

`TTCN3Timer(TTCN3TimerManager parent)`

Die Laufzeitdauer muss vor oder beim Start gesetzt werden.

FUNKTIONEN

`HashCode()`

wird für die Timerlisten des Managers benötigt. Sie liefert eine eindeutige ID des Timers.

`Equals(Object obj)`

liefert den Wahrheitswert *true*, wenn der aktuelle Timer in *obj* referenziert ist.

`CompareTo(Object o)`

erhält eine Timerreferenz und vergleicht deren Ablaufszeit. Läuft der aktuelle Timer schneller, so wird von der Funktion der Wert -1 geliefert. Laufen beide Timer gleichzeitig, so wird der Wert 0 geliefert. Wenn der abgegebene Timer schneller abläuft, wird schließlich der Wert 1 geliefert.

`Parent()`

ermittelt den Timermanager.

`SetDuration(int duration)`
setzt die Laufzeitdauer des Timers fest.
`GetDuration()`
ermittelt die gesetzte Laufzeitdauer.
`SetExpires(long expires) throws WrongParamException`
setzt den Ablaufszeitpunkt des Timers fest. Der angegebene Wert wird nur übernommen, wenn der Timer sich nicht im laufenden Zustand befindet. Andernfalls wird die Ausnahme `WrongParamException` ausgeworfen.
`GetExpires()`
ermittelt den Ablaufszeitpunkt des Timers.

```
GetElapsed(),
GetElapsed(int snapshot) throws WrongParamException
```

ermitteln die verstrichene Laufzeit, entweder aktuell (Aufruf ohne Parameter) oder die Zeit, die bei dem Timer zum entsprechenden Zeitpunkt abgelaufen ist, als der Snapshot mit der angegebenen Nummer aufgenommen wurde. Existiert kein Snapshot unter dieser Nummer, wird die Ausnahmemeldung vom Typ `WrongParamException` ausgeworfen.
`SetRunning(boolean running)`
setzt oder löscht den Zustand *laufend*.

```
IsRunning(),
isRunning(int snapshot) throws WrongParamException
```

ermitteln den aktuellen Zustand des Timers. Zum Einen wird überprüft, ob der Timer sich im laufenden Zustand befindet oder ob er zu der Zeit der Snapshotaufnahme aktuell war. Existiert kein Snapshot mit der angegebenen Nummer, wird die Ausnahme `WrongParamException` ausgeworfen.
`SetTimedout(boolean timedout)`
setzt den Zustand auf *unterbrochen*.

```
IsTimedout(),
isTimedout(int snapshot) throws WrongParamException
```

ermitteln, ob der Timer aktuell unterbrochen ist oder ob er zu der Zeit der Snapshotaufnahme aktuell war. Existiert kein Snapshot mit der angegebenen Nummer, so wird die Ausnahme `WrongParamException` ausgeworfen.

```
Start() throws WrongParamException, TimerUnknownException
start(int duration) throws WrongParamException,
TimerUnknownException
```

starten den Timer, geben die Fehlermeldung `WrongParamException` aus, wenn die Laufzeitdauer nicht gesetzt oder nicht positiv ist. Erkennt jedoch der Timermanager den Timer nicht bzw. wird der Timer nicht von dem Timermanager kontrolliert, so wird die Fehlermeldung `TimerUnknownException` ausgegeben.

Der Timer führt den Startvorgang nicht selbst aus, sondern veranlasst den Timermanager, den er über die Eigenschaft *owner* erkennt, für diese Aktion (siehe Abbildung 5.4) diesen zu starten. Auf diese Anfrage ruft der Timer seinen Manager auf und übergibt die Referenz auf sich selbst als Timerreferenz zum Starten.

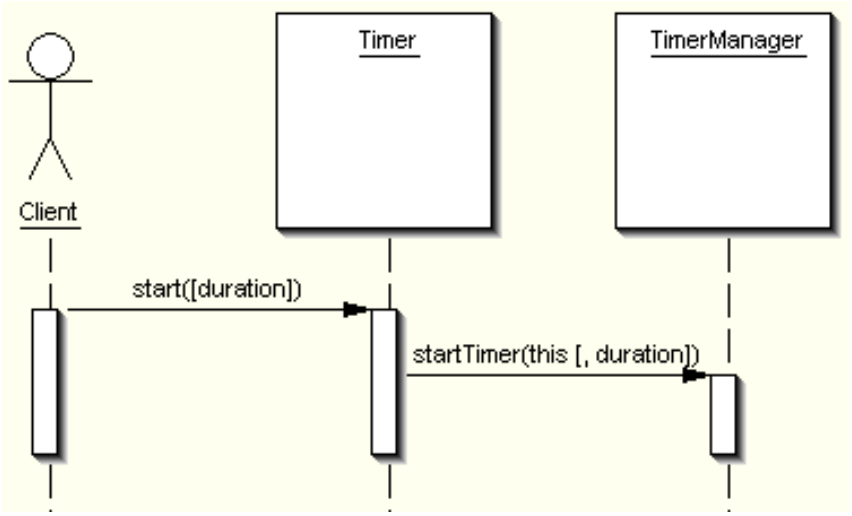

Abbildung 5.4.: Timer starten.

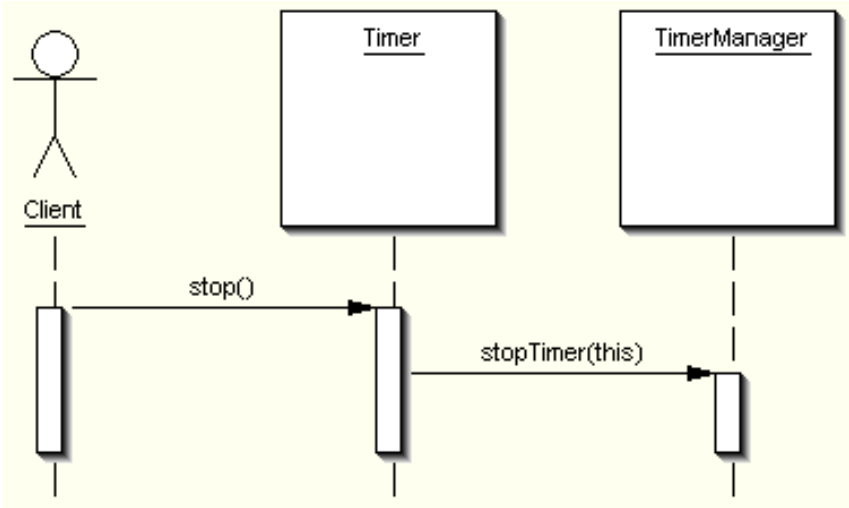

Abbildung 5.5.: Timer anhalten.

Da es in TTCN-3 keinen Timermanager gibt, sondern Timer direkt gestartet werden, bietet die Timerklasse die Funktion *start()* an, die eine Abkürzung für den eigentlichen Startvorgang durch den Manager darstellt (siehe Abbildung 5.11).

`Stop() throws TimerUnknownException`

Hält den Timer an. Der Vorgang wird mit der Ausnahme `TimerUnknownException` unterbrochen, wenn der Timermanager diesen Timer nicht verwaltet. Der Timer führt diesen Befehl ebenfalls nicht selbst aus, sondern ruft den Timermanager dazu auf (siehe Abbildung 5.5). Dieses ist ebenfalls eine Abkürzung für den eigentlichen Vorgang, den der Timermanager ausführt (Abbildung 5.11.) Der Timer selbst übergibt lediglich eine Referenz auf sich, um als Timer gestartet zu werden.

`RegisterListener(Object listener)`

trägt das angegebenen Objekt als Listener in die Liste ein (siehe Abbildung 5.6).

Jedes Objekt kann sich als *Listener* bei einem Timer eintragen lassen. Es erfolgt vom Timer eine Benachrichtigung, sobald der Timer unterbrochen wird oder abgelaufen ist.

`UnregisterListener(Object listener)`

entfernt den angegebenen Listener aus der Liste (siehe Abbildung 5.6). Der Listener erhält

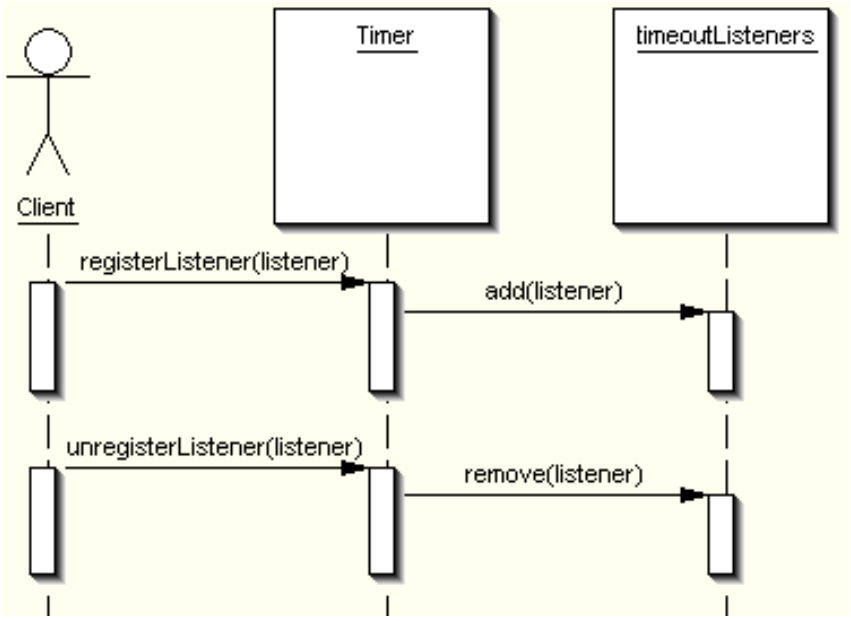

Abbildung 5.6.: Listenerobjekte eintragen und austragen.

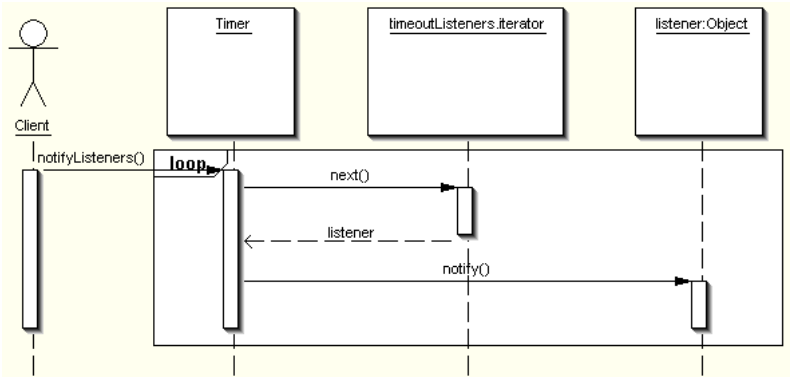

Abbildung 5.7.: Listenerobjekte benachrichtigen.

ab diesem Zeitpunkt keine Benachrichtigung mehr über das Unterbrechen oder Ablaufen des Timers.

`NotifyListeners()`

benachrichtigt jeden registrierten Listener (siehe Abbildung 5.7).

Diese Funktion wird vom Timermanager aufgerufen, und zwar innerhalb der Funktion *timeoutTimer()* (siehe Abbildung 5.14), was wiederum geschieht, wenn der Timer abgelaufen ist (der Timerdämon wurde aktiv) oder unterbrochen wird.

`NewSnapshot()`

Speichert relevante Timerdaten in einen neuen *Snapshot*, Abbildung 5.8. Jeder Timer verwaltet eine eigene Liste von Snapshots, in denen er nicht alle, sondern nur für TTCN-3 relevante Daten speichert.

`TakeSnapshot(int number)`

Nimmt ein neuen Snapshot und speichert diesen unter der angegebenen Nummer. Ein Snapshot unter derselben Nummer wird dabei überschrieben.

`removeSnapshot(int number)`

entfernt den Snapshot mit der angegebenen Nummer. Nach dieser Operation werden Befehle, aus einem Snapshot des Timers mit dieser Nummer auszulesen, mit `WrongParamException`

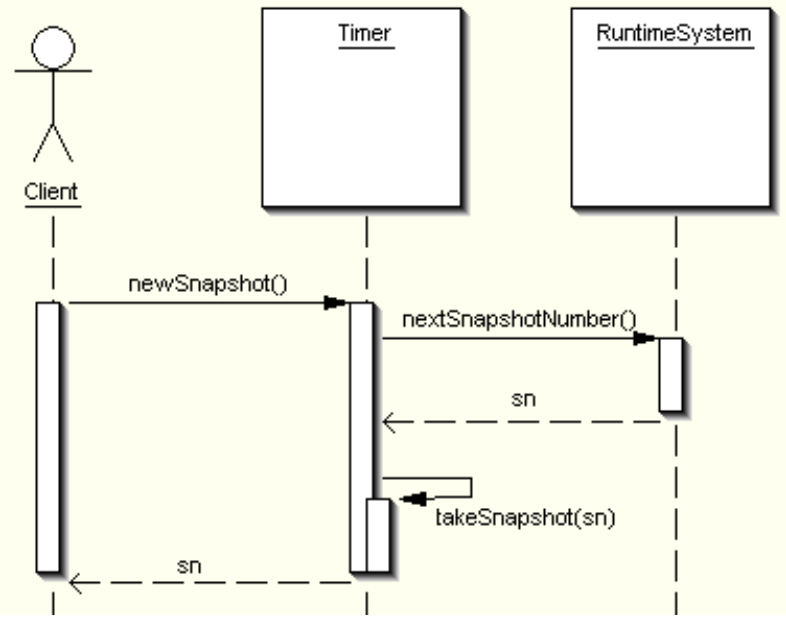

Abbildung 5.8.: Snapshot des Timers anlegen.

abgewiesen.

Jeder Zeitpunkt der Snapshotaufnahme besitzt eine globale ID, *snapNum*. Diese globale Nummer kann nur vom Testsystem angefordert werden. Mit dieser globalen Nummer kann eine Anfrage nach dem Snapshot gestellt werden, der den Zustand des Timers oder Timermanagers zu diesem Zeitpunkt darstellt.

Der Timer speichert folgende Daten in seinem Snapshot:

- Seinen Status (Eigenschaften *running*, *timeout*),
- Abgelaufene Zeit, sofern er im laufenden Zustand ist.

Diese Daten lassen sich mittels der Funktionen *isRunning()*, *isTimedout()* und *getElapsed()* unter Angabe der Snapshotnummer (der ID des Snapshots) auslesen. Der Aufruf der Funktionen *newSnapshot()*, *takeSnapshot()* und *removeSnapshot()* ist lokal, d.. es wird nur der Snapshot des Timers erstellt und gespeichert – in Gegensatz zu gleichen Funktionen des Timermanagers – keine Zustände des Timermanagers, sowie anderer Timer.

5.3.1.1.2 Klasse TTCN3TimerManager

Die Ablaufkontrolle und Verwaltung von einzelner Timer übernimmt der Timermanager. Hierzu verwaltet der Timermanager zwei Timerlisten – Eine Liste der verwalteten (bzw. aller Timer, *availTimerlist*) und eine Liste der laufenden Timer (*runningTimerlist*.) In der Liste 'laufenden Timer' befinden sich nur Timer, die aktuell laufen und den Status *laufend* besitzen. Die Liste aller Timer wird benötigt, da jeder Timer im Falle, dass alle Timer gestoppt werden sollen, seinen Status verlieren muss. Auch wenn er *unterbrochen* ist, verliert er diesen Status und bekommt den neuen Status *angehalten*, Abbildung 5.13.

Diese Liste erlaubt es außerdem, dass mehrere Instanzen der Managerklasse ihre eigene Timermengen parallel verwalten können. Hierfür ist es allerdings erforderlich, dass bevor der Status eines Timers durch den Manager geändert werden kann, überprüft wird, ob dieser Timer

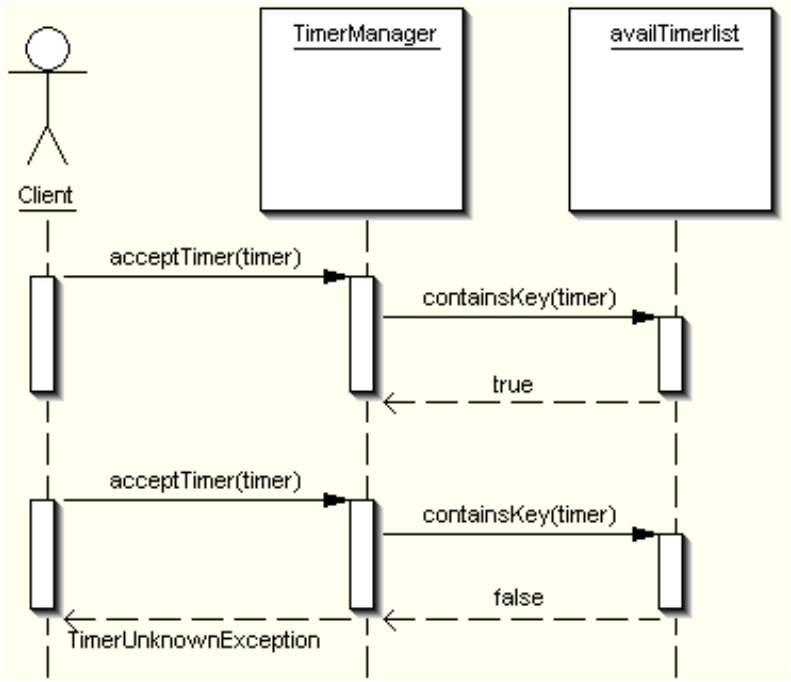

Abbildung 5.9.: Test ob der Timer existiert

von diesem Timermanager überhaupt verwaltet wird (Trennung der Zuständigkeitsbereiche,) was durch die Funktion *acceptTimer()* geschieht, Abbildung 5.9

Diese Funktion wird innerhalb der Funktionen *startTimer()*, *stopTimer()*, *timeoutTimer()*, (5.11, 5.12 und 5.14) als erstes ausgeführt. Hierbei wird überprüft, ob der angegebene Timer in der Liste der verwalteten Timer aufgeführt ist. Wenn ja, wird die aktuelle Funktion normal ausgeführt. Falls dieses nicht der Fall ist, wird die Ausführung mit der Ausnahmemeldung `TimerUnknownException` abgebrochen.

Des weiteren ist noch anzumerken, dass Timer immer in der Liste aller Timer bleiben (Timer können nicht gelöscht werden). In der Liste laufenden Timer dagegen nur, wenn sie aktuell laufen.

KONSTRUKTOR

`TTCN3TimerManager()`

Erstellt ein neues Timermanagement-Objekt.

FUNKTIONEN

```
NewTimer(),
newTimer(int duration)
```

erstellt einen neuen Timer. Dieser wird mit oder ohne initialer Laufzeitdauer gesetzt, Abbildung 5.10.

Das Erstellen eines neuen Timers geschieht nur durch den Timermanager. Dabei erstellt der Timermanager eine neue Timerinstanz, fügt diese seiner Liste der verwalteten Timer hinzu und liefert Referenz auf die Timerinstanz zurück.

```
acceptTimer(TTCN3Timer timer)

startTimer(TTCN3Timer timer) throws
          WrongParamException, TimerUnknownException
```

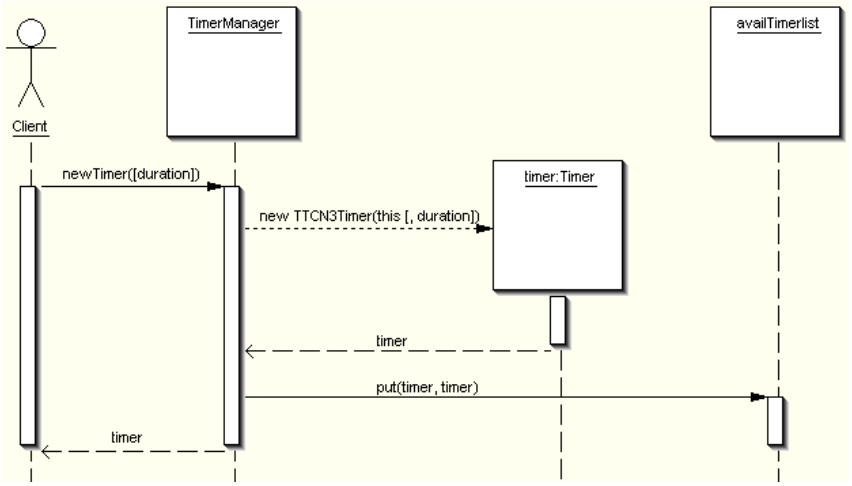

Abbildung 5.10.: Timer erstellen

```
startTimer(TTCN3Timer timer, int duration) throws
          WrongParamException, TimerUnknownException
```

Startet den angegebenen Timer oder startet den Timer neu. Dieses geschieht entweder mit oder ohne setzen einer neuen Laufzeitdauer, siehe Abbildung 5.11.

Nach der Prüfung, ob der Timer verwaltet wird (*accept(Timer)*, wenn nicht – Abbruch mit `TimerUnknownException`) wird überprüft, ob eine gültige Laufzeitdauer gesetzt oder angegeben ist, wenn nicht – Abbruch mit `WrongParamException`. Ist alles in Ordnung, wird der Timerstatus so aktualisiert, dass er *laufend* und *nicht unterbrochen* ist (unabhängig von seinem vorherigen Status) und der Timer in die Liste der laufenden Timer hinzugefügt.

`StopTimer(TTCN3Timer timer) throws TimerUnknownException`

Hält den Timer an, Abbildung 5.12.

Nach der Prüfung ob der Timer verwaltet wird (*accept(Timer)*, wenn nicht – Abbruch mit `TimerUnknownException`), wird der Timerstatus so aktualisiert, dass er *nicht laufend* und *nicht unterbrochen* ist (unabhängig von seinem vorigen Status) und der Timer aus der Liste der laufenden Timer entfernt, falls er dort vorhanden war.

`StopAllTimers()`

Hält alle laufende Timer an, Abbildung 5.13.

Hierzu wird die Liste der laufenden Timer geleert und der Status jedes *verwalteten* Timers so aktualisiert, dass dieser *angehalten* und *nicht unterbrochen* ist, unabhängig von seinem vorherigen Status.

`TimeoutTimer(TTCN3Timer timer) throws TimerUnknownException`

unterbricht den Timer, Abbildung 5.14.

Die Methode *timeoutTimer()* ermöglicht es, unter anderen dem Timerdämon, einen Timer zu unterbrechen. Der Ablauf ist auf die Benachrichtigung der Listener identisch mit *startTimer()* und *stopTimer()*: Nach der Prüfung, ob der Timer verwaltet wird (*accept(Timer)*, wenn nicht – Abbruch mit `TimerUnknownException`), wird der Timerstatus so aktualisiert, dass er *nicht laufend* und *unterbrochen* ist (unabhängig von seinem vorigen Status) und der Timer aus die Liste der laufenden Timer entfernt, falls er dort vorhanden war.

```
IsAnyTimerRunning(),
isAnyTimerRunning(int snapshot) throws
WrongParamException
```

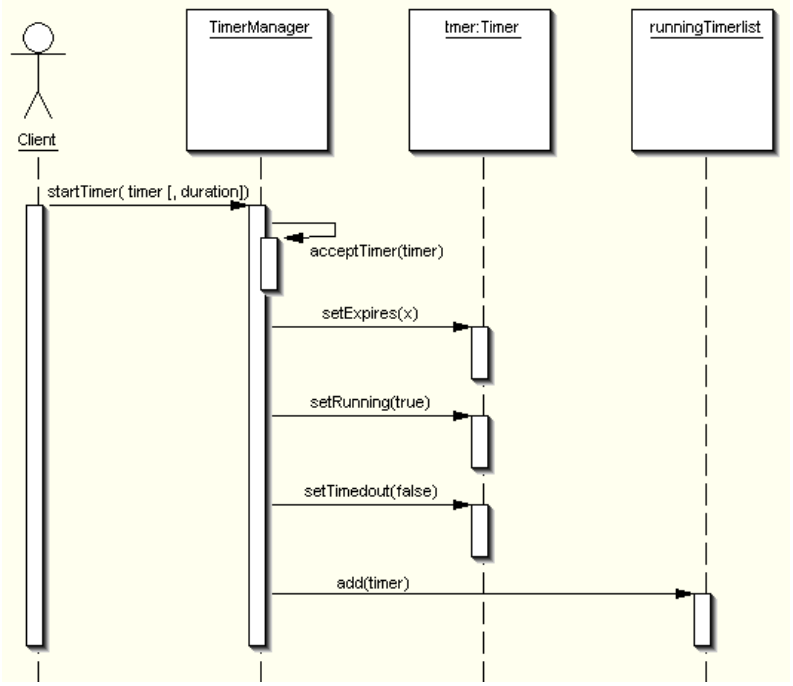

Abbildung 5.11.: Timer (neu-)starten

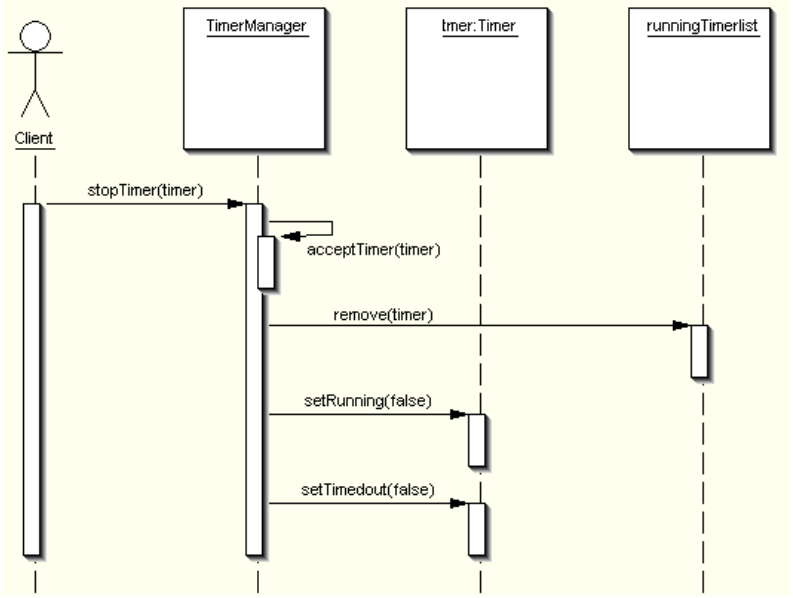

Abbildung 5.12.: Timer anhalten

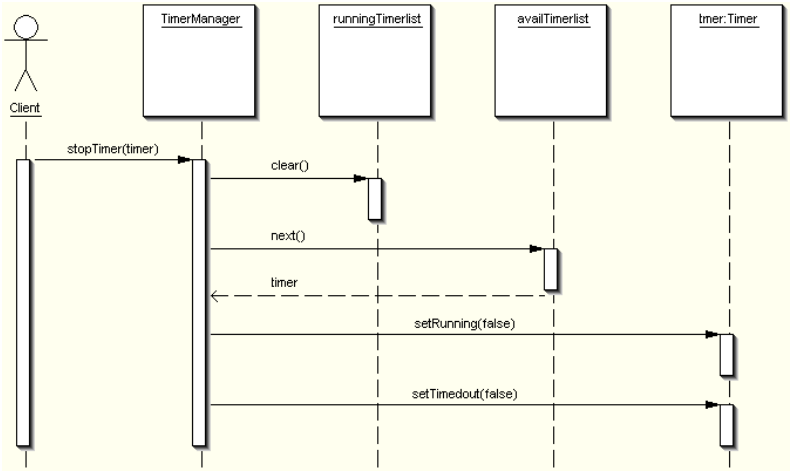

Abbildung 5.13.: Alle Timer anhalten

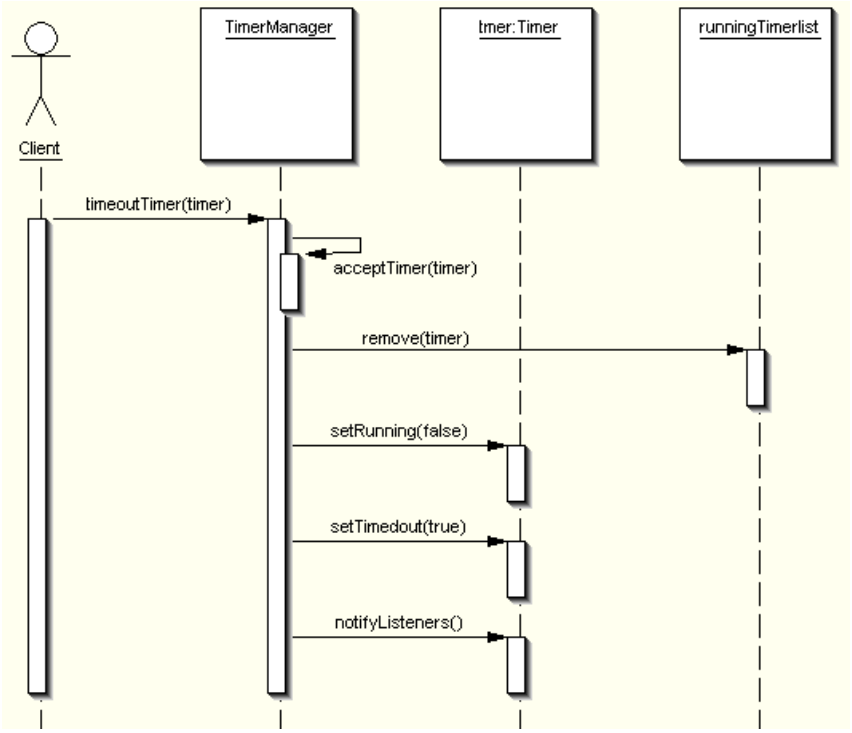

Abbildung 5.14.: Timer unterbrechen

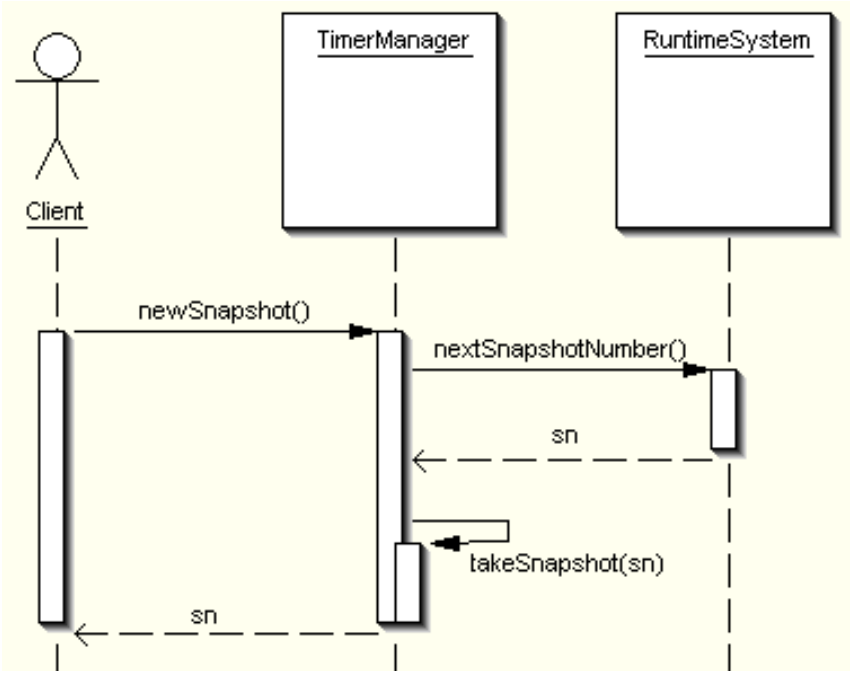

Abbildung 5.15.: Neuen Snapshot anlegen.

ermittelt, ob irgendein Timer aktuell läuft oder laufend war, als der Snapshot mit der angegebenen Nummer aufgenommen wurde. Existiert kein Snapshot mit der angegebenen Nummer, wird die Ausnahme `WrongParamException` geworfen.

```
IsAnyTimerTimedout(),
isAnyTimerTimedout(int snapshot) throws
WrongParamException
```

ermittelt, ob irgendein Timer aktuell unterbrochen ist oder unterbrochen war, als der Snapshot mit der angegebenen Nummer aufgenommen wurde. Existiert kein Snapshot mit der angegebenen Nummer, wird die Ausnahme `WrongParamException` geworfen.

`ManagedTimers()`
ermittelt die Anzahl der verwalteten Timer, dies entspricht exakt der Größe der Timerliste *availTimerlist*.

`RunningTimers()`
ermittelt die Anzahl der aktuell laufenden Timer, entspricht exakt der Größe von *runningTimerlist*.

`GetFirstExpiresTimer()`
ermittelt den Timer, der als nächster abläuft.

`NewSnapshot()`
erstellt einen neuen Snapshot, unter einer neuen Nummer und veranlasst jeden verwalteten Timer, einen Snapshot unter derselben Nummer anzulegen, Abbildung 5.15, wobei zum Speichern des Snapshots die Funktion *takeSnapshot()* aufgerufen wird – sowohl des Timers, als auch des Managers selbst.

Wenn man den Timermanager ein Snapshot anlegen lässt, erstellt dieser einen Snapshot seiner eigenen Zustände. Des Weiteren ruft er jeden verwalteten Timer auf, ein Snapshot für dessen eigene Zustände anzulegen. Alle Snapshots werden unter derselben ID gespeichert und sind über diese abrufbar. Dadurch ist es später möglich, Eigenschaften jedes einzelnen Timers abzufragen, ohne für jeden Timer die zugehörige Snapshot-ID notieren zu müssen. Wenn ein Timer

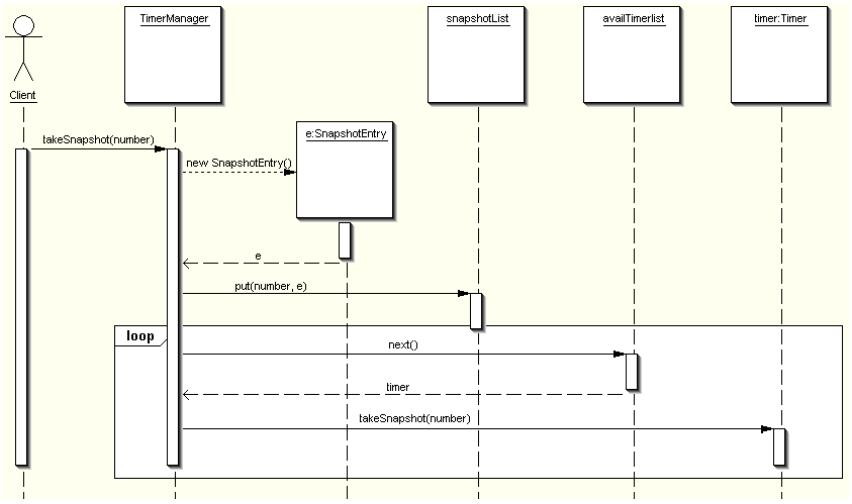

Abbildung 5.16.: Snapshot des Managers speichern.

einen Snapshot anlegt, werden nur Zustände des Timers im angelegten Snapshot gespeichert. Dies kann dafür genutzt werden, nur Snapshots von wenigen relevanten Timer anlegen zu lassen.

