Fachgebiet Programmierung eingebetteter Systeme
Fakultät IV - Elektrotechnik und Informatik
Technische Universität Berlin

PES

SysCIR: Eine zielsprachenunabhängige Zwischenrepräsentation für SystemC/TLM

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science Informatik

Florian Greiner

20.11.2012

Betreuer:

Prof. Dr. Sabine Glesner
Dipl.-Inform. Marcel Pockrandt
Dr.-Ing. Paula Herber

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 20.11.2012

_____

Inhaltsverzeichnis

1 Einleitung

Vom Institut PES an der Technischen Universität Berlin wurde ein Ansatz zur automatisierten Verifikation von Hardware/Software Systemen entwickelt. Kernidee ist dabei eine Transformation von SystemC-Modellen nach UPPAAL-Timed-Automata. Allerdings wird eine Intermediate Representation verwendet, die sehr UPPAAL nah ist. Hierdurch wird die Verwendung anderer Verifikationswerkzeuge, beispielsweise UCLID oder NuSMV erschwert. Außerdem sind bei diesem Ansatz Optimierungsschritte die unabhängig von der Zielsprache sind, auf der Zwischenrepräsentation schwer durchzuführen.

1.1 Zielstellung

Im Rahmen der Bachelorarbeit soll eine zielsprachenunabhängige Zwischenrepräsentation definiert werden. Anschließend soll eine Transformation erstellt werden, die diese Zwischenrepräsentation automatisch aus einem SystemC-Modell generiert.
Die Struktur des ursprünglichen SystemC-Modells soll dabei erhalten bleiben. Beispielsweise muss die Repräsentation eines SystemC-Moduls Ports, Sockets, globale Variablen, Funktionen, Expressions enthalten können. Außerdem muss es in der Lage sein, Hierarchien abzubilden. Ebenfalls muss die Kommunikation der SystemC-Modelle korrekt abgebildet werden.

1.2 Motivation

In vielen Bereichen unserer heutigen Gesellschaft kommen vielfach eingebettete Systeme zum Einsatz. Darunter befinden sich auch viele sicherheitskritische Einsatzgebiete, beispielsweise Airbag-Systeme in Autos oder die Steuerung von Kraftwerken. Hier kommt es darauf an, ein fehlerfreies Funktionieren zu gewährleisten. Nur durch Testen und Simulieren ist es jedoch nicht möglich, die Fehlerfreiheit und das reibungslose Zusammenspiel der einzelnen Komponenten zu gewährleisten. Testen ist per Definition unvollständig, da es kaum möglich ist alle möglichen Testfälle abzudecken. Die Komplexität wird zusätzlich durch die Konzepte der Nebenläufigkeit, Parallelität und Synchronisation erhöht, die in eingebetteten Systemen Verwendung finden. Es können also durch Testen und Simulieren nicht alle Fehler entdeckt werden. Man benötigt also formale Verifikationstechniken wie das Modelchecking, die den gesamten Zustandsraum erfassen.
Mit SystemC to Timed Automata Transformation Engine, kurz STATE [HFG08] [HPG11] [PHG11] wurde ein Ansatz vorgestellt, welcher SystemC-Modelle in UPPAAL-Timed-Automata-Modelle [BLL+95] [BDL04] umwandelt. Diese Timed Automata können dann mithilfe des UPPAAL Modelcheckers auf die Einhaltung temporal-logischer Eigenschaften hin verifiziert werden.

UPPAAL ist ein Tool zur grafischen Bearbeitung von Timed Automata. Außerdem beinhaltet UPPAAL einen Modelchecker um die Timed Automata auf die Einhaltung von CTL-Formeln hin zu verifizieren. Die Stärken von UPPAAL liegen in der Verifikation von Kommunikaitonsprotokollen und Softwareprozessen. Die Hardwareverifikation hingegen ist sehr ressourcenaufwendig. Es gibt andere Techniken, die die Funktionalität von UPPAAL gut ergänzen würden.
STATE verwendet allerdings eine UPPAAL-nahe Intermediate Representation, wodurch die Transformation in Eingabesprachem für andere Verifikationswerkzeuge, beispielsweise UCLID oder NuSMV, erschwert wird. Eine zielsprachenunabhängige Zwischenrepräsentation vereinfacht eine entsprechende Transformation deutlich. Weiterhin ist es möglich, auf diese Weise Optimierungen zu entwickeln und umzusetzen, die direkt auf der Intermediate Representation arbeiten und dementsprechend für alle daraus generierten Modelle bzw. Zielsprachen anwendbar sind. Eine zielsprachenunabhängige Zwischenrepräsentation hat auch den Vorteil, dass bei Änderungen der SystemC-Syntax unter Umständen nicht jede Transformation angepasst werden muss. Wenn diese Änderungen auf bestehende Konstrukte abgebildet werden können, dann ist es ausreichend, die Zwischenrepräsentation anzupassen.

1.3 Verwandte Arbeiten

Es gibt bereits Arbeiten, die SystemC in formale Semantiken umwandeln. Zum Beispiel stellen Müller et al. [MRR03] und Ruf [RHG+01] eine Definition einer formalen Semantik vor, die auf abstrakten statemachines basiert. Dabei wird das SystemC-Design an sich nicht berücksichtigt, so existieren dort keine Module, Prozesse und Kanäle. Salem [Sal03] kann zumindest Prozesse darstellen hat jedoch nur für ein eingeschränktes subset. Ähnlich Große et al. [GKD06], er benutzt Binary Decision Diagrams (BDDs) und beschränktes modlechecking aber auch hier kann nur ein eingeschränktes subset verifiziert werden. In anderen Ansätzen, beispielsweise Habibi et al. [HT05], [HMT06] werden ebenfalls statemachines verwendet, doch im Gegensatz zu unserem Ansatz kann dabei keine Zeit simuliert werden.
Traulsen et al. [TCMM07] stellen ein Mapping von SystemC nach PROMELA vor, jedoch können sie das SystemC-Design nur auf einer recht abstrakten Ebene darstellen und unterstützen keine primitiven Kanäle. Zhang et al. [ZVM07] führt den Formalismus von SystemC waiting-state automata ein. Damit kann man SystemC-Modelle formal auf der Ebene der Deltazyklen darstellen. Allerdings muss das formale Modell manuell spezifiziert werden und die komplexe Interaktion zwischen Prozessen wurde nicht berücksichtigt.

1.4 Gliederung

Diese Arbeit ist wie folgt strukturiert:
In Kapitel 2 geben wir einen Überblick über SystemC. In Kapitel 3 stellen wir die von uns entwickelte Zwischenrepräsentation vor. Die Implementierung werden wir in Kapitel 4 erläutern. Im Kapitel 5 evaluieren wir unsere Zwischenrepräsentation. In Kapitel 6 fassen wir unsere Ergebnisse zusammen und präsentieren Erweiterungsmöglichkeiten.

2 Grundlagen

Eingebettete Systemen müssen anderen Aspekten genügen als reguläre Software. So müssen eingebettete Systeme echtzeitfähig sein, sowie in einem größeren Maße verlässlicher sein als reguläre Software. Das liegt daran, dass eingebettete Systeme häufig in sicherheitskritischen Bereichen eingesetzt werden. Im klassischen Entwurf wird der Hardware und Softwareanteil eines eingebetteten Systems getrennt konstruiert und entworfen. Dies führt zu Problemen bei der Zusammenführung der einzelnen Teile.
Im Hardware/Software-Codesign werden die Hardware- und Softwareanteile zusammen entwickelt und getestet und im Laufe der Entwicklung schrittweise verfeinert. Somit ist es auch möglich, durch Testen und Simulieren, Fehler frühzeitig zu erkennen. Dieses Konzept wurde mit SystemC [IEE05] umgesetzt.
SystemC ist eine Hardware/Software-Beschreibungssprache und besonders geeignet für die Entwicklung von komplexen eingebetteten Systemen. Dabei ist SystemC keine eigenständige Sprache. SystemC ist eine C++-Klassenbibliothek mit Simulationskernel.