`TakeSnapshot(int number)`

ist die eigentliche Funktion, die einen Snapshot anlegt und unter der angegebenen Nummer speichert, Abbildung 5.16.

Speichert einen neuen Snapshot unter der gegebenen Nummer. Sie kann somit auch einen zuvor unter derselben Nummer gespeicherten Snapshot überschreiben – auch bei jedem verwalteten Timer. In einem Snapshot wird aufgenommen:

- die Angabe, ob irgendein Timer zur Snapshotaufnahme unterbrochen oder abgelaufen war.
- die Angabe, ob irgendein Timer zur Snapshotaufnahme laufend war.

Diese Daten lassen sich mittels der Funktionen *isAnyTimerRunning()* und *isAnyTimerTimedout()* mit Angabe der Snapshotnummer als Parameter auslesen. Liegt kein Snapshot unter der angegebenen Nummer vor, wird die Ausnahmemeldung `WrongParamException` geworfen.

`RemoveSnapshot(int number)`

entfernt einen Snapshot mit der angegebenen Nummer aus der eigenen Snapshotliste. Des weiteren veranlasst die Funktion jeden verwalteten Timer dazu, den unter der angegebenen Nummer gespeicherten Snapshot zu löschen.

`Suspend()`

Suspendiert den Timermanager und hält Timer sauber an. Dabei werden folgende Schritte eingeleitet:

1. Der Zeitpunkt der Suspendierung wird notiert.
2. Der Flag *suspended* wird gesetzt, die Funktion *isSuspended()* liefert nun *true*.
3. Die Aufrufe *startTimer()*, *stopTimer()*, *stopAllTimers()* und *timeoutTimer()* werden ohne Aktion ignoriert.

`Resume()`

löst die Suspendierung wieder auf und startet das Timermanagement wieder, wobei

1. bei jedem laufenden Timer die Ablaufszeit um die Zeit erweitert, in der der Manager suspendiert war.

2. das Flag *suspended* wird gelöscht, die Funktion *isSuspended()* liefert nun *false*.

3. Aufrufe *startTimer()*, *stopTimer()*, *stopAllTimers()* und *timeoutTimer()* werden wieder verarbeitet.

`IsSuspended()`
ermittelt, ob der Timermanager gerade suspendiert ist (true) oder nicht.

Die Funktionen *suspend()*, *resume()* und *isSuspended()* erlauben, Haltepunkte in der Ausführung von TTCN-3 Code zu setzen: Beim Eintritt eines Haltepunktes soll der Timermanager suspendiert werden, dann können Zustände aller Timer überprüft werden, ohne dass ein Timer dabei abläuft. Vor dem Verlassen des Haltepunktes muss der Timermanager wieder aufgenommen werden. Nach der Korrektur, die er an der Ablaufszeit von Timer vornimmt, laufen die Timer in derselben Reihenfolge ab, wie vor der Suspendierung und räumen dem System genau soviel Zeit für seine Aktivitäten ein, wie es ohne die Suspendierung hätte (Das System merkt dadurch die Verweilzeit im Haltepunkt nicht.)

5.3.1.1.3 Klasse TTCN3TimerDaemon

Der Timerdämon (5.17) ist ein Thread, der (und nur der) das Ablaufen von Timer sicherstellt. Er wird vom Timermanager erstellt, und wird im Hintergrund parallel zum restlichen System ausgeführt. Sobald er aktiv ist, überprüft der Dämon, ob es einen Timer gibt, der abgelaufen ist. Im positiven Fall bedient er sich der Methode *timeoutTimer()* (5.14) des Timermanagers, um dessen entsprechende Aktionen auszulösen und fragt erneut nach dem abgelaufenen Timer (usw. in der Schleife). Falls kein Timer (mehr) abgelaufen ist, dann *schläft* (Selbstaufruf von *sleep()*) der Timerdämon bis zum Ablauf des nächsten Timers. Der Timerdämon wird erst aufwachen bzw. aufgeweckt, wenn es sich an der Liste der laufender Timer etwas ändern soll, d.. wenn er was zu tun hat.

5.3.1.2. Portverwaltung

Der für Ports relevante Teil des Laufzeitsystem ist in der Abbildung 5.18 dargestellt.

Ports funktionieren wie eine Postfiliale, sie nehmen Nachrichten entgegen, verpacken sie zum Transport und legen sie auf eine Eingangsqueue der Zielports. Jeder dieser Ports benachrichtigt seine registrierte Listener, dass eine neue Nachricht angekommen ist, so dass jeder Listener darauf reagieren – die Nachricht abholen oder überprüfen kann. Jeder Port gehört zu einer Komponente, die er als Besitzer kennt und die Referenz darauf in der Eigenschaft *owner* speichert.

Bei der Verwaltung der Ports spielt die Klasse `TTCN3message` (Abschn. 5.3.1.2.2) eine entscheidende Rolle und dient damit zum Verpacken der Nachrichtendaten: Eine Instanz dieser Klasse speichert das versendete Datum und eine Referenz auf das Objekt, das dieses Datum versendet hat, was mit einem Briefumschlag vergleichbar ist.

Das Datum, das versendet wird, ist eine Instanz der Klasse `TTCN3Type`, welche wiederum das Interface `TTCN3Template` implementiert, das als das Nachrichtendatum vom Interface erwartet wird.

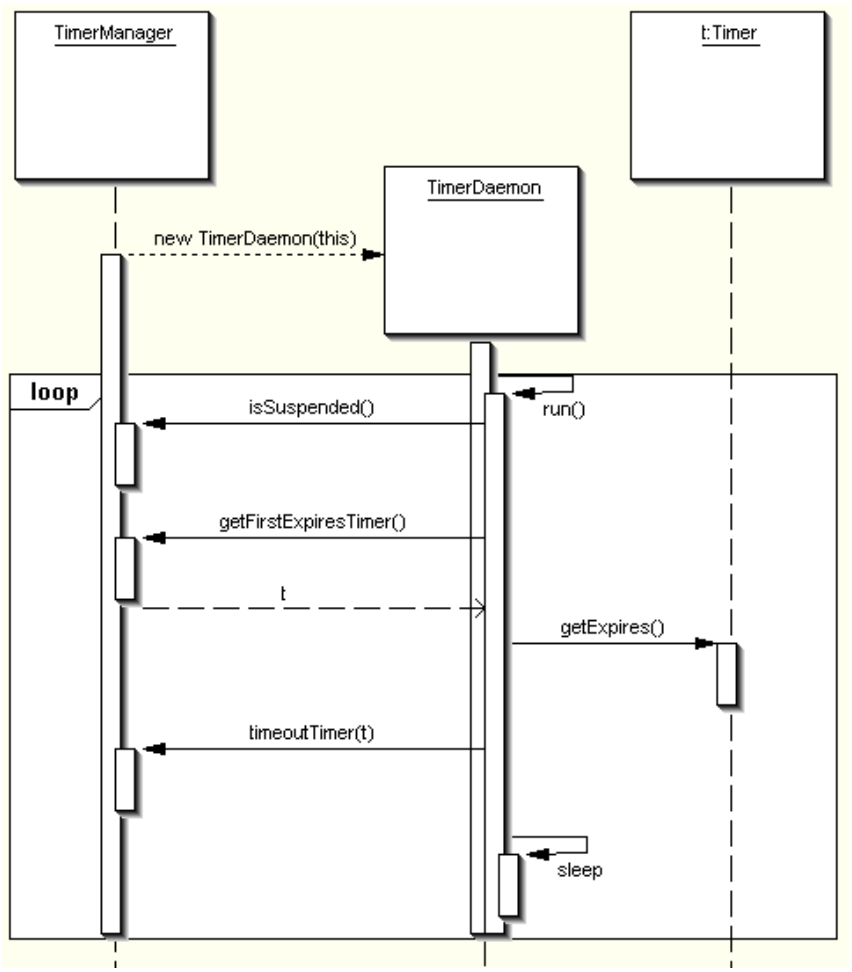

Abbildung 5.17.: Arbeitsablauf des Timerdämon

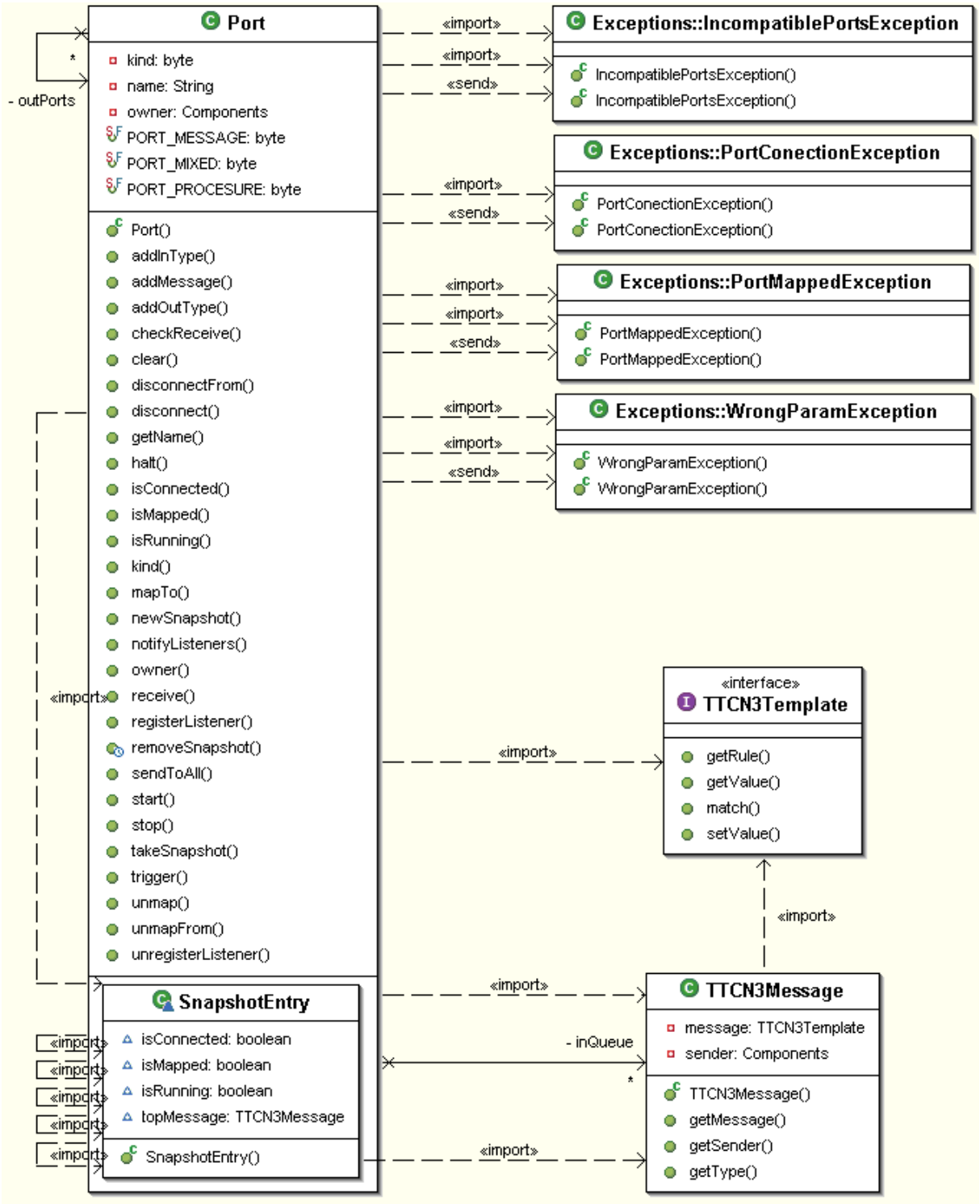

Abbildung 5.18.: Klassendiagramm, Portmanagement

5.3.1.2.1 Klasse Port

dient dazu, das zu versendete Nachrichtendatum anzunehmen, für den Benutzer transparent zum Zielport zu befördern, und an der anderen Seite zum Empfang bereitzustellen.

KONSTANTEN

`PORT_MESSAGE,`
`PORT_PROCESURE,`
`PORT_MIXED,`

bestimmen die Art des Ports: Ein Port kann ein nachrichtenbasierter, ein prozedurbasierter Port bzw. gemischter Port sein. Ein nachrichtenbasierter Port versendet Daten an andere Ports. Ein prozedurbasierter Port bildet hingegen die Ausführung von Aktionen an entfernten Komponenten nach und versendet somit Aufrufe der entfernten Methoden so, wie Rückgabewerte der entfernten Aufrufe. Ein gemischter Port beherrscht beide Betriebsarten.

Ein Port besitzt die Eigenschaft *kind*, die einen der obigen Werte annimmt, um die Art des Ports anzugeben. Momentan speichert diese Eigenschaft konstant den Wert `PORT_MESSAGE`, da derzeit nur nachrichtenbasierte Ports implementiert sind. Unterscheidung in Portarten dient der Aufwärtskompatibilitä der Entwicklung.

KONSTRUKTOREN

```
Port(Components owner) throws WrongParamException},
Port(Components owner, String name) throws WrongParamException,
Port(Components owner, Collection inTypes, Collection outTypes)
    throws WrongParamException,
```

erstellen einen neuen Port. Parameter:

- *owner* - referenziert die Komponente, zu welcher der Port gehört.
- *name* - gibt den Portnamen an.
- *inTypes* - Liste der Nachrichtentypen, die der Port empfangen kann (sog. *empfangbare* Nachrichtentypen.)
- *outTypes* - Liste der Nachrichtentypen, die der Port senden kann (sog. *sendbare* Nachrichtentypen.)

Alle Konstruktoren werfen die Ausnahme `WrongParamException`, wenn die Besitzerkomponente nicht existiert (d.. mit *null* angegeben ist).

FUNKTIONEN

`Kind()`
ermittelt die Art des Ports. Derzeit liefert diese Funktion konstant den Wert `PORT_MESSAGE`, da ausschließlich nachrichtbasierte Ports implementiert sind.

`Owner()`
gibt die Referenz auf die Besitzerkomponente des Ports zurück.

`AddInType(Class type)`
fügt der Liste der empfangbaren Nachrichtentypen einen neuen Nachrichtentyp hinzu.

`AddOutType(Class type)`
fügt der Liste der Sendbaren Nachrichtentypen einen neuen Nachrichtentyp hinzu.

```
connectTo (Port port) throws PortConectionException,
                      PortMappedException,
                      IncompatiblePortsException,

connectTo (Port... ports) throws PortConectionException,
                      PortMappedException,
                      IncompatiblePortsException,
```

verbindet diesen Port zu Einem oder mehreren Anderen. Entspricht dem TTCN-3 Operator *connect()*. Nach der *erfolgreichen* Ausführung dieser Funktion, können verbundene Ports Nachrichten einander senden.
Prüft auf folgende Fehlerfälle und unterbricht die Ausführung mit der Ausnahmemeldung:

- `PortConectionException` - falls versucht wird, einen Port mit zwei Ports einer Komponente zu verbinden,
- `PortMappedException` - falls eine Verbindung zu einem Port hergestellt wird, der bereits mit dem Systeminterface verbunden (gemappt) ist,
- `IncompatiblePortsException` - Wenn Typenlisten der Ports inkompatibel sind. Hierbei muss die Liste der sendbaren Typen eines Ports die echte Teilmenge der Liste empfangbarer Nachrichtentypen des anderen Ports sein.

Die Funktion *connectTo (ports)* erhält eine Portliste und versucht den Port mit mehreren angegebenen Ports zu verbinden. Sie prüft die obige Bedingungen für jede Einzelverbindung nach und unterbricht im Fehlerfall die weitere Ausführung, sodass eine Verbindung zu einem oder mehreren Ports noch erfolgreich hergestellt werden kann (wenn erst die zweite oder die weitere Verbindung fehlschlägt.) Diese wird bzw. werden nicht explizit wieder geschlossen.

```
IsConnected(),
isConnected(int snapshot)
```

gibt an, ob der Port zu irgendeinem Port aktuell verbunden ist oder bei der Aufnahme eines Snapshots mit der angegebenen Nummer verbunden war.

```
MapTo(Port port) throws PortMappedException,
PortConectionException,
                IncompatiblePortsException, WrongParamException,

mapTo(Port... ports) throws PortMappedException, PortConectionException,
                IncompatiblePortsException, WrongParamException
```

entspricht dem TTCN-3 Operator *map()*. Dieser verbindet den Componentenport mit einem Port des Systeminterface bzw. einen Port des Systeminterface mit mehreren Componentenports. Er prüft auf folgende Fehlerfälle:

- `PortMappedException` - Wenn ein Port gemappt sein soll, der bereits gemappt ist.
- `PortConectionException` - Wenn ein Port gemappt werden soll, der bereits mit einem anderen Port verbunden ist.

- `WrongParamException` - Wenn eine Verbindung zwischen keinem oder mehreren Ports des Systeminterface hergestellt werden soll.
- `IncompatiblePortsException` - Wenn Nachrichtentypenlisten, der zu verbindenden Ports, inkompatibel sind.

```
  isMapped(),
isMapped(int snapshot)
```

gibt an, ob der Port zu einem Port des Systeminterface verbunden ist, oder verbunden war, als ein Snapshot mit der angegebenen Nummer aufgenommen wurde.

```
disconnect(), disconnectFrom (Port port),
disconnectFrom (Port...
ports)
```

entsprechen dem TTCN-3 Operator *disconnect()*. Sie lösen alle Verbindungen zu bestimmten Ports.

```
unmap(), unmapFrom(Port port),
unmapFrom(Port... ports
```

entsprechen dem TTCN-3 Operator *unmap()*. Sie lösen die Verbindungen zum Port des Systeminterfaces.

```
Send(TTCN3Template message, Components... recepients),
sendToAll(TTCN3Template message)
```

Der TTCN-3 Operator *send()* sendet eine Nachricht an angegebene Empfänger oder an alle Ports mit denen dieser verbunden ist.

Beim Senden der Nachricht wird dem Port die Nachricht und eine Liste der Empfängerkomponenten übergeben (Abbildung 5.19). Aus dieser Liste ermittelt der Port die Liste aller Ports, an die er die Nachricht schicken soll. Daraufhin erstellt er ein neues Objekt der Klasse `TTCN3-message`, das das Nachrichtendatum und die Referenz auf die Besitzerkomponente des Ports als Sender des Datums speichert und übergibt dieses Objekt an jeden Port der ermittelten Portliste (weist jedem Port die Nachricht zu).

```
  addMessage(TTCN3Message message)
```

Wenn eine Nachricht von einem anderen Port, mittels der Methode *addMessage()*, zugewiesen wird (Abbildung 5.20), dann speichert der Port diese Nachricht in der Nachrichtenliste und benachrichtigt jeden seiner registrierten Listener, so dass diese sich die Nachricht abholen können. Alternativ: Wenn der Port nicht zu einer normalen Komponente, sondern zum Systeminterface gehört, findet kein Abspeichern in der Nachrichtenliste statt. Die Nachricht wird direkt an das TRI weitergeschickt.

```
receive(), receive(Components from), receive(TTCN3Template
message), receive(TTCN3Template message, Components from)
```

TTCN-3 Operator *receive()*, empfängt eine Nachricht (Abbildung 5.21). Die zuletzt ankommende Nachricht wird von der Nachrichtenliste gelesen. Anschließend wird überprüft, ob die Nachricht den Funktionsparametern entspricht. Diese Parameter dienen als Template, um die Nachricht zu verifizieren und eine Referenz auf eine Senderkomponente zu besitzen, von der

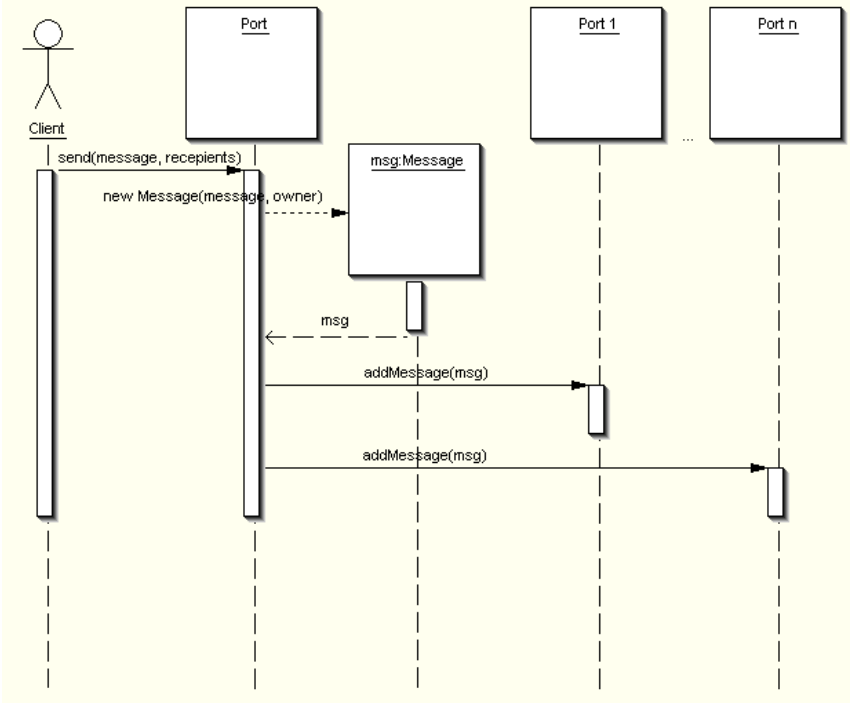

Abbildung 5.19.: Nachricht senden.

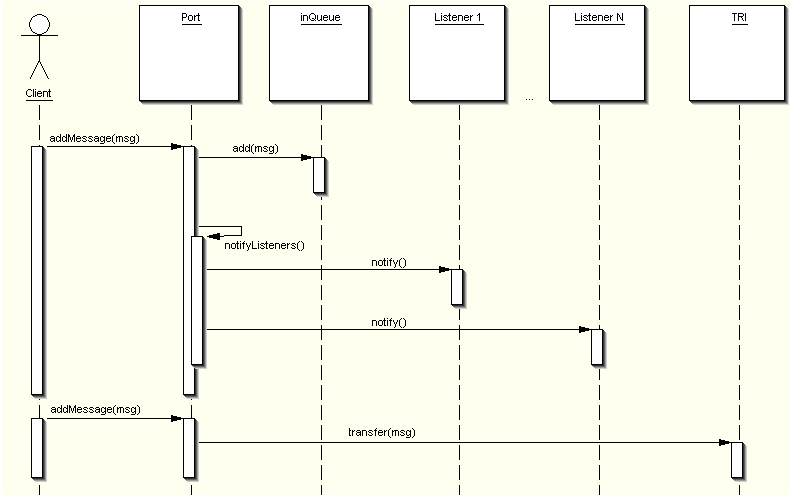

Abbildung 5.20.: Nachricht zuweisen.

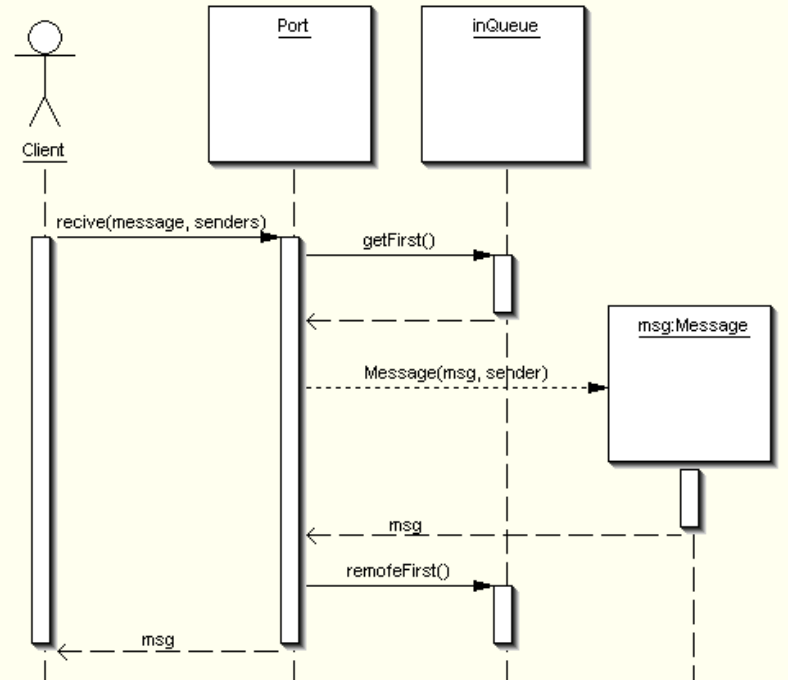

Abbildung 5.21.: Nachricht empfangen.

die Nachricht abgeschickt werden soll. Entspricht diese Nachricht den Parametern, wird die Nachricht durch diesen Operator zurückgegeben und von der Nachrichtenliste entfernt. Die Parameter sind optional, d.. die Überprüfung findet durch einen oder beide angegebenen Parameter nicht statt. So wird die zuletzt angekommene Nachricht einfach zurückgegeben, wenn beide Parameter ausgelassen worden sind. Wurde nur der Parameter *sender* übergeben, wird die zuletzt angekommene Nachricht nur zurückgegeben, wenn diese von der angegebenen Senderkomponente geschickt wurde. Dabei wird nicht die empfangene Nachricht zurückgegeben, sondern nur ein neues Objekt der `TTCN3message` Klasse erstellt, das das empfangene Datum speichert und die gesendete Komponente als Senderreferenz angibt.

```
trigger(), trigger(TTCN3Template message), trigger(Components
from), trigger(TTCN3Template message, Components from)
```

Der TTCN-3 Operator *trigger()* entfernt eine Nachricht (Abbildung 5.22:). Dabei wird die Nachricht von der Nachrichtenliste sofort entfernt und bei Übereinstimmung mit den Parametern, wie beim Empfang der Nachricht, zurückgegeben.

```
checkReceive(), checkReceive(TTCN3Template message),
checkReceive(Components from), checkReceive(TTCN3Template message,
Components from),

checkReceive(snapshot) throws WrongParamException,
checkReceive(message, snapshot) throws WrongParamException,
checkReceive(from,snapshot) throws WrongParamException,
checkReceive(message, from, snapshot) throws WrongParamException,
```

prüfen auf eine Nachricht (Abbildung 5.23). Die Methode in allen Varianten entspricht dem Operator *check(recieve)* von `TTCN-3` und dient zur Überprüfung, ob die Nachricht vorhanden ist

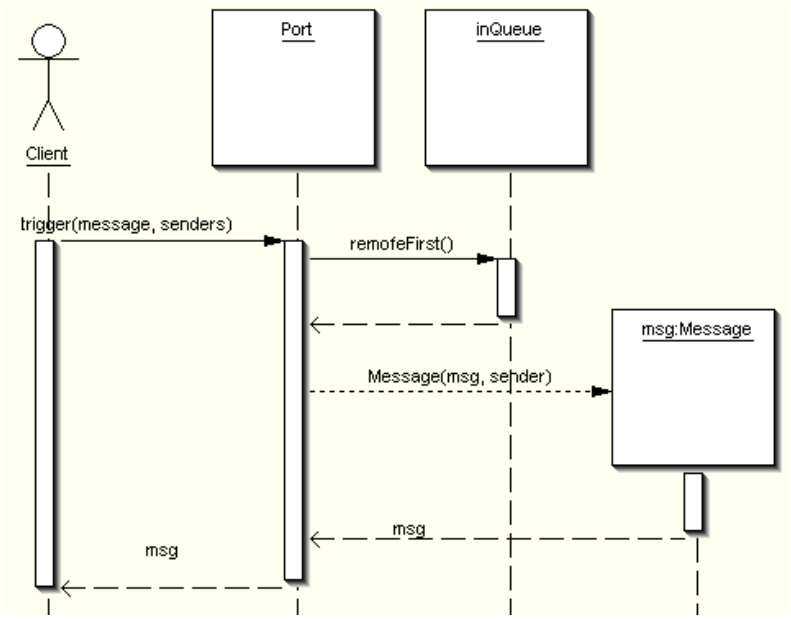

Abbildung 5.22.: Nachricht entfernen.

oder nicht. Durch diese Methode wird die zuletzt angekommene Nachricht nicht von der Nachrichtenliste entfernt, sondern auf Übereinstimmung mit den Funktionsparametern überprüft und im positiven Fall wie beim Empfang (5.21) der Nachricht zurückgegeben.

`clear()`

Der TTCN-3 Operator *clear()* leert die Nachrichtenliste des Port (Abbildung 5.24), sodass keine zuvor angekommene Nachrichten mehr durch Empfangsoperatoren gelesen werden können.

`halt()`

Der TTCN-3 Operator *halt()* deaktiviert den Port (Abbildung 5.25). Die zuvor angekommenen Nachrichten stehen zum Empfang bereit, die Annahme neuer Nachrichten durch die Methode *addMessage()* wird jedoch verweigert.

`stop()`

Der TTCN-3 Operator *stop()* deaktiviert den Port und leert die Nachrichtenliste (Abbildung 5.25), so dass auch der Empfang von zuvor angenommenen und noch nicht verarbeiteten Nachrichten nicht mehr möglich ist.

`start()`

Der TTCN-3 Operator *start()*aktiviert den Port (Abbildung 5.26), wobei die Nachrichtenliste geleert wird.

```
IsRunning(),
isRunning(int snapshot)
```

Liefert die Überprüfung, ob der Port aktiv ist, geschieht mit der Methode *isRunning()*.

`RegisterListener(Object listener)`

Registriert einen neuen Listener. Der Listener ist in eine Listenerliste aufgenommen und wird benachrichtigt, wenn eine neue Nachricht angekommen ist.

`UnregisterListener(Object listener)`

Entfernt den angegebenen Listener von der Liste, sodass er nicht mehr über einen Nachrichtenempfang informiert wird.

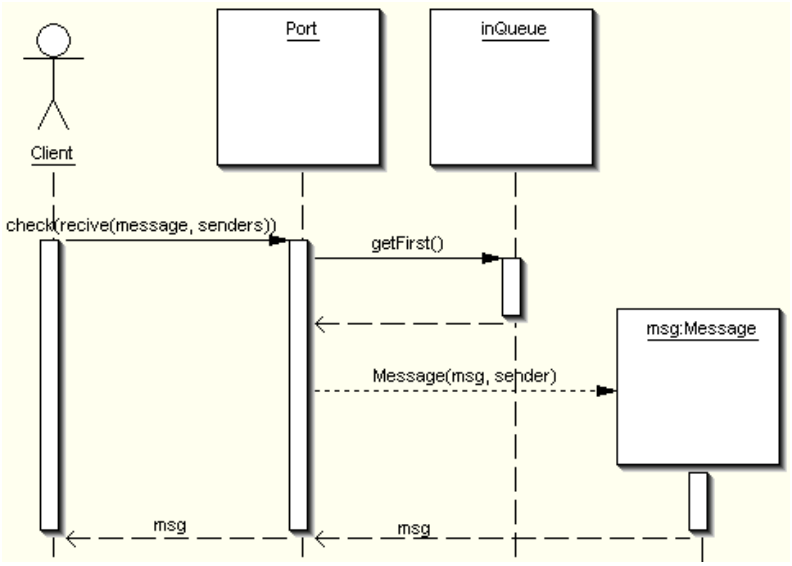

Abbildung 5.23.: Auf Nachricht prüfen.

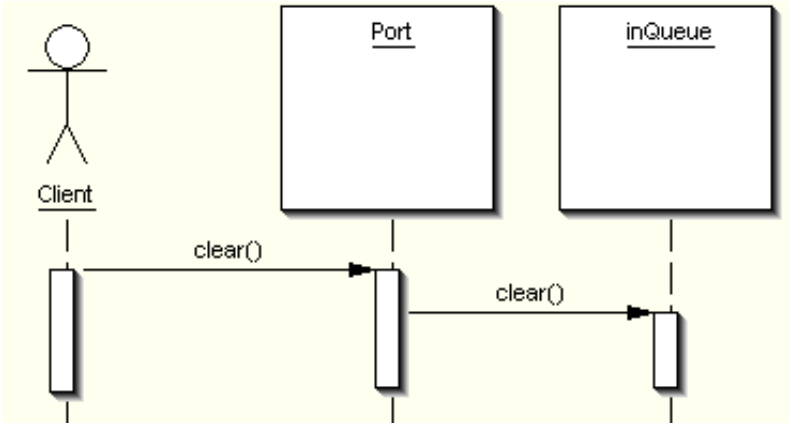

Abbildung 5.24.: Nachrichtenliste leeren.

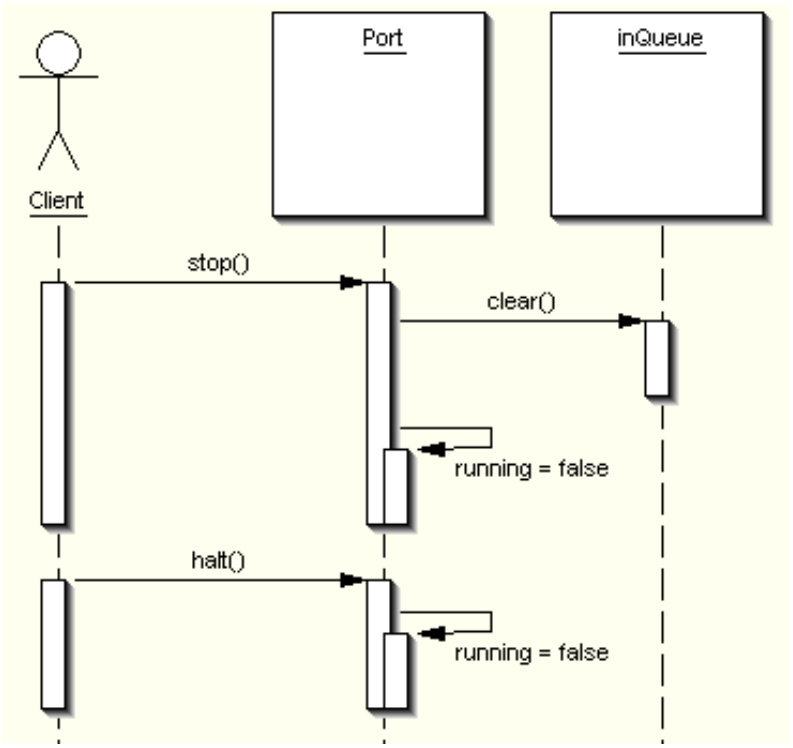

Abbildung 5.25.: Port deaktivieren.

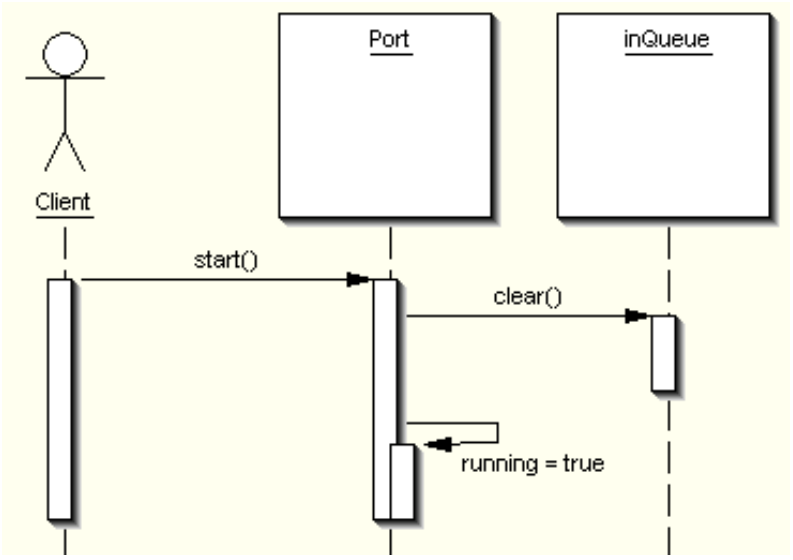

Abbildung 5.26.: Port aktivieren.

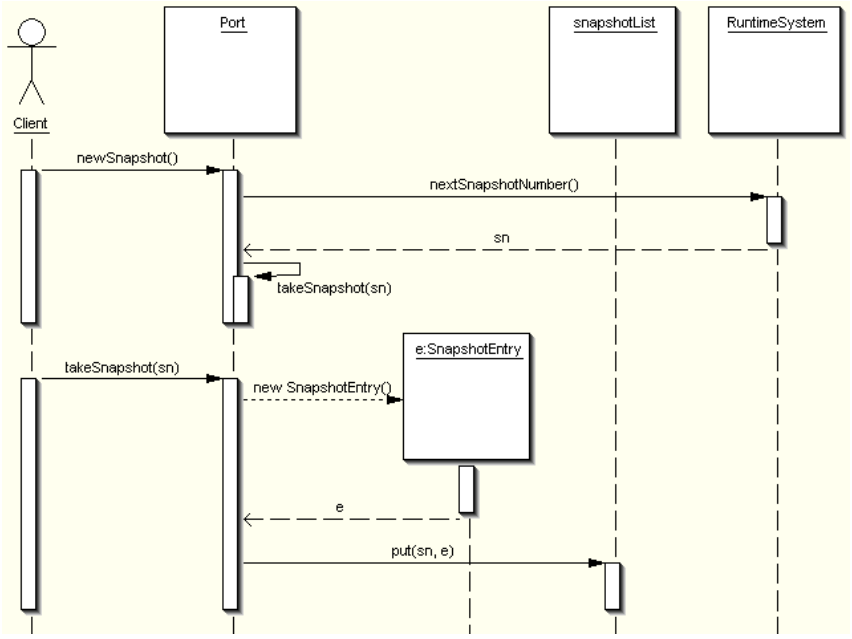

Abbildung 5.27.: Aktuellen Zustand speichern.

`NewSnapshot(),`
`takeSnapshot(number)`

Bei der Methode *newsnapshot()*, 5.27 wird eine neue Snapshotnummer vom Laufzeitsystem angefordert und ein neuer Snapshot unter dieser Nummer gespeichert. Das Speichern des Snapshots, unter der angegebenen Nummer, erfolgt mit der Methode *takesnapshot()*. Die dabei gespeicherten Daten geben den Zustand des Ports, ob er „aktiv“ ist oder „nicht“, ob er mit einem anderen Port oder dem Systeminterface verbunden ist. Des weiteren speichert diese Methode die zuletzt angekommene Nachricht. Mittels der Snapshotnummer ist es möglich, diese Daten später auszulesen.

`RemoveSnapshot(number)`

Um den unter der gegebenen Nummer gespeicherten Snapshot zu entfernen, bedient man dich der Methode *removeSnapshot()*.

`GetName()`

Ermittelt den Portnamen.

5.3.1.2.2 Klasse TTCN3Message

Hat zwei Aufgaben: Den Versand und Empfang der Nachricht.

Beim Versand speichert ein Objekt dieser Klasse das Datum, das zu versenden ist – Objekt der Klasse `TTCN3Template`. Des Weiteren enthält es die Referenz auf die Komponente, die das Datum versendet hat. Dieses Objekt wird sodann auf die Eingangsqueue des Empfängerports gelegt (Methode *addMesage()*.)

Beim Empfang wird ein Objekt dieser Klasse erstellt, das wiederum das empfangene Datum, sowie die Referenz auf dessen Sender speichert. Dieses Objekt wird dann von den Empfangsfunktionen *receive()*, *trigger()* und *checkReceive()* zurückgegeben.

KONSTRUKTOR

`TTCN3Message(TTCN3Template message, Components sender)`

Erstellt ein neues Nachrichtenobjekt und speichert die übergebene Nachrichtendatum und die Senderreferenz.

FUNKTIONEN

`GetMessage()`
liest das Nachrichtendatum wieder.

`GetSender()`
ermittelt die gespeicherte Senderreferenz.

`GetType()`
ermittelt den Datentyp des gespeicherten Nachrichtendatums.

5.3.1.3. Alt-Statement

Da die Port und Timer Operationen in TTCN3 blockierende Operationen sind und das Auftreten der Ereignisse, die diese Operationen aktivieren, nicht vorhergesagt werden können, ist es notwendig, mehrere dieser Operationen parallel ausführen zu können. Diese Aufgabe erfüllt das Alt-Statement. Es erlaubt den parallelen Aufruf von solchen Operationen, wobei beim gleichzeitigen Auftreten von entsprechenden Ereignissen eine Prioritätsreihenfolge gegeben ist, so dass Operationen, die weiter oben im Alt-Statement stehen, bevorzugt werden. Außerdem bietet das Alt-Statement noch die Möglichkeit, jede Operation mit einem booleschen Ausdruck zu versehen, der die Aktivierung der Alternativen zusätzlich einschränken kann.

Bei der Implementierung des Alt-Statements bekommen die einzelnen Alternativen keine eigenen Threads, sondern alles wird sequentiell in einem Thread durch eine entsprechende Folge von if-Anweisungen abgearbeitet. Dies erleichtert die Synchronisation zur Einhaltung der Prioritäten, allerdings erfordert diese Vorgehensweise auch, dass die Port und Timer Operationen nicht-blockierend sind. Das wiederum erfordert eine Art Producer-Consumer System zum Empfang der Ereignisse, welches in Java einfach durch die wait und notify Methoden realisiert werden kann. Beispiele: 5.7, 5.8.

```
alt {
    [] pt.receive(c_i) { setverdict( pass ) };
    [] t_guard.timeout {setverdict( fail ) }
}
```

Beispiel 5.7: Ein einfaches Beispiel eines Alt Statements in TTCN3

5.3.1.4. Typen

Typen spielen in TTCN-3 und Typenklassen im Laufzeitsystem eine enorme Rolle, da sie die Struktur von Daten definieren, die zwischen einzelnen Komponenten ausgetauscht werden.

5.3.1.4.1 Typen und Subtyping in TTCN-3

Strukturtypen wie *record* oder *set of* enthalten Felder eines bestimmten Typs, ob eines primitiven oder eines Strukturtyps. Daher lassen sie sich rekursiv auf primitive Typen zurückführen.

Implementiert sind folgende Grundtypen von TTCN-3: *integer*, *float*, *boolean*, *verdicttype*, *bitstring*, *hexstring*, *octetstring* und *charstring*.

Definiert durch TTCN-3 sind sie wie folgt: Die Grundtypen *integer*, *float* nehmen unbeschränkte Zahlenwerte auf. *Boolean* erlaubt zwei Wahrheitswerte, *verdicttype* – fünf Werte.

```
Object waiter = new Object();
synchronized (waiter) {
    Registrierung des waiter Objekts bei den Ports/Timern
    do {
        Take Snapshot...
        if (_b.getBooleanValue() && !chosen[0]) {
            TTCN3Message msg = null;
            if ((msg = comp.getPort("_pt").receive(_c_i, null,
                    ss)) != null) {
                chosen[0] = true;
                {
                    comp.getverdict().setVerdict(1);
                }
            }
        }
        if (!chosen[0]) {
            if (comp.getTimer("_t_guard").isTimedout(ss)) {
                chosen[0] = true;
                {
                    comp.getverdict().setVerdict(3);
                }
            }
        }
        waiter.wait();
    } while (nichts ausgew"ahlt oder repeat);
}
```

Beispiel 5.8: Die entsprechende Übersetzung in Java/Pseudo Code

Charstring nimmt Stings unbegrenzter Länge von ASCII-Zeichen. Bitmustertypen *bitstring*, *hexstring*, *octetstring* sind Strings unbeschränkter Länge, wobei die Belegungen Bitmuster sind in einer Binär-, Hexadezimaldarstellung und einer Hexadezimaldarstellung zu Basis 256 (zwei Hexadezimalzeichen) [14, S. 25].

Subtyping ist ein Konzept von TTCN-3, das erlaubt weitere Typen zu definieren, die Werte eines Grundtyps sind und in einem beschränkten Wertebereich liegen dürfen (Subtyping ist eine Wertebereichsdefinition). Es werden vier Arten des Subtyping in TTCN-3 definiert, wobei unterschiedliche Grundtypen unterschiedliche Arten, auch in Kombination, unterstützen:

- *Werteliste*. Es dürfen nur bestimmte, definierte Werte einer Variable zugewiesen werden.
- *Bereichsdefinition*. Erlaubte Werte müssen in einem bestimmten Wertebereich liegen.
- *Längendefinition*. Strings dürfen eine bestimmte Anzahl von Zeichen besitzen.
- *Stringmuster*. Werte, die einer Stringvariable zugewiesen werden dürfen, müssen einem regulären Ausdruck entsprechen.

Subtyping ähnelt sehr der Vererbung und ist transitiv: Ist ein Typ von einem Grundtyp durch Subtyping definiert worden, so kann von diesem Typ ein weiterer durch Subtyping definiert werden. Dabei ist der Wertebereich eines so definierten Typs stets eine Teilmenge des Wertebereichs des Typs von dem dieser Typ definiert wurde. Dies soll nun näher erläutert werden.

Ein Typ wird definiert durch den Code, hier z.. ein Subtyp von *integer*:

```
type integer MyInteger;
```

Danach können Variablen und Konstanten mit diesem Typ erstellt werden:

```
var   MyInteger MyIntegerVar;
var   MyInteger MyIntegerVariable := 25;
const MyInteger MyIntegerConst := 125;
```

Dabei wurde im Grunde ein Integer-Subtyp *MyInteger* ohne Wertebereichseinschränkung definiert, die obige Initialisierung ist somit mit dieser gleichwertig:

```
var   integer MyIntegerVar;
var   integer MyIntegerVariable := 25;
const integer MyIntegerConst := 125;
```

Wertliste ist die einfachste Bereichsdefinition und wird von *allen* Typen in TTCN-3, nicht nur von Grundtypen unterstützt:

```
type integer MyVLInteger (2,46,80);
```

Nun kann eine Variable dieses Typs nur die angegebene Werte annehmen:

```
var   MyVLInteger MyIntegerVar;            // ok
      MyIntegerVar := 46;                  // ok
var   MyVLInteger MyIntegerVariable := 2;  // ok
const MyVLInteger MyIntegerConst := 125;   // Laufzeitfehler
```

Wertebereich ist universeller

```
type integer MyVRInteger (2..8);
```

und kann (beim *integer* - Grundtyp) mit einer Liste kombiniert werden:

```
type integer MyVLRInteger (2..8, 25, 40, 30..50);
```

Längen- und *Stringmusterdefinition* gibt es nur für Strings.

```
// alle Strings der L"ange 2 bis 8 Zeichen.
type charstring MyString length (2..8);

//Alle Strings mit Zeichen a bis q,
//z.. "aal" oder "abba", aber nicht "ABBA"
type charstring MyString (pattern '[a-z]#(2,9)');

//Wie vorhin, nur Strings der L"ange 5 bis 9 Zeichen, weil
//der Stringmuster Strings nur bis zur L"ange von 9 Zeichen erlaubt.
type charstring MyString (pattern '[a-z]#(2,9)') length (5..18);
```

Dass die Musterdefinition in den Klammern steht, soll verdeutlichen, dass es wieder eine Listendefinition ist. Zwar eine etwas klügere, diese kann aber nicht mit Listen und Bereichen kombiniert werden, nur mit der Längendefinition.

Wie bereits erwähnt, erlauben nicht alle Typen alle Bereichsdefinitionen zu kombinieren, [14, S. 25]. So erlauben *integer*, *float* nur Wertelisten und Wertebereiche zu definieren und zu kombinieren. *Boolean* und *verdicttype* unterstützen *nur* Wertelistendefinition. Bitmustertypen *bitstring*, *hexstring* und *octetstring* unterstützen nur Wertelisten- und Längendefinition (kombinierbar) und der Stringtyp *charstring* erlaubt alle Definitionsarten.

5.3.1.4.2 Implementierung in Java

Für den Typ *float* von TTCN-3 gibt steht eine Klasse `FloatType`, die intern mit Werten aus `Double` arbeitet.

Für den *integer* von TTCN-3 gibt es zwei Klassen, `IntType` und `IntegerType`. Die erste arbeitet mit Werten des `Long` - Bereiches und der ist zwar groß, aber beschränkt. Die `IntegerType` Klasse arbeitet intern mit `BigInteger` und verarbeitet unbeschränkte Variablenwerte. Die Klasse `IntType` ist somit schneller und kann für Subtypen verwendet werden, die sowieso beschränkt definiert werden. Beide Klassen sind inkompatibel, Werte des Typs `IntegerType` lassen sich weder der Variable von `IntType` zuweisen, noch für eine Operation mit einer Variable dieses Typs verwenden, und umgekehrt.

Ein Typ ist eine Klasse mit der eine Variable oder Konstante erstellt werden soll. Dabei muss diese Klasse bereits erlaubte Werte vorgelegt bekommen, ein nachträgliches Definieren wäre so nicht sinnvoll.

Dazu stellt `TTCN3Type` zwei Methoden bereit:

```
defValues(),
defLength()
```

Welche nichts tun und von den konkreten Typen überschrieben werden müssen, sofern TTCN-3 eine entsprechende Definition dafür vorsieht. Ein Subtyp wird als Klasse definiert, *die einfach vom Grundtyp erbt* und die obigen Methoden werden im Konstruktor aufgerufen. Sobald eine

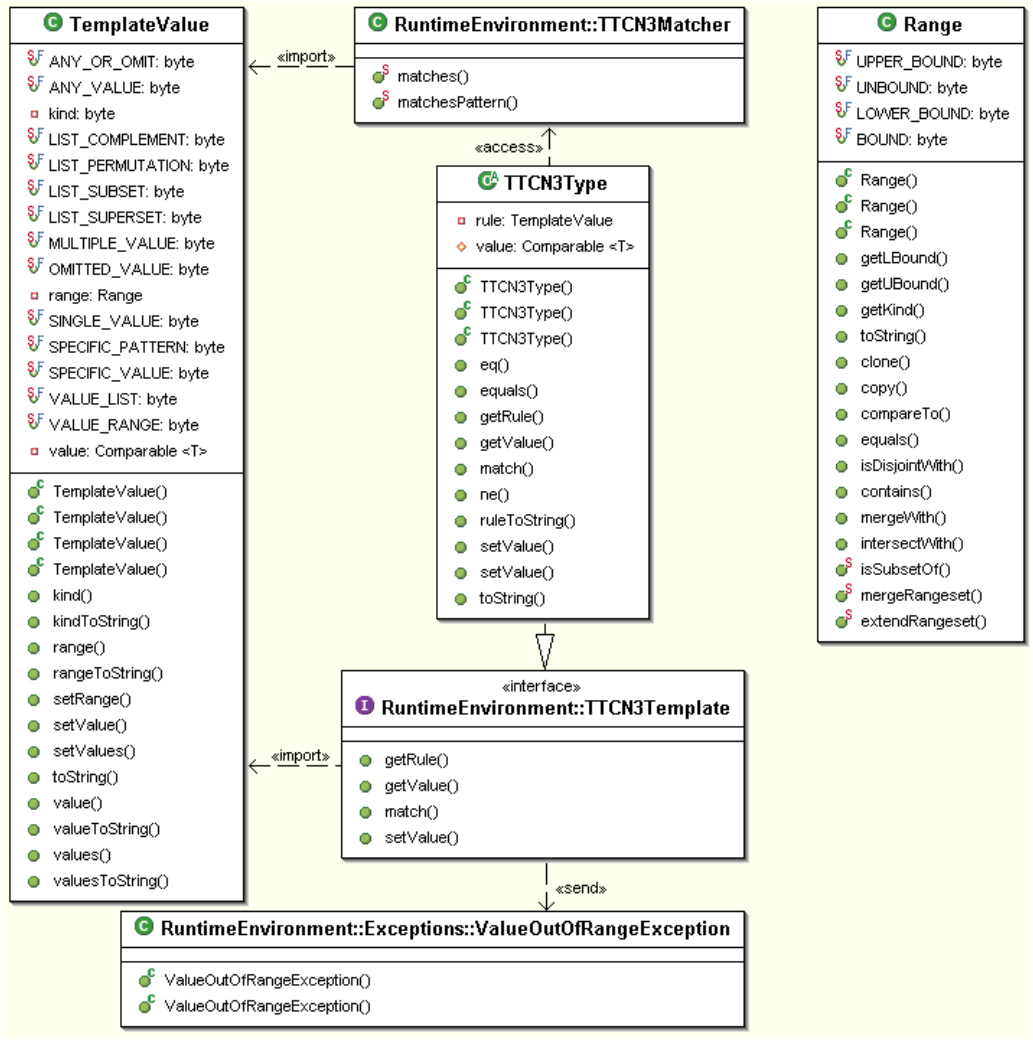

Abbildung 5.28.: Kernimplementierung, Klassendiagramm.

Variable dieses Typs erstellt wird, sind die Werte dann bereits belegt. Die Stringmusterdefinition ist eine Art Listendefinition. Aus diesem Grund wurde diese Aufgabe ebenfalls an die Funktion *defValues()* übertragen.

Als erstes sollen die Coreklassen vorgestellt werden. Danach werden einzelne konkrete Grundtypen und deren Einsatz erläutert (Abschnitt 5.3.1.4.9).

5.3.1.4.3 Kernimplementierung

Die Coreklassen sind im Klassendiagramm dargestellt, Abbildung 5.28

5.3.1.4.4 Interface TTCN3Template

Die Klasse `TTCN3Template` dient dazu, Variablenwerte zu speichern oder andere Templates bezüglich der gespeicherte Werte zu verifizieren. Die Differenzierung beider Rollen ist jedoch nicht eindeutig, da ein Template, welches ein Integerwert speichert, bereits ein anderes Template verifizieren kann – darauf verifizieren, ob dieses den gleichen Integerwert speichert oder nicht.

ABSTRAKTE FUNKTIONEN

`SetValue(Comparable value) throws ValueOutOfRangeException;`
speichert einen Variablenwert in das Template.
`SetValue(TemplateValue rule);`
speichert die Verifizierungsregel in das Template. Das sind Objekte der Klasse TemplateValue, die Regeln definieren, mit denen sich wiederum Werte anderer Templates verifizieren lassen.
`GetValue();`
liefert den gespeicherten Variablenwert zurück.
`GetRule();`
liefert die gespeicherte Verifizierungsregel zurück.
`match(TTCN3Template template);`
verifiziert das angegebene Template bezüglich der gespeicherten Verifizierungsregel, sofern diese nicht gesetzt ist, bezüglich der Gleichheit mit dem gespeicherten Variablenwert.

5.3.1.4.5 Klasse TTCN3Matcher

Die Klasse `TTCN3Matcher` definiert zwei statische Verifikationsmethoden. Sie wird statisch verwendet und muss daher nicht instanziiert werden.

Statische Funktionen

`Matches(Comparable obj, TemplateValue rule)`
übernimmt ein Objekt *obj* und ein Regelobjekt der Klasse TemplateValue *rule* und prüft, ob das Objekt dem definierten Regelobjekt genügt.
`MatchesPattern(Comparable obj, Comparable pattern)`
implementiert das Stringmatching. Es übernimmt zwei Strings (internes Casting `Comparable` nach `String`) und ermittelt, ob der String *obj* dem Stringmuster *pattern* entspricht.

5.3.1.4.6 Klasse TTCN3Type

Die abstrakte Klasse `TTCN3Type` implementiert das Interface `TTCN3Template` und ist eine Superklasse für alle TTCN-3 Typen. Die Klasse `TTCN3Type` systematisiert außerdem das Subtyping. Objekte der Klasse `TTCN3Type` und deren Erben lassen sich in zwei Arten unterteilen: *Wertevariable* speichern ein Variablenwert, beispielsweise Integer oder String und beschränken die Templateverifizierung nur auf die Gleicheitsprüfung. *Templatevariablen* hingegen speichern die Verifizierungsregel (Objekt der Klasse `TemplateValue`) und führen eine vollständige Templateverifizierung durch.

KONSTRUKTOREN

`TTCN3Type(),`
`TTCN3Type(Comparable value),`
`TTCN3Type(TemplateValue rule)`
Erstellen eine neue Variable des gegebenen Typs. Ein parameterloser Konstruktor erstellt eine leere Variable. Der zweite und dritte Konstruktor übernehmen einen Variablenwert *value* bzw. den Regelobjekt *rule* und erstellen dementsprechend eine Wertevariable bzw. eine Templatevariable.

FUNKTIONEN

`SetValue(Comparable value) throws ValueOutOfRangeException`
setzt und ändert den gespeicherten Variablenwert (Abbildung 5.29). Vor dem setzen wird der neue Wert gegen den Wertebereich verifiziert, der durch den Subtyping definiert ist. Befindet

sich der Wert außerhalb des Wertebereichs, wird die Ausnahmemeldung vom Typ ValueOutOfRangeException ausgeworfen, ansonsten wird er übernommen. Das bereits gespeicherte Regelobjekt wird dadurch entfernt, so dass die Variable zu einer Datenvariable umgewandelt wird.

`SetValue(TemplateValue rule)`

übernimmt das Regelobjekt und macht die Variable zur Templatevariable, indem der gespeicherte Variablenwert entfernt wird (Abb. 5.29).

`GetValue()`

liefert den gespeicherten Variablenwert zurück.

`GetRule()`

liefert das gespeicherte Regelobjekt zurück.

`CheckValue(Comparable value)`

prüft den angegebenen Wert gegen den definierten Wertebereich. Diese Funktion ist abstrakt und muss von konkreten Typenklassen überschrieben werden, um die Bereichsprüfung dem Typ entsprechend durchführen zu können.

`match(TTCN3Template template),`
`equals(TTCN3Template template)`

Die Methode *match()* verifiziert den Wert, der im angegebenen Template gespeichert ist, gegen das eigens gespeicherte Regelobjekt, sofern dieses definiert (gesetzt) ist. Ansonsten prüft es die Gleichheit mit dem eigens gespeicherten Variablenwert. Die Methode *equals()* ruft intern die Methode *match()* auf und ist somit deren Stellvertreter.

`ToString(Comparable value)`

übernimmt einen Wert und stellt ihn als String dar. Diese Funktion muss von den konkreten Klassen überschrieben werden, um eine typentsprechende Wertedarstellung zu erreichen.

`ToString()`

ruft *toString()* mit dem eigens gespeicherten Variablenwert als Parameter auf und erhält somit dessen typentsprechende Darstellung, sofern jene Funktion von der konkreten Typklasse überschrieben wurde. Ist das Variablenwert nicht gesetzt, wird der String `"{undefined}"` zurückgegeben.

`RuleToString()`

ermittelt die Stringdarstellung des gespeicherten Regelobjektes. Ist dieses nicht gesetzt, wird der String `"{undefined}"` zurückgegeben.

`eq(TTCN3Type op)`

Der TTCN-3 Operator ==, der für jeden TTCN-3 Typ definiert ist. Prüft auf die Gleichheit des eigenen Variablenwertes mit dem der angegebenen Variable.

`ne(TTCN3Type op)`

Der TTCN-3 Operator `!=`, der ebenfalls für jeden TTCN-3 Typ definiert ist. Prüft die Ungleichheit des eigenen Variablenwertes mit dem der angegebenen Variable.

5.3.1.4.7 Klasse TemplateValue

Die Klasse `TemplateValue` definiert Regeln, mit denen die in Templates gespeicherte Variablenwerte verifiziert werden.

KONSTANTEN

Konstanten definieren die Art der Regel. Sie bestimmen, wie weitere, im Regelobjekt gespeicherte Werte zu interpretieren sind.

`SPECIFIC_VALUE`

entspricht dem Templatewert '-': Prüfen auf die Gleichheit mit dem gesetzten Wert (*value*).

`OMITTED_VALUE`

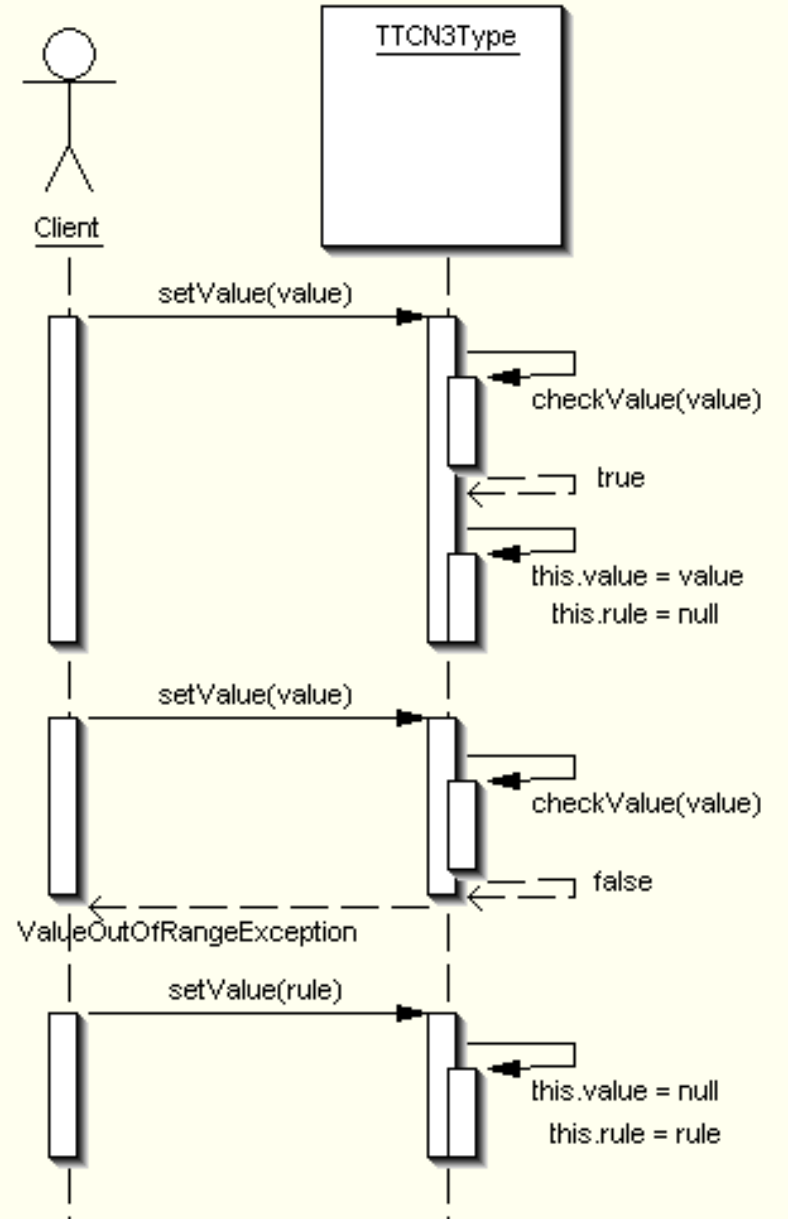

Abbildung 5.29.: Wert oder Regelobjekt setzen.

Der Templatewert '*omit*': Der zu prüfende Wert ist ausgelassen, der Templatewert soll immer akzeptiert werden.

`SPECIFIC_PATTERN`

Der zu prüfende Wert muss gegen das Stringmuster verifiziert werden, das in der Eigenschaft *value* gespeichert ist.

`ANY_VALUE`

Der Templatewert '*?*': Er soll jeder definierte Wert akzeptiert werden (alles außer *null*).

`ANY_OR_OMIT`

Der Templatewert '*': Es soll auch der undefinierte Wert akzeptiert werden (auch *null*).

`VALUE_LIST`

Der Templatewert '*(...)*': Der zu prüfende Wert muss in der Werteliste vorhanden sein, der mit der Eigenschaft *values* definiert ist.

`LIST_COMPLEMENT`

Der Templatewert '*complement(...)*': Der zu prüfende Wert darf nicht in der Werteliste vorhanden sein, die mit der Eigenschaft *values* definiert ist.

`VALUE_RANGE`

der Templatewert '*(lowerBound .. upperBound)*': Der zu prüfende Wert muss innerhalb des mit der Eigenschaft *range* definierten Wertebereich liegen.

`LIST_SUPERSET`

Der Templatewert '*superset(...)*'

`LIST_SUBSET`

Der Templatewert '*subset(...)*'

`SINGLE_VALUE`

Der Templatewert '*?*' innerhalb Wertebelegungen

`MULTIPLE_VALUE`

Der Templatewert '*' innerhalb Wertebelegungen:

`LIST_PERMUTATION`

Der Templatewert '*permutation(...)*':

KONSTRUKTOREN

`TemplateValue(byte kind),`
`TemplateValue(byte kind, Comparable value),`
`TemplateValue(byte kind, Collection values),`
`TemplateValue(byte range)`

erstellen ein neues Regelobjekt einer bestimmten Art (siehe *Konstanten*) und belegen es mit Zusatzinformationen, die von der Regelart (*kind*) interpretiert werden müssen.

FUNKTIONEN

`SetValues(Comparable value)`

setzt einen einfachen Templatewert.

`SetRange(Range range)`

setzt den zu akzeptierenden Wertebereich.

`SetValue(Collection values)`

setzt eine Werteliste.

`Kind()`

Ermittelt die Art des Regelobjektes.

`Value()`

ermittelt den gesetzten Templatewert.

`Values()`

ermittelt die gesetzte Werteliste.

`Range()`
ermittelt den gesetzten Wertebereich
`ValueToString()`,
`valuesToString()`,
`rangeToString()`,
`kindToString()`,
`toString()`
berechnen Stringdarstellungen des gesetzten Wertes, der Werteliste, des gesetzten Wertebereichs, der Art und die gesamte Stringdarstellung des Regelobjektes.

5.3.1.4.8 Klasse Range

Die Klasse `Range` wird für Bereichsdefinitionen verwendet. Zum Einen kennt TTCN-3 beschränkte Rangdefinitionen und einseitig unbeschränkte Bereichsdefinitionen, zum Anderen benötigen Typenklassen Werte verschiedenes Javatyps, was den Einsatz dieser Klasse erfordert. Als Bereichsgrenzen arbeitet diese Klasse mit Typen, die das Java Interface `Comparable` implementiert. Dies sind alle erforderliche Javaklassen, wie `Integer`, `Boolean` oder `String`. Dieser Umstand macht die Klasse `Range` universell einsetzbar.

KONSTANTEN
definieren, welche Seite des Wertebereichs unbeschränkt ist. Diese werden der Eigenschaft *kind* zugewiesen.
`BOUND`
Der Wertebereich ist beidseitig beschränkt, „`(2..25)`“ beispielsweise.
`UPPER_BOUND`
Der Wertebereich ist nach oben beschränkt, nach unten nicht, „`(-infinity..25)`“ beispielsweise.
`LOWER_BOUND`
Der Wertebereich ist nach unten beschränkt, nach oben nicht, „`(2..infinity)`“ beispielsweise.
`UNBOUND`
Der Wertebereich ist beidseitig unbeschränkt, „`(-infinity..infinity)`“ wäre es dann.
KONSTRUKTOREN
`Range()`
definiert einen beidseitig unbeschränkten Bereich. Bereichsgrenzen sind in diesem Fall undefiniert (*getLBound()* und *getUBound()* liefern beide *null*) und die Art ist auf `UNBOUND` gesetzt.
`Range(Comparable lBound, Comparable uBound)`
definiert einen beschränkten (falls keins der Parameter *null* ist) oder einen unbeschränkten Bereich, wobei die unbeschränkte Grenze auf *null* zu setzen ist, und zwar:

- Ist keine Grenze mit *null* angegeben, liefert *getKind()* den Wert `BOUND`, *getLBound()* und *getUBound()* den Wert von *lBound* und *uBound*, wobei durch einen Vergleich sichergestellt wird, dass *getLBound()* $\leq$ *getUBound()* gilt.
- Ist *lBound* auf *null* gesetzt, liefert *getKind()* den Wert `UPPER_BOUND` und beide Funktionen *getLBound()* , *getUBound()* den Wert von *uBound*.
- Ist *uBound* auf *null* gesetzt, liefert *getKind()* den Wert `LOWER_BOUND` und beide Funktionen *getLBound()* , *getUBound()* den Wert von *lBound*.
- Sind beide Schranken *null*, liefert *getKind()* ebenfalls den Wert `UNBOUND`.