Abbildung 1: Struktur der SystemC-Modelle

Die SystemC-Syntax kennt Module, die die eigentlichen Bausteine des Systems sind. In den Modulen stecken die Funktionalität und die Berechnung.
Wie in Abbildung 1 zu sehen, können Module untereinander mit Ports über ein sogenanntes Kanalobjekt verbunden werden. Über diese Verbindungen findet die Kommunikation in SystemC-Modellen statt. Neben Ports und Kanälen können auch Sockets und Interfaces eingesetzt werden um die Kommunikation zu simulieren. SystemC bietet dafür einige Standartobjekte und Interfaces an.
Die schon erwähnte Berechnung findet sowohl in Funktionen als auch in Prozessen statt. Die Funktionen lösen nicht auf Ereignisse aus, sonderen werden von Prozessen aufgerufen. Die Prozesse hingegen sind spezielle Funktionen. Bei der Deklaration eines Prozesses wird eine Funktion referenziert und eine Mege von Ereignissen. Die Prozesse werden ausgelöst wenn eines der entsprechenden Ereignisse stattfindet. Danach laufen sie, bis sie die Kontrolle an den Scheduler abgeben, beispielsweise durch ein „wait()“. Anschließend können sie durch Events wieder

fortgeführt werden, diese Event werden können durch ein „notify()“ ausgelöst werden. Beides, sowohl das wait() als auch das notify() kann mit einer Zeitangabe versehen werden. Damit können Zeitverbrauch und Wartezeit simmuliert werden. Die Abgabe der Kontrolle muss geschehen, da es sich um einen kooperativen Scheduler handelt.

Außerdem ist es erlaubt, Module in Modulen zu schachteln. Damit ist es in SystemC möglich, eine Hierarchien zu modellieren.
Weiterhin können - wie in C++ auch - Structs verwendet werden, also selbstdefinierte Datenstrukturen. Auch diese können Bestandteile von Modulen sein und definieren so zusammen mit anderen Variablen den Zustand des Moduls.
Ein weiterer Aspekt bei SystemC-Modellen ist das Zeitverhalten. SystemC verfügt über zeitbehaftete Objekte und Clocks, womit die Modellierung eines genauen Zeitverhaltens möglich ist. SystemC ist in der Lage exakte Zeitwerte anzugeben und somit Timeouts und Zeitspannen anzugeben, die genau eingehalten werden.
Die erwähnten Prozesse sind so definiert, dass sie nebenläufig arbeiten. Da dies jedoch in der Regel beim Entwurf nicht zu realisieren ist, wurde in SystemC ein Scheduler eingeführt. Dieser Scheduler ist in jedem SystemC-Modell implizit vorhanden und muss nicht extra implementiert werden. Er behandelt Ereignisse, führt Prozesse aus, organisiert die Reihenfolge der Ausführung und verwaltet und aktualisiert die Kanäle und damit die Kommunikation. Diese Vorgänge werden in sogenannten Deltazyklen abgearbeitet. Die schematische Darstellung dieser Deltazyklen ist in Abbildung 2 zu sehn.

Der Scheduler wechselt, nachdem er initiiert wurde, in den Ausführungszustand. Dabei werden aktive Prozesse ausgeführt, bis sie die Kontrolle abgebeben. Informationen werden auf die Kanäle geschrieben usw. Die Reihenfolge, in der die Prozesse ausgeführt werden, entscheidet der Scheduler deterministisch. In der ISO zu SystemC ist der Scheduler zwar als nichtdeterministisch definiert, in der Referenzimplementierung wurde jedoch ein deterministischer Scheduler angelegt.
Wenn kein Prozess mehr auszuführen ist, wechselt der Scheduler in die Update-Phase. In dieser Phase werden die globalen Modulvariablen aktualisiert, sofern es keine Konflikte gibt. Konflikte können entstehen, wenn zwei Prozesse, die nebenläufi laufen können, die selbe globale Variable geändert haben. In soeinem Fall bricht der Scheduler die Ausführung ab und überlässt es dem Entwickler derartige Fälle abzufange. Weiterhin fidnet die eigentliche Kommunikation findet statt, das heißt, die Kanäle, welche die Ports verbinden, werden aktualisiert, sofern es sich um primitive Kanäle handelt. Die Daten, die über Ports von Prozessen auf diese Kanäle geschrieben wurden, können nun ebenfalls über Ports von den Prozessen abgerufen werden. Nun wechselt der Scheduler wieder in die Ausführungsphase und führt die Prozesse aus, die nun ausführbar sind. Danach findet wieder eine

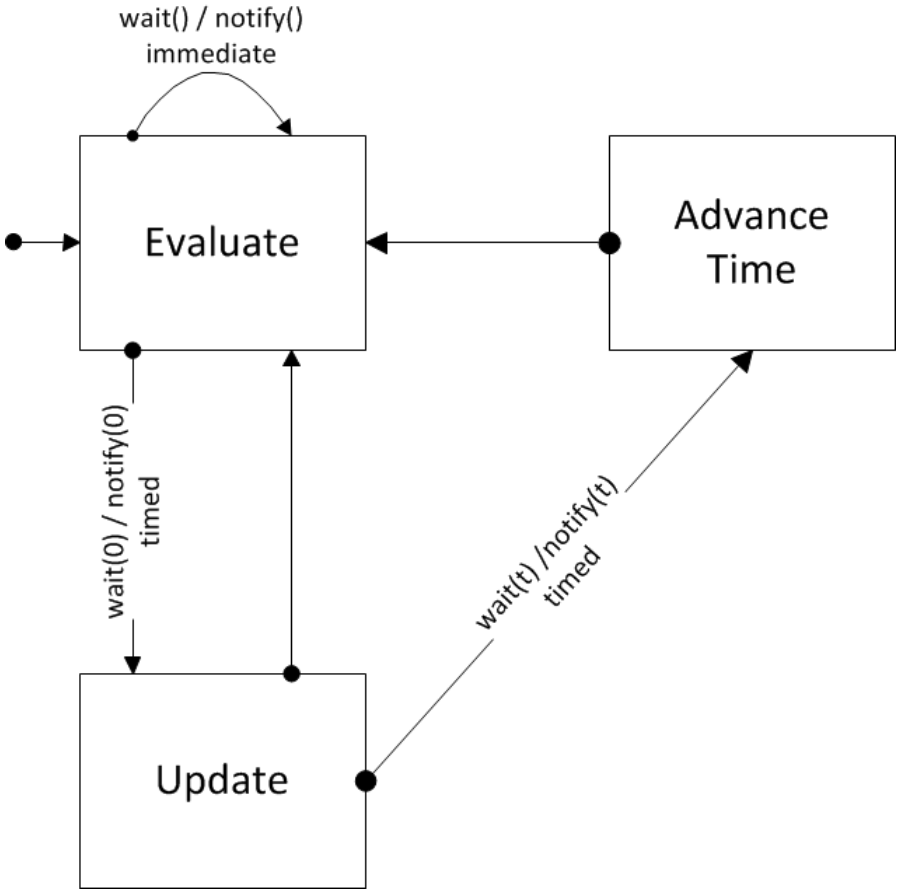


Abbildung 2: Schematische Darstellung der Deltazyklen

Update-Phase statt. Das wird solange fortgeführt, bis kein Prozess mehr ausführbar ist.
In der Evaluation und in der Updatephase wird keine Zeit verbraucht, das heißt, das eigentliche System befindet sich immer noch an dem Zeitpunkt, zu dem die erste Ausführungsphase gestartet wurde. Wenn jedoch kein Prozess mehr zur Ausführung bereit ist, weil alle auf irgendwelche Ereignisse warten oder darauf warten, dass Zeit vergeht, dann wird ein Zeitschritt durchgeführt. Nach diesem Zeitschritt wechselt der Scheduler wieder in die Ausführungsphase. Durch dieses Vorgehen wird in SystemC-Modellen Nebenläufigkeit simuliert.