`Range(Comparable aBound, boolean upperBound)`
definiert einen einseitig beschränkten Bereich. *getLBound()* , *getUBound()* liefern den Wert von *aBound*, *getKind()* liefert `UPPER_BOUND` (nach oben beschränkter Wertebereich), falls der boolesche Parameter *upperBound* auf *true* gesetzt ist und den Wert `LOWER_BOUND`, falls Das Objekt mit *upperBound* = *false* initialisiert wurde.

FUNKTIONEN

`GetLBound()`
liefert die untere Grenze des Wertebereichs, falls dieser beidseitig beschränkt ist und den Wert der einzig definierten Schranke, wenn der Bereich einseitig beschränkt ist. Für beidseitig unbeschränkte Wertebereiche wird *null* zurückgegeben.

`GetUBound()`
liefert die obere Grenze des Wertebereichs, falls dieser beidseitig beschränkt ist und den Wert der einzig definierten Schranke, wenn der Bereich einseitig beschränkt ist. Für beidseitig unbeschränkte Wertebereiche wird *null* zurückgegeben.

`GetKind()`
gibt die Art des Wertebereichs des Rückgabewertes an.

- `BOUND` für einen beidseitig beschränkten,
- `LOWER_BOUND` für nach unten beschränkten, nach oben unbeschränkten,
- `UPPER_BOUND` für nach oben beschränkten, nach unten unbeschränkten,
- `UNBOUND` für beidseitig unbeschränkten Wertebereich.

`toString()`
Stellt die Art und die Schranken als String dar. Die Rückgabe hat die Form („2“ und „25“ sind stellvertretend für Bereichsschranken im konkreten Fall).

- „`[-infinity..infinity]`“ für einen unbeschränkten Bereich,
- „`[2..infinity]`“ für einen nach oben unbeschränkten Wertebereich,
- „`[-infinity..25]`“ für einen nach oben beschränkten Wertebereich,
- „`[2..25]`“ für einen beidseitig beschränkten Wertebereich.

`Clone()`
liefert einen Wertebereich desselben Datentyps, derselben Art und mit gleichen Schranken, wie der aktuelle Bereichsobjekt.

`Copy(Range range)`
erstellt einen Bereichsobjekt des eigenen Datentyps und kopiert die Art und Bereichsgrenzen vom angegebenen Bereichsobjekt.

`CompareTo(Object obj)`
vergleicht den aktuellen mit dem angegebenen Bereichsobjekt und liefert eine *0*, wenn beide Objekte gleich sind. Es liefert eine *-1*, falls dieses Objekt kleiner und *1*, falls es größer ist als das angegebene Wertebereichsobjekt. Sind *A*, Wertebereichsobjekte, so gilt *A*<, falls einer der Fälle zutrifft:

- *A* ist nach unten nicht beschränkt (`UPPER_BOUND` oder `UNBOUND`) und ist nach unten beschränkt (`LOWER_BOUND` oder `BOUND`).

- Beide sind nach unten beschränkt und die Untere Grenze von *A* ist kleiner, als die von .
- Beide sind nach unten beschränkt, haben gleiche untere Grenze, *A* ist nach oben beschränkt (`BOUND`), nicht (`LOWER_BOUND`).
- Beide sind beidseitig beschränkt, haben gleiche untere Grenze, obere Grenze von *A* ist kleiner, als die von .

Sind beide Objekte beidseitig unbeschränkt oder beidseitig beschränkt mit identischen Grenzwerten, liefert die Funktion eine *0* (Objekte sind gleich). Dabei wird der Datentyp beider Objekte nicht beachtet, da die Grenzen das Interface `Comparable` implementieren und vergleichbar sind.

`Equals(Object obj)`

vergleicht den aktuellen mit dem angegebenen Bereichsobjekt und liefert `true`, wenn die Art und Grenzwerte gleich sind (bzw. wenn `compareTo(o)` eine *0* liefern würde).

`IsDisjointWith(Range range)`

liefert `true`, wenn beide Bereichsobjekte disjunkt sind, d.. wenn ein Bereich vollständig außerhalb des anderen liegt. Dies ist dann der Fall, wenn die untere Grenze eines Bereichs echt größer ist als die ober Grenze des Anderen.

`Contains(Comparable obj)`

übernimmt ein Objekt der `Comparable`, implementiert es und liefert *true*, wenn es im Bereich des aktuellen Objektes liegt. Das ist dann der Fall, wenn die untere Grenze des aktuellen Bereichs nicht definiert oder kleiner als der angegebene Wert ist, *und* wenn die obere Grenze des aktuellen Bereiches undefiniert oder größer als der angegebene Wert ist.

`MergeWith(Range range)`

sind beide disjunkte Bereichsobjekte (das aktuelle und das angegebene). Sie liefern in dieser Funktion entweder *null* oder ein Bereichsobjekt, das eine Vereinigung beider Wertebereiche speichert.

`IntersectWith(Range range)`

sind beide disjunkte Bereichsobjekte (das aktuelle und das angegebene). Sie liefern in dieser Funktion *null* oder ein Bereichsobjekt, das ein Durchschnitt beider Wertebereiche speichert.

`isSubsetOf(TreeSet subset, TreeSet superset)`

Diese statische Funktion liefert `true` zurück, wenn die Menge *subset* der Bereichsobjekte eine Teilmenge von *superset* ist. Das ist dann der Fall, wenn jeder Bereich aus *subset* in mindestens einem Bereich aus *superset* vollständig enthalten ist.

`mergeRangeset (TreeSet set)`

Diese statische Funktion bildet eine Menge von Bereichsobjekten und stellt sicher, dass alle Bereiche in dieser Menge disjunkt sind, indem sie die sich überlappende Bereiche vereint.

`extendRangeset (TreeSet set, TreeSet bySet)`

Übernimmt zwei Mengen von Bereichsobjekten. Die Ergebnismenge enthält Bereiche, die in mindestens einem Bereich aus *set* und mindestens einem Bereich aus *bySet* vollständig enthalten ist (oder als Bereinigung der beiden Bereichsmengen verstanden werden kann). Das Ergebnis ist wieder in *set* gespeichert.

5.3.1.4.9 Konkrete Typenimplementierung

Kommen wir jetzt zur konkreten Typenimplementierung. Da es viele Klassen sind, werden diese in Gruppen vorgestellt, wobei einzelne Gruppen durch Vertreterklassen erklärt werden können.

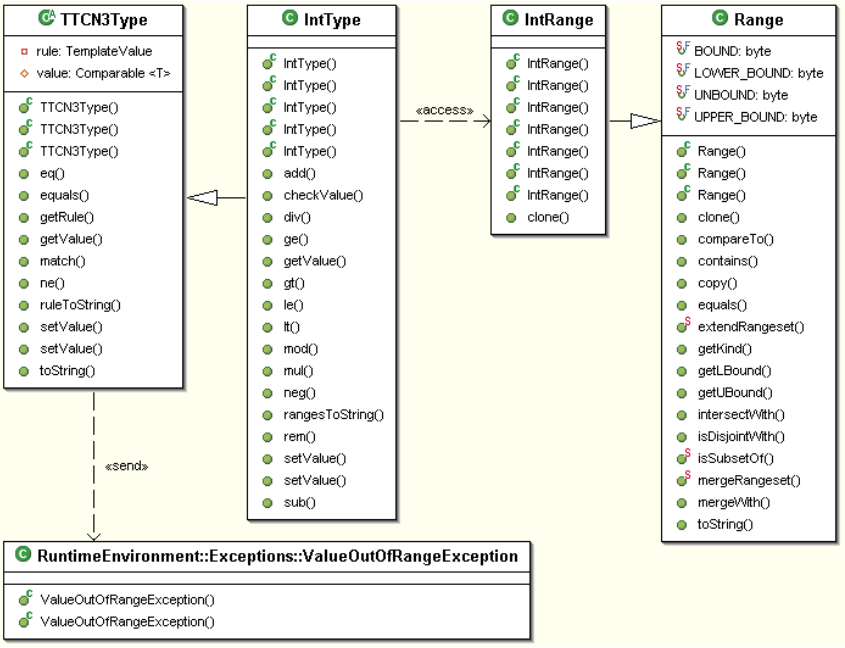

Abbildung 5.30.: Klassen um `IntType` – Klassendiagramm

5.3.1.4.10 Zahlenklassen

Das Klassenstruktur (Abbildung 5.30) lässt sich anhand der `IntType` erklären und auf `IntegerType` und `FloatType` einfach übertragen.

5.3.1.4.11 Klasse IntRange

Wird für Bereichsdefinitionen für `IntType` benötigt. Die Grenzwerte sind vom Typ `Long`. Die Aufgabe der Klasse `IntRange` ist vor Allem, die Grenzwerte des Primitiven Typs *long* in den Typ `Long` zu konvertieren und somit einfachere Konstruktoren anzubieten.

KONSTRUKTOREN

`IntRange()`

Unbeschränkter Wertebereich.

`IntRange(long lBound, long uBound)`

Beidseitig beschränkter Bereich.

`IntRange(long lBound, boolean upperBound)`, `IntRange(Long aBound, boolean upperBound)`

Einseitig beschränkter Bereich. Bei *upperBound* = *true* ist untere Grenze unbeschränkt, bei *upperBound* = *false* ist es die obere Grenze.

`IntRange(Long lBound, Long uBound)`

Beidseitig beschränkter, beidseitig unbeschränkter oder einseitig beschränkter Bereich: Als eine unbeschränkte Grenze wird die angenommen, welche mit *null* initialisiert ist.

`IntRange(Long lBound, long uBound)`

Nach oben beschränkter Bereich. Diesen Konstruktor wählt der Java Compiler, wenn das Bereichsobjekt mit `new IntRange(null,25)` initialisiert wird.

`IntRange(long lBound, Long uBound)`

Nach unten beschränkter Bereich. Diese Konstruktor wählt der Java Compiler, wenn das Bereichsobjekt mit `new`IntRange(25, null) initialisiert wird.

FUNKTIONEN

`clone()`

Erstellt ein Bereichsobjekt des Javatyps `IntRange` mit identischen Wertebelegungen.

5.3.1.4.12 Klasse IntType

KONSTRUKTOREN

`IntType()`

Erstellt eine neue leere Variable.

`IntType(IntType ivalue)`

Erstellt eine neue Wertevariable und übernimmt den Wert der angegebenen Variable.

```
IntType(long value), IntType(Long value)
```

Erstellen eine neue Wertevariable mit dem angegebenen Wert.

`IntType(TemplateValue rule)`

Erstellt eine neue Templatevariable mit der angegebenen Templateregel.

```
setValue(Long value) throws ValueOutOfRangeException,
setValue(long value) throws ValueOutOfRangeException
```

Übernehmen den angegebenen Wert, falls dieser im eigenen Wertebereich liegt, wenn nicht wird eine Ausnahmemeldung ausgegeben.

`getValue ()`

Ermittelt den gesetzten Variablenwert.

```
defValues(long[] values), defValues(long[] values)
```

Wertelistendefinition: Als `protected` deklarierte Funktion, ist nur für die erbende Klassen sichtbar; sie dient zur Wertebereichsdefinition und schränkt den in der Superklasse definierten Wertebereich auf die Liste der angegebenen Werte weiter ein.

`defValues(IntRange[] ranges)`

Bereichsdefinition: Als `protected` deklarierte Funktion, dient zur Wertebereichsdefinition und schränkt den in der Superklasse definierten Wertebereich auf die Liste der angegebenen Wertebereiche weiter ein.

```
defValues(Long[] values, IntRange[] ranges), defValues(long[]
values, IntRange[] ranges)
```

Kombination der Wertelisten- und Bereichsdefinition: Als `protected` deklarierte Funktion, dient zur Wertebereichsdefinition und schränkt den in der Superklasse definierten Wertebereich auf die Liste der angegebenen Werte und Wertebereiche weiter ein.

`checkValue (Comparable value)`

Übernimmt einen Wert und überprüft, ob dieser im definierten Wertebereich liegt. Die Methode überschreibt jene vom `TTCN3Type` und wird von ihm aufgerufen, bevor der neue Wert in *setValue()* übernommen wird.

`rangesToString()`

Stellt den definierten Wertebereich als String dar.

`neg()`

Unäre Operation - von TTCN-3. Negiert den aktuellen Wert und liefert eine Variable zurück, die das Ergebnis speichert.

`add(IntType op)`

TTCN-3 Operation +: addiert zum aktuellen Wert den der angegebenen Variable und liefert eine Variable zurück, die das Ergebnis speichert.

`sub(IntType op)`

TTCN-3 Operation -: subtrahiert vom aktuellen Wert den der angegebenen Variable und liefert eine Variable zurück, die das Ergebnis speichert.

`mul(IntType op)`

TTCN-3 Operation *: Multipliziert den aktuellen und den Wert der angegebenen Variable und liefert das Ergebnis in einer neuen Variable zurück.

`div(IntType op)`

TTCN-3 Operation /: Dividiert *ganzzahlig* den Aktuellen durch den Wert der angegebenen Variable und liefert das Ergebnis in einer neuen Variable zurück.

`rem(IntType op)`

TTCN-3 Operation *rem*: Das Ergebnis ist eine Variable mit dem Divisionsrest des aktuellen und des Wertes der angegebenen Variable.

`mod(IntType op)`

TTCN-3 Operation *mod*: Das Ergebnis ist eine Variable mit dem *positiven* Divisionsrest des aktuellen und des Wertes der angegebenen Variable.

`gt(IntType op)`

TTCN-3 Operation >: Das Ergebnis ist eine Variable vom Typ `BoolenType`, die wahr ist, falls der aktuelle Wert größer ist, als der der angegebenen Variable.

`lt(IntType op)`

TTCN-3 Operation <: Das Ergebnis ist eine Variable vom Typ `BoolenType`, die wahr ist, falls der aktuelle Wert kleiner ist, als die der angegebenen Variable.

`ge(IntType op)`

TTCN-3 Operation $\geq$: Das Ergebnis ist eine Variable vom Typ `BoolenType`, die wahr ist, falls der aktuelle Wert größer oder gleich ist, als die der angegebenen Variable.

le(IntType op)

TTCN-3 Operation $\leq$: Das Ergebnis ist eine Variable vom Typ `BoolenType`, die wahr ist, falls der aktuelle Wert kleiner oder gleich ist, als die der angegebenen Variable.

Eine Anmerkung an dieser Stelle zur Funktion *defValues* in allen drei Varianten. Ein einzelner Wert kann als ein auf diesen Wert kollabierter Wertebereich behandelt werden. So lässt sich eine Liste von Werten als eine Liste von Wertebereichen behandeln und beide Listen lassen sich miteinander mischen.

Die Klasse `IntType` wird mit einem unbeschränkten Wertebereich definiert. Deren Erbe sehen die Funktion *defValues* und können eine eigene Liste von Werten und Bereichen übergeben, um ihren Wertebereich zu definieren. Ein Erbe dieser Klasse kann eine weitere Liste von Werte und Bereiche übergeben, um den Wertebereich weiter einzuschränken. Die Klasse Range unterstützt mit der Funktion *extendRangeset* genau diesen Vorgang.

Dies ist die Weise, auf die das Subtyping in den Zahlenklassen `IntType`, `IntegeType` und `FloatType` implementiert ist.

RECHNUNG MIT INTTYPE UND VOR ALLEM MIT DEREN ERBEN

Eine Wertliste, wie

```
type integer MyInteger (2,46,80);,
```

lässt sich folgendermaßen definieren:

```
class MyInteger extends IntType {
    public MyInteger () {
        defValues(new Long[]{new Long(2), new Long(46), new Long(80)});
    }
}
```

Oder alternativ mit primitiven `long` - Zahlenobjekten:

```
class MyInteger extends IntType {
    public MyInteger () {
        defValues(new long[]{2, 46, 80});
    }
}
```

Für die Definition einzelner Wertebereiche mit denen ein `IntType` was anfangen kann, steht die Klasse `IntRange` bereit. Ein Bereich, der mit

`new IntRange(2, 8);`

definiert wurde, wird auf die Werte zwischen 2 und 8 beschränkt (In `TTCN-3` Code als `2..8` definiert). Ist ein Bereich mit

`new IntRange(2, true);`

definiert, ist er nach oben auf 2 beschränkt, dies entspricht somit `- infinity ..2` und ein Bereich

`new IntRange(2, false);`

entspricht einem nach unten auf 2 Beschränkten `2..infinity` Bereich.

Alternativ erlauben Bereiche bei der Definition auch den Wert *null* für eine unendliche Grenze.

Die Definitionen

```
new IntRange(2, true);
new IntRange(null, 2);
new IntRange(null, new Long(2));
```

führen somit zum selben Ergebnis `infinity..2` und

```
new IntRange(2, false);
new IntRange(2, null);
new IntRange(new Long(2), null);
```

führen zum folgenden Ergebnis von `2..infinity`.

Ein Parameterloses Konstruktor legt einen beidseitig unbeschränkten Wertebereich an.

Um also das folgende Beispiel

`type integer MyInteger (-infinity..8);`

zu definieren, reicht dieser Code

```
class MyInteger extends IntType {
    public MyInteger() {
        defValues(new IntRange[]{new IntRange(null, 8)});
    }
}
```

Für die Kombination steht die dritte Variante der Funktion *defValues()* zur Verfügung. Dem Code

```
type integer MyInteger (2..8, 25, 40, 30..50);
```

entspricht der Java-Code

```
class MyInteger extends IntType {
    public MyInteger () {
        defValues(
            new Long[]{new Long(25), new Long(40)},
            new IntRange[]{new IntRange(2, 8), new IntRange(30, 50)}
        );
    }
}
```

oder alternativ

```
class MyInteger extends IntType {
    public MyInteger () {
        defValues(
            new long[]{25, 40},
            new IntRange[]{new IntRange(2, 8), new IntRange(30, 50)}
        );
    }
}
```

Kann aber folgender Code funktionieren?

```
class MyInteger extends IntType {
    public MyInteger () {
        defValues(
            new long[]{25, 40}
        );
        defValues(
            new IntRange[]{new IntRange(2, 8), new IntRange(30, 50)}
        );
    }
}
```

Nein: Obwohl dieser identisch aussieht: Ein Aufruf von zwei getrennten *defValues()*, je für Liste und Bereiche, würde nicht korrekt funktionieren. Der Erste würde zwar Werte setzen (intern wird die Werteliste eigentlich zu einer Wertebereichs-Liste umgewandelt), der Zweite Aufruf jedoch würde einen Durchschnitt beider Bereichsdefinitionen setzen. Die Gesamtmenge der Bereiche würde in diesem Fall nur dem Bereich `40..40` bzw. dem Wert `40` entsprechen. Dies ergibt sich zwangsläufig aus der folgenden Anwendung:

Steht beispielsweise in `TTCN-3` folgender Code:

```
type integer MyInteger (25,40);
:
type MyInteger MySubInteger (2..8, 30..50);
```

erlaubt der Typ `MyInteger` nur beide Werte 25, 40 (er erbt vom `IntType` und schränkt sein unbeschränktes Bereich auf die beide Werte ein). Der `MySubInteger` schränkt diesen Wertebereich weiter auf den Bereich `(30..50)` ein und nur `(40..40)` ist der Durchschnitt beider Bereiche bzw. nur der Wert `40` liegt im Bereich des zweiten Typs.

In Java würde der obige Code, wie folgt, übersetzt werden:

```
class MyInteger extends IntType {
    public MyInteger () {
        defValues(
            new long[]{25, 40}
        );
    }
}
:
class MySubInteger extends MyInteger {
    public MySubInteger () {
        defValues(
            new IntRange[]{new IntRange(2, 8), new IntRange(30, 50)}
        );
    }
}
```

Der Konstruktor von `MySubInteger` würde also genau den Code aus der Fragestellung ausführen. Aus diesem Grund muss für eine Kombination von Wertelisten und Wertebereiche die kombinierte Variante von *defValues()* verwendet werden.

Dieser Code soll vorerst bleiben. Nach seiner Ausführung kann eine Variable des `MySubInteger` erstellt und gesetzt werden:

```
MySubInteger A = new MySubInteger();
A.setValue(0);  // A.getValue() = null, ValueOutOfRangeException
A.setValue(4);  // A.getValue() = 4    (Das Wert "ubernommen)
A.setValue(14); // A.getValue() = 4, ValueOutOfRangeException
```

5.3.1.4.13 Klasse IntegerType

Bis auf Konstruktoren und Wertzuweisungen ist sie identisch mit **`IntType`** verwendbar. Die Klasse `IntegerType` verwendet intern die Klasse `BigInteger` und kann auf unterschiedliche Arten initialisiert werden. Von diesen wurden drei auf **`IntegerType`** übertragen und einige zusätzlich hinzugefügt:

- Durch einen Wert des Typ `BigInteger`.
- Durch einen Objekt des Typ `IntegerType`.
- Über einen String zu Basis 10.
- Über einen String zur beliebigen Basis.
- Über einen Wert des Typs `Long` bzw. *long*, was zwar keine beliebige Zahlen erlaubt, aber die Übersetzung von `TTCN-3` Quelltexte nach Java vereinfacht.

KONSTRUKTOREN

`IntegerType()`

Legt eine neue leere Variable an.

```
IntegerType(IntegerType ivalue), \\
IntegerType(BigInteger value), \\
IntegerType(String value), \\
IntegerType(String value, int radix), \\
IntegerType(Long value), \\
IntegerType(long value)
```

Legt eine neue Wertevariable an und belegt diese mit dem angegebenen Wert.

```
setValue(BigInteger value) throws ValueOutOfRangeException, \\
setValue(String value) throws ValueOutOfRangeException, \\
setValue(String value, int radix) throws ValueOutOfRangeException,
\\
setValue(Long value) throws ValueOutOfRangeException setValue(long
value) throws ValueOutOfRangeException
```

Setzt einen neuen Variablenwert und setzt die Variable als Wertevariable, sofern dieser Wert im definierten Wertebereich liegt. Andernfalls wird eine Ausnahmemeldung aufgerufen.

```
defValues(BigInteger[] values), \\
defValues(IntegerType[] values), \\
defValues(String[] values), \\
defValues(String[] values, int radix), \\
defValues(Long[] values) defValues(long[] values)
```

Die als `protected` deklarierte Funktion definiert den Wertebereich über eine Wertebereichsliste. Ist die Werteliste als Array von Strings gegeben, gilt dieselbe Basis (10 oder mit *radix* angegebene) für jeden Stringwert.

`defValues(IntegerRange[] ranges)`

```
defValues(BigInteger[] values, IntegerRange[] ranges), \\
defValues(IntegerType[] values, IntegerRange[] ranges), \\
defValues(String[] values, IntegerRange[] ranges), \\
defValues(String[] values, int radix, IntegerRange[] ranges), \\
defValues(Long[] values, IntegerRange[] ranges) defValues(long[]
values, IntegerRange[] ranges)
```

Die als `protected` deklarierte Funktion definiert den Wertebereich über eine Kombination der Werte- und Wertebereichsliste.

Alle weitere Funktionen entsprechen denen des `IntType`.

CODEBEISPIEL

```
// Stringdefinition zu Basis 10
IntegerType A = new IntegerType("1986");     // A = 1986

// Stringdefinition zu Basis 16
```

```
IntegerType  = new IntegerType("1986", 16); // A = 6534

// Definiton "uber eine belegte Variable
IntegerType C = new IntegerType();          // A = 6534

// Definition und Belegung "uber die Typen Long und long
IntegerType D = new IntegerType(new Long(1986));
IntegerType E = new IntegerType(1986);

D.setValue(new Long(1986));
E.setBalue(1986);
```

Alle vier Anweisungen belegen die Variablen mit gleichem Wert. Bei Verwendung der Typen `Long` und *long*, ist es (bedingt durch den Java Variablentypen) nicht möglich, einen unbeschränkten Wert zuzuweisen.

5.3.1.4.14 Klasse IntegerRange

Die Klasse IntegerRange wird für Wertebereichsdefinitionen des Typs `IntegerType` und für die Längendefinitionen der Erbklassen des `StringTypes` verwendet.

Es gelten gleiche Initialisierungsmöglichkeiten, wie beim `Integertype`:

KONSTRUKTOREN

```
IntegerRange(), \\
IntegerRange(BigInteger lBound, BigInteger uBound), \\
IntegerRange(IntegerType lBound, IntegerType uBound), \\
IntegerRange(long lBound, long uBound) IntegerRange(String lBound,
String uBound) IntegerRange(String lBound, String uBound, int
radix), \\

IntegerRange(BigInteger aBound, boolean upperBound), \\
IntegerRange(long aBound, boolean upperBound) IntegerRange(String
aBound, boolean upperBound) IntegerRange(String aBound, int radix,
boolean upperBound), \\

IntegerRange(long lBound, BigInteger uBound), \\
IntegerRange(BigInteger lBound, long uBound)
```

Eine Ausnahme besteht jedoch. Während es für `IntRange` möglich war, beide Schranken mit *null* zu initialisieren. Mit (`new IntRange(null, null)`) ist das für IntegerRange nicht möglich, da der Java-Compiler solche Initialisierungen nicht eindeutig einer Funktion zuordnen kann. In diesem Fall sollte ein parameterloser Konstruktor `new IntegerRange()` anstatt `new IntegerRange(null, null)` verwendet werden.

5.3.1.4.15 Klasse FloatType

Die Klasse `FloatType` speichert Werte im Bereich `Double` und erwartet für Wertebelegungen und Bereichsdefinitionen Zahlen im Fließkomma-Format. Die restlichen Funktionen sind identisch mit denen des `IntTypes`.

KONSTRUKTOREN

```
FloatType(), \\
FloatType(FloatType fvalue), \\
FloatType(double value), \\
FloatType(Double value)
```

Erstellt eine leere Variable bzw. eine Wertevariable mit der angegebenen Belegung des Javatyps `Double`.

`Funktionen`

```
setValue(double value) throws ValueOutOfRangeException, \\
setValue(Double value) throws ValueOutOfRangeException
```

Übernimmt einen neuen Wert und macht die Variable zur Wertevariable, falls der angegebene Wert im Wertebereich der Variable liegt.

```
defValues(Double[] values), \\
defValues(double[] values)
```

Die als `protected` deklarierte Funktion, definiert einen Wertebereich durch eine Werteliste.

`defValues(FloatRange[] ranges)`

Als `protected` deklarierte Funktion, definiert einen Wertebereich durch Wertebereichsliste.

```
defValues(Double[] values, FloatRange[] ranges), \\
defValues(double[] values, FloatRange[] ranges)
```

Als `protected` deklarierte Funktion, definiert einen Wertebereich durch Werte- und Wertebereichslisten.

Alle weiteren Funktionen entsprechen denen des IntType, mit Ausnahme der Funktionen *mod* und *rem*. Diese Operationen sind für den TTCN-3 Typ `float` nicht definiert.

5.3.1.4.16 Klasse FloatRange

Wird für die Bereichsdefinition der Klasse `FloatType` verwendet und verwendet Werte des Javatyps `Double` als Grenzwertangaben.

KONSTRUKTOR

Entsprechen denen des `IntType`:

`FloatRange()`

Unbeschränkter Wertebereich.

```
FloatRange(double lBound, boolean upperBound), \\
FloatRange(Double aBound, boolean upperBound)
```

Einseitig beschränkter Bereich.

```
FloatRange(FloatType lBound, FloatType uBound), \\
FloatRange(Double lBound, Double uBound), \\
FloatRange(double lBound, double uBound)
```

TTCN-3 Code	Java-Code	IntType	IntegerType	FloatType
C := -A	C = A.neg();	IntType	IntegerType	FloatType
C := A +	C = A.add();	IntType	IntegerType	FloatType
C := A -	C = A.sub();	IntType	IntegerType	FloatType
C := A *	C = A.mul();	IntType	IntegerType	FloatType
C := A /	C = A.div();	IntType	IntegerType	FloatType
C := A mod	C = A.mod();	IntType	IntegerType	-
C := A rem	C = A.rem();	IntType	IntegerType	-
C := A <	C = A.lt();	BooleanType	BooleanType	BooleanType
C := A <=	C = A.le();	BooleanType	BooleanType	BooleanType
C := A ==	C = A.eq();	BooleanType	BooleanType	BooleanType
C := A !=	C = A.ne();	BooleanType	BooleanType	BooleanType
C := A >=	C = A.ge();	BooleanType	BooleanType	BooleanType
C := A >	C = A.gt();	BooleanType	BooleanType	BooleanType

Tabelle 5.1.: `Zahlentypen`: Implementierung und Ergebnistyp

Unbeschränkter, einseitig oder beidseitig beschränkter Bereich. Die unbeschränkte Seite gibt der Wert *null* an. Auch hier gilt, die Initialisierung mit `new FloatRange(null, null)` wird von Java nicht akzeptiert.

BEISPIELCODE

Der TTCN-3 Definition

```
type float FloatSubType (1, 1.4, 2.5 .. 5, 3, 12.5 .. 15.3);
```

entspricht folgender Code:

```
class FloatSubType extends FloatType {
    public FloatSubType() {
        defValues(new double[]{1.0, 1.4, 3.0},
                  new FloatRange[] {new FloatRange(2.5, 5.0),
                                    new FloatRange(12.5, 15.3)});
        }
    }
```

5.3.1.4.17 Operationen mit TTCN-3 Zahlenvariablen

Die Zahlentypen `IntType` und `IntegerType` erlauben arithmetische und vergleichende Operationen, `FloatType` unterstützt dabei die Operationen *mod* und *rem* nicht. Alle Operationen liefern eine neue Variable mit dem berechneten Wert zurück, wobei arithmetische Operatoren den Typ beibehalten, Vergleichsoperatoren liefern Wert von `BooleanType` zurück. Die Tabelle 5.1 listet definierte TTCN-3 Operationen auf, deren Übersetzung in Java und den Ergebnistyp für Operanden des Typs `IntType`, `IntegerType` und `FloatType` darstellen.

5.3.1.4.18 Stringklassen

Stringklassen lassen sich in Kernklassen `StringRange`, `StringType` und in TTCN-3 Typklassen `CharstringType`, `BitstringType`, `OctstringType` unterteilen.

5.3.1.4.19 Klasse StringRange

Sie wird von der Klasse `StringType` und somit von allen Stringklassen verwendet, um Zeichenbereiche zu definieren.

KONSTRUKTOREN

`StringRange()`

Unbeschränkter Zeichenbereich. Die Funktion *getKind()* liefert jetzt `UNBOUND`.

```
StringRange(String lBound, String uBound); \\
StringRange(StringType lBound, StringType uBound)
```

Einseitig oder beiderseitig beschränkter Zeichenbereich. Die unbeschränkte Seite muss mit *null* ausgezeichnet werden. Da der Compiler eine Wahl hätte, ist es nicht möglich, beide Schranken mit *null* auszuzeichnen und somit mit (`new StringRange(null, null)`) einen unbeschränkten Zeichenbereich zu erstellen.

- Ist keine Seite *null*, liefert *getKind()* den Wert `BOUND`, `getLBound()` den kleinsten und *getUBound()* den größten der beiden angegebenen Schrankenwerte.
- Ist *lBound* = *null*, liefert *getKind()* den Wert `UPPER_BOUND` und beide Eigenschaftsfunktionen `getLBound()` und *getUBound()* den Wert von *uBound*.
- Ist *uBound* = *null*, liefert *getKind()* den Wert `LOWER_BOUND` und beide Eigenschaftsfunktionen `getLBound()`, *getUBound()* den Wert von *lBound*.

`StringRange(String aBound, boolean upperBound)`

Einseitig beschränkter Zeichenbereich:

- Ist *upperBound* = *true*, liefert *getKind()* den Wert `UPPER_BOUND` und beide Eigenschaftsfunktionen `getLBound()`, *getUBound()* den Wert von *uBound*.
- Ist *upperBound* = *false*, liefert *getKind()* den Wert `LOWER_BOUND` und beide Eigenschaftsfunktionen `getLBound()`, *getUBound()* den Wert von *lBound*.

FUNKTIONEN

`clone()`

Erstellt eine vollständige Kopie des Zeichenbereiches.

5.3.1.4.20 Klasse StringType

Die Klasse `StringType` ist eine Superklasse für die Stringtypklasse `CharstringType` und Bitmusterklassen `BitstringType`, `HexstringType` und `OctstringType` und implementiert deren gemeinsame Funktionen.

KONSTRUKTOREN

`StringType(int digitlength)`

Erstellt eine leere Variable. Ohne Wert- oder Regelbelegung.

`StringType(String value, int digitlength)`

Erstellt eine Wertevariable und initialisiert sie mit dem angegebenen Stringwert.

`StringType(int digitlength, TemplateValue rule)`

Erstellt eine Templatevariable und belegt sie mit der angegebenen Templateregel.

Alle Konstruktoren benötigen den Parameter *digitlength*. Dieser gibt die Ziffernlänge in Zeichen, sofern die Variable die Superklasse von Bitmusterklassen darstellt: Die Klassen `CharstringType`, `BitstringType` und `HexstringType` belegen diesen Parameter mit 1 und `OctstringType` mit 2 (Ziffernlänge für `octetstring` ist 2 Zeichen lang.).

Der Parameter wird für Verschiebe- und Rotationsoperatoren verwendet (alle Bitmustervariablen verschieben bzw. rotieren um eine Ziffer) und für die Wertebereichsprüfung (Ziffern für bsp. `hexstring` liegen sie zwischen '0' und 'F', für `octetstring` dagegen zwischen '00' und 'FF').

FUNKTIONEN

`setValue(String value) throws ValueOutOfRangeException`

Stellt sicher, dass der angegebene Wert die ganze Zahl von Zeichen (*digitlength*) enthält und die Wertebereichsbeschränkung nicht verletzt. Im negativen Fall wird `ValueOutOfRangeException` geworfen, im Positiven der Wert übernommen und die Variable zur Wertevariable deklariert.

`getValue ()`

Ermittelt den gesetzten Wert.

`defValues(String[] values)`

Als *protected* deklarierte Funktion *defValues()* übernimmt eine Liste von Werten und schränkt den Wertebereich auf die Elemente der Liste.