3 Zwischenrepräsentation

Im Zuge dieser Arbeit soll eine zielsprachenunabhängige Zwischenrepräsentation für die Systembeschreibungssprache SystemC entstehen. Wie wir bereits im vorrangegangenen Kapitel erwähnt haben, ist SystemC eine Klassenbibliothek von C++. Das stellt eine besondere Herausforderung dar. SystemC besitzt wenig eigenen Konstrukte, jedoch müssen wir auch die große Anzahl an C++-Konstrukten abbilden können. Weiterhin ist besonders darauf zu achten, dass die Modulstruktur und die Kommunikation des SystemC-Modells korrekt in der Zwischenrepräsentation dargestellt werden. Im Grundlagenkapitel haben wir bereits erwähnt, dass die Kommunikation in SystemC zwischen Modulen über dedizierte Kommunikationsschnittstellen, Ports, Sockets, Kanäle und Interfaces, stattfindet. Dieses Konzept muss in unserer Zwischenrepräsentation ebenfalls dargestellt werden können.

3.1 Anforderungen und Ziele

An unsere Zwischenrepräsentation stellen wir eine Reihe von Bedingungen.
Wir wollen einen großen Teil aller SystemC-Modelle abbilden können. Um dies zu gewährleisten, muss die Zwischenrepräsentation alle wichtigen und häufig vorkommenden SystemC- und C++-Konstrukte abbilden können.
Zusätzlich sehen wir als eine weitere, notwendige Bedingung, dass die Zwischenrepräsentation leicht erweiterbar ist und somit auf einfachem Weg neue Konstrukte hinzugefügt werden können.
Weiterhin wollen wir, dass bereits vorhandene SystemC-Modelle automatisiert in die Zwischenrepräsentation übertragen werden können. Dazu soll der existierende Programmcode verwendet werden und als Eingabe für die Generierung der Zwischenrepräsentation dienen. Da dieser Programmcode an sich schwer zu verabreiten ist, soll er vorher in einen abstrakten Syntaxbaum umgewandelt werden.
Außerdem wollen wir die Struktur von SystemC-Modellen so exakt wie möglich nachbilden. Damit is auch gemeint, dass die Zwischenrepräsentation einen einfachen udn direkten Zugriff auf die Objekte, aus denen sie besteht, gewährleisten.
Wir wollen auch, dass innerhalb der Zwischenrepräsentation einzelne Konstrukte leicht durch andere ersetzt werden können. Dies ermöglicht es uns beispielsweise Schleifentypen ineinander zu überführen, was notwenig sein kann, da einige Verifikationswerkzeuge nicht alle Schleifen unterstützen.
Um derartige Anpassungen oder Optimierungen effizient durchzuführen, ist es notwendig, die Zwischenrepräsentation speichern zu können. Das müssen wir ebenfalls ermöglichen.

3.2 Annahmen

Unser Ziel ist, einen großen Sprachumfang von SystemC abzubilden und somit eine große Anzahl von SystemC-Modellen darstellen zu können. Um dies zu erreichen, treffen wir zwei Annahmen über das SystemC-Modell, das in die Zwischenrepräsentation abgebildet werden soll:

Die erste ist, dass das Eingabemodell syntaktisch korrekt ist. Damit meinen wir, dass es sich kompilieren lässt. Unsere Zwischenrepräsentation soll in Formate umgewandelt werden, die bei verschiedensten Verifikationswerkzeugen als Eingabe dienen können. Jedoch ist eine Verifikation von syntaktisch inkorrekten Modellen mit den meisten formalen Verifikationswerkzeugen ohnehin nicht möglich und nicht sinnvoll.
Die zweite Annahme, die wir treffen, ist das keine Namenskonflikte auftreten. Das heißt, alle Variablen und Typen sind überschattungsfrei. Im Scope einer Variable darf damit keine Variablen gleichen Namens existieren. Damit vereinfachen wir die Erstellung der Zwischenrepräsentation, da somit Variablen und Typen leichter zu identifizieren sind. Jedes SystemC-Modell kann durch entsprechende Umbenennungen von derartigen Namenskonflikten befreit werden, sodass diese Annahme nur eine geringe Einschränkung darstellt.

Alle Modelle, die diesen beiden Annahmen genügen und nur von uns unterstützte Konstrukte verwenden, lassen sich in unsere Zwischenrepräsentation abbilden.

3.3 Unterstützte Konstrukte

Grundsätzlich können wir Module, Structs, Ports, Sockets, Funktionen, Prozesse, Arrays, Pointer, payload-event-queue's, Events und Zeitobjekte in unserer Zwischenrepräsentation abbilden.
Weitere Objekte können unter Umständen ebenfalls abgebildet werden, sofern die Implementierung dieser Objekte sich durch ein Modul oder ein Struct beschreiben lässt. Dann kann diese Implementierung in einen abstrakten Syntaxbaum umgewandelt werden und als KnownType gekennzeichnet werden. Diese Datei wird dann bei der Transformation geladen, sofern sie benötigt wird. Wir haben dieses Konzept verwendet, um so die Kanalobjekte darzustellen.
Es ist darüber hinaus möglich, Hierarchien abzubilden und Vererbungen zu kennzeichnen.
Die in den Funktionen enthaltenen Algorithmen können aus Operationen, Zuweisungen, Adressierung von Objekten, Funktionsaufrufen, Schleifen, Verzweigungen und Steuerbefehlen bestehen.
Wir können arithmetische, logische und shift-Operationen abbilden, auch in Verbindung mit Zuweisungen. Adressierte Objekte können sein: Module, Structs, Ports

bzw. Sockets, Variablen, egal ob ein oder mehrdimensional. Verwendbare Steuerbefehle haben wir auch return, break und continue eingeschränkt.
Letztendlich sind wir in der Lage Konstrukte abzubilden, die in der Postfix-Notation vorliegen.

3.4 Konzepte

Eine zielsprachenunabhängige Zwischenrepräsentation muss bestimmte Kriterien erfüllen. Grundsätzlich dürfen bei der Umwandlung in die Zwischenrepräsentation keine Informationen des ursprünglichen Modelles verloren gehen. Die Zwischenrepräsentation muss in der Lage sein, aus der Quellsprache eines Modelles alle Objekte abbilden zu können, sowie alle Informationen zu diesen Objekten besitzen. Weiterhin muss sie einen einfachen, direkten und eindeutigen Zugriff auf diese Objekte anbieten, um diese direkt zu referenzieren. Sie muss in der Lage sein, die Funktionalität dieser Objekte bzw. des Modells darzustellen, um das Modell komplet verifizieren zu können. Insbesondere muss sie alle Kontrollstrukturen(If-Then-Else, Schleifen, u.ä.) darstellen können. Außerdem muss die Zwischenrepräsentation zielsprachenunabhängig sein, damit eine Überführung in alle Zielsprachen einfacher ist. Sie muss also allgemein aufgebaut sein und darf nicht speziell für ein Verifikationswerkzeug angepasst sein.

Diese Anforderungen und auch die in unserer Zielstellung müssen von uns umgesetzt werden. Wir haben uns bei unserer Zwischenrepräsentation auf die am häufigsten verwendeten SystemC-Konstrukte konzentriert, da dadurch die meisten SystemC-Modelle abbildbar sind. In unserer Zwischenrepräsentation bilden wir diese Konstrukte auf Objekte der Zwischenrepräsentation ab. Diese Objekte speichern außerdem die Vererbungshierarchie, also die Vaterobjekte, sowie die Modellhierarchie. Wir behalten dadurch auch die Struktur des SystemC-Modells bei. Damit haben wir die ersten beiden Anforderungen, die wir an die Zwischenrepräsentation stellen, erfüllt.

Da wir die Bestandteile von Objekten an den Objekten selbst speichern, bieten wir eine Möglichkeit für einen einfachen und direkten Zugriff. Dadurch ist es auch einfacher, später gewisse Konstrukte zu ersetzen.
Die Funktionalität von Methoden speichern wir in Form von Expressions. Somit haben wir eine große Freiheit in der Hinsicht, jeden beliebigen Teilausdruck abbilden zu können. Unsere Expressions können geschachtelt aufgebaut sein, d. h. eine Expression kann wiederum Expressions enthalten. Damit erreichen wir, dass jeder Teilausdruck durch andere ersetzt werden kann und somit einfach Änderungen und Optimierungen durchzuführen sind.