`defValues(StringRange[] ranges)`

Als *protected* deklarierte Funktion *defValues()* übernimmt eine Liste von Zeichenbereichen und schränkt die Liste erlaubter Zeichenbereiche auf den Durchschnitt mit den bereits definierten Zeichenbereichen ein, so wie den Wertebereich auf solche Werte, deren Zeichen in so definierten Zeichenbereich liegen.

```
defLength(IntegerRange length), \\
defLength(String lBound, String uBound), \\
defLength(String length) defLength(long lBound, long uBound)
defLength(long length)
```

Setzt eine Längenbeschränkung. Ist bereits eine Stringlänge definiert, wird der Durchschnitt mit der gesetzten Stringlänge übernommen. Die Menge der erlaubter Werte wird auf solche eingeschränkt, deren Stringlänge im so definierten Längenbereich liegen. Als Längenbereich dient ein Objekt der Klasse `IntegerRange`, 5.3.1.4.14, das auf unterschiedliche Art initialisiert werden kann. Die Funktion *defLength()* übersetzt deren Parameter in die Initialisierungsparameter des internen Stringlängenobjektes.

`checkValue (Comparable value)`

Prüft den angegebenen Wert auf die Einhaltung des gesetzten Wertebereichs: Er muss in der Werteliste liegen, sofern definiert und nur aus erlaubten Zeichen bestehend (sofern Zeichenbereich definiert ist).

`length()`

Ermittelt die Länge des gesetzten Strings.

`substring(IntegerType a, IntegerType )`

Berechnet den Stringausschnitt aus dem gesetzten Wert mit angegebenen Grenzen.

`valuesToString()`

Berechnet eine typgerechte Stringdarstellung der gesetzten Werteliste. Typgerecht bedeutet: Eine Variable des `BitstringType` ermittelt den String wie „`0110`,'1101'“. Eine Variable des `HexstringType` ermittelt den String „`'A12', '77ABA'`“ und `OctstringType` liefert „`'AB12'O, '777ABA'O`“. Um diese Darstellung zu berechnen, ruft diese Funktion für jeden

Wert die Funktion *toString()* von `TTCN3Type` auf, die wiederum von jeder Stringklasse überschrieben wird und zur Klasse passende Darstellung berechnet.

`lengthToString()`

Stellt die definierte Stringlänge als String dar.

`rangesToString()`

Stellt den definierten Zeichenbereich als String dar.

`con(StringType op)`

TTCN-3 Operator `&`. Konkateniert den Wert dieser Variable mit dem der angegebenen und liefert das Ergebnis in einer neuen Variable zurück.

`lsh(int op)`

Operator <<. Linksverschiebung: Verschiebt den Wert dieser Variable um *op Zeichen* nach links (fehlende Plätze werden mit `0`-en aufgefüllt) und liefert das Ergebnis in einer neuen Variable zurück.

`rsh(int op)`

Operator >>. Rechtsverschiebung: Verschiebt den Wert dieser Variable um *op Zeichen* nach rechts (freie Plätze werden mit `0`-en aufgefüllt) und liefert das Ergebnis in einer neuen Variable zurück.

`lrt(int op)`

Operator `<@`. Linksrotation: Rotiert den Wert dieser Variable um *op Zeichen* nach links und liefert das Ergebnis in einer neuen Variable zurück.

`rrt(int op)`

Operator `@>`. Rechtsrotation: Rotiert den Wert dieser Variable um *op Zeichen* nach rechts und liefert das Ergebnis in einer neuen Variable zurück.

5.3.1.4.21 Klasse CharstringType

Implementiert den TTCN-3 Typ `charstring`.

5.3.1.4.22 Klasse BitstringType

Implementiert den TTCN-3 Typ `bitstring` und ist stellvertretend für Bitmustertypen `BitstringType`, `HexstringType` und `OctstringType`, weshalb die beiden anderen nur kurz behandelt werden.

KONSTRUKTOREN

`BitstringType()`

Leere Variable. Die Ziffernlänge wird mit 1 initialisiert und der Zeichenbereich auf Werte zwischen „0“ und „1“, wodurch nur Stringwerte zugewiesen werden können, die eine binäre Zahlendarstellung speichern.

`BitstringType(String value) throws ValueOutOfRangeException`

Wertevariable. Die Ziffernlänge wird mit 1 initialisiert und der Zeichenbereich auf Werte zwischen „0“ und „1“. Der Wert wird übernommen, sofern er im definierten Wertebereich liegt. Wenn nicht wird `ValueOutOfRangeException` geworfen.

BitstringType(TemplateValue rule)

Templatevariable. Die Ziffernlänge wird mit 1 initialisiert und der Zeichenbereich auf Werte zwischen „0“ und „1“ gesetzt. Die Templateregel wird übernommen.

FUNKTIONEN

`not4b()`

Der Operator `not4B` (Bitweises *not*) liefert eine neue Variable, die einen Wert speichert, der wiederum durch das Negieren jedes Bits aus dem Wert dieser Variable hervorgeht.

```
and4b(BitstringType op),
and4b(BitstringType... )
```

Der Operator `and4B` (Bitweises *and*) führt eine bitweise *and*-Verknüpfung des Wertes dieser Variable und der angegebenen und liefert das Ergebnis in einer neuen Variable zurück.

```
or4b(BitstringType op),
or4b(BitstringType... )
```

Der Operator `or4b` (Bitweises *or*) führt eine bitweise *or*-Verknüpfung des Wertes dieser Variable und der angegebenen und liefert das Ergebnis in einer neuen Variable zurück.

```
xor4b(BitstringType op),
xor4b(BitstringType... )
```

Der Operator `xor4b` (Bitweises *xor*) führt eine bitweise *xor*-Verknüpfung des Wertes dieser Variable mit denen der angegebenen und liefert das Ergebnis in einer neuen Variable zurück.

5.3.1.4.23 Klasse HexstringType

Implementiert den TTCN-3 Typ `hexstring`. Die Ziffernlänge definiert, wie `BitstringType`, 5.3.1.4.22 als 1 Zeichen, der Zeichenbereich dagegen als `'0'` bis `'F'`. Aus Performancegründen werden gesetzte Werte stets in Großbuchstaben konvertiert, weil es keinen unterschied in der Wertebelegung zwischen `'12abcd'`, `'12aBCd'` und `'12ABCD'` gibt.

5.3.1.4.24 Klasse OctstringType

Implementiert den TTCN-3 Typ `octetstring`. Die Ziffernlänge definiert als 2 Zeichen, der Zeichenbereich als `'00'` bis `'FF'`. Aus Performancegründen werden gesetzte Werte, wie bei `HexstringType`, 5.3.1.4.23 stets in Großbuchstaben konvertiert.

5.3.1.4.25 Operationen mit Stringtypen

Alle Stringtypen liefern eine Variable des selben Typs, wie die aktuelle und evtl. der Operand. Die Tabellen 5.2, 5.3 zeigt Operationen mit Stringvariablen auf und entsprechende Stringdarstellung der Ergebnisvariablen. Die „`X`“ in den beiden ersten Zuweisungen steht für den entsprechende Auszeichnungsbuchstaben, so wie die Variable `A` vom Typ `HexstringType` beispielsweise, mit `A := '100000'` belegt werden. Außerdem wird vorausgesetzt, dass die beide ersten Zuweisungen fehlerfrei waren (Werte für `A` und liegen in deren Wertebereichen).

5.3.1.4.26 Klasse BooleanType

Implementiert den booleschen TTCN-3 Typ `boolean`.
KONSTRUKTOREN

TTCN-3 Code	Java-Code	CharstringType
`A := "100000"` `  := "10"`	`A = "100000";` `  = "10";`	`"100000"` `"10"`

`C := A & ;`
`C := A <@ 2;`
`C := A @> 2;`
	`C = A.con();`
	`C = A.lrt(2);`
	`C = A.rrt(2);`
		`"10000010"`
		`"000010"`
		`"001000"`

Tabelle 5.2.: TTCN-3 `CharstringType`: Operationen, Implementierung und Darstellung

TTCN-3 Code	Java-Code	BitstringType	HexstringType	OctstringType
`A := '100000'X` `  := '10'X`	`A = "100000";` `  = "10";`	`'100000'` `'10'`	`'100000'` `'10'`	`'100000'` `'10'`

`C := A & ;`
`C := A << 2;`
`C := A >> 2;`
`C := A <@ 2;`
`C := A @> 2;`
	`C = A.con();`
	`C = A.lsh(2);`
	`C = A.rsh(2);`
	`C = A.lrt(2);`
	`C = A.rrt(2);`
		`'10000010'`
		`'000000'`
		`'001000'`
		`'000010'`
		`'001000'`
			`'10000010'`
			`'000000'`
			`'001000'`
			`'000010'`
			`'001000'`
				`'10000010'O`
				`'000000'`
				`'000010'`
				`'001000'`
				`'000010'`

Tabelle 5.3.: TTCN-3 `Bitmustertypen` : Operationen, Implementierung und Darstellung

```
BooleanType(),
BooleanType(BooleanType value),
BooleanType(booleanvalue),
BooleanType(Boolean value)
```

Wertevariable. Entweder uninitialisiert oder initialisiert mit dem Wert der angegebenen booleschen Variable.

`BooleanType(TemplateValue rule)`

Templatevariable mit der angegebenen Regelinstanz und mit einem undefinierten Wert.

FUNKTIONEN

```
void setValue(Boolean value) throws ValueOutOfRangeException, void
setValue(boolean value) throws ValueOutOfRangeException
```

Übernimmt den angegebenen Wert. Wirft die `ValueOutOfRangeException` aus, wenn dieses nicht im Wertebereich des konkreten Typs liegt.

```
Boolean getValue(),
Boolean getBooleanValue()
```

Ermitteln den gesetzten booleschen Wert.

```
protected void defValues(Boolean[] values),
protected void
defValues(boolean[] values)
```

Definiert die Wertliste des booleschen Subtyps.

`boolean checkValue (Comparable value)`

Prüft den angegebenen Wert darauf, ob es ein boolescher Wert ist und im Wertebereich des aktuellen Typs liegt.

`String valuesToString()`

Liefert die Stringrepräsentation der gespeicherten Werteliste.

```
BooleanType not(),
BooleanType and(BooleanType op),
BooleanType
and(BooleanType... op),
BooleanType or(BooleanType op),
BooleanType
or(BooleanType... op),
BooleanType xor(BooleanType op)
```

Liefern das Ergebnis der logischen Operationen **not**, **and**, **or**, **xor**. Das Ergebnis ist eine neue Instanz von `BooleanType`, die den Wert der entsprechenden Operation enthält,

5.3.1.4.27 Klasse Enum

Dient der Definition von Aufzählungstypen (Abschnitt 5.3.1.4.28), und zwar zur Definition von Aufzählungen, insbesondere vornummerierter Aufzählungen.

KONSTRUKTOREN

`Enum(String value)`

Definiert eine Aufzählung. Das Parameter `value` legt den Namen der Aufzählung fest.

```
Enum(String value, Integer key),
Enum(String value, int key)
```

Definiert eine vornummerierte Aufzählung. Der Parameter *value* definiert den Namen und *key* die Nummer der Aufzählung.

FUNKTIONEN

`String value()`

Liefert den Namen der Aufzählung zurück.

`Integer key()`

Liefert die Nummer der Aufzählung zurück.

`int compareTo(Object obj)`

Vergleicht die Nummer der aktuellen Aufzählung mit der Nummer der Angegebenen. Das Ergebnis entspricht dem Vergleich zweier vergleichbarer Daten: Ein negativer Wert wird ermittelt, wenn die aktuelle Aufzählung eine kleinere Nummer besitzt. Ein positiver Wert, wenn die aktuelle Aufzählung eine größere Nummer besitzt, als die angegebene Aufzählung und 0 wenn beide Aufzählungen die gleiche Nummer besitzen. Namen werden nicht verglichen. Die Operation wirft die `NullPointerException`, wenn eine der Aufzählung `null` als Nummer angibt und `ClassCastException`, wenn das übergebene Objekt nicht eine Instanz von `Enum` ist.

```
type enumerated myEnumType {
  Monday, Tuesday, Wednesday, Thursday, Friday
};
type enumerated myMixedEnumType {
  Tuesday, Wednesday, Friday, Monday (0) , Thursday (3)
};
```

Beispiel 5.9: Definition der Aufzählungstypen

5.3.1.4.28 Klasse EnumeratedType

Implementiert den Aufzählungstyp `enumerated` von TTCN-3. Ein Aufzählungstyp wird in TTCN-3 definiert wie im Beispiel 5.9 angegeben [14, s. 37].

Die Vornummerierung ist nur erforderlich, wenn einer Aufzählung eine andere Nummer zugeordnet werden soll, als solche, die automatisch aus der Angabereihenfolge berechnet werden. Beide Typdefinitionen sind daher gleichwertig.

Danach wird es möglich, Aufzählungsvariablen zu definieren und zu vergleichen, Beispiel 5.10

```
var myEnumType myMonday := Monday,
               myFriday := Friday;

if (myMonday < myFriday) {
  // Bedingung ist wahr - Block wird ausgef"uhrt.
}
```

Beispiel 5.10: Anwendung der Aufzählungstypen

Der Vergleich findet nach der Aufzählungsnummer statt. Die Nummer ist immer gegeben, ob sie bei der Definition angegeben ist oder automatisch berechnet wurde. In Java wird die Klassendefinition ohne die Nummernangabe wie im Beispiel 5.11 aussehen.

Die vornummerierte Version instanziiert Aufzählungen mit Zahlenparameter, Beispiel 5.12

Die so definierte Java-Klassen können nun TTCN-3 konform genutzt werden, Beispiel 5.13.

KONSTRUKTOREN

`EnumeratedType()`

instanziiert die Klasse und erstellt eine leere Variable.

`EnumeratedType(String value) throws ValueOutOfRangeException`

instanziiert die Klasse und belegt sie mit dem angegebenen Wert der Aufzählung. Erstellt somit eine neue Wertevariable.

`EnumeratedType(TemplateValue rule)`

Erstellt eine neue Templatevariable, indem die neue Instanz mit dem Regelobjekt belegt ist.

FUNKTIONEN

```
protected void defValues(Enum[] values),
protected void
defValues(String[] values)
```

Wird im Konstruktor einer neuen Aufzählungsklasse aufgerufen. Definiert Aufzählungen des neuen Typs, wie im Abschnitt 5.3.1.4.28 gezeigt wurde.

```
class myEnumType extends EnumeratedType {
  public myEnumType() {
    defValues(
      new Enum[]{
        new Enum("Monday"),
        new Enum("Tuesday"),
        new Enum("Wednesday"),
        new Enum("Thursday"),
        new Enum("Friday")
      }
    );
  }
  public myEnumType(String value) throws ValueOutOfRangeException {
    this();
    setValue(value);
  }
}
```

Beispiel 5.11: Javaklasse `myEnumType`

```
class myMixedEnumType extends EnumeratedType {
  public myMixedEnumType() {
    defValues(
      new Enum[]{
        new Enum("Tuesday"),
        new Enum("Wednesday"),
        new Enum("Friday"),
        new Enum("Monday", 0),
        new Enum("Thursday", 3)
      }
    );
  }
  public myMixedEnumType(String value) throws ValueOutOfRangeException {
    this();
    setValue(value);
  }
}
```

Beispiel 5.12: Javaklasse `myMixedEnumType`

```
myEnumType myMonday = new myEnumType("Monday"),
           myFriday = new myEnumType("Friday");

if (myMonday.lt(myFriday)) {
  // Bedingung ist wahr - Block wird ausgef"uhrt.
}
```

Beispiel 5.13: Anwendung der Javaklasse `myEnumType`

```
void setValue(Integer key) throws ValueOutOfRangeException
void setValue(int key) throws ValueOutOfRangeException
```

Setzt den Wert der Variable auf eine Aufzählung, die mit der angegebenen Nummer definiert wurde. Ausnahme wird geworfen, wenn die Nummer bei der Typdefinition nicht (manuell oder automatisch) vergeben wurde.

`void setValue(String value) throws ValueOutOfRangeException`

Setzt den Wert der Variable auf eine Aufzählung, gegeben durch den Namen. Die Ausnahme wird geworfen, wenn keine Aufzählung mit diesem Namen definiert wurde.

`boolean checkValue(Comparable value)`

Prüft, ob der angegebene Name in der Liste definierter Aufzählungen vorhanden ist.

```
String keyToValue(Integer key),
Integer valueToKey (String value)
```

Für eine gegebene Nummer gibt den entsprechenden Namen der Aufzählung zurück und umgekehrt.

```
String getValue(),
Integer getKey()
```

Liefern den Namen bzw. die Nummer der aktuell gesetzten Aufzählung zurück.

`String valuesToString()`

Listet die Namen der definierten Aufzählungen auf.

`int compareTo(EnumeratedType v)`

Vergleicht die aktuelle und die angegebene Aufzählungsvariable bezüglich der Aufzählungsnummer der gesetzten Werte.

```
BooleanType eq(EnumeratedType op)
BooleanType ne(EnumeratedType op)
BooleanType le(EnumeratedType op)
BooleanType ge(EnumeratedType op)
BooleanType lt(EnumeratedType op)
BooleanType gt(EnumeratedType op)
```

Die Vergleichsoperationen berechnen die TTCN-3 Operationen '==', '!=', '<=', '>=', '<', '>' (entsprechend der Reihenfolge), indem zum Vergleich der aktuellen und der angegebenen Variable die Funktion *compareTo* aufgerufen wird.

5.3.1.4.29 Klasse VerdictType

Der TTCN-3 Typ `verdicttype` wird als Ergebnistyp für Testurteile verwendet. Kennt folgende Urteilswerte: *none*, *pass*, *inconc*, *fail*, *error*. [14, s. 143]

Die Ersetzungsprioritä definiert auch die Ordnung der Urteilswerte: Das Testurteil *pass* kann durch *fail* ersetzt werden, umgekehrt aber nicht. [14, s. 143]

Diese Definition erlaubt es, den TTCN-3 Typ `verdichttype` als eine Art des Aufzählungstyps zu sehen. Dementsprechend ist die Klasse `VerdictType` als Erbe der `EnumeratedType` mit fürnf vorbelegten Aufzählungen `"none"`, `"pass"`, `"inconc"`, `"fail"`, `"error"` definiert, die in dieser Reihenfolge auch der Ersetzungsprioritä genügen müssen.

KONSTRUKTOREN

`VerdictType()`
Erstellt eine neue Wertevariable mit einer vollständigen Werteliste ohne einen Urteilswert zu setzen.

```
VerdictType(String verdict) throws ValueOutOfRangeException
VerdictType(int verdict) throws ValueOutOfRangeException
```

Erstellt eine neue Wertevariable mit einer vollständigen Werteliste und belegt diese mit einem Urteilswert. Wirft die Ausnahme aus, wenn der angegebene Wert nicht in der Liste der aktuellen Urteilswerte liegt.
FUNKTIONEN

```
void setVerdict(int verdict),
void setVerdict(VerdictType v)
```

Setzt einen neuen Urteilswert. Wirft die Ausnahme aus, wenn der angegebene Wert nicht in der aktuellen Liste der Urteilswerte liegt. Der Urteilswert kann sowohl per Name (`"none"`, `"pass"`, `"inconc"`, `"fail"`, `"error"`), als auch per Nummer (0 bis 4 entsprechend) gesetzt werden.
`int getVerdict()`
Liefert die Nummer des gesetzten Urteilswertes zurück.
`void max(VerdictType v)`
Setzt TTCN-3 konform einen neuen Urteilswert. In der festgelegten Wertordnung darf sich der Urteilswert erhöhen, was durch die Funktion `max()` sichergestellt wird.
`String toString()`
Gibt die Stringrepräsentation des gesetzten Urteilswertes zurück, was auch eine Entsprechung zu `getVerdict()` bezüglich des Namens ist.

5.3.1.4.30 Strukturtypen

Zwar ordnet ETSI auch `verdicttype` und `enumeratedtype` unter Strukturtypen ([14, S. 26]), in vorliegenden Dokument werden als Strukturtypen solche bezeichnet, die durch eine Datenstruktur definiert werden, die aus Feldern besteht, die wiederum eines beliebigen TTCN-3 Typs, inklusive eines Strukturtyps, definiert werden. Folgende TTCN-3 Typen sind derzeit implementiert: `record`, `set`, `union`. Das dafür benötigte Klassengerüst ist in der Abbildung 5.31 dargestellt.
Der Zusammenhang des TTCN-3 Code und der zugehörigen Java Klassen soll anhand des Typs `record` bzw. der Klasse `RecordType`, so wie den weiteren Involvierten erläutert werden.
DEKLARATION
Ein Record wird wie im Beispiel 5.14 definiert.

```
type record myRecordType {
    integer      field1,
    myUnionType  field2 optional,
    charstring   field3
}
```

Beispiel 5.14: TTCN-3 Typ `myRecordType`

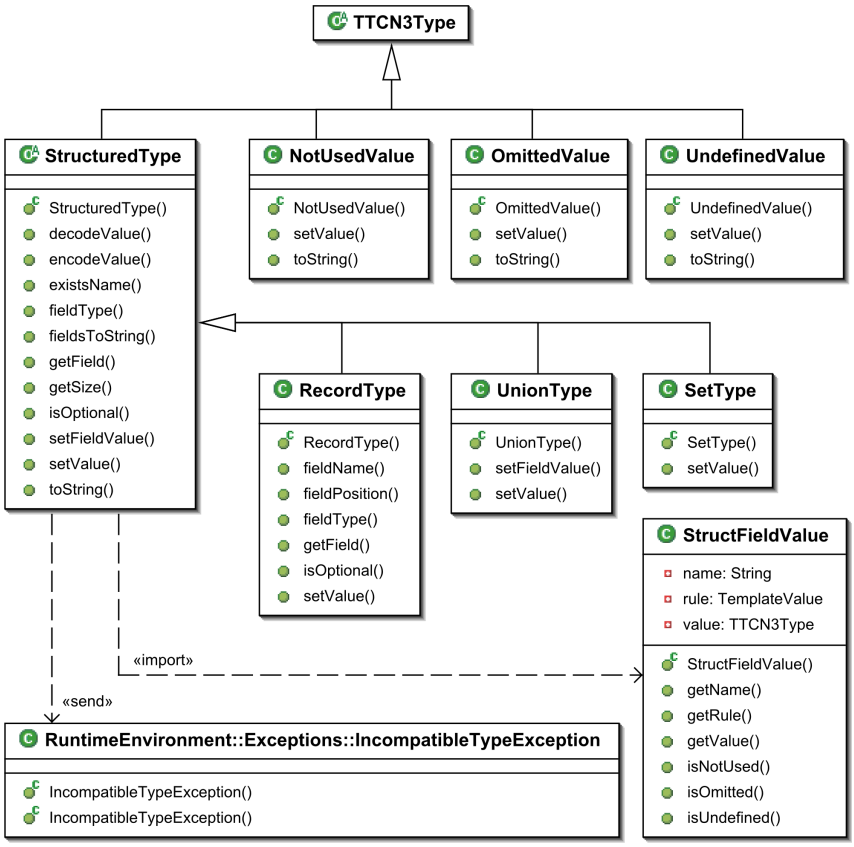

Abbildung 5.31.: Strukturtypen, Klassendiagramm

Dabei definiert der Typ `myRecordType` drei Felder. Das Feld *field2* ist optional und ist vom Typ `myUnionType`, der wiederum ein Strukturtyp mit weiteren Feldern ist. Für diese Ausführung wird dies nicht weiter verfolgt und dieser Typ als gegeben betrachtet.

ZUWEISUNGEN

Eine Strukturvariable kann in vier Arten zugewiesen werden. Drei davon werden von den implementierten Klassen unterstützt: *Wertelisten-*, *Zuweisung-* und *Feldnotation*. Beide erste Notationen können auch in der Variableninitialisierung auftreten.

In der *Wertelistennotation* („Value list notation“) wird einer Strukturvariable die Liste von Werte zugewiesen, ohne Angabe der Felder. Die Strukturvariable muss dann entscheiden, welches Feld welchen Wert zugewiesen bekommt, Beispiel 5.15.

```
// varU ist eine Variable vom Typ myUnionType.

var myRecordType A;           // A = {{undef}, {undef}, {undef}}
A := {111, varU, "A string"}; // A = {111, varU, "A string"}

var myRecordType  := {111, varU, "A string"}; //  = A

 := {-, omit, "A new string"}; //  = {111, {undef}, "A new string"}

var myRecordType C := {222, -, -}; // C = {222, {undef}, {undef}}
```

Beispiel 5.15: Strukturtyp. Zuweisungen in der Wertlistennotation

Die angegebene Werteliste muss Werte für alle Felder angeben, wobei die Werte in der Reihenfolge der Felder in der Typdefinition angegeben werden müssen. Der erste Wert wird dem ersten Feld der Strukturvariable zugewiesen, der zweite Wert dem zweiten Feld usw.

In der ersten Anweisung wurde die Variable `A` erstellt und in der Zweiten mit Werten belegt. Dritte Anweisung initialisiert die Variable , so dass beide identisch belegt sind.

Die vierte Anweisung verändert die Varible und die letzte initialisiert die Variable C.

Dabei wurden Symbole '`-`' und '`omit`' verwendet. Das Symbol '`-`' lässt das entsprechende Feld unverändert. So behält das Feld `.field1` den Wert '111' in der vierten Anweisung bei und in der Letzten bleiben beide letzten Feldern der Variable `C` undefiniert.

Das Symbol '`omit`' setzt das entsprechende Feld zurück. Das Feld `.field2` wurde so auf '*undef*iniert' zurückgesetzt. Das Symbol darf nur für optionale Felder verwendet werden, andernfalls wird die Ausnahmemeldung `IncompatibleTypeException` ausgeworfen.

In der *Zuweisungsnotation* („Assignment notation“) wird eine Wertliste mit Angabe der Feldnamen zugewiesen, Beispiel 5.16.

```
// varU ist eine Variable vom Typ myUnionType.

var myRecordType A;
A := {field1 := 111, field3 := "A string"};

// A = {111, {undef}, "A string"}

var myRecordType  := {field3 := "A string", field2:= varU};

//  = {{undef}, varU, "A string"}

 := {field1 := 333, field2:= omit};

//  = {333, {undef}, "A string"}
```

Beispiel 5.16: Strukturtyp. Zuweisungen in der Zuweisungsnotation

In der Zuweisungsnotation müssen nicht alle Felder angegeben werden und die Feldreihenfolge kann beliebig sein. Beide Notationen dürfen nicht gemischt werden. Es dürfen keine unbekannte Feldnamen verwendet werden.

In der *Feldnotation* („Dot notation“) wird jedes Feld direkt angesprochen. Es können nicht mehrere Felder gleichzeitig angesprochen werden, Beispiel 5.17. In der selben Notation kann jedes Feld auch ausgelesen werden.

IMPLEMENTIERUNG

Die Klassendefinition soll mindestens Felder definieren können. Der Java-Code für das Beispiel 5.14 ist im Beispiel 5.18 angegeben.

Jedes Feld wird mittels der Funktion *defineField()* definiert, diese erwartet drei Parameter: Eine Instanz der Klasse für die Felder, einen Feldnamen und die boolesche Optionalitätsangabe. Der letzte Parameter darf ausgelassen werden und wird als *false* angenommen, so dass es nur dann (mit *true*) angegeben werden muss, wenn im TTCN-3 Code das Schlüsselwort '*optional*' verwendet wurde.

Die Klasse `StructuredType` erlaubt das Setzen der Feldwerte anhand einer Werteliste und eines anderen Strukturtyps, was durch die Klasse `RecordType` übernommen wird und in der

```
// varU ist eine Variable vom Typ myUnionType.

var myRecordType A;
A.field1 := 111;
A.field3 := "A string";
A.field2 := varU;

// A = {111, varU, "A string"}

A.field2 := omit; // Zur"ucksetzen
A.field3 := -;    // Unver"andert lassen

// A = {111, {undef}, "A string"}

A.field1 := omit; // Dies ist nicht erlaubt -> IncompatibleTypeException

var myRecordType ;
.field1 := A.field1;

//  = {111, {undef}, {undef}}
```

Beispiel 5.17: Strukturtyp. Zuweisungen in der Feldnotation

```
type myRecordType extends RecordType {
    public myRecordType () {
        defineField (new IntegerType(),    "field1");
        defineField (new myUnionType(),    "field2", true);
        defineField (new CharstringType(), "field3");
    }
}
```

Beispiel 5.18: Klasse `myRecordType`, minimale Deklaration

neuen Recordklasse verwendet werden muss. Die vollständige Klassendefinition zeigt das Beispiel 5.19.

```
type myRecordType extends RecordType {
    public myRecordType () {
        defineField (new IntegerType(),    "field1");
        defineField (new myUnionType(),    "field2", true);
        defineField (new CharstringType(), "field3");
    }
    public myRecordType (StructFieldValue... values) {
        this();
        setValue(values);
    }
    public myRecordType (TTCN3Type value) {
        this();
        setValue(value);
    }
}
```

Beispiel 5.19: Klasse `myRecordType`, vollständige Deklaration

Der zweite Konstruktor initialisiert die neue Variable anhand einer Werteliste und der Letzte anhand einer Strukturvariable, die als `TTCN3Type` getarnt ist.

Die Initialisierung und Zuweisung in der *Wertelistennotation* zeigt das Beispiel 5.15. Den zugehörigen Java-Code zeigt das Beispiel 5.20.

Die Werteliste wird mittels eines Instanzenarrays der Klassen `StructFieldValue` angegeben, wobei jede davon mit dem entsprechenden Wert initialisiert wird. die Schlüsselsymbole '`-`' und '`omit`' werden durch Instanzen der Klassen `NotUsedValue` und `OmittedValue` entsprechend angegeben.

Da die *Zuweisungsnotation* sehr der Wertelistennotation ähnelt, übernimmt sie ebenfalls ein Instanzenarray von `StructFieldValue`, wobei diesmal auch die Feldnamen angegeben werden müssen. Dem Beispiel 5.15 entspricht der Code in 5.21

Für die *Feldnotation* bietet die Klasse `StructuredType` zwei Funktionen an, *getField()* zum Auslesen und *setField()* zum Setzen des Feldwertes, Beispiel 5.22 entspricht dem 5.17.

Schließlich erlaubt `StructuredType` das Setzen und Initialisieren durch eine andere Strukturvariable. Dies wurde durch den dritten Konstruktor von `myRecordType` in 5.19 auch übernommen. Dabei kann als Quelle eine beliebige Strukturvariable dienen. Die Zuweisung wird als eine in der Zuweisungsnotation gehandhabt, wobei als Feldnamen die Feldnamen und als Werte die Feldwerte in der Quellvariable verwendet werden. Es darf also keine Variable angegeben werden, die ein Feld definiert, dessen Name nicht im aktuellen Typ definiert wurde, Beispiel 5.23.

5.3.1.4.31 Klasse StructuredType

Die Klasse StructuredType ist die Grundklasse für alle Strukturtypen, und implementiert das für Strukturtypen gemeinsame Verhalten. Sie ist abstrakt und kann nicht direkt instanziiert werden.

KONSTRUKTOREN

```
myRecordType A = new myRecordType();
A.setValue (
    new StructFieldValue[] {
         new StructFieldValue (new IntegerType(111)),
         new StructFieldValue (myUnionType),
         new StructFieldValue (new CharstringType("A string"))
    }
);

myRecordType  = new myRecordType (
    new StructFieldValue[] {
         new StructFieldValue (new IntegerType(111)),
         new StructFieldValue (varU),
         new StructFieldValue (new CharstringType("A string"))
    }
);

.setValue (
    new StructFieldValue[] {
         new StructFieldValue (new NotUsedValue()),                // -
         new StructFieldValue (new OmittedValue()),                // omit
         new StructFieldValue (new CharstringType("A new string"))
    }
);

A.setValue (
    new StructFieldValue[] {
         new StructFieldValue (new IntegerType(222)),
         new StructFieldValue (new NotUsedValue()),
         new StructFieldValue (new NotUsedValue())
    }
);
```

Beispiel 5.20: Strukturtyp. Zuweisungen in der Wertlistennotation, Java

```
var myRecordType A;
A.setValue (
    new StructFieldValue[] {
         new StructFieldValue ("field1", new IntegerType(111)),
         new StructFieldValue ("field3", new CharstringType("A string"))
    }
);

myRecordType  = new myRecordType (
    new StructFieldValue[] {
         new StructFieldValue ("field3", new CharstringType("A string")),
         new StructFieldValue ("field2", varU)
    }
);

A.setValue (
    new StructFieldValue[] {
         new StructFieldValue ("field1", new IntegerType(333)),
         new StructFieldValue ("field2", new OmittedValue())
    }
);
```

Beispiel 5.21: Strukturtyp. Zuweisungen in der Zuweisungsnotation, Java

```
var myRecordType A;
A.setField("field1", new IntegerType(111));
A.setField("field3", CharstringType("A string"));
A.setField("field2", varU);

A.setField("field2", new OmittedValue());
A.setField("field3", new NotUsedValue());

var myRecordType A;
.setField("field1", A.getField("field1"));
```

Beispiel 5.22: Strukturtyp. Zuweisungen in der Feldnotation, Java

```
// Nach der folgenden Initialisierung ...

myRecordType A = new myRecordType (
    new StructFieldValue[] {
         new StructFieldValue (new IntegerType(111)),
         new StructFieldValue (varU),
         new StructFieldValue (new NotUsedValue())
    }
);

// ... entspricht ...

myRecordType  = new myRecordType (A);

// ... dieser Anweisung:

myRecordType  = new myRecordType (
    new StructFieldValue[] {
         new StructFieldValue ("field1", new IntegerType(111)),
         new StructFieldValue ("field2", varU)
    }
);
```

Beispiel 5.23: Strukturtyp. Zuweisungen anhand einer Strukturvariable, Java

```
StructuredType(),
StructuredType(StructFieldValue[] values),
StructuredType(Comparable value)
```

erstellen eine neue Variable ohne Felder und ohne Regelobjekt. Jegliche Parameter, wie z.. Feldbelegungen, werden ignoriert, weil noch keine Felder definiert sind, die die übergebene Feldwerte aufnehmen können und auch kein einfacher Wert, weil dieser nur ein Strukturtyp und keinen einfachen Werte aufnehmen kann. Der Konstruktor in der dritten Form ist da, weil ihn die Superklasse `TTCN3Type` (5.3.1.4.6) definiert.

`StructuredType(TemplateValue rule)`
erstellt eine Templatevariable, indem das Regelobjekt übernommen wird.

FUNKTIONEN

```
protected void defineField(TTCN3Type type, String name, boolean
optional),
protected void defineField(TTCN3Type Type, String name)
```

Definiert ein neues Feld der Datenstruktur. Ist der Parameter `optional` ausgelassen, wird es als **false** angenommen.

`TTCN3Type getField(String name)`
liefert den Wert eines Feldes mit dem angegebenen Namen zurück.

`boolean existsName (String name)`
prüft, ob ein Feld mit dem angegebenen Namen definiert worden ist.

`Class<TTCN3Type> fieldType (String name)`

liefert die Javaklasse zurück, die für die Definition des Feldes mit dem angegebenen Namen bei der Strukturtypendeklaration (mittels der Funktion `defineField()`) verwendet worden ist.

`boolean isOptional(String name)`

ermittelt, ob das Feld mit dem angegebenen Namen als optional oder obligatorisch definiert wurde.

`int getSize()`

liefert die Anzahl von Feldern in der Datenstruktur.

```
void setField(String name, TTCN3Type value)
     throws IncompatibleTypeException
```

setzt den Feldwert in der Feldnotation. Entspricht der Wertetyp nicht der Feldtypdefinition oder soll das obligatorische Feld zurückgesetzt werden, wird die Ausnahme geworfen.

```
void setValue(StructFieldValue... values)
     throws IncompatibleTypeException
```

setzt Feldwerte in den Wertelisten- und in der Zuweisungsnotation. Entspricht der Wertetyp eines Feldes nicht seiner Typdefinition oder soll ein obligatorisches Feld zurückgesetzt werden, wird die Ausnahme geworfen.

`void setValue(Comparable value) throws ValueOutOfRangeException`

Ist ein Artefakt der Superklasse. Es überschreibt die Funktion in der Klasse `TTCN3Type` (5.3.1.4.6) und ignoriert die Anfrage, da Strukturtypen keine einfachen Werte Annehmen können. Wirft sofort und bedingungslos die Ausnahmemeldung aus.

`void setValue(TTCN3Type val) throws IncompatibleTypeException`

versucht, die Felder anhand einer anderen Strukturvariable zu setzen. Die Ausnahmemeldung wird ausgeworfen, wenn

- das Parameter `val` gar keine Strukturvariable ist.
- `val` Felder enthält, die nicht in der aktuellen Struktur definiert sind (nur Namen werden beachtet).
- Felder mit demselben Namen in beiden Strukturen eines inkompatiblen Typ sind.
- ein Feld in der angegebenen Strukturvariable ist, mit einem undefinierten Wert belegt und ein Feld mit dem gleichen Namen im aktuellen Strukturtyp obligatorisch ist.

Wenn keine der obigen Bedingungen zutrifft, werden Feldwerte der angegebenen Struktur `val` den Feldern mit demselben Namen in der aktuellen Strukturvariable zugewiesen.

`String fieldsToString()`

liefert die Stringrepräsentation der Felddefinitionen. Eine mögliche Ausgabe ist

„`{field1:IntegerType [, field2:myUnionType], field3:CharstringType}`“,

wenn die entsprechende Stuktur mit drei Feldern und das Feld `field2` als optional definiert wurde.

`byte[] encodeValue()`

dient der Nachrichtenübertragung an das SUT. Das Ergebnis ist ein Bytestrom, der Werte der Feldbelegungen repräsentiert und an das TRI geschickt wird.

`void decodeValue(byte[] value) throws ValueOutOfRangeException`

dient dem Datenempfang bei der Kommunikation mit dem SUT. Es entschlüsselt den Bytestrom der von der TRI empfangen wurde und setz die Feldwerte. Wirft eine Ausnahmemeldung aus, wenn Felder nicht die gegebenen Werte aufnehmen können bzw. dürfen.

5.3.1.4.32 Klasse StructFieldValue

dient der Felddefinition für die Klasse `StructuredType` (Abschnitt 5.3.1.4.31) und somit für weitere Strukturtypen.

KONSTRUKTOREN

`StructFieldValue(TTCN3Type value)`

erstellt einen Feldwert in der Wertelistennotation (keine Feldnamenangabe)

`StructFieldValue(String name, TTCN3Type value)`

erstellt einen Feldwert in der Zuweisungsnotation.

`StructFieldValue(String name, TemplateValue rule)`

erstellt einen Templatefeldwert und belegt ihn mit dem Regelobjekt.

FUNKTIONEN

`boolean isOmitted()`

ermittelt, ob der Feldwert zurückgesetzt ist. Dies ist der Fall, wenn das Feld mit einer Instanz der Klasse `OmittedValue` (5.3.1.4.33) belegt ist.

`boolean isNotUsed()`

ermittelt, ob der Feldwert unbenutzt ist. Dies ist der Fall, wenn das Feld mit einer Instanz der Klasse `NotUsedValue` (5.3.1.4.34) belegt ist.

`boolean isUndefined()`

ermittelt, ob der Feldwert unbenutzt ist. Dies ist der Fall, wenn das Feld mit einer Instanz der Klasse `UndefinedValue` (5.3.1.4.35) belegt ist.

`TTCN3Type getValue()`

liefert den gespeicherten Feldwert zurück.

`String getName()`

liefert den gespeicherten Namen zurück. Ist das Feldobjekt in der Wertelistennotation erstellt worden, liefert diese Funktion **null** zurück.

`TemplateValue getRule()`

liefert das gespeicherte Regelobjekt zurück. Bei Feldvariablen ist das Ergebnis **null**.

5.3.1.4.33 Klasse OmittedValue

dient den Feldzuweisungen eines Strunkturtyps, wie `RecordType` (5.3.1.4.36) und zwar zum Zurücksetzen eines optionalen Feldes. Eine solche Zuweisung geschieht in TTCN-3 mittels des Schlüsselwortes **omit**, Beispiele 5.15, 5.20.

KONSTRUKTOREN

```
OmittedValue(),
OmittedValue(Comparable
value),
OmittedValue(TemplateValue rule)
```

erstellen eine leere Variable. Das angegebenen Werte- oder Regelobjekt wird verworfen, so dass mittels Instanzen der Klasse `OmittedValue` keine Werte übertragen oder überprüft werden können.

FUNKTIONEN

```
void setValue(Comparable value),
void setValue(TemplateValue rule)
```

überschreiben die Funktionen der Klasse `TTCN3Type` (5.3.1.4.6) und verwerfen das übergebene Werte- bzw. Regelobjekt. Auf diese Weise kann `OmittedValue` nicht nachträglich als eine Werte- oder Templatevariable zweckentfremdet werden.

`boolean checkValue(Comparable value)`
liefert konstant **false**, unabhängig vom angegebenen Wert zurück.

`String toString()`
liefert die Strindarstellung der Variable, und zwar konstant `"{omitted}"`.

5.3.1.4.34 Klasse NotUsedValue

dient den Feldzuweisungen eines Strunkturtyps, wie `RecordType` (5.3.1.4.36) und zwar zum Auslassen der Wertzuweisung. Eine solche Zuweisung geschieht in TTCN-3 mittels des Schlüsselzeichens **-**, Beispiele 5.15, 5.20.

Die Klasse `NotUsedValue` ist bis auf die Funktion `toString()` identisch mit der Klasse `OmittedValue` (5.3.1.4.33) implementiert:

KONSTRUKTOREN

```
NotUsedValue(),
NotUsedValue(Comparable value),
NotUsedValue(TemplateValue rule)
```

erstellen eine leere Variable, das angegebene Werte- oder Regelobjekt wird verworfen.

FUNKTIONEN

```
void setValue(Comparable value),
void setValue(TemplateValue rule)
```

überschreiben die Funktionen der Klasse `TTCN3Type` (5.3.1.4.6) und verwerfen das übergebene Werte- bzw. Regelobjekt.

`boolean checkValue(Comparable value)`
liefert konstant **false**, unabhängig vom angegebenen Wert zurück.

`String toString()`
liefert die Strindarstellung der Variable, und zwar konstant `"{-}"`.

5.3.1.4.35 Klasse UndefinedValue

wird anstelle eines undefinierten Feldwertes eines Strukturtypes wie `RecordType` (5.3.1.4.36) gespeichert. Ist ein Strukturtyp instanziiert, enthalten alle seine Felder bis zu deren Initialisierung Instanzen des Typs `UndefinedValue`. Ebenso werden Feldwerte durch Instanzen dieser Klasse überschrieben, sobald das Feld bei der Zuweisung mittels des Schlüsselwortes '`omit`' zurückgesetzt wurde (5.3.1.4.33).

Die Klasse `UndefinedValue` ist bis auf die Funktion `toString()` identisch mit der Klasse `OmittedValue` (5.3.1.4.33) implementiert:

KONSTRUCTOREN

```
UndefinedValue(),
UndefinedValue(Comparable value),
UndefinedValue(TemplateValue rule)
```

erstellen eine leere Variable, das angegebene Werte- oder Regelobjekt wird verworfen.

FUNKTIONEN

```
void setValue(Comparable value),
void setValue(TemplateValue rule)
```

überschreiben die Funktionen der Klasse `TTCN3Type` (5.3.1.4.6) und verwerfen das übergebene Werte- bzw. Regelobjekt.

`boolean checkValue(Comparable value)`

liefert konstant **false**, unabhängig vom angegebenen Wert zurück.

`String toString()`

liefert die Strindarstellung der Variable, und zwar konstant `"{undefined}"`.

Im Weiteren, bei der Dokumentation der einzelnen Strukturtypen, wird nur auf deren zusätzliches Verhalten eingegangen.

5.3.1.4.36 Klasse RecordType

Die Klasse `RecordType` implementiert den TTCN-3 Typ `record` und kann so behandelt werden, wie im Abschnitt 5.3.1.4.30 dargestellt. Da die Reihenfolge der Felder definiert ist, ist die Wertelistennotation erlaubt und einzelne Felder besitzen eine Position. Die Klasse `RecordType` veröffentlicht deshalb auch geschützte Funktionen von `Structurtype`, die die Variablenfelder mit ihrer Position ansprechen.

KONSTRUCTOREN

```
RecordType(),
RecordType(StructFieldValue[] values),
RecordType(Comparable value)
```

Leere Recordvariable. Die Parameter werden, wie auch bei `StructuredType` (5.3.1.4.31), ignoriert, aus den dort erläuterten Gründen.

`RecordType(TemplateValue rule)`

Eine Templatevariable vom Typ `RecordType`.

FUNKTIONEN

`getField(Integer number)`

ermittelt den Wert eines Feldes, das in der Typdeklaration an der angegebenen Position definiert wurde. Das erste Feld ist an der Position `0` definiert.

`Integer fieldPosition(String name)`

ermittelt die Position eines Feldes, das in der Typdeklaration mit dem angegebenen Namen definiert wurde. Das erste Feld ist an der Position `0` definiert.

`String fieldName(Integer position)`

ermittelt den Namen eines Feldes, das in der Typdeklaration an der angegebenen Position definiert wurde. Das erste Feld ist an der Position `0` definiert.

`Class<TTCN3Type> fieldType (Integer position)`

ermittelt den Typ eines Feldes, das in der Typdeklaration an der angegebenen Position definiert wurde. Das erste Feld ist an der Position `0` definiert.

`Boolean isOptional(Integer position)`

ermittelt, ob das in der Typdeklaration an der gegebenen Position definierte Feld optional definiert ist oder nicht. Das erste Feld ist an der Position `0` definiert.

5.3.1.4.37 Klasse SetType

Die Klasse `SetType` implementiert den TTCN-3 Typ `set`. Dessen Definition unterstützt keine Reihenfolge der Felder, so dass auch Zuweisungen in der Wertelistennotation zu einem Fehler führen.

KONSTRUKTOREN

```
SetType(),
SetType(StructFieldValue[] values),
SetType(Comparable
value)
```

Leere Setvariable. Die Parameter werden, wie auch bei `StructuredType` (5.3.1.4.31) ignoriert, aus den dort erläuterten Gründen.

`SetType(TemplateValue rule)`

Eine Templatevariable.

FUNKTIONEN

`setValue(StructFieldValue... values) throws IncompatibleTypeException`

überschreibt die Funktion `StructuredType.setValue()` und zwar deshalb, weil der TTCN-3 Typ `set` nur Zuweisungen in der Zuweisungsnotation erlaubt. Die Reihenfolge der Felder spielt keine Rolle mehr und die Wertelistennotation ist nicht mehr eindeutig. [14, S. 35]

5.3.1.4.38 Klasse UnionType

Die Klasse `UnionType` implementiert den TTCN-3 Typ `union`. Dieser Typ unterscheidet sich von `StructuredType` dahingehend, dass zu jeder Zeit nur ein Feld belegt sein darf. Wertezuweisungen zu einem Feld setzt das zuvor definierte Feld zurück. Eine weitere Einschränkung ist, dass Unions keine optionalen Felder kennen. [14, S. 38]

KONSTRUCTOREN

```
UnionType(),
UnionType(StructFieldValue[]
values),
UnionType(Comparable value)
```

Leere Variable. Die Parameter werden, wie auch bei `StructuredType` (5.3.1.4.31), aus den dort erläuterten Gründen ignoriert.

`UnionType(TemplateValue rule)`

Eine Templatevariable.

FUNKTIONEN

`void defineField(TTCN3Type type, String name, boolean optional)`

überschreibt die Funktion `StructuredType.defineField()` der Superklasse und ignoriert die Anfrage, sobald ein optionales Feld definiert werden soll (keine Optionale Felder bei Unions erlaubt).

```
void setValue(StructFieldValue... values)
     throws IncompatibleTypeException
```

überschreibt die Funktion der Superklasse und gibt zusätzlich einen Fehler aus, falls in der übergebenen Werteliste mehr als ein belegter Wert vorhanden ist oder wenn ein Feld zurückgesetzt werden soll, denn dies ist bei obligatorischen Feldern nicht erlaubt.

`void setValue(TTCN3Type val) throws IncompatibleTypeException`

überschreibt die Funktion der Superklasse und gibt zusätzlich einen Fehler aus, falls die angegebene Strukturvariable mehr als ein Feld belegt.

```
void setField(String name, TTCN3Type value)
     throws IncompatibleTypeException
```

überschreibt die Funktion der Superklasse, setzt das angegebene Feld auf den gegebenen Wert und setzt alle anderen Feldern auf '*undefiniert*' zurück.

5.4. Automatische Testfallgenerierung

Dieser Teil der Projektgruppe begleitet und unterstützt andere Projektteile.

Es ist wichtig, bei fortschreitender Entwicklung des Projektes, zeitnah auf Fehlerarmut zu testen. Hierzu wird eine Sammlung von Testfällen benötigt, die einen möglichst großen Teil der Funktionalität abdecken. Bei der Erstellung dieser Testfälle gibt es zwei Arbeitsparadigmen:

- Proaktiv den Fortschritt zu beobachten und durch maßgeschneiderte Testfälle zu unterbauen.
- Reaktiv gefundene Fehlerfälle zu isolieren und in die Testfallsammlung zu integrieren.

Das reaktive Erstellen von Testfällen ist ein natürlich stattfindender Vorgang, der dem Arbeitsfluss der Projektfortentwicklung entspringt. Hierfür sind keine weiteren Vorkehrungen notwendig.

Das proaktive Erstellen von Testfällen hingegen ist insofern problematisch, als dass die theoretisch mögliche Anzahl von Testfällen praktisch unbegrenzt ist, wobei hingegen die Anzahl der real manuell erstellbaren Testfälle durch die maximal aufwendbare Arbeitszeit eng begrenzt wird. Deshalb kann nur eine kleine Untermenge der denkbaren Tests durchgeführt werden.

Um diese Einschränkung etwas zu lockern, hat die Testfallgenerierungsgruppe an einer Lösung gearbeitet, die die Methoden des maschinellen Lernens umfasst. Hierfür wurde uns eine am Lehrstuhl 5 entwickelte Bibliothek („LearnLib“) zur Verfügung gestellt, die in der Lage ist, mit wenig Aufwand zu einem automatisierten Lerner für deterministische finite Automaten (DFAs) erweitert zu werden.

Hierfür wurde ein Alphabet über die Sprache TTCN3 (bestehend z.. aus Klammern, Keywords, Bezeichnern etc.) erstellt, sowie einige Beispiele über dieses Alphabet, mit denen der automatische Lerner ein internes Anfangsmodell generieren kann. Fortan gibt der Lerner Beispiele aus, die mit Hilfe des internen Modells erstellt werden. Diese Beispiele müssen als zur Sprache gehörig oder ungehörig markiert werden, so dass der Lerner mit Hilfe dieser neuen Informationen sein internes Modell revidieren und verbessern kann.

Dies ist eine iterative Prozedur, welche endet, wenn ein konsistentes internes Modell erarbeitet worden ist. Die während der Lernphase generierten Beispiele, werden mitprotokolliert und stehen fortan für Tests im Rahmen der Projektgruppe zur Verfügung.

Die Klassifikation, ob ein Beispiel gültig oder nicht gültig ist, übernimmt in unserem Falle der (kommerzielle) Compiler aus der TTWorkbench der Firma Testing Tech, der von uns als Referenzimplementation von TTCN3 betrachtet wird. Somit kann das Lernverfahren vollautomatisch durchgeführt werden.

5.4.1. Verteilung des Lernprozesses auf einen Cluster

Um ein konsistentes Modell zu erstellen, muss der maschinelle Lernalgorithmus einige zehntausend Beispiele generieren, die zur Fortführung des Lernprozesses korrekt klassifiziert werden müssen. Hierbei rücken Überlegungen zur Performanz des Klassifizierers in den Blickwinkel unserer Betrachtungen.

Der von uns als Klassifizierer eingesetzte Compiler der TTWorkbench benötigt mehrerer Sekunden, um ein Beispiel bearbeitet und eingeordnet zu haben. Unter Berücksichtigung der Vielzahl der Eingaben, die zu bearbeiten sind, wird schnell klar, dass somit ein Lerndurchlauf eine Zeitspanne in Anspruch nimmt, die jenseits akzeptabler Grenzen liegt.

Um trotzdem das beschriebene Verfahren einsetzen zu können, bedarf es einer signifikanten Beschleunigung des Klassifizierungsprozesses. Hierbei hat sich herausgestellt, dass die einzige Option, die Performanz hinreichend zu erhöhen, darin besteht, die Klassifizierung parallel auf einer Vielzahl von Rechner durchzuführen. Die Universität Dortmund verfügt über einen Verbund von Rechnern (Cluster), der hierzu geeignet erscheint.

Jeder am Rechenverbund beteiligte Rechner verfügt über Zugriff auf den TTWorkbench Compiler. Eine zentraler Rechner, der das eigentliche Lernverfahren steuert und über die Anbindung an die Lernverfahren der LearnLib verfügt, sammelt die generierten Beispiele und verteilt diese anschließend über Mechanismen des Remote Method Invocation (RMI) an die im Verbund befindlichen Rechner, welche sich zuvor (ebenfalls über RMI) als dienstbereite Rechner bei dem zentralen Rechner angemeldet haben. Nach Erhalt der Klassifizierungen gibt der Zentralrechner diese an den Lernalgorithmus weiter und erwartet die Generierung neuer Beispiele.

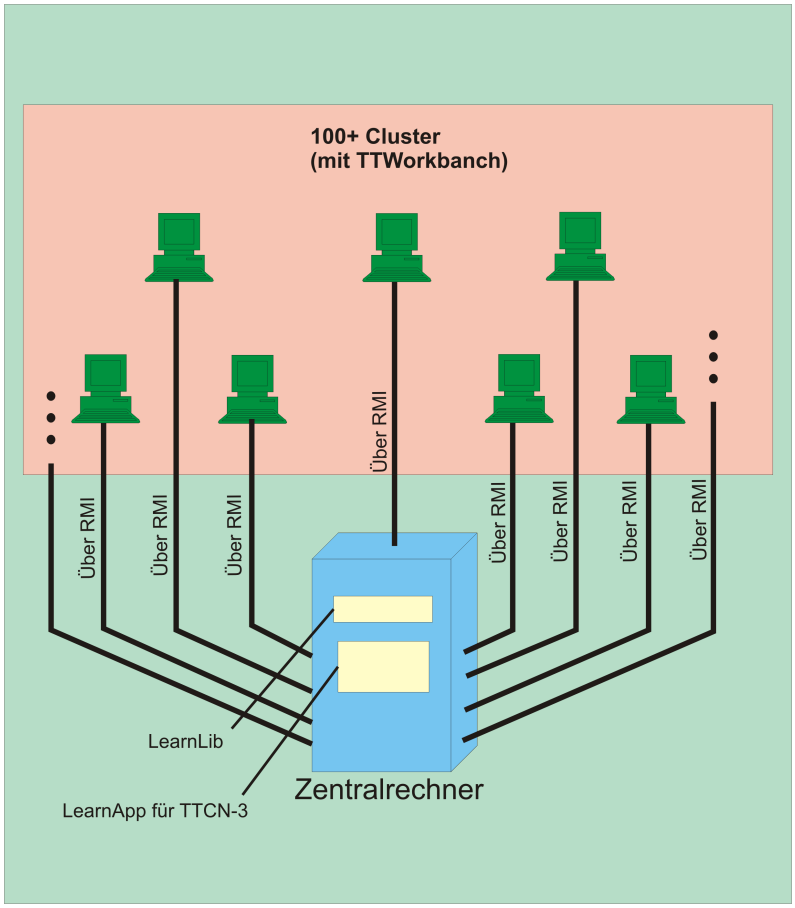

Abbildung 5.32.: Verteilung des Lernprozesses auf einen Cluster

6. Fazit

Abschließend steht in diesem Kapitel eine Zusammenfassung mit Hinblick auf das Minimalziel der PG und dem tatsächlich erreichten Stand. Auch Aspekte zur Weiterentwicklung der entwickelten Tools werden betrachtet.

Ziel der Projektgruppe war es, die zuvor skizzierte Bluetooth-Anwendung in einem gegebenen Szenario erfolgreich zu realisieren. Im Einzelnen umfasste diese Anforderung damit insbesondere das Erstellen einer Entwicklungsumgebung basierend auf jABC, die es erlaubt, die Bluetooth-Anwendung graphisch zu visualisiern. Weiterhin mussten die angesprochenen Testfälle realisiert werden. Zudem sollte noch eine Ausführungsplattform für eine Teilmenge von TTCN-3 inklusive einer Anbindung an das Runtime-Interface entwickeln und implementiert werden.

Die Entscheidung für Java als Zielsprache des TTCN-3-Tools fiel sehr schnell, da Java eine plattformunabhängige Sprache ist, die aktiv weiterentwickelt wird und eine akzeptierte Plattform für akademische und auch professionelle Entwicklungen darstellt.

Die Aufgaben der Parsergruppe ist gewesen, einen Scanner und Parser für TTCN-3 zu entwickeln (es sollte die vollständige Sprache implementiert), was durch die Anwendung des ANTLR Parsergenerators auch gelungen ist. Des Weiteren wurden Selbsttests für das Zieltool erarbeitet. Zu diesem Zweck sind mehrere TTCN-3-Miniskripte erstellt worden, die das erfolgreiche Kompilieren von korrekten bzw. Abweisen von inkorrekten TTCN-3 Codes bestätigen können und diesen Test zu jedem Entwicklungszeitpunkt erlauben.

Somit hat die Parsergruppe ihre Teilaufgaben, wie sie im Zeitplan, Abschnitt 1.5 angegeben sind, erfüllt.

Die Selbsttestfähigkeit wurde ausgebaut. Während mehrere korrekte und inkorrekte Codeabschnitte die Korrektheit der Implementierung des Zieltools überprüft haben, wobei diese Codeabschnitte wiederum manuell erstellt werden mussten, hat nun dieses Erstellen ein maschineller Lerner übernommen, der auf die Syntax und Semantik von TTCN-3 trainiert wurde und auf Änderungen der TTCN-3 Spezifikation flexibler reagierte.

Da die Parsergruppe ihre Aufgabe bereits im 1. Semester erfüllt hatte, wurden die Mitglieder in die anderen Gruppen integriert und haben dort neue Aufgaben übernommen. Die Aufgabe der Instandhaltung, Erweiterung und Pflege des Parserteils des TTCN-3-Tools blieb jedoch weiterhin bestehen.

Die Semantikchecker-Gruppe hat ihr Analysetool in zwei Stufen unterteilt (Abschnitt 4.1.2). Die erste Stufe war im 1. Semester weitgehend abgeschlossen. Es wurden zum einen noch Imports und Templates abgeschlossen, und andererseits noch Fehler, die in Programmtests noch auftreten können, korrigiert.

In der zweiten Ausbaustufe des Sematikcheckers wurden komplizierte Typen, wie Ports, Komponenten und Templates in die Struktur integriert. Ausdrücke im if-, else- und for- Statement wurden ebenfalls erstellt.

Um das entwickelte Programm zu testen, waren eine Vielzahl von Testfällen nötig. Die manuelle Erstellung von Testfällen ist sehr aufwendig. Daher wurde mit dem Ansatz der automatischen Testfallgenerierung dieses Problem gelöst.

Nachdem sich die Compiler-Gruppe im ersten Semester ausgiebig mit den verschiedensten

Architekturen auseinandergesetzt hat, ist die Entscheidung über den Bau eines Compilers getroffen worden. Mit der Implementierung der Architektur, wie in Kapitel 2.2 beschrieben, ist daraufhin auch gleich begonnen worden. Bis zum Ende des ersten Semesters wurde die erste Ausbaustufe des Compilers umgesetzt, die eine kleine Grammatik von TTCN-3 in Java übersetzt.

In der zweiten Ausbaustufe ist die Gruppe im 2. Semester, neben der ständigen Korrektur des Sourcecodes, mit dem Ausbau der Grammatik von TTCN-3 weiter fortgefahren. Es wurden komplizierte Konstrukte wie Ports, Timer, Typen und Komponenten in das Laufzeitsystem integriert.

Abschließend ist zu erwähnen, dass die Projektgruppe es erfolgreich geschafft hat, die Bluetooth-Anwendungen zu realisieren. Da jedoch einzelne Aspekte von TTCN-3 für die Belange der Projektgruppe nicht relevant waren, bestehen hier Möglichkeiten zu weiteren Arbeiten.

6.1. Ausblick

Im Rahmen des Projektes ist eine funktionsfähige Ausführungsplattform für die Sprache TTCN-3 entstanden. Weitere Teile der Spezifikation wurden umgesetzt und sind robust genug implementiert, um auf der Basis des Erreichten weitere Arbeiten durchzuführen. Eine Offenlegung des Quelltextes der im Rahmen des Projektes erstellten Tools ist vorgesehen. Hieraus ergibt sich die Möglichkeit, eine komplette und spezifikationsgetreue Implementation des TTCN-3 Standards innerhalb eines akademisch oder kommerziell vorangetriebenen Projektes unter Verwendung der Ergebnisse der TTCP-Projektgruppe zu erstellen. Auf jeden Fall ist das Arbeitsergebnis der Projektgruppe dazu geeignet, Neueinsteigern eine kostenlose Umgebung zu bieten, mit der sie sich die Grundzüge von TTCN-3 aneignen können.

Teile der Spezifikation, die aufgrund fehlender Notwendigkeit für das erreichen der Ziele der Projektgruppe noch nicht umgesetzt worden sind, zeigen Arbeitsgebiete auf, die noch in Angriff zu nehmen sind. Dies trifft zum Beispiel auf die prozedurenbasierte Kommunikation zu, die bisher komplett außer Acht gelassen wurde. Auch sind Anbindungen für andere Testszenarien als dem von der Projektgruppe verfolgten Bluetooth-Testszenario denkbar. Parallel zu eventuellen Erweiterungen müssen geeignete Testfälle entwickelt werden, um den Entwicklungsfortschritt durch kontinuierliches Testen der Implementation zu untermauern und somit Rückschritte in Bezug auf Funktionalität und Fehlerarmut zu verhindern.

A. Anhang

A.1. BNF

A.1.1. TTCN-3 BNF

A.1.1.1. General

Im Anhang wird die Definition von TTCN-3 mittels des Erweiterten BNF gegeben (*EBNF*,) weiterhin einfach *BNF* bezeichnet. Obschon der erste Teil des Anhangs ins Deutsche übersetzt ist, wird der BNF-Teil unverändert wiedergegeben und entspricht der Definition von ETSI, [14, Anhang A, s.155].

A.1.1.2. Konvention für die Syntaxbeschreibung

Die Tabelle A.1 definiert die Metanotation für die Grammatikdefiniton von TTCN-3.

A.1.2. Ausdruckabschlußsymbol

Im Allgemeinen, alle TTCN-3 Sprachkonstrukte (d.. Definitionen, Deklarationen, Ausdrücke und Operationen) werden mit einem Semikolon ';' abgeschlossen. Der Semikolon ist optional wenn der Konstrukt davor mit einer schließenden geschweiften Klammer '}' endet oder dem Semikolon die schließende geschweifte Klammer folgen soll, das heiß wenn der Sprachkonstrukt vor dem Semikolon der letzte Ausdruck in einem Block von Ausdrücke, Operationen und Deklarationen.

```
abc ::= xyz     'ist' definiert als 'xyz'
abc xyz         'abc' gefolgt von 'xyz'
|                Alternative
[abc]            0 oder 1 Instanz von 'abc'
{abc}            0 oder mehr Instanzen von 'abc'
{abc}+           1 oder mehr Instanzen von 'abc'
(...)            Textuelle Gruppierung
Abc              Nichtterminalsymbol 'Abc'
"Abc"            Terminalsymbol 'Abc'
```

Tabelle A.1.: BNF, Metanotation der Syntax

A.1.3. Bezeichner

Bezeichner in TTCN-3 unterscheiden die Schreibweise (*case sensitive*), müssen nur aus Klein- und Großbuchstaben, Zahlen und dem Unterstrich '_' bestehen, und nur mit einem Buchstaben beginnen.

A.1.4. Kommentare

Kommentare können an beliebigen Orten in der TTCN-3 Spezifikation vorkommen.

Blockkommentare werden mit dem Symbol '/*' geöffnet und mit dem Symbol '*/' geschlossen. Der eingeschlossene Text gilt als Kommentar und darf den schließenden Kommentarsymbol nicht enthalten. Dies erlaubt die Kommentarverschachtelung nicht, Beispiel .

```
/* This is a block comment
spread over two lines */

/* This is not /* a legal */ comment */
```

Beispiel A.1: Blockkommentare in TTCN-3

Zeilenkommentare werden mit einem Slashpaar '//' geöffnet und mit dem Zeilenende beendet. Zeilenkommentare können den Programmelementen folgen, dürfen aber nicht in Programmelementen eingebettet werden, Beispiel A.2.

```
// This is a line comment
// spread over two lines

// The following is not legal const
// This is MyConst integer MyConst := 1;

// The following is legal
const integer MyConst := 1; // This is MyConst
```

Beispiel A.2: Zeilenkommentare in TTCN-3

A.1.5. TTCN-3 Terminalsymbole

Terminalsymbole und reservierte Wörter in TTCN-3 sind in den Tabellen A.2 und A.3.

Namen der vordefinierten Funktionen, die in [14, Tabelle 10] definiert und in [14, Anhang C] beschrieben sind, müssen ebenfalls als reservierte Bezeichner behandelt werden.

Die in der Tabelle A.3 aufgelistete Terminale dürfen nicht als Bezeichner in einem TTCN-3 Modul verwendet werden. Sie werden klein geschrieben.