Des Weiteren haben wir eine Möglichkeit vorgesehen, die Zwischenrepräsentation zu speichern und zu einem späteren Zeitpunkt wieder zu laden. Damit können beispielsweise Optimierungsschritte, die auf der Zwischenrepräsentation ausgeführt werden, zwischengespeichert werden. Das ist sinnvoll, da damit nicht für jedes Verifikationswerkzeug eine neue Zwischenrepräsentation generiert werden muss und somit leicht sichergestellt werden kann, dass alle Verifikationswerkzeuge mit demselben Modell arbeiten. Außerdem können somit Optimierungen getestet werden und bei einem Misserfolg muss das Modell nicht neu erzeugt werden, sondern es kann dirket die Vorgängerversion geladen werden. Damit ist die Voraussetzung gegeben, die Zwischenrepräsentation später direkt in unterschiedliche Verifikationswerkzeuge einzubinden.

Module

Ein SystemC-Modell besteht in der obersten Ebene aus instanziierten Modulen und der Kommunikation zwischen diesen. In dem Objekt, welches in unserer Zwischenrepräsentation das System darstellt, speichern wir die Objekte, die die Module und die Kommunikationskanäle dazwischen repräsentieren. Wir stellen die Kommunikation auf verschiedene Arten dar, da sie in SystemC auch auf verschiedene Arten implementiert werden kann. Es gibt in SystemC Ports und Sockets. Während Ports nur über Kanäle miteinander verbunden werden können, müssen Sockets direkt miteinander verbunden werden. Diese beiden Verbindungsarten können wir speichern. Damit haben wir die Möglichkeit, jede Kommunikationsart zwischen den Modulen abzubilden, die SystemC vorsieht.

Prozesse

Die Module, die in einem SystemC-Modell instanziiert werden, können Konstrukte verschiedenster Arten enthalten. Darunter sind die bereits erwähnten Ports und Sockets. Weiterhin können Module Funktionen und Prozesse enthalten. Prozesse sind Funktionen, die auf ein oder mehrere Ereignisse hin ausgelöst werden. Um dies abzubilden, speichern wir an dem Prozessobjekt eine Referenz zu dem entsprechenden Funktionsobjekt. Außerdem enthält jedes Prozessobjekt eine Liste von Ereignissen, auf die es reagieren kann.

Funktionen

Eine Funktion besteht aus dem Funktionsrumpf und dem Funktionskopf. Der Kopf besteht aus dem Funktionsnamen, den Parametern und dem Rückgabetyp. Wir

speichern die Parameter und den Rückgabetyp. Außerdem müssen wir Referenzen der lokalen Variablen in dem Funktionsobjekt ablegen. Wir wollen eine eindeutige Referenzierung herstellen. Aus diesem Grund müssen wir zu jedem Konstrukt, das in einem Modell erzeugt wird, ein Objekt erstellen, welches dieses abbildet. Diese Objekte müssen wir uns dann merken, sodass wir später nur eine Referenz zu dem Objekt speichern, wenn beispielsweise in Funktionsrümpfen auf eines dieser Konstrukte referenziert wird. Dadurch, dass wir Referenzen speichern, können wir einen direkten und eindeutigen Zugriff gewährleisten. Zusammen mit unserer 2. Annahme, können wir so sicherstellen, dass ein Name immer auf dasselbe Objekt verweist.
Der Rumpf einer Funktion beinhaltet die Funktionalität. Diese Funktionalität speichern wir mittels Expressions. Dabei müssen wir beachten, dass hier die Reihenfolge von größter Wichtigkeit ist. Wir haben daher eine Datenstruktur gewählt, mit der wir diese speichern können. Von den Expressions haben wir verschiedene Typen angelegt, die die verschiedensten Konstrukte darstellen. In einigen dieser Expression-Objekte speichern wir eine Referenz auf ein Objekt, welches ein Konstrukt des SystemC-Modells abbildet. Auch damit erreichen wir wieder einen direkten Zugriff auf die SystemC-Konstrukte. Die meisten Expressions können wieder auf Expressions referenzieren. Auf diese Weise gelingt es uns, jede Kombination aus Expressions darzustellen. Somit setzen wir die Anforderung um, dass eine Zwischenrepräsentation alle möglichen Konstrukte aus der Quellsprache abdecken muss.

Datentypen

In SystemC-Modellen können Variablen der verschiedensten Typen auftreten. Viele davon sind einfache Datentypen, so wie Integer, andere sind komplexer. Die einfachen Datentypen bilden wir alle auf ein Objekt ab, von dem wir entsprechende Instanzen erstellen. Dort wird der Typ gespeichert, sodass diese Information nicht verloren geht.
Weiterhin gibt es in SystemC auch Arrays. Diese stellen wir ebenfalls in Objekten dar, wobei die eigentlichen Elemente des Arrays wieder Variablen sind. Auf diese Weise gelingt es uns auch, mehrdimensionale Arrays abzuspeichern.
Es ist in C++ und damit auch in SystemC möglich, Structs zu deklarieren. Das sind selbstdeklarierte Datenstrukturen. Wir bilden die eigentliche Implementierung dieses Struct in einem Objekt ab und referenzieren auf dieses in einer Struct-Instanz, die ebenfalls eine Variable ist.
Die Events die in SystemC existieren, müssen wir ebenfalls abbilden, da diese für die Prozesse und der Ausführung derselben wichtig ist. Allerdings können Prozesse nicht nur auf Events reagieren, sondern auch auf Änderungen bei Ports und Variablen. Um das abzubilden mussten wir weitere Objekte schaffen, die auch diese

Art von Events repräsentieren. Diese verweisen dann auf den Port bzw. Socket oder auf die Variable, die für das entsprechende Event relevant sind. Damit haben wir einen direkten Zugriff auf diese Objekte, die für die Ausführung der Prozesse relevant sind.
Wir müssen es auch möglich machen, während der transformation zusätzliche Objekte in die Zwischenrepräsentation einzufügen. Ein Beispiel hierfür sind die Kanalobjekte, deren Funktionsweise bzw. Definition in der Zwischenrepräsentation enthalten sein muss. Das erreichen wir, indem wir diese Objekte durch ein Modul oder ein Struct definieren. Dieses wird durch eine SystemC-Datei beschrieben, welche in einen abstrakten Syntaxbaum umgewandelt wurde. Wenn bei der Transformation ein entsprechender Datentyp gefunden wird, lesen wir diesen Syntaxbaum ein, erstellen daraus ein Modul beziehungsweise ein Struct und verweisen darauf.
Allgemein legen wir jedes Objekt nur einmal an. Das geschieht bei allen Objekten bei der Deklaration. Da wir, gemäß unseren Annahmen, keine Überschattung erlauben, können wir so sicherstellen, dass alle Objekte eindeutig referenziert werden können. Somit gelingt es uns auch, einen einfachen und eindeutigen Zugriff zu gewährleisten und somit dieses Ziel zu erreichen.

Der Scheduler

SystemC bietet zusätzlich die Möglichkeit für einen hierarchischen Entwurf, indem die Bestandteile von Modulen wieder Module sein können. Weiterhin können in dem Modul Prozesse enthalten sein. Diese sind für uns relevant, da die Zielsprache einen Scheduler generieren muss.
Wie schon im Grundlagenkapitel erwähnt, besitzt jedes SystemC-Modell implizit einen Scheduler. Die Zwischenrepräsentation soll nicht ausführbar sein. Demzufolge wird auch kein Scheduler in der Zwischenrepräsentation benötigt. Hinzu kommt, dass der Scheduler stark von der Zielsprache abhängig ist und demzufolge bei der Umwandlung der Repräsentation in die Zielsprache erstellt werden muss. Das müssen wir unterstützen.
In dem Scheduler werden außer den Prozessen, auch noch Modulvariablen sowie Kanäle verwaltet. Um die Generierung des Schedulers zu erleichtern, behandeln wir die Modulinstanzen innerhalb der Module nicht so wie die anderen Variablen, zu denen die Modulinstanzen eigentlich dazugehören, sondern speichern sie gesondert. Somit muss bei der Generierung des Schedulers nicht nach den Modulinstanzen gesucht werden, sondern es kann direkt auf sie zugegriffen werden. Es müssen also nicht alle Modulvariablen überprüft werden, ob eine Modulinstanz in ihnen enthalten ist. Ähnlich sind wir mit den Prozessen verfahren. Da Prozesse eigentlich nur gekapselte Funktionen sind, hätten wir dies an der entsprechenden Funktion kennzeichnen können. Da allerdings die Prozesse für die Generierung des Schedulers

benötigt werden, die Funktionen jedoch nicht, haben wir uns dafür entschieden, die Prozesse in einem eigenen Objekt abzubilden und nur auf die Funktion zu verweisen.
Durch diese beiden Entscheidungen decken wir unsere Bedingung ab, dass wir einen einfachen und direkten Zugriff auf die einzelnen Objekte gewähren müssen.

Im Zuge einer Umwandlung in ein Modell, das verifiziert werden soll, sind häufig Optimierungen nötig. Um für eine neue Optimierung nicht immer die gesamte Toolchain durchlaufen zu müssen, haben wir unsere Zwischenrepräsentation so gestaltet, dass jedes Konstrukt aus dem SystemC-Modell von einem einzelnen Objekt aus erreicht werden kann. Wir haben also die Hierachie von SystemC-Modellen abgebildet, welche dieses Konzept unterstützt. Dieses übergeordnete Objekt ist bei uns dasjenige, welches das Gesamtsystem abbildet. In diesem Objekt sind alle Modulinstanzen und globalen Objekte enthalten, wovon wiederum jedes Objekt Repräsentationen aller Konstrukte enthält, die sein Äquivalent im SystemC Modell besitzt. Dadurch ermöglichen wir das Speichern und Laden von Zwischenzuständen, da wir somit nur einen Einstiegspunkt haben, von dem aus wir alle Bestandteile des Objektes erreichen können. Das leichte Speichern und Laden, durch den wir unsere Anforderung, dass die Zwischenrepräsentation leicht zu speichern ist, umsetzen.
Die Struktur, die uns das Speichern und Laden erleichtert muss auch so gestaltet sein, dass die Zwischenrepräsentation leicht erweitert werden kann. Dies gelingt uns einerseits durch eine klar strukturierte Objekthierarchie und andererseits durch den Einsatz von abstrakten Objekten, die selbst kein SystemC-Konstrukt abbilden, aber als Zusammenfassung für Objekte dienen die ähnliche Konstrukte abbilden, beispielsweise die Expressions oder die Variablen. Auf weitere Konzepte die wir umgesetzt haben, um die Erweiterbarkeit, als eines unserer Ziele, zu unterstützen, gehen wir im Implementierungskapitel ein.

Wir haben in diesem Kapitel erläutert, wie unsere Zwischenrepräsentation aufgebaut ist und wie die an sie gestellten Bedingungen umgesetzt wurden. Wir sind dabei sowohl auf die Probleme der Darstellung eingegangen, als auch auf die Besonderheiten von SystemC, die ebenfalls abgebildet wurden. Auf die genauere Umsetzung der Zwischenrepräsentation gehen wir im nachfolgenden Implementierungskapitel ein.

4 Implementierung

Nachfolgend legen wir die Implementierung unserer Zwischenrepräsentation, SystemC Intermediate Representation, kurz SysCIR, dar. Dabei gehen wir besonders auf die Struktur unseres Klassendiagramms ein und zeigen außerdem, wie wir unsere in 3.1 präsentierten Anforderungen umgesetzt haben.

4.1 Java

Die Modularisierung von SystemC-Modellen ist mit einem objektorientierten Ansatz gut umzusetzen. Aus diesem Grund haben wir uns für Java entschieden. Außerdem ist Java plattformunabhängig. Somit ist die Zwischenrepräsentation universell einsetzbar. Ein weiterer Vorteil ist, dass Java Interfaces und Bibliotheken mitliefert, die uns die Transformation in die Zwischenrepräsentation und das Speichern derselben erleichtern beziehungsweise überhaupt erst ermöglichen. Zum Beispiel bietet Java einen XML-Parser für die Verarbeitung des Eingabeformates und das Serializable-Interface zum Speichern. Weiterhin gibt es mit Eclipse ein sehr komfortables Entwicklungswerkzeug für Java.

4.2 Ablauf der Transformation

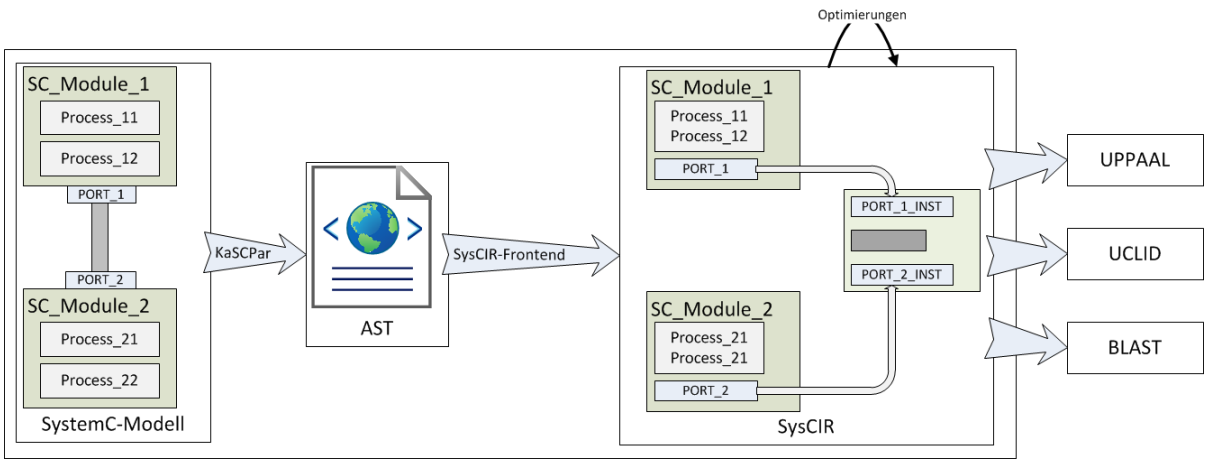


Abbildung 3: Toolchain

Als Eingabeformat haben wir uns für einen abstrakten Syntaxbaum entschieden. Das Forschungszentrum Informatik in Karlsruhe hat für die Umwandlung von SystemC-Modellen in einen abstrakten Syntaxbaum ein Werkzeug entwickelt. Dieses Werkzeug heißt KaSCPar [FZI]. Die Ausgabedatei wird von uns als Eingabe-

format verwendet und dann in die Zwischenrepräsentation überführt. Auf dieser können dann Optimierungen ausgeführt werden. Danach kann die Zwischenrepräsentation in die verschiedensten Modelle überführt werden, die dann verifiziert werden können. Dieser Ablauf ist auch in Abbildung 3 dargestellt.

4.3 Klassenstruktur der Zwischenrepräsentation

Wir haben, wie im Klassendiagramm (Anhang A) zu sehen, die SystemC-Konstrukte auf entsprechende Java-Klassen abgebildet. Auch die Struktur der SystemC-Konstrukte ist dort abgebildet. So enthält die SCSystem-Klasse alle Modulinstanzen, die in ihr deklariert werden, sowie die globalen Funktionen und restlichen Variablen. Außerdem sind auch die Bindungsinformationen zwischen den Modulen in dieser Klasse enthalten. Man sieht auch, dass Ports und Sockets instanziiert werden. Das muss geschehen, da die Ports bzw. Sockets in SystemC in Modulen deklariert werden, was auch aus dem Klassendiagramm ersichtlich ist. Da jedoch Module instanziiert werden, müssen auch ihre Bestandteile instanziiert werden. Für Variablen, Funktionen und Prozesse ist das in der Zwischenrepräsentation nicht notwendig. Die Ports bzw. Sockets werden jedoch für die Bindungsinformationen benötigt.
Wie im Klassendiagramm auch zu sehen, haben wir die Vererbungshierarchie der SystemC-Konstrukte teilweise nachgebildet. Das zeigen wir anhand des Beispiels der SCModule-Klasse. In SystemC erbt ihr Gegenstück von dem C++-Konstrukt class. In SysCIR hingegen haben wir Class und Struct in SCStruct gekapselt, da sie in bezug auf die Zwischernrepräsentation dasselbe sind. Somit haben wir die Vererbung dieses Konstruktes abgebildet.