A.1.6 TTCN-3 Syntaxproduktionen der BNF

A.1.6.0 TTCN-3 Module

```
1. TTCN3Module ::= TTCN3ModuleKeyword TTCN3ModuleId
```

t

Begin/end block symbols	{ }
Begin/end list symbols	( )
Alternative symbols	[ ]
To symbol (in a range)	’..’
Line comments and Block comments	/* */ //
Line/statement terminator symbol	;
Arithmetic operator symbols	+ / −
String concatenation operator symbol	&
Equivalence operator symbols	!= == >= <=
String enclosure symbols	“
Wildcard/matching symbols	? *
Assignment symbol	:=
Communication operation assignment	− >
Bitstring, hexstring and Octetstring values	**O**
Float exponent	**E**

Tabelle A.2.: Spezielle Terminalsymbole von TTCN-3

```
                    "{"
                    [ModuleDefinitionsPart]
                    [ModuleControlPart]
                    "}"
                    [WithStatement] [SemiColon]
2. TTCN3ModuleKeyword ::= "module"
3. TTCN3ModuleId ::= ModuleId
4. ModuleId ::= GlobalModuleId [LanguageSpec]
```

/ STATIC SEMANTICS - LanguageSpec may only be omitted if the referenced module contains TTCN-3 notation */*

```
5. GlobalModuleId ::= ModuleIdentifier
6. ModuleIdentifier ::= Identifier
7. LanguageSpec ::= LanguageKeyword FreeText
8. LanguageKeyword ::= "language"
```

A.1.6.1 Module definitions part

A.1.6.1.0 General

```
9. ModuleDefinitionsPart ::= ModuleDefinitionsList
10. ModuleDefinitionsList ::= {ModuleDefinition [SemiColon]}+
11. ModuleDefinition ::= (TypeDef |
                          ConstDef |
                          TemplateDef |
                          ModuleParDef |
                          FunctionDef |
                          SignatureDef |
                          TestcaseDef|
```

action	fail	noblock	select
activate	false	none	self
address	float	not	send
alive	for	not4b	sender
all	from	nowait	set
alt	function	null	setverdict
altstep			signature
and	getverdict	octetstring	start
and4b	getcall	of	stop
any	getreply	omit	subset
anytype	goto	on	superset
	group	optional	system
bitstring		or	
boolean	hexstring	or4b	template
		out	testcase
case	if	override	timeout
call	ifpresent		timer
catch	import	param	to
char	in	pass	trigger
charstring	inconc	pattern	true
check	infinity	port	type
clear	inout	procedure	
complement	integer		union
component	interleave	raise	universal
connect		read	unmap
const	kill	receive	
control	killed	record	value
create		rem	valueof
	label	repeat	var
deactivate	language	reply	variant
default	length	return	verdicttype
disconnect	log	running	
display		runs	while
do	map		with
done	match		
	message		xor
else	mixed		xor4b
encode	mod		
enumerated	modifies		
error	module		
except	modulepar		
exception	mtc		
execute			
extends			
extension			
external			

Tabelle A.3.: Reservierte Bezeichner von TTCN-3

```
                        AltstepDef |
                        ImportDef |
                        GroupDef |
                        ExtFunctionDef |
                        ExtConstDef) [WithStatement]
```

A.1.6.1.1 Typedef definitions

```
12. TypeDef ::= TypeDefKeyword TypeDefBody
13. TypeDefBody ::= StructuredTypeDef | SubTypeDef
14. TypeDefKeyword ::= "type"
15. StructuredTypeDef ::= RecordDef |
                          UnionDef |
                          SetDef |
                          RecordOfDef |
                          SetOfDef |
                          EnumDef |
                          PortDef |
                          ComponentDef
16. RecordDef ::= RecordKeyword StructDefBody
17. RecordKeyword ::= "record"
18. StructDefBody ::= (StructTypeIdentifier [StructDefFormalParList]
                                            | AddressKeyword)
                      "{" [StructFieldDef {"," StructFieldDef}] "}"
19. StructTypeIdentifier ::= Identifier
20. StructDefFormalParList ::= "(" StructDefFormalPar
                           {"," StructDefFormalPar} ")"
21. StructDefFormalPar ::= FormalValuePar
```

/ STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */*

```
22. StructFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier
                                              [ArrayDef] [SubTypeSpec]
                       [OptionalKeyword]
23. NestedTypeDef ::= NestedRecordDef |
                      NestedUnionDef |
                      NestedSetDef |
                      NestedRecordOfDef |
                      NestedSetOfDef |
                      NestedEnumDef
24. NestedRecordDef ::= RecordKeyword "{" [StructFieldDef
                                             {"," StructFieldDef}] "}"
25. NestedUnionDef ::= UnionKeyword "{" UnionFieldDef
                                           {"," UnionFieldDef} "}"
26. NestedSetDef ::= SetKeyword "{" [StructFieldDef
                    {"," StructFieldDef}] "}"
27. NestedRecordOfDef ::= RecordKeyword [StringLength] OfKeyword
                          (Type | NestedTypeDef)
28. NestedSetOfDef ::= SetKeyword [StringLength] OfKeyword
```

```
                        (Type | NestedTypeDef)
29. NestedEnumDef ::= EnumKeyword "{" EnumerationList "}"
30. StructFieldIdentifier ::= Identifier
31. OptionalKeyword ::= "optional"
32. UnionDef ::= UnionKeyword UnionDefBody
33. UnionKeyword ::= "union"
34. UnionDefBody ::= (StructTypeIdentifier [StructDefFormalParList]
                     | AddressKeyword)
                     "{" UnionFieldDef {"," UnionFieldDef} "}"
35. UnionFieldDef ::= (Type | NestedTypeDef) StructFieldIdentifier
                      [ArrayDef] [SubTypeSpec]
36. SetDef ::= SetKeyword StructDefBody
37. SetKeyword ::= "set"
38. RecordOfDef ::= RecordKeyword [StringLength] OfKeyword
                    StructOfDefBody
39. OfKeyword ::= "of"
40. StructOfDefBody ::= (Type | NestedTypeDef) (StructTypeIdentifier
                        | AddressKeyword) [SubTypeSpec]
41. SetOfDef ::= SetKeyword [StringLength] OfKeyword StructOfDefBody
42. EnumDef ::= EnumKeyword (EnumTypeIdentifier | AddressKeyword)
                "{" EnumerationList "}"
43. EnumKeyword ::= "enumerated"
44. EnumTypeIdentifier ::= Identifier
45. EnumerationList ::= Enumeration {"," Enumeration}
46. Enumeration ::= EnumerationIdentifier ["("[Minus] Number ")"]
47. EnumerationIdentifier ::= Identifier
48. SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword)
                   [ArrayDef] [SubTypeSpec]
49. SubTypeIdentifier ::= Identifier
50. SubTypeSpec ::= AllowedValues [StringLength] | StringLength
```

/ STATIC SEMANTICS - AllowedValues shall be of the same type as the field being subtyped */*

```
51. AllowedValues ::= "(" (ValueOrRange {"," ValueOrRange})
                      | CharStringMatch ")"
52. ValueOrRange ::= RangeDef | ConstantExpression
```

/ STATIC SEMANTICS - RangeDef production shall only be used with integer, charstring, universal charstring or float based types */ /* STATIC SEMANTICS - When subtyping charstring or universal charstring range and values shall not be mixed in the same SubTypeSpec */*

```
53. RangeDef ::= LowerBound ".." UpperBound
54. StringLength ::= LengthKeyword "(" SingleConstExpression
                     [".." UpperBound] ")"
```

/ STATIC SEMANTICS - StringLength shall only be used with String types or to limit set of and record of. SingleConstExpression and UpperBound shall evaluate to non-negative integer values (in case of UpperBound including infinity) */*

```
55. LengthKeyword ::= "length"
56. PortType ::= [GlobalModuleId Dot] PortTypeIdentifier
57. PortDef ::= PortKeyword PortDefBody
58. PortDefBody ::= PortTypeIdentifier PortDefAttribs
59. PortKeyword ::= "port"
60. PortTypeIdentifier ::= Identifier
61. PortDefAttribs ::= MessageAttribs | ProcedureAttribs
                     | MixedAttribs
62. MessageAttribs ::= MessageKeyword
                     "{" {MessageList [SemiColon]}+ "}"
63. MessageList ::= Direction AllOrTypeList
64. Direction ::= InParKeyword | OutParKeyword | InOutParKeyword
65. MessageKeyword ::= "message"
66. AllOrTypeList ::= AllKeyword | TypeList
```

/ NOTE: The use of AllKeyword in port definitions is deprecated */*

```
67. AllKeyword ::= "all"
68. TypeList ::= Type {"," Type}
69. ProcedureAttribs ::= ProcedureKeyword
                       "{" {ProcedureList [SemiColon]}+ "}"
70. ProcedureKeyword ::= "procedure"
71. ProcedureList ::= Direction AllOrSignatureList
72. AllOrSignatureList ::= AllKeyword | SignatureList
73. SignatureList ::= Signature {"," Signature}
74. MixedAttribs ::= MixedKeyword
                   "{" {MixedList [SemiColon]}+ "}"
75. MixedKeyword ::= "mixed"
76. MixedList ::= Direction ProcOrTypeList
77. ProcOrTypeList ::= AllKeyword | (ProcOrType {"," ProcOrType})
78. ProcOrType ::= Signature | Type
79. ComponentDef ::= ComponentKeyword ComponentTypeIdentifier
                   [ExtendsKeyword ComponentType
                   {"," ComponentType}]
                   "{" [ComponentDefList] "}"
80. ComponentKeyword ::= "component"
81. ExtendsKeyword ::= "extends"
82. ComponentType ::= [GlobalModuleId Dot] ComponentTypeIdentifier
83. ComponentTypeIdentifier ::= Identifier
84. ComponentDefList ::= {ComponentElementDef [SemiColon]}
85. ComponentElementDef ::= PortInstance | VarInstance
                          | TimerInstance | ConstDef
86. PortInstance ::= PortKeyword PortType PortElement
                   {"," PortElement}
87. PortElement ::= PortIdentifier [ArrayDef]
88. PortIdentifier ::= Identifier
```

A.1.6.1.2 Constant definitions

```
89. ConstDef ::= ConstKeyword Type ConstList
```

/ STATIC SEMANTICS - Type shall follow the rules given in clause 9 of ES 201 873-1.*/*

```
90. ConstList ::= SingleConstDef {"," SingleConstDef}
91. SingleConstDef ::= ConstIdentifier [ArrayDef] AssignmentChar
                       ConstantExpression
```

/ STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated type for the constants */*

```
92. ConstKeyword ::= "const"
93. ConstIdentifier ::= Identifier
```

A.1.6.1.3 Template definitions

```
94. TemplateDef ::= TemplateKeyword BaseTemplate [DerivedDef]
                    AssignmentChar TemplateBody
95. BaseTemplate ::= (Type | Signature) TemplateIdentifier
                     ["(" TemplateFormalParList ")"]
96. TemplateKeyword ::= "template"
97. TemplateIdentifier ::= Identifier
98. DerivedDef ::= ModifiesKeyword TemplateRef
99. ModifiesKeyword ::= "modifies"
100. TemplateFormalParList ::= TemplateFormalPar
                               {"," TemplateFormalPar}
101. TemplateFormalPar ::= FormalValuePar | FormalTemplatePar
```

/ STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */*

```
102. TemplateBody ::= (SimpleSpec | FieldSpecList |
                       ArrayValueOrAttrib) | [ExtraMatchingAttributes]
```

/ STATIC SEMANTICS - Within TeplateBody the ArrayValueOrAttrib can be used for array, record, record of and set of types. */*

```
103. SimpleSpec ::= SingleValueOrAttrib
104. FieldSpecList ::= "{"[FieldSpec {"," FieldSpec}] "}"
105. FieldSpec ::= FieldReference AssignmentChar TemplateBody
106. FieldReference ::= StructFieldRef | ArrayOrBitRef | ParRef
```

/ STATIC SEMANTICS - Within FieldReference ArrayOrBitRef can be used for record of and set of templates/template fields in modified templates only*/*

```
107. StructFieldRef ::= StructFieldIdentifier| PredefinedType
                        | TypeReference
```

/ STATIC SEMANTICS - PredefinedType and TypeReference shall be used for anytype value notation only. PredefinedType shall not be AnyTypeKeyword.*/*

```
108. ParRef ::= SignatureParIdentifier
```

/ STATIC SEMANTICS - SignatureParIdentifier shall be a formal parameter Identifier from the associated signature definition */*

```
109. SignatureParIdentifier ::= ValueParIdentifier
110. ArrayOrBitRef ::= "[" FieldOrBitNumber "]"
```

/ STATIC SEMANTICS - ArrayRef shall be optionally used for array types and ASN.1 SET OF and SEQUENCE OF and TTCN-3 record of and set of. The same notation can be used for a Bit reference inside an ASN.1 or TTCN-3 bitstring type */*

```
111. FieldOrBitNumber ::= SingleExpression
```

/ STATIC SEMANTICS - SingleExpression will resolve to a value of integer type */*

```
112. SingleValueOrAttrib ::= MatchingSymbol |
                             SingleExpression |
                             TemplateRefWithParList
```

/ STATIC SEMANTIC - VariableIdentifier (accessed via singleExpression) may only be used in in-line template definitions to reference variables in the current scope */*

```
113. ArrayValueOrAttrib ::= "{" ArrayElementSpecList "}"
114. ArrayElementSpecList ::= ArrayElementSpec
                               {"," ArrayElementSpec}
115. ArrayElementSpec ::= NotUsedSymbol | PermutationMatch
                        | TemplateBody
116. NotUsedSymbol ::= Dash
117. MatchingSymbol ::= Complement |
                        AnyValue |
                        AnyOrOmit |
                        ValueOrAttribList |
                        Range |
                        BitStringMatch |
                        HexStringMatch |
                        OctetStringMatch |
                        CharStringMatch |
                        SubsetMatch |
                        SupersetMatch
118. ExtraMatchingAttributes ::= LengthMatch | IfPresentMatch
                                | LengthMatch IfPresentMatch
119. BitStringMatch ::= "'" {BinOrMatch} "'" ""
120. BinOrMatch ::= Bin | AnyValue | AnyOrOmit
121. HexStringMatch ::= "'" {HexOrMatch} "'" ""
122. HexOrMatch ::= Hex | AnyValue | AnyOrOmit
123. OctetStringMatch ::= "'" {OctOrMatch} "'" "O"
124. OctOrMatch ::= Oct | AnyValue | AnyOrOmit
125. CharStringMatch ::= PatternKeyword Cstring
126. PatternKeyword ::= "pattern"
```

```
127. Complement ::= ComplementKeyword ValueList
128. ComplementKeyword ::= "complement"
129. ValueList ::= "(" ConstantExpression
                   {"," ConstantExpression} ")"
130. SubsetMatch ::= SubsetKeyword ValueList
```

/ STATIC SEMANTIC - Subset matching shall only be used with the set of type */*

```
131. SubsetKeyword ::= "subset"
132. SupersetMatch ::= SupersetKeyword ValueList
```

/ STATIC SEMANTIC - Superset matching shall only be used with the set of type */*

```
133. SupersetKeyword ::= "superset"
134. PermutationMatch ::= PermutationKeyword PermutationList
135. PermutationKeyword ::= "permutation"
136. PermutationList ::= "(" TemplateBody
                         { "," TemplateBody } ")"
```

/ STATIC SEMANTICS: Restrictions on the content of TemplateBody are given in clause .1.3.3 */*

```
137. AnyValue ::= "?"
138. AnyOrOmit ::= "*"
139. ValueOrAttribList ::= "(" TemplateBody
                           {"," TemplateBody}+ ")"
140. LengthMatch ::= StringLength
141. IfPresentMatch ::= IfPresentKeyword
142. IfPresentKeyword ::= "ifpresent"
143. Range ::= "(" LowerBound ".." UpperBound ")"
144. LowerBound ::= SingleConstExpression | Minus
                    InfinityKeyword
145. UpperBound ::= SingleConstExpression | InfinityKeyword
```

/ STATIC SEMANTICS - LowerBound and UpperBound shall evaluate to types integer, charstring, universal charstring or float. In case LowerBound or UpperBound evaluates to types charstring or universal charstring, only SingleConstExpression may be present and the string length shall be 1*/*

```
146. InfinityKeyword ::= "infinity"
147. TemplateInstance ::= InLineTemplate
148. TemplateRefWithParList ::= [GlobalModuleId Dot]
                                TemplateIdentifier
                                [TemplateActualParList]
                                | TemplateParIdentifier
149. TemplateRef ::= [GlobalModuleId Dot] TemplateIdentifier
                   | TemplateParIdentifier
150. InLineTemplate ::= [(Type | Signature) Colon]
                        [DerivedRefWithParList AssignmentChar]
                        TemplateBody
```

/ STATIC SEMANTICS - The type field may only be omitted when the type is implicitly unambiguous */*

```
151. DerivedRefWithParList ::= ModifiesKeyword TemplateRefWithParList
152. TemplateActualParList ::= "(" TemplateActualPar
                               {"," TemplateActualPar} ")"
153. TemplateActualPar ::= TemplateInstance
```

/ STATIC SEMANTICS - When the corresponding formal parameter is not of template type the TemplateInstance production shall resolve to one or more SingleExpressions */*

```
154. TemplateOps ::= MatchOp | ValueofOp
155. MatchOp ::= MatchKeyword "(" Expression "," TemplateInstance")"
```

/ STATIC SEMANTICS - The type of the value returned by the expression must be the same as the template type and each field of the template shall resolve to a single value */*

```
156. MatchKeyword ::= "match"
157. ValueofOp ::= ValueofKeyword "(" TemplateInstance")"
158. ValueofKeyword ::= "valueof"
```

A.1.6.1.4 Function definitions

```
159. FunctionDef ::= FunctionKeyword FunctionIdentifier
                     "("[FunctionFormalParList] ")" [RunsOnSpec]
                     [ReturnType]
                     StatementBlock
160. FunctionKeyword ::= "function"
161. FunctionIdentifier ::= Identifier
162. FunctionFormalParList ::= FunctionFormalPar
                               {"," FunctionFormalPar}
163. FunctionFormalPar ::= FormalValuePar |
                           FormalTimerPar |
                           FormalTemplatePar |
                           FormalPortPar
164. ReturnType ::= ReturnKeyword [TemplateKeyword] Type
```

/ STATIC SEMANTICS - The use of the template keyword shall conform to restrictions in clause 16.1.0 of ES 201 873-1*/*

```
165. ReturnKeyword ::= "return"
166. RunsOnSpec ::= RunsKeyword OnKeyword ComponentType
167. RunsKeyword ::= "runs"
168. OnKeyword ::= "on"
169. MTCKeyword ::= "mtc"
170. StatementBlock ::= "{" [FunctionStatementOrDefList] "}"
171. FunctionStatementOrDefList ::= {FunctionStatementOrDef
                                    [SemiColon]}+
172. FunctionStatementOrDef ::= FunctionLocalDef |
                                FunctionLocalInst |
```

```
                                FunctionStatement
173. FunctionLocalInst ::= VarInstance | TimerInstance
174. FunctionLocalDef ::= ConstDef | TemplateDef
175. FunctionStatement ::= ConfigurationStatements |
                           TimerStatements |
                           CommunicationStatements |
                           BasicStatements |
                           BehaviourStatements |
                           VerdictStatements |
                           SUTStatements
176. FunctionInstance ::= FunctionRef "("
                          [FunctionActualParList] ")"
177. FunctionRef ::= [GlobalModuleId Dot] (FunctionIdentifier
                     | ExtFunctionIdentifier )
                     | PreDefFunctionIdentifier
178. PreDefFunctionIdentifier ::= Identifier
```

/ STATIC SEMANTICS - The Identifier will be one of the pre-defined TTCN-3 Function Identifiers from Annex C of ES 201 873-1*/*

```
179. FunctionActualParList ::= FunctionActualPar {","
                               FunctionActualPar}
180. FunctionActualPar ::= TimerRef |
                           TemplateInstance |
                           Port |
                           ComponentRef
```

/ STATIC SEMANTICS - When the corresponding formal parameter is not of template type the TemplateInstance production shall resolve to one or more SingleExpressions i.e. equivalent to the Expression production */*

A.1.6.1.5 Signature definitions

```
181. SignatureDef ::= SignatureKeyword SignatureIdentifier
                      "("[SignatureFormalParList] ")" [ReturnType
                      | NoBlockKeyword]
                      [ExceptionSpec]
182. SignatureKeyword ::= "signature"
183. SignatureIdentifier ::= Identifier
184. SignatureFormalParList ::= SignatureFormalPar {","
                                SignatureFormalPar}
185. SignatureFormalPar ::= FormalValuePar
186. ExceptionSpec ::= ExceptionKeyword "(" ExceptionTypeList ")"
187. ExceptionKeyword ::= "exception"
188. ExceptionTypeList ::= Type {"," Type}
189. NoBlockKeyword ::= "noblock"
190. Signature ::= [GlobalModuleId Dot] SignatureIdentifier
```

A.1.6.1.6 Testcase definitions

```
191. TestcaseDef ::= TestcaseKeyword TestcaseIdentifier
                     "("[TestcaseFormalParList] ")"
                     ConfigSpec
                     StatementBlock
192. TestcaseKeyword ::= "testcase"
193. TestcaseIdentifier ::= Identifier
194. TestcaseFormalParList ::= TestcaseFormalPar {","
                               TestcaseFormalPar}
195. TestcaseFormalPar ::= FormalValuePar |
                           FormalTemplatePar
196. ConfigSpec ::= RunsOnSpec [SystemSpec]
197. SystemSpec ::= SystemKeyword ComponentType
198. SystemKeyword ::= "system"
199. TestcaseInstance ::= ExecuteKeyword "(" TestcaseRef "("
                          [TestcaseActualParList] ")"
                          ["," TimerValue] ")"
200. ExecuteKeyword ::= "execute"
201. TestcaseRef ::= [GlobalModuleId Dot] TestcaseIdentifier
202. TestcaseActualParList ::= TestcaseActualPar {","
                               TestcaseActualPar}
203. TestcaseActualPar ::= TemplateInstance
```

/ STATIC SEMANTICS - When the corresponding formal parameter is not of template type the TemplateInstance production shall resolve to one or more SingleExpressions i.e. equivalent to the Expression production */*

A.1.6.1.7 Altstep definitions

```
204. AltstepDef ::= AltstepKeyword AltstepIdentifier
                    "("[AltstepFormalParList] ")" [RunsOnSpec]
                    "{" AltstepLocalDefList AltGuardList "}"
205. AltstepKeyword ::= "altstep"
206. AltstepIdentifier ::= Identifier
207. AltstepFormalParList ::= FunctionFormalParList
```

/ STATIC SEMANTICS - altsteps that are activated as defaults shall only have in parameters, port parameters, or timer parameters */*

/ STATIC SEMANTICS -altsteps that are only invoked as an alternative in an alt statement or as stand-alone statement in a TTCN-3 behaviour description may have in, out and inout parameters. */*

```
208. AltstepLocalDefList ::= {AltstepLocalDef [SemiColon]}
209. AltstepLocalDef ::= VarInstance | TimerInstance
                       | ConstDef | TemplateDef
```

*/*STATIC SEMANTICS - AltstepLocalDef shall conform to restrictions in clause 16.2.2.1 of ES 201 873-1*/*

```
210. AltstepInstance ::= AltstepRef "("
                         [FunctionActualParList] ")"
```

/ STATIC SEMANTICS - all timer instances in FunctionActualParList shall be declared as component local timers (see also production ComponentElementDef) */*

```
211. AltstepRef ::= [GlobalModuleId Dot] AltstepIdentifier
```

A.1.6.1.8 Import definitions

```
212. ImportDef ::= ImportKeyword ImportFromSpec (AllWithExcepts
                   | ("{" ImportSpec "}"))
213. ImportKeyword ::= "import"
214. AllWithExcepts ::= AllKeyword [ExceptsDef]
215. ExceptsDef ::= ExceptKeyword "{" ExceptSpec "}"
216. ExceptKeyword ::= "except"
217. ExceptSpec ::= {ExceptElement [SemiColon]}
```

/ STATIC SEMANTICS: Any of the production components (ExceptGroupSpec, ExceptTypeDefSpec etc.) may be present only once in the ExceptSpec production */*

```
218. ExceptElement ::= ExceptGroupSpec |
                       ExceptTypeDefSpec |
                       ExceptTemplateSpec |
                       ExceptConstSpec |
                       ExceptTestcaseSpec |
                       ExceptAltstepSpec |
                       ExceptFunctionSpec |
                       ExceptSignatureSpec |
                       ExceptModuleParSpec
219. ExceptGroupSpec ::= GroupKeyword
                         (ExceptGroupRefList
                         | AllKeyword)
220. ExceptTypeDefSpec ::= TypeDefKeyword
                           (TypeRefList
                           | AllKeyword)
221. ExceptTemplateSpec ::= TemplateKeyword
                            (TemplateRefList
                            | AllKeyword)
222. ExceptConstSpec ::= ConstKeyword
                         (ConstRefList
                         | AllKeyword)
223. ExceptTestcaseSpec ::= TestcaseKeyword
                            (TestcaseRefList
                            | AllKeyword)
224. ExceptAltstepSpec ::= AltstepKeyword
                           (AltstepRefList
                           | AllKeyword)
225. ExceptFunctionSpec ::= FunctionKeyword
```

```
                            (FunctionRefList
                            | AllKeyword)
226. ExceptSignatureSpec ::= SignatureKeyword
                             (SignatureRefList
                             | AllKeyword)
227. ExceptModuleParSpec ::= ModuleParKeyword
                             (ModuleParRefList
                             | AllKeyword)
228. ImportSpec ::= {ImportElement [SemiColon]}
229. ImportElement ::= ImportGroupSpec |
                       ImportTypeDefSpec |
                       ImportTemplateSpec |
                       ImportConstSpec |
                       ImportTestcaseSpec |
                       ImportAltstepSpec |
                       ImportFunctionSpec |
                       ImportSignatureSpec |
                       ImportModuleParSpec
230. ImportFromSpec ::= FromKeyword ModuleId [RecursiveKeyword]
```

/ NOTE: The use of RecursiveKeyword is deprecated*/*

```
231. RecursiveKeyword ::= "recursive"
232. ImportGroupSpec ::= GroupKeyword (GroupRefListWithExcept
                         | AllGroupsWithExcept)
233. GroupRefList ::= FullGroupIdentifier {"," FullGroupIdentifier}
234. GroupRefListWithExcept ::= FullGroupIdentifierWithExcept
                                {"," FullGroupIdentifierWithExcept}
235. AllGroupsWithExcept ::= AllKeyword [ExceptKeyword GroupRefList]
236. FullGroupIdentifier ::= GroupIdentifier {Dot GroupIdentifier}
237. FullGroupIdentifierWithExcept ::= FullGroupIdentifier [ExceptsDef]
238. ExceptGroupRefList ::= ExceptFullGroupIdentifier
                            {"," ExceptFullGroupIdentifier}
239. ExceptFullGroupIdentifier ::= FullGroupIdentifier
240. ImportTypeDefSpec ::= TypeDefKeyword
                           (TypeRefList
                           | AllTypesWithExcept)
241. TypeRefList ::= TypeDefIdentifier {"," TypeDefIdentifier}
242. AllTypesWithExcept ::= AllKeyword [ExceptKeyword TypeRefList]
243. TypeDefIdentifier ::= StructTypeIdentifier |
                           EnumTypeIdentifier |
                           PortTypeIdentifier |
                           ComponentTypeIdentifier |
                           SubTypeIdentifier
244. ImportTemplateSpec ::= TemplateKeyword (TemplateRefList
                            | AllTemplsWithExcept)
245. TemplateRefList ::= TemplateIdentifier {"," TemplateIdentifier}
246. AllTemplsWithExcept ::= AllKeyword [ExceptKeyword
                             TemplateRefList]
```

```
247. ImportConstSpec ::= ConstKeyword (ConstRefList
                         | AllConstsWithExcept)
248. ConstRefList ::= ConstIdentifier {"," ConstIdentifier}
249. AllConstsWithExcept ::= AllKeyword [ExceptKeyword ConstRefList]
250. ImportAltstepSpec ::= AltstepKeyword (AltstepRefList
                           | AllAltstepsWithExcept)
251. AltstepRefList ::= AltstepIdentifier {"," AltstepIdentifier}
252. AllAltstepsWithExcept ::= AllKeyword [ExceptKeyword
                               AltstepRefList]
253. ImportTestcaseSpec ::= TestcaseKeyword
                            (TestcaseRefList
                            | AllTestcasesWithExcept)
254. TestcaseRefList ::= TestcaseIdentifier {"," TestcaseIdentifier}
255. AllTestcasesWithExcept ::= AllKeyword [ExceptKeyword
                                TestcaseRefList]
256. ImportFunctionSpec ::= FunctionKeyword (FunctionRefList
                          | AllFunctionsWithExcept)
257. FunctionRefList ::= FunctionIdentifier {"," FunctionIdentifier}
258. AllFunctionsWithExcept ::= AllKeyword [ExceptKeyword
                                FunctionRefList]
259. ImportSignatureSpec ::= SignatureKeyword (SignatureRefList
                             | AllSignaturesWithExcept)
260. SignatureRefList ::= SignatureIdentifier {","
                          SignatureIdentifier}
261. AllSignaturesWithExcept ::= AllKeyword [ExceptKeyword
                                 SignatureRefList]
262. ImportModuleParSpec ::= ModuleParKeyword (ModuleParRefList
                             | AllModuleParWithExcept)
263. ModuleParRefList ::= ModuleParIdentifier {","
                          ModuleParIdentifier}
264. AllModuleParWithExcept ::= AllKeyword [ExceptKeyword
                                ModuleParRefList]
```

A.1.6.1.9 Group definitions

```
265. GroupDef ::= GroupKeyword GroupIdentifier
                  "{" [ModuleDefinitionsPart] "}"
266. GroupKeyword ::= "group"
267. GroupIdentifier ::= Identifier
```

A.1.6.1.10 External function definitions

```
268. ExtFunctionDef ::= ExtKeyword FunctionKeyword
                        ExtFunctionIdentifier
                        "("[FunctionFormalParList] ")"
                        [ReturnType]
269. ExtKeyword ::= "external"
270. ExtFunctionIdentifier ::= Identifier
```

A.1.6.1.11 External constant definitions

```
271. ExtConstDef ::= ExtKeyword ConstKeyword
                     Type ExtConstIdentifier
```

/ STATIC SEMANTICS - Type shall follow the rules given in clause 9 of ES 201 873-1.*/*

```
272. ExtConstIdentifier ::= Identifier
```

A.1.6.1.12 Module parameter definitions

```
273. ModuleParDef ::= ModuleParKeyword ( ModulePar |
                      ("{" MultitypedModuleParList "}"))
274. ModuleParKeyword ::= "modulepar"
275. MultitypedModuleParList ::= { ModulePar SemiColon }+
276. ModulePar ::= ModuleParType ModuleParList
```

/ STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated type for the Parameter */*

```
277. ModuleParType ::= Type
```

/ STATIC SEMANTICS - Type shall not be of component, default or anytype. Type shall only resolve to address type if a definition for the address type is defined within the module */*

```
278. ModuleParList ::= ModuleParIdentifier [AssignmentChar
                       ConstantExpression] {","ModuleParIdentifier
                       [AssignmentChar ConstantExpression]}
279. ModuleParIdentifier ::= Identifier
```

A.1.6.2 Control part

A.1.6.2.0 General

```
280. ModuleControlPart ::= ControlKeyword
                           "{" ModuleControlBody "}"
                           [WithStatement] [SemiColon]
281. ControlKeyword ::= "control"
282. ModuleControlBody ::= [ControlStatementOrDefList]
283. ControlStatementOrDefList ::= {ControlStatementOrDef
                                    [SemiColon]}+
284. ControlStatementOrDef ::= FunctionLocalDef |
                               FunctionLocalInst |
                               ControlStatement
285. ControlStatement ::= TimerStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SUTStatements |
                          StopKeyword
```

/ STATIC SEMANTICS - Restrictions on use of statements in the control part are given in table 11 */*

A.1.6.2.1 Variable instantiation

```
286. VarInstance ::= VarKeyword ((Type VarList)
                     | (TemplateKeyword Type TempVarList))
287. VarList ::= SingleVarInstance {"," SingleVarInstance}
288. SingleVarInstance ::= VarIdentifier [ArrayDef]
                           [AssignmentChar VarInitialValue]
289. VarInitialValue ::= Expression
290. VarKeyword ::= "var"
291. VarIdentifier ::= Identifier
292. TempVarList ::= SingleTempVarInstance {","
                     SingleTempVarInstance}
293. SingleTempVarInstance ::= VarIdentifier [ArrayDef]
                               [AssignmentChar TempVarInitialValue]
294. TempVarInitialValue ::= TemplateBody
295. VariableRef ::= (VarIdentifier | ValueParIdentifier)
                     [ExtendedFieldReference]
```

A.1.6.2.2 Timer instantiation

```
296. TimerInstance ::= TimerKeyword TimerList
297. TimerList ::= SingleTimerInstance {"," SingleTimerInstance}
298. SingleTimerInstance ::= TimerIdentifier [ArrayDef]
                             [AssignmentChar TimerValue]
299. TimerKeyword ::= "timer"
300. TimerIdentifier ::= Identifier
301. TimerValue ::= Expression
```

/ STATIC SEMANTICS - When Expression resolves to SingleExpression it must resolve to a value of type float. Expression shall only resolves to CompoundExpression in the initialization in default timer value assignment for timer arrays */*

```
302. TimerRef ::= (TimerIdentifier | TimerParIdentifier)
                  {ArrayOrBitRef}
```

A.1.6.2.3 Component operations

```
303. ConfigurationStatements ::= ConnectStatement |
                                 MapStatement |
                                 DisconnectStatement |
                                 UnmapStatement |
                                 DoneStatement |
                                 KilledStatement |
                                 StartTCStatement |
                                 StopTCStatement |
                                 KillTCStatement
304. ConfigurationOps ::= CreateOp | SelfOp | SystemOp |
                          MTCOp | RunningOp | AliveOp
305. CreateOp ::= ComponentType Dot CreateKeyword
```

```
                ["(" SingleExpession ")"] [AliveKeyword]
```

/ STATIC SEMANTICS - Restrictions on SingleExpession see in clause 22.1 */*

```
306. SystemOp ::= SystemKeyword
307. SelfOp ::= "self"
308. MTCOp ::= MTCKeyword
309. DoneStatement ::= ComponentId Dot DoneKeyword
310. KilledStatement ::= ComponentId Dot KilledKeyword
311. ComponentId ::= ComponentOrDefaultReference |
                     (AnyKeyword | AllKeyword) ComponentKeyword
312. DoneKeyword ::= "done"
313. KilledKeyword ::= "killed"
314. RunningOp ::= ComponentId Dot RunningKeyword
315. RunningKeyword ::= "running"
316. AliveOp ::= ComponentId Dot AliveKeyword
317. CreateKeyword ::= "create"
318. AliveKeyword ::= "alive"
319. ConnectStatement ::= ConnectKeyword SingleConnectionSpec
320. ConnectKeyword ::= "connect"
321. SingleConnectionSpec ::= "(" PortRef "," PortRef ")"
322. PortRef ::= ComponentRef Colon Port
323. ComponentRef ::= ComponentOrDefaultReference | SystemOp |
                      SelfOp | MTCOp
324. DisconnectStatement ::= DisconnectKeyword
                             [SingleOrMultiConnectionSpec]
325. SingleOrMultiConnectionSpec ::= SingleConnectionSpec |
                                    AllConnectionsSpec |
                                    AllPortsSpec |
                                    AllCompsAllPortsSpec]
326. AllConnectionsSpec ::= "(" PortRef ")"
327. AllPortsSpec ::= "(" ComponentRef ":" AllKeyword PortKeyword ")"
328. AllCompsAllPortsSpec ::= "(" AllKeyword ComponentKeyword ":"
                              AllKeyword PortKeyword ")"
329. DisconnectKeyword ::= "disconnect"
330. MapStatement ::= MapKeyword SingleConnectionSpec
331. MapKeyword ::= "map"
332. UnmapStatement ::= UnmapKeyword [SingleOrMultiConnectionSpec]
333. UnmapKeyword ::= "unmap"
334. StartTCStatement ::= ComponentOrDefaultReference Dot
                          StartKeyword "(" FunctionInstance ")"
```

/ STATIC SEMANTICS the Function instance may only have in parameters */ /* STATIC SEMANTICS the Function instance shall not have timer parameters */*

```
335. StartKeyword ::= "start"
336. StopTCStatement ::= StopKeyword | (ComponentReferenceOrLiteral
                         Dot StopKeyword) |
                         (AllKeyword ComponentKeyword Dot StopKeyword)
```

```
337. ComponentReferenceOrLiteral ::= ComponentOrDefaultReference
                                     | MTCOp | SelfOp
338. KillTCStatement ::= KillKeyword | (ComponentIdentifierOrLiteral
                         Dot KillKeyword) |
                         (AllKeyword ComponentKeyword Dot KillKeyword)
339. ComponentOrDefaultReference ::= VariableRef | FunctionInstance
```

/ STATIC SEMANTICS - The variable associated with VariableRef or the return type associated with FunctionInstance must be of component type when used in configuration statements and shall be of default type when used in the deactivate statement. */*

```
340. KillKeyword ::= "kill"
```