Zur Kapselung unserer Zwischenrepräsentation haben wir das Konzept der Packages benutzt. Die Zwischenrepräsentation ist bei uns vollständig im Package sc_model enthalten. Wie in Abbildung im Klassendiagramm (Anang A) zu sehen, enthält dieses Package jedoch noch 2 weitere Packages, „expressions" und „variables", in denen wir die verschiedenen Expressions und Variablentypen zusammengefasst haben. In der obersten Ebene sind alle Konstrukte enthalten, die für die Struktur des SystemC-Modells notwendig sind. In den untergeordneten Packages sind dann einerseits die Expressions gekapselt, die zwar für das Model wichtig sind, jedoch nicht für die Struktur. Andererseits sind die Variablentypen in einem eigenen Package gekapselt. Sie sind zwar relevant für die Struktur, wurden aber zur Erhaltung der Übersicht gekapselt und werden in der oberen Ebene durch das Interface vertreten, das sie implementieren.

4.4 Verwendete Datenstrukturen

Sobald in einem Objekt, welches ein SystemC-Konstrukt repräsentiert, mehrere Objekte desselben Typs enthalten sein können, werden diese zusammen in einer Listenstruktur gespeichert. Ein Beispiel hierfür ist die Klasse SCStruct. In ihr ist eine Liste enthalten mit den Referenzen auf alle SCFunctions des jeweiligen Structs. Wir haben uns für eine Liste entschieden, da wir damit sowohl die Flexibilität hinsichtlich der Größe erhalten, als auch die Reihenfolge in der die entsprechenden Konstrukte im SystemC-Modell angelegt wurden. Bei Funktionen ist die Reihenfolge in der Regel egal. Bei einigen Verifikationswerkzeugen kann es jedoch notwendig sein, das Funktionen deklariert werden, bevor sie aufgerufen werden.
Bei Variablen oder den Funktionsrümpfen sieht das jedoch anders aus, dort ist die Reihenfolge immer relevant. Anstatt in diesen Situationen verschiedene Datenstrukturen zu nutzen, haben wir uns auf die Listenstruktur festgelegt, um die Weiterverwendung der Zwischenrepräsentation zu erleichtern. Nur bei der SCArray-Klasse sind wir davon abgewichen, da es uns sinnvoll erschien, Werte eines Arrays auch als Array zu speichern.

4.5 Erweiterbarkeit und Wiederverwendbarkeit

Wir wollen, dass die Zwischenrepräsentation erweiterbar ist. Um dies umzusetzen, haben wir die Konzepte der Objektorientierung verwendet. Beispielsweise verwenden wir abstrakte Klassen um die Variablen und Expressions zu gruppieren. Das erleichtert die Erweiterung, da nun bei einem neuen Konstrukt nur die dabei neuen Bestandteile implementiert werden müssen.
Ein weiteres Ziel für unsere Zwischenrepräsentation ist diese einfach zu speichern und zu laden. Wir haben das einerseits durch die Struktur der Zwischenrepräsentation sichergestellt - diesen Punkt haben wir bereits im vorangegangenen Kapitel erläutert. Anderseits implementieren unsere Klassen das Serializable-Interface. In der SCSystem-Klasse, die nach der Transformation alle Komponenten des SystemC-Modells enthält, haben wir zwei Funktionen für das Laden und Sichern implementiert. Da diese Klasse für jeden weiteren Schritt zur Verfügung stehen muss, haben wir damit auch gewährleistet, dass das System in jedem weiteren Schritt geladen und gesichert werden kann.

In diesem Kapitel haben wir einige Implementierungsdetails erläutert. Wir sind dabei auf die Struktur der Zwischenrepräsentation eingegangen, sowie die Maßnahmen die wir unternommen haben, um die Zwischenrepräsentation erweiterbar und speicherbar zu gestalten. wir haben die verwendeten Datenstrukturen erläutert und begründet, warum wir uns bei der Umsetzung für Java entschieden haben.

5 Evaluierung

Die Überprüfung der Korrektheit der Zwischenrepräsentation ist bei uns nur bedingt möglich. Lediglich durch Debugging kann überprüft werden, ob ein SystemC-Modell korrekt in die Zwischenrepräsentation überführt wurde. Wir haben die Zwischenrepräsentation mit den Beispielen getestet, die in STATE enthalten sind. Wir können die Funktionen und auch die Expressions in einen String umwandeln und können so nachweisen, dass die Funktionen vor und nach der Transformation gleich sind. Durch das Debugging-Werkzeug von Eclipse können wir außerdem zeigen, dass die Modellstruktur erhalten geblieben ist. Für einen Korrektheitsnachweis ist allerdings eine Überführung der Zwischenrepräsentation in eine Zielsprache notwendig.

Die getesteten Beispiele decken alle von uns unterstützen Konstrukte ab.
Am Anfang haben wir ein einfaches Beispiel getestet. Dieses bestand aus 2 Modulen, die über einen Kanal verbunden sind. Jedes dieser Module enthält 2 Ports, ein Prozess und eine weitere Funktion. Ein Modul generiert eine Zahl und schreibt diese über seinen Port auf einen Kanal. Das andere Modul liest über seinen Port diesen Kanal aus und erhält somit die generierte Zahl und gibt diese aus. Beide Module sind durch eine Clock getaktet. Die Prozesse reagieren also sensitiv auf eine Änderung der Taktflanke.
Mit diesem Beispiel haben wir die Grundlagen getestet. Wir haben überprüft, ob Module, Funktionen, Prozesse, Ports, sowie einige Variablen korrekt angelegt werden. Weiterhin konnten wir damit überprüfen, ob die Verbindungen von Ports und Kanälen korrekt gespeichert werden. An Programmkonstrukten konnten wir eine einfache Verzweigung, Listing 1, und eine while-Schleife testen.

```
1          a = ((wait_time == 100) ? 0 : 1);
```

Listing 1: Questionmarkexpression

Außerdem waren in dem Beispiel einige arithmetische Operationen, Zuweisungen, einfache Funktionsaufrufe sowie der Steuerbefehl „return" enthalten. Weiterhin konnten wir durch die Kanäle testen, ob das Einlesen von KnownType-Implementierungen funktioniert.

Das nächste Beispiel, das wir getestet haben, macht dasselbe wie das vorrangegangene. Allerdings wird hierbei kein SystemC-Kanalobjekt verwendet, sondern ein selbstdefinierte. Die Implementierung des selbstdefinierten Kanals ist im Beispiel enthalten. In dieser Implementierung ist sowohl das Konzept der Vererbung enthalten, als auch Arrays. Wir konnten damit überprüfen, ob die Referenzen auf die Vaterklassen korrekt gesetzt werden. Weiterhin konnten wir testen, ob Arrays korrekt angelegt werden und ob die Adressierung der Elemente korrekt funktio-

niert. Weiterhin konnten wir das Anlegen Events überprüfen und das Auslösen derselben.

```
1        w_event.notify();
```

Listing 2: Auslösen von Events

Wie in Listing 2 zu sehen, konnten wir damit auch die erste einfache PostFix-Expression testen.

In dem nächsten Beipielblock wurden Structs intensiv überprüft. Grundsätzlich haben die einzelnen Beispiele immer dieselbe Funktionsweise wie die vorrangegangenen, aber diesmal werden Structs verwendet. So gibt es ein Beispiel, dass die generierten Daten in einem Struct abgelegt werden und dann vor dem Schreiben auf den Kanal von dort abgerufen werden.
In dem weiteren Beispiel werden Structs verschachtelt, aber auch hier werden die Daten in Structs abgelegt und von dort geladen.
Das dritte Beispiel benutzt Structs mit Funktionen. Anstatt die Variablen der Structs direkt auszulesen, werden diese mit Funktionen abgefragt.
Das nächste Beispiel überträgt keine Integer-Daten, wie alle vorrangegangenen, sondern eine Variable, die eine Instanz eines Structs ist.
Prinzipiell wurden also verschiedene Verwendungszwecke von Structs getestet, auch Übergabe als Funktionsparameter oder als Rückgabewert. Ebenfalls wurde die Verwendung von Structs in Bedingungen von Schleifen oder Verzweigungen wurde mit diesem Beispielblock getestet.

Im nächsten Beispiel haben wir das switch-Konstrukt getestet. Dieses Beispiel besteht nur aus einem Modul, in diesem sind jedoch mehrere verschiedene switch-Konstrukte enthalten. Wir haben mit diesem Beispiel die verschiedenen Fälle getestet, die auftreten können, wie das Vorhandensein von dem Steuerbefehl „break" beziehungsweise das Fehlen desselben, mehrere verschiedene Cases und einem Default-case.

Ein weiterer Beispielblock wurde verwendet, um Pointer zu testen. Die Beispiele bestehen aus jeweils einem Modul, und teilweise mit einem zusätzlichen Struct. In diesem Modul werden dann verschiedenste Verwendungen von Pointern getestet, so zum Beispiel, neben Pointern, die auf normale Datentypen verweisen, auch Pointer die auf Structs oder andere komplexe Datenstrukturen zeigen. Auch Pointer in Structs, wobei diese auch verschachtelt auftraten, wurden getestet.

Unter den getesteten Beispielen, simulierte auch eines einen Prozessor. Dort wurde das Konzept des hierarchischen Entwurfs verwendet. Wir konnten somit also auch testen, ob Hierarchien in unserer Zwischenrepräsentation richtig abgebildet werden.

In dem letzten Beispielblock wurden intensiv Sockets verwendet. Die Beispiele an sich simulieren ein Bussystem. Neben Sockets und Multisockets wurden auch Konstrukte getestet, die bisher noch nicht überprüft wurden. Darunter zählen If-Then-Else-Konstrukte, Do-While-Schleifen, For-Schleifen sowie diverse komplexe PostFix-Ausdrücke. Außerdem konnten wir in diesen Beispielen auch die diversen Bindungsmöglichkeiten testen, die in SystemC benutzt werden, um Module über Ports bzw. Sockets zu verknüpfen und somit eine Kommunikation ermöglichen.

Wir haben also alle SystemC-Konstrukte, die wir unterstützen, getestet. Damit haben wir, mit unseren Mitteln, überprüft ob diese Konstrukte korrekt abgebildet werden. Für eine endgültige Verifikation müssen diese Beispiele jedoch noch in eine Zielsprache überführt werden und dort getestet werden.

6 Fazit und Ausblick

Im Zuge dieser Arbeit haben wir eine zielsprachenunabhängige Zwischenrepräsentation für SystemC erstellt. Diese Zwischenrepräsentation trägt den Namen SysCIR, SystemC Intermediate Representation. Diese Zwischenrepräsentation erfüllt die folgenden Anforderungen.

Sie ist in der Lage, die wichtigsten und häufigsten SystemC- und C++-Konstrukte abzubilden. Somit können die meisten SystemC-Modelle vollständig dargestellt werden. Das haben wir erreicht, indem wir die Konstrukte aus SystemC, die für die Struktur beziehungsweise die Hierarchie eines Modells wichtig sind, auf eigene Objekte in der Zwischenrepräsentation abbilden. Weiterhin bilden wir die Programmkonstrukte aus den Funktionsrümpfen in Expressions ab. Die Struktur der Expression erlaubt es, miteinander verknüpft zu werden. Damit können wir alle wesentlichen Codekonstrukte aus SystemC in der Zwischenrepräsentation abbilden. Darunter zählen wir Schleifen und Verzweigungen sowie arithmetische, logische und shift-Operationen ebenso wie Zuweisungen. Außerdem können wir die Adressierung von Variablen, einschließlich Arrays, Module, Structs, Ports und Sockets abbilden. Auch Funktionsaufrufe sind möglich. Für Steuerbefehle wie return, break und continue haben wir eine Möglichkeit geschaffen, diese darzustellen. Selbst Konstrukte in der Postfix-Notation können wir abbilden.

Wir haben alle relevanten Konstrukte aus SystemC in der Zwischenrepräsentation abgebildet. Zusätzlich haben wir unsere Zwischenrepräsentation erweiterbar gestaltet, damit neue SystemC-Konstrukte ebenfalls abgebildet werden können. Das haben wir erreicht, indem wir die Objekte der Zwischenrepräsentation klar strukturiert haben, sowie die Konzepte der Objektorientierung sinnvoll angewandt haben.
Einige der Objekte bilden kein Konstrukt aus SystemC ab, sondern fungieren als Gruppierung für solche. Die übergeordneten enthalten Member und Funktionen für die untergeordneten Objekte. Ein Beispiel hierfür ist die Expressionklasse. Diese enthält alle Member die für eine Expression notwendig sind. Die Klassen, die von dieser Klasse erben, enthalten dann nur noch die Member, die für die entsprechende Expression notwendig ist. Auf diese Weise können neue Expressions leicht hinzugefügt werden. Bei den Variablen sind wir ähnlich vorgegangen. Außerdem sind all unsere Objekte so aufgebaut, das Klassen die von diesen erben auf alle Member der Vaterklasse zugreifen können. Damit haben wir das Hinzufügen von neuen Klassen weiter erleichtert.

Die Zwischenrepräsentation kann automatisiert benutzt werden. Wir verwenden den bereits existierenden Programmcode, der geschrieben wurde, um das SystemC-

Modell zu beschreiben, um daraus die Zwischenrepräsentation für das Modell zu generieren. Wir verwenden jedoch nicht den reinen Programmcode, sonderen lassen ihn erst in einen abstrakten Syntaxbaum umwandeln. Für die Transformation dieses Baumes in die Zwischenrepräsentation haben wir Transformationsklassen entwickelt, die jede ein bestimmtes Konstrukt aus SystemC in die Zwischenrepräsentation überführt. Als Eingabe dient hier ein abstrakter Syntaxbaum, der mithilfe von KaSCPar aus den SystemC-Codedateien generiert wurde. Somit ist es nicht notwendig, ein SystemC-Modell per Hand in der Zwischenrepräsentation nachzubilden.
Diese Automatisierung war außerdem notwendig, da die Zwischenrepräsentation in eine Toolchain einbettbar sein muss. Aus dieser Bedingung folgt auch, dass die Zwischenrepräsentation speicherbar sein muss und dann auch wieder geladen werden kann. Das haben wir erreicht, indem wir einerseits dafür gesorgt haben, dass alle Bestandteile des SystemC-Modells in der Zwischenrepräsentation von einem Objekt aus zu erreichen sind. Wir haben also die Zwischenrepräsentation klar strukturiert und mit Verweisen auf die einzelnen Bestandteile gearbeitet. Andererseits haben wir auf der Programmebene das Java-Interface Serializable implementiert und in dem obersten Objekt eine load- und eine store-Methode implementiert. Damit können wir jedes Modell welches in die Zwischenrepräsentation überführt wurde, abspeichern und später wieder laden. Außerdem haben wir die Zwischenrepräsentation und den Parser getrennt, sodass die Werkzeuge, die die Zwischenrepräsentation weiterverwenden, nur die Pakete der Zwischenrepräsentation benötigen. Die Transformationsklassen sind nicht notwendig.

Wir wollen, dass die Objekte, die in Ihrer Gesamtheit das SystemC-Modell darstellen, einfach zu erreichen sind. Die für das Laden und Speichern benötigte Struktur erfüllt diese Bedingung. Dass auf die abgebildeten Konstrukte ein einfacher und direkter Zugriff existiert, erreichen wir einerseits durch die schon erwähnte Struktur und andererseits durch durch die Arbeit mit Referenzen. Jedes Objekt in SystemC legen wir nur ein einziges Mal an. Sollte in SystemC-Code dann auf das Konstrukt, das durch dieses Objekt abgebildet wird, referenziert werden, so suchen wir in unserer Zwischenrepräsentation das entsprechende Objekt und speichern die Referenz darauf.
Auf diese Weise gelingt es uns auch, die Struktur von SystemC-Modellen zu erhalten. Das war ebenfalls eine unserer Bedingungen an die Zwischenrepräsentation.