A.1.6.2.4 Port operations

```
341. Port ::= (PortIdentifier | PortParIdentifier) {ArrayOrBitRef}
342. CommunicationStatements ::= SendStatement |
                                 CallStatement |
                                 ReplyStatement |
                                 RaiseStatement |
                                 ReceiveStatement |
                                 TriggerStatement |
                                 GetCallStatement |
                                 GetReplyStatement |
                                 CatchStatement |
                                 CheckStatement |
                                 ClearStatement |
                                 StartStatement |
                                 StopStatement
343. SendStatement ::= Port Dot PortSendOp
344. PortSendOp ::= SendOpKeyword "(" SendParameter ")" [ToClause]
345. SendOpKeyword ::= "send"
346. SendParameter ::= TemplateInstance
347. ToClause ::= ToKeyword AddressRef |
                  AddressRefList |
                  AllKeyword ComponentKeyword
```

/ STATIC SEMANTICS - AddressRef should not contain matching mechanisms */*

```
348. AddressRefList ::= "(" AddressRef {"," AddressRef} ")"
349. ToKeyword ::= "to"
350. AddressRef ::= TemplateInstance
```

/ STATIC SEMANTICS - TemplateInstance must be of address or component type */*

```
351. CallStatement ::= Port Dot PortCallOp [PortCallBody]
352. PortCallOp ::= CallOpKeyword "(" CallParameters ")" [ToClause]
353. CallOpKeyword ::= "call"
354. CallParameters ::= TemplateInstance ["," CallTimerValue]
```

/ STATIC SEMANTICS only out parameters may be omitted or specified with a matching attribute */*

```
355. CallTimerValue ::= TimerValue | NowaitKeyword
```

/ STATIC SEMANTICS Value must be of type float */*

```
356. NowaitKeyword ::= "nowait"
357. PortCallBody ::= "{" CallBodyStatementList "}"
358. CallBodyStatementList ::= {CallBodyStatement [SemiColon]}+
359. CallBodyStatement ::= CallBodyGuard StatementBlock
360. CallBodyGuard ::= AltGuardChar CallBodyOps
361. CallBodyOps ::= GetReplyStatement | CatchStatement
362. ReplyStatement ::= Port Dot PortReplyOp
363. PortReplyOp ::= ReplyKeyword "(" TemplateInstance
                     [ReplyValue]")" [ToClause]
364. ReplyKeyword ::= "reply"
365. ReplyValue ::= ValueKeyword Expression
366. RaiseStatement ::= Port Dot PortRaiseOp
367. PortRaiseOp ::= RaiseKeyword "(" Signature ","
                     TemplateInstance ")" [ToClause]
368. RaiseKeyword ::= "raise"
369. ReceiveStatement ::= PortOrAny Dot PortReceiveOp
370. PortOrAny ::= Port | AnyKeyword PortKeyword
371. PortReceiveOp ::= ReceiveOpKeyword ["(" ReceiveParameter ")"]
                       [FromClause] [PortRedirect]
```

/ STATIC SEMANTICS: the PortRedirect option may only be present if the ReceiveParameter option is also present */*

```
372. ReceiveOpKeyword ::= "receive"
373. ReceiveParameter ::= TemplateInstance
374. FromClause ::= FromKeyword AddressRef
375. FromKeyword ::= "from"
376. PortRedirect ::= PortRedirectSymbol (ValueSpec [SenderSpec]
                      | SenderSpec)
377. PortRedirectSymbol ::= "->"
378. ValueSpec ::= ValueKeyword VariableRef
379. ValueKeyword ::= "value"
380. SenderSpec ::= SenderKeyword VariableRef
```

/ STATIC SEMANTIC Variable ref must be of address or component type */*

```
381. SenderKeyword ::= "sender"
382. TriggerStatement ::= PortOrAny Dot PortTriggerOp
383. PortTriggerOp ::= TriggerOpKeyword ["(" ReceiveParameter ")"]
                       [FromClause] [PortRedirect]
```

/ STATIC SEMANTICS: the PortRedirect option may only be present if the ReceiveParameter option is also present */*

```
384. TriggerOpKeyword ::= "trigger"
385. GetCallStatement ::= PortOrAny Dot PortGetCallOp
386. PortGetCallOp ::= GetCallOpKeyword ["(" ReceiveParameter ")"]
                       [FromClause]
                       [PortRedirectWithParam]
```

/ STATIC SEMANTICS: the PortRedirectWithParam option may only be present if the ReceiveParameter option is also present */*

```
387. GetCallOpKeyword ::= "getcall"
388. PortRedirectWithParam ::= PortRedirectSymbol
                               RedirectWithParamSpec
389. RedirectWithParamSpec ::= ParamSpec [SenderSpec] |
                               SenderSpec
390. ParamSpec ::= ParamKeyword ParamAssignmentList
391. ParamKeyword ::= "param"
392. ParamAssignmentList ::= "(" (AssignmentList | VariableList) ")"
393. AssignmentList ::= VariableAssignment {"," VariableAssignment}
394. VariableAssignment ::= VariableRef AssignmentChar
                            ParameterIdentifier
```

/ STATIC SEMANTICS: the parameterIdentifiers must be from the corresponding signature definition */*

```
395. ParameterIdentifier ::= ValueParIdentifier
396. VariableList ::= VariableEntry {"," VariableEntry}
397. VariableEntry ::= VariableRef | NotUsedSymbol
398. GetReplyStatement ::= PortOrAny Dot PortGetReplyOp
399. PortGetReplyOp ::= GetReplyOpKeyword ["(" ReceiveParameter
                       [ValueMatchSpec] ")"]
                       [FromClause]
                       [PortRedirectWithValueAndParam]
```

/ STATIC SEMANTICS: the PortRedirectWithParam option may only be present if the ReceiveParameter option is also present */*

```
400. PortRedirectWithValueAndParam ::= PortRedirectSymbol
                                       RedirectWithValueAndParamSpec
401. RedirectWithValueAndParamSpec ::= ValueSpec [ParamSpec]
                                       [SenderSpec] |
                                       RedirectWithParamSpec
402. GetReplyOpKeyword ::= "getreply"
403. ValueMatchSpec ::= ValueKeyword TemplateInstance
404. CheckStatement ::= PortOrAny Dot PortCheckOp
405. PortCheckOp ::= CheckOpKeyword ["(" CheckParameter ")"]
406. CheckOpKeyword ::= "check"
407. CheckParameter ::= CheckPortOpsPresent
                       | FromClausePresent
                       | RedirectPresent
```

```
408. FromClausePresent ::= FromClause
                           [PortRedirectSymbol
                           SenderSpec]
409. RedirectPresent ::= PortRedirectSymbol SenderSpec
410. CheckPortOpsPresent ::= PortReceiveOp | PortGetCallOp
                             | PortGetReplyOp | PortCatchOp
411. CatchStatement ::= PortOrAny Dot PortCatchOp
412. PortCatchOp ::= CatchOpKeyword ["("CatchOpParameter ")"]
                     [FromClause] [PortRedirect]
```

/ STATIC SEMANTICS: the PortRedirect option may only be present if the CatchOpParameter option is also present */*

```
413. CatchOpKeyword ::= "catch"
414. CatchOpParameter ::= Signature "," TemplateInstance
                          | TimeoutKeyword
415. ClearStatement ::= PortOrAll Dot PortClearOp
416. PortOrAll ::= Port | AllKeyword PortKeyword
417. PortClearOp ::= ClearOpKeyword
418. ClearOpKeyword ::= "clear"
419. StartStatement ::= PortOrAll Dot PortStartOp
420. PortStartOp ::= StartKeyword
421. StopStatement ::= PortOrAll Dot PortStopOp
422. PortStopOp ::= StopKeyword
423. StopKeyword ::= "stop"
424. AnyKeyword ::= "any"
```

A.1.6.2.5 Timer operations

```
425. TimerStatements ::= StartTimerStatement | StopTimerStatement
                         | TimeoutStatement
426. TimerOps ::= ReadTimerOp | RunningTimerOp
427. StartTimerStatement ::= TimerRef Dot StartKeyword
                             ["(" TimerValue ")"]
428. StopTimerStatement ::= TimerRefOrAll Dot StopKeyword
429. TimerRefOrAll ::= TimerRef | AllKeyword TimerKeyword
430. ReadTimerOp ::= TimerRef Dot ReadKeyword
431. ReadKeyword ::= "read"
432. RunningTimerOp ::= TimerRefOrAny Dot RunningKeyword
433. TimeoutStatement ::= TimerRefOrAny Dot TimeoutKeyword
434. TimerRefOrAny ::= TimerRef | AnyKeyword TimerKeyword
435. TimeoutKeyword ::= "timeout"
```

A.1.6.3 Type

```
436. Type ::= PredefinedType | ReferencedType
437. PredefinedType ::= BitStringKeyword |
                        BooleanKeyword |
                        CharStringKeyword |
```

```
                        UniversalCharString |
                        IntegerKeyword |
                        OctetStringKeyword |
                        HexStringKeyword |
                        VerdictTypeKeyword |
                        FloatKeyword |
                        AddressKeyword |
                        DefaultKeyword |
                        AnyTypeKeyword
438. BitStringKeyword ::= "bitstring"
439. BooleanKeyword ::= "boolean"
440. IntegerKeyword ::= "integer"
441. OctetStringKeyword ::= "octetstring"
442. HexStringKeyword ::= "hexstring"
443. VerdictTypeKeyword ::= "verdicttype"
444. FloatKeyword ::= "float"
445. AddressKeyword ::= "address"
446. DefaultKeyword ::= "default"
447. AnyTypeKeyword ::= "anytype"
448. CharStringKeyword ::= "charstring"
449. UniversalCharString ::= UniversalKeyword CharStringKeyword
450. UniversalKeyword ::= "universal"
451. ReferencedType ::= [GlobalModuleId Dot] TypeReference
                       [ExtendedFieldReference]
452. TypeReference ::= StructTypeIdentifier[TypeActualParList] |
                      EnumTypeIdentifier |
                      SubTypeIdentifier |
                      ComponentTypeIdentifier
453. TypeActualParList ::= "(" TypeActualPar {"," TypeActualPar} ")"
454. TypeActualPar ::= ConstantExpression
455. ArrayDef ::= {"[" ArrayBounds [".." ArrayBounds] "]"}+
456. ArrayBounds ::= SingleConstExpression
```

/ STATIC SEMANTICS - ArrayBounds will resolve to a non negative value of integer type */*

A.1.6.4 Value

```
457. Value ::= PredefinedValue | ReferencedValue
458. PredefinedValue ::= BitStringValue |
                         BooleanValue |
                         CharStringValue |
                         IntegerValue |
                         OctetStringValue |
                         HexStringValue |
                         VerdictTypeValue |
                         EnumeratedValue |
                         FloatValue |
                         AddressValue |
```

```
                        OmitValue
459. BitStringValue ::= Bstring
460. BooleanValue ::= "true" | "false"
461. IntegerValue ::= Number
462. OctetStringValue ::= Ostring
463. HexStringValue ::= Hstring
464. VerdictTypeValue ::= "pass" | "fail" | "inconc" |
                          "none" | "error"
465. EnumeratedValue ::= EnumerationIdentifier
466. CharStringValue ::= Cstring | Quadruple
467. Quadruple ::= CharKeyword "(" Group "," Plane ","
                   Row "," Cell ")"
468. CharKeyword ::= "char"
469. Group ::= Number
470. Plane ::= Number
471. Row ::= Number
472. Cell ::= Number
473. FloatValue ::= FloatDotNotation | FloatENotation
474. FloatDotNotation ::= Number Dot DecimalNumber
475. FloatENotation ::= Number [Dot DecimalNumber] Exponential
                        [Minus] Number
476. Exponential ::= "E"
477. ReferencedValue ::= ValueReference [ExtendedFieldReference]
478. ValueReference ::= [GlobalModuleId Dot] (ConstIdentifier |
                        ExtConstIdentifier |
                        ModuleParIdentifier ) |
                        ValueParIdentifier |
                        VarIdentifier
479. Number ::= (NonZeroNum {Num}) | "0"
480. NonZeroNum ::= "1" | "2" | "3" | "4" | "5" |
                    "6" | "7" | "8" | "9"
481. DecimalNumber ::= {Num}+
482. Num ::= "0" | NonZeroNum
483. Bstring ::= "'" {Bin} "'" ""
484. Bin ::= "0" | "1"
485. Hstring ::= "'" {Hex} "'" ""
486. Hex ::= Num | "A" | "" | "C" | "D" | "E" | "F"|
                 "a" | "" | "c" | "d" | "e" | "f"
487. Ostring ::= "'" {Oct} "'" "O"
488. Oct ::= Hex Hex
489. Cstring ::= """ {Char} """
490. Char ::= /* REFERENCE - A character defined by the relevant
CharacterString type. For harstring a character from the character
set defined in ISO/IEC 646. For universal charstring a character
from any character set defined in ISO/IEC 10646 */

491. Identifier ::= Alpha{AlphaNum | Underscore}
492. Alpha ::= UpperAlpha | LowerAlpha
```

```
493. AlphaNum ::= Alpha | Num
494. UpperAlpha ::= "A" | "" | "C" | "D" | "E" | "F" | "G" |
                    "" | "I" | "J" | "K" | "L" | "M" | "N" |
                    "O" | "P" | "Q" | "R" | "S" | "" | "U" |
                    "V" | "W" | "X" | "Y" | "Z"
495. LowerAlpha ::= "a" | "" | "c" | "d" | "e" | "f" | "g" |
                    "" | "i" | "j" | "k" | "l" | "m" | "n" |
                    "o" | "p" | "q" | "r" | "s" | "" | "u" |
                    "v" | "w" | "x" | "y" | "z"
496. ExtendedAlphaNum ::= /* REFERENCE - A graphical character from
the BASIC LATIN or from the LATIN-1 SUPPLEMENT character sets defined
in ISO/IEC 10646 (characters from char (0,0,0,32) to char (0,0,0,126),
from char (0,0,0,161) to char (0,0,0,172) and from char (0,0,0,174)
to char (0,0,0,255)*/

497. FreeText ::= """ {ExtendedAlphaNum} """
498. AddressValue ::= "null"
499. OmitValue ::= OmitKeyword
500. OmitKeyword ::= "omit"
```

A.1.6.5 Parameterization

```
501. InParKeyword ::= "in"
502. OutParKeyword ::= "out"
503. InOutParKeyword ::= "inout"
504. FormalValuePar ::= [(InParKeyword | InOutParKeyword |
                         OutParKeyword)] Type ValueParIdentifier
505. ValueParIdentifier ::= Identifier
506. FormalPortPar ::= [InOutParKeyword] PortTypeIdentifier
                       PortParIdentifier
507. PortParIdentifier ::= Identifier
508. FormalTimerPar ::= [InOutParKeyword] TimerKeyword
                        TimerParIdentifier
509. TimerParIdentifier ::= Identifier
510. FormalTemplatePar ::= [(InParKeyword | OutParKeyword |
                            InOutParKeyword )] TemplateKeyword
                            Type TemplateParIdentifier
511. TemplateParIdentifier ::= Identifier
```

A.1.6.6 With statement

```
512. WithStatement ::= WithKeyword WithAttribList
513. WithKeyword ::= "with"
514. WithAttribList ::= "{" MultiWithAttrib "}"
515. MultiWithAttrib ::= {SingleWithAttrib [SemiColon]}
516. SingleWithAttrib ::= AttribKeyword [OverrideKeyword]
                          [AttribQualifier] AttribSpec
517. AttribKeyword ::= EncodeKeyword |
                       VariantKeyword |
```

```
                    DisplayKeyword |
                    ExtensionKeyword
518. EncodeKeyword ::= "encode"
519. VariantKeyword ::= "variant"
520. DisplayKeyword ::= "display"
521. ExtensionKeyword ::= "extension"
522. OverrideKeyword ::= "override"
523. AttribQualifier ::= "(" DefOrFieldRefList ")"
524. DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}
525. DefOrFieldRef ::= DefinitionRef | FieldReference | AllRef
```

/ STATIC SEMANTICS: the DefOrFieldRef must refer to a definition or field which is within the module, group or definition to which the with statement is associated */*

```
526. DefinitionRef ::= StructTypeIdentifier |
                       EnumTypeIdentifier |
                       PortTypeIdentifier |
                       ComponentTypeIdentifier |
                       SubTypeIdentifier |
                       ConstIdentifier |
                       TemplateIdentifier |
                       AltstepIdentifier |
                       TestcaseIdentifier |
                       FunctionIdentifier |
                       SignatureIdentifier |
                       VarIdentifier |
                       TimerIdentifier |
                       PortIdentifier |
                       ModuleParIdentifier |
                       FullGroupIdentifier
527. AllRef ::= ( GroupKeyword AllKeyword [ExceptKeyword "{"
                GroupRefList "}"]) |
                ( TypeDefKeyword AllKeyword [ExceptKeyword "{"
                TypeRefList "}"]) |
                ( TemplateKeyword AllKeyword [ExceptKeyword "{"
                TemplateRefList "}"]) |
                ( ConstKeyword AllKeyword [ExceptKeyword "{"
                ConstRefList "}"]) |
                ( AltstepKeyword AllKeyword [ExceptKeyword "{"
                AltstepRefList "}"]) |
                ( TestcaseKeyword AllKeyword [ExceptKeyword "{"
                TestcaseRefList "}"]) |
                ( FunctionKeyword AllKeyword [ExceptKeyword "{"
                FunctionRefList "}"]) |
                ( SignatureKeyword AllKeyword [ExceptKeyword "{"
                SignatureRefList "}"]) |
                ( ModuleParKeyword AllKeyword [ExceptKeyword "{"
                ModuleParRefList "}"])
528. AttribSpec ::= FreeText
```

A.1.6.7 Behaviour statements

```
529. BehaviourStatements ::= TestcaseInstance |
                             FunctionInstance |
                             ReturnStatement |
                             AltConstruct |
                             InterleavedConstruct |
                             LabelStatement |
                             GotoStatement |
                             RepeatStatement |
                             DeactivateStatement |
                             AltstepInstance |
                             ActivateOp
```

/ STATIC SEMANTICS: TestcaseInstance shall not be called from within an existing executing testcase or function chain called from a testcase i.e. testcases can only be instantiated from the control part or from functions directly called from the control part */ /* STATIC SEMANTICS - ActivateOp shall not be called from within the module control part */*

```
530. VerdictStatements ::= SetLocalVerdict
531. VerdictOps ::= GetLocalVerdict
532. SetLocalVerdict ::= SetVerdictKeyword "(" SingleExpression ")"
```

/ STATIC SEMANTICS -SingleExpression must resolve to a value of type verdict */ /* STATIC SEMANTICS - the SetLocalVerdict shall not be used to assign the Value error */*

```
533. SetVerdictKeyword ::= "setverdict"
534. GetLocalVerdict ::= "getverdict"
535. SUTStatements ::= ActionKeyword "(" [ActionText ]
                       {StringOp ActionText} ")"
536. ActionKeyword ::= "action"
537. ActionText ::= FreeText | Expression
```

*/*STATIC SEMANTICS - Expression shall have the base type charstring or universal charstring */*

```
538. ReturnStatement ::= ReturnKeyword [Expression]
539. AltConstruct ::= AltKeyword "{" AltGuardList "}"
540. AltKeyword ::= "alt"
541. AltGuardList ::= {GuardStatement | ElseStatement [SemiColon]}
542. GuardStatement ::= AltGuardChar (AltstepInstance
                        [StatementBlock] | GuardOp StatementBlock)
543. ElseStatement ::= "["ElseKeyword "]" StatementBlock
544. AltGuardChar ::= "[" [BooleanExpression] "]"
```

*/*STATIC SEMANTICS - BooleanExpression shall conform to restrictions in clause 20.1.2 of ES 201 873-1*/*

```
545. GuardOp ::= TimeoutStatement |
                 ReceiveStatement |
                 TriggerStatement |
                 GetCallStatement |
                 CatchStatement |
                 CheckStatement |
                 GetReplyStatement |
                 DoneStatement |
                 KilledStatement
```

/ STATIC SEMANTICS - GuardOp used within the module control part shall only contain the timeoutStatement */*

```
546. InterleavedConstruct ::= InterleavedKeyword "{"
                              InterleavedGuardList "}"
547. InterleavedKeyword ::= "interleave"
548. InterleavedGuardList ::= {InterleavedGuardElement
                              [SemiColon]}+
549. InterleavedGuardElement ::= InterleavedGuard
                                 InterleavedAction
550. InterleavedGuard ::= "[" "]" GuardOp
551. InterleavedAction ::= StatementBlock
```

/ STATIC SEMANTICS - the StatementBlock may not contain loop statements, goto, activate, deactivate, stop, return or calls to functions */*

```
552. LabelStatement ::= LabelKeyword LabelIdentifier
553. LabelKeyword ::= "label"
554. LabelIdentifier ::= Identifier
555. GotoStatement ::= GotoKeyword LabelIdentifier
556. GotoKeyword ::= "goto"
557. RepeatStatement ::= "repeat"
558. ActivateOp ::= ActivateKeyword "(" AltstepInstance ")"
559. ActivateKeyword ::= "activate"
560. DeactivateStatement ::= DeactivateKeyword ["("
                             ComponentOrDefaultReference ")"]
561. DeactivateKeyword ::= "deactivate"
```

A.1.6.8 Basic statements

```
562. BasicStatements ::= Assignment | LogStatement |
                         LoopConstruct | ConditionalConstruct |
                         SelectCaseConstruct
563. Expression ::= SingleExpression | CompoundExpression
```

/ STATIC SEMANTICS - Expression shall not contain Configuration, activate operation or verdict operations within the module control part */*

```
564. CompoundExpression ::= FieldExpressionList | ArrayExpression
```

/ STATIC SEMANTICS - Within CompoundExpression the ArrayExpression can be used for Arrays, record, record of and set of types. */*

```
565. FieldExpressionList ::= "{" FieldExpressionSpec {","
                             FieldExpressionSpec} "}"
566. FieldExpressionSpec ::= FieldReference AssignmentChar
                             NotUsedOrExpression
567. ArrayExpression ::= "{" [ArrayElementExpressionList] "}"
568. ArrayElementExpressionList ::= NotUsedOrExpression {","
                                    NotUsedOrExpression}
569. NotUsedOrExpression ::= Expression | NotUsedSymbol
570. ConstantExpression ::= SingleConstExpression |
                            CompoundConstExpression
571. SingleConstExpression ::= SingleExpression
```

/ STATIC SEMANTICS - SingleConstExpression shall not contain Variables or Module parameters and shall resolve to a constant Value at compile time */*

```
572. BooleanExpression ::= SingleExpression
```

/ STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean */*

```
573. CompoundConstExpression ::= FieldConstExpressionList |
                                 ArrayConstExpression
```

/ STATIC SEMANTICS - Within CompoundConstExpression the ArrayConstExpression can be used for Arrays, record, record of and set of types. */*

```
574. FieldConstExpressionList ::= "{" FieldConstExpressionSpec
                                  {","
                                  FieldConstExpressionSpec}
                                  "}"
575. FieldConstExpressionSpec ::= FieldReference AssignmentChar
ConstantExpression
576. ArrayConstExpression ::= "{"
                              [ArrayElementConstExpressionList]
                              "}"
577. ArrayElementConstExpressionList ::= ConstantExpression
                                         {","
                                         ConstantExpression}
578. Assignment ::= VariableRef AssignmentChar
                    (Expression |
                    TemplateBody)
```

/ STATIC SEMANTICS - The Expression on the right hand side of Assignment shall evaluate to an explicit Value of a type compatible with the type of theleft hand side for value variables and shall evaluate to an explicit Value, template (literal or a template instance) or a matching mechanism compatible with the type of the left hand side for template variables. */*

```
579. SingleExpression ::= XorExpression { "or" XorExpression }
```

/ STATIC SEMANTICS - If more than one XorExpression exists, then the XorExpressions shall evaluate to specific values of compatible types */*

```
580. XorExpression ::= AndExpression { "xor" AndExpression }
```

/ STATIC SEMANTICS - If more than one AndExpression exists, then the AndExpressions shall evaluate to specific values of compatible types */*

```
581. AndExpression ::= NotExpression { "and" NotExpression }
```

/ STATIC SEMANTICS - If more than one NotExpression exists, then the NotExpressions shall evaluate to specific values of compatible types */*

```
582. NotExpression ::= [ "not" ] EqualExpression
```

/ STATIC SEMANTICS - Operands of the not operator shall be of type boolean (TTCN or ASN.1) or derivatives of type Boolean. */*

```
583. EqualExpression ::= RelExpression { EqualOp RelExpression }
```

/ STATIC SEMANTICS - If more than one RelExpression exists, then the RelExpressions shall evaluate to specific values of compatible types */*

```
584. RelExpression ::= ShiftExpression [ RelOp ShiftExpression ]
```

/ STATIC SEMANTICS - If both ShiftExpressions exist, then each ShiftExpression shall evaluate to a specific integer, Enumerated or float Value (these values can either be TTCN or ASN.1 values) or derivatives of these types */*

```
585. ShiftExpression ::= BitOrExpression { ShiftOp
                         BitOrExpression }
```

/ STATIC SEMANTICS - Each Result shall resolve to a specific Value. If more than one Result exists the right-hand operand shall be of type integer or derivatives and if the shift op is '«' or '»' then the left-hand operand shall resolve to either bitstring, hexstring or octetstring type or derivatives of these types. If the shift op is '<@' or '@>' then the left-hand operand shall be of type bitstring, hexstring, charstring or universal charstring or derivatives of these types */*

```
586. BitOrExpression ::= BitXorExpression { "or4b"
                         BitXorExpression }
```

/ STATIC SEMANTICS - If more than one BitXorExpression exists, then the BitXorExpressions shall evaluate to specific values of compatible types */*

```
587. BitXorExpression ::= BitAndExpression { "xor4b"
                          BitAndExpression }
```

/ STATIC SEMANTICS - If more than one BitAndExpression exists, then the BitAndExpressions shall evaluate to specific values of compatible types */*

```
588. BitAndExpression ::= BitNotExpression { "and4b"
                          BitNotExpression }
```

/ STATIC SEMANTICS - If more than one BitNotExpression exists, then the BitNotExpressions shall evaluate to specific values of compatible types */*

```
589. BitNotExpression ::= [ "not4b" ] AddExpression
```

/ STATIC SEMANTICS - If the not4b operator exists, the operand shall be of type bitstring, octetstring or hexstring or derivatives of these types. */*

```
590. AddExpression ::= MulExpression { AddOp MulExpression }
```

/ STATIC SEMANTICS - Each MulExpression shall resolve to a specific Value. If more than one MulExpression exists and the AddOp resolves to StringOp then the MulExpressions shall resolve to same type which shall be of bitstring, hexstring, octetstring, charstring or universal charstring or derivatives of these types. If more than one MulExpression exists and the AddOp does not resolve to StringOp then the MulExpression shall both resolve to type integer or float or derivatives of these types.*/*

```
591. MulExpression ::= UnaryExpression { MultiplyOp
                       UnaryExpression}
```

/ STATIC SEMANTICS - Each UnaryExpression shall resolve to a specific Value. If more than one UnaryExpression exists then the UnaryExpressions shall resolve to type integer or float or derivatives of these types. */*

```
592. UnaryExpression ::= [ UnaryOp ] Primary
```

/ STATIC SEMANTICS - Primary shall resolve to a specific Value of type integer or float or derivatives of these types.*/*

```
593. Primary ::= OpCall | Value | "(" SingleExpression ")"
594. ExtendedFieldReference ::= {(Dot ( StructFieldIdentifier
                                | TypeDefIdentifier))
                                | ArrayOrBitRef }+
```

/ STATIC SEMANTIC - The TypeDefIdentifier shall be used only if the type of the VarInstance or ReferencedValue in wich the ExtendedFieldReference is used is anytype.*/*

```
595. OpCall ::= ConfigurationOps |
                VerdictOps |
                TimerOps |
                TestcaseInstance |
                FunctionInstance |
                TemplateOps |
                ActivateOp
596. AddOp ::= "+" | "-" | StringOp
```

/ STATIC SEMANTICS - Operands of the "+ör öperators shall be of type integer or float or derivations of integer or float (i.e. subrange) */*

```
597. MultiplyOp ::= "*" | "/" | "mod" | "rem"
```

/ STATIC SEMANTICS - Operands of the "*", "/", rem or mod operators shall be of type integer or float or derivations of integer or float (i.e. subrange). */*

```
598. UnaryOp ::= "+" | "-"
```

/ STATIC SEMANTICS - Operands of the "+ör öperators shall be of type integer or float or derivations of integer or float (i.e. subrange). */*

```
599. RelOp ::= "<" | ">" | ">=" | "<="
```

/ STATIC SEMANTICS - the precedence of the operators is defined in Table 7 */*

```
600. EqualOp ::= "==" | "!="
601. StringOp ::= "&"
```

/ STATIC SEMANTICS - Operands of the string operator shall be bitstring, hexstring, octetstring or character string */*

```
602. ShiftOp ::= "<<" | ">>" | "<@" | "@>"
603. LogStatement ::= LogKeyword "(" LogItem { ","
                      LogItem } ")"
604. LogKeyword ::= "log"
605. LogItem ::= FreeText | TemplateInstance
606. LoopConstruct ::= ForStatement |
                       WhileStatement |
                       DoWhileStatement
607. ForStatement ::= ForKeyword "(" Initial SemiColon
                      Final SemiColon Step ")"
                      StatementBlock
608. ForKeyword ::= "for"
609. Initial ::= VarInstance | Assignment
610. Final ::= BooleanExpression
611. Step ::= Assignment
612. WhileStatement ::= WhileKeyword "(" BooleanExpression ")"
                        StatementBlock
613. WhileKeyword ::= "while"
614. DoWhileStatement ::= DoKeyword StatementBlock
                          WhileKeyword "(" BooleanExpression ")"
615. DoKeyword ::= "do"
616. ConditionalConstruct ::= IfKeyword "(" BooleanExpression ")"
                              StatementBlock
                              {ElseIfClause}[ElseClause]
617. IfKeyword ::= "if"
618. ElseIfClause ::= ElseKeyword IfKeyword "("
                      BooleanExpression ")" StatementBlock
619. ElseKeyword ::= "else"
620. ElseClause ::= ElseKeyword StatementBlock
621. SelectCaseConstruct ::= SelectKeyword "(" SingleExpression ")"
```

```
                              SelectCaseBody
622. SelectKeyword ::= "select"
623. SelectCaseBody ::= "{" { SelectCase }+ "}"
624. SelectCase ::= CaseKeyword ( '(' TemplateInstance {","
                    TemplateInstance } ')' | ElseKeyword)
                    StatementBlock
625. CaseKeyword ::= "case"
```

A.1.6.9 Miscellaneous productions

```
626. Dot ::= "."
627. Dash ::= "-"
628. Minus ::= Dash
629. SemiColon ::= ";"
630. Colon ::= ":"
631. Underscore ::= "_"
632. AssignmentChar ::= ":="
```

Literaturverzeichnis

[1] Eclipse. http://www.eclipse.org/, 18. Mai 2006. Homepage.

[2] Eclipse. http://www.eclipse.org/swt/, 18. Mai 2006. Homepage.

[3] European Telecomunication Standards Institute. http://www.etsi.org/, 11. Januar 2006. Homepage.

[4] Java ABC Framework. http://jabc.cs.uni-dortmund.de/, 15. Mai 2006. Homepage.

[5] Omondo. http://www.omondo.com/, 09. Mai 2006. Homepage.

[6] Test & Testing Control Notation. http://www.ttcn3.org/, 11. Januar 2006. Homepage.

[7] Testing Tech. http://www.testingtech.com/, 20. Mai 2006. Homepage.

[8] Lehrstuhl 5, Universität Dortmund. http://ls5-www.cs.uni-dortmund.de/, Februar 2006.

[9] NOKIA. http://www.nokia.com, Februar 2006.

[10] Uta Dienst. Konzeption und prototypische Implementierung eines Werkzeuges für den funktionalen Klassentest. Master's thesis, Universität Ulm, Fakultät für Informatik, 2004.

[11] E. W. Dijkstra. Notes on structured programming. In *Structured Programming*, pages 1–81. Academic Press, New York, 1972.

[12] Scott Hudson. CUP Parser Generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[13] European Telecommunications Standards Institute. Methods for testing and Specification (MTS); Part 5: TTCN-3 Runtime Interface (TRI). Technical report, European Telecommunications Standards Institute, 06 2005.

[14] European Telecommunications Standards Institute. Methods for testing and Specification (MTS); Part 1: TTCN-3 Core Language. Technical report, European Telecommunications Standards Institute, 06.2005.

[15] Stephen C. Johnson. Yet Another Compiler-Compiler. http://dinosaur.compilertools.net.

[16] Gerwin Klein. JFlex - The Fast Scanner Generator for Java. http://jflex.de/index.html.

[17] O. Niese, A. Hagerer, T. Margaria, M. Nagelmann, G. Brune H.-D.Ide, and B. Steffen. *An automated testing environment for cti systems using concepts forspecification and verification of workflows*. 2000.

[18] Terence Parr. ANother Tool for Language Recognition. http://www.antlr.org/.

[19] A. Hagerer H. Hungar T. Margaria O. Niese B. Steffen and H.-D.Ide. *Demonstration of an operational procedure for the model-based testingof cti systems*. April 2002.

[20] B. Steffen and T. Margaria. *Metaframe in practice: Design of intelligent network services*. Oktober 1999.

[21] Wikipedia. Bluetooth. http://de.wikipedia.org/wiki/Bluetooth.

[22] Wikipedia. Mooresches Gesetz. http://de.wikipedia.org/wiki/Mooresches_Gesetz, 11. Januar 2006. Wikipedia article.

[23] Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. John Wiley & Sons, Ltd, 2005.

[24] Jens Zech. Ttcn-3 Testing and Test Control Notation. http://swt.cs.tu-berlin.de/lehre/seminar/ss03/ausarbeitungen/ttcn-3_ausarbeitung.pdf, Jan. 2002.

[25] Wei Zhao. Entwicklung eines Parsers für TTCN-3 Version 3 unter Verwendung des Parsergenerators ANTLR. http://www.swe.informatik.uni-goettingen.de/publications/WZ/gaug-zfi-bm-2005-15.pdf, 08.