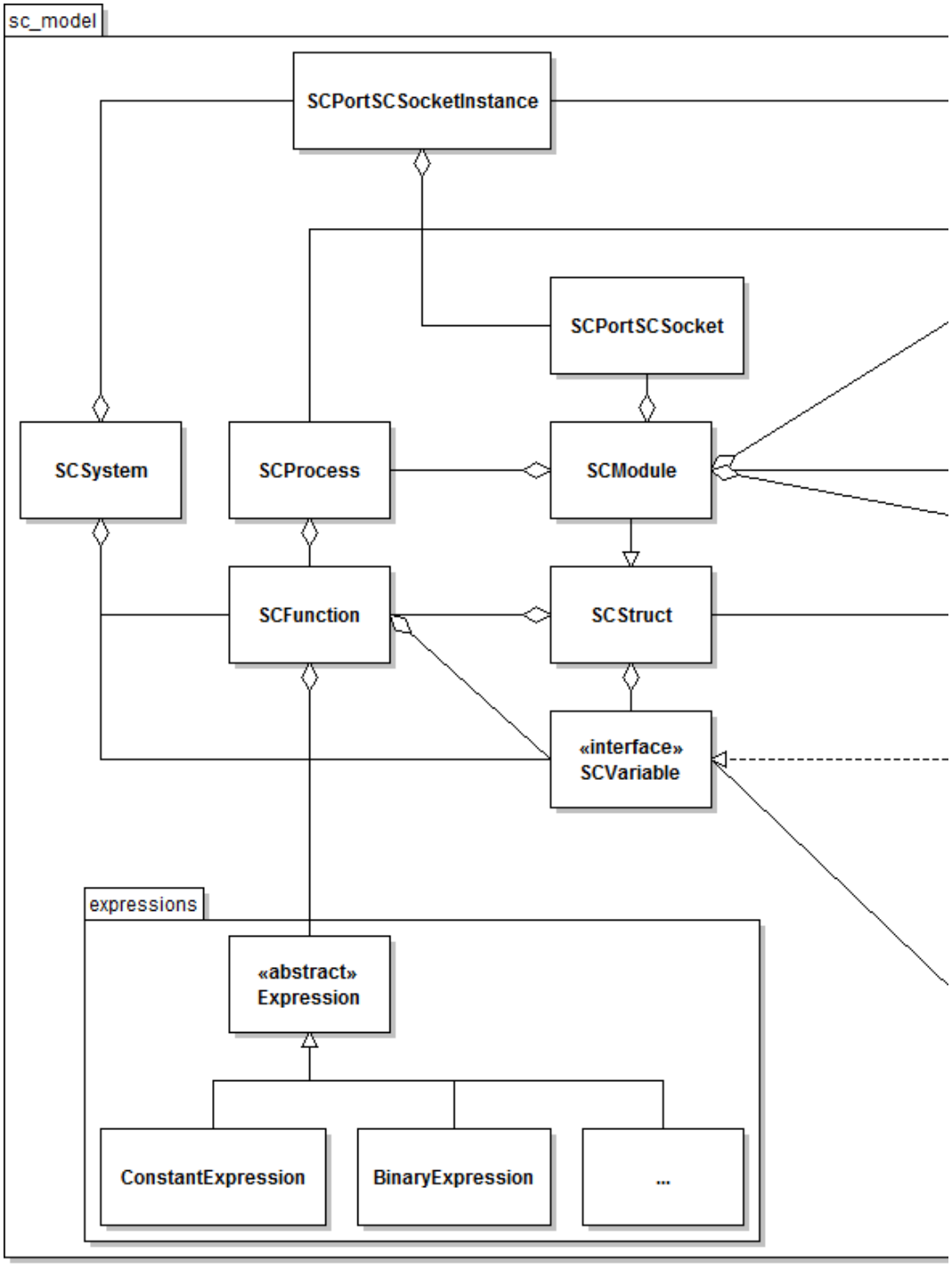
Besondere Schwierigkeiten ergaben sich, da SystemC, wie bereits im Grundlagenkapitel erwähnt, eine Klassenbibliothek von C++ ist. Das führt einerseits dazu, dass wir nicht nur SystemC-Konstrukte abbilden müssen, sondern zum Beispiel auch Structs, die zur Deklarierung eigener Datenstrukturen verwendet werden. Dies haben wir umgesetzt, indem wir auch für solche Konstrukte Objekte oder

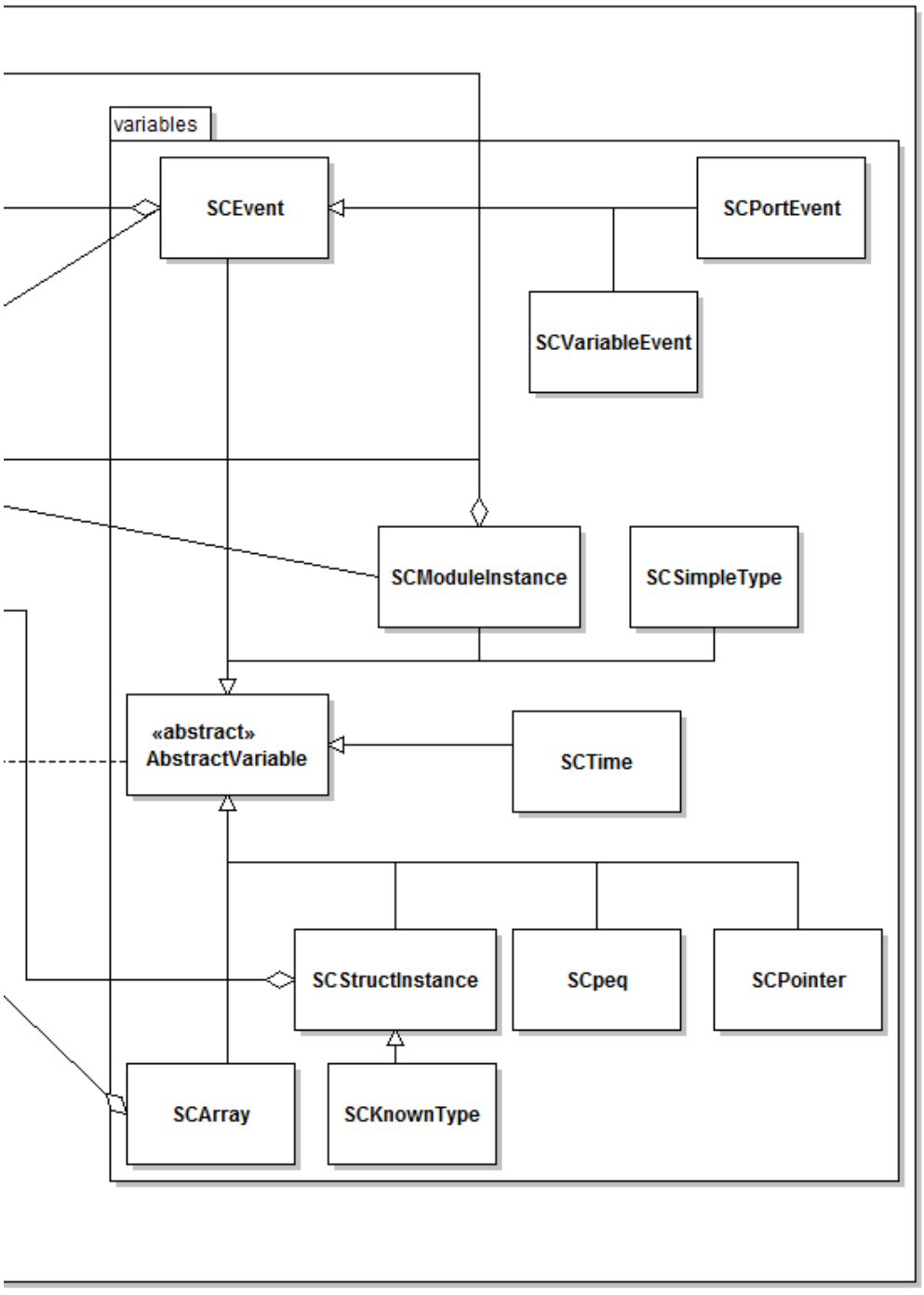
Expressions angelegt haben.
Eine weitere Besonderheit von SystemC ist die Kommunikation. Diese wird in SystemC über Ports und Sockets, sowie Kanäle und Interfaces umgesetzt. Wir haben das realisiert, indem wir für die Ports und Sockets sowie für die Kanäle und Interfaces jeweils eigene Objekte angelegt haben, die diese repräsentieren. In einem weiteren Objekt werden dann die Bindungsinformationen gespeichert.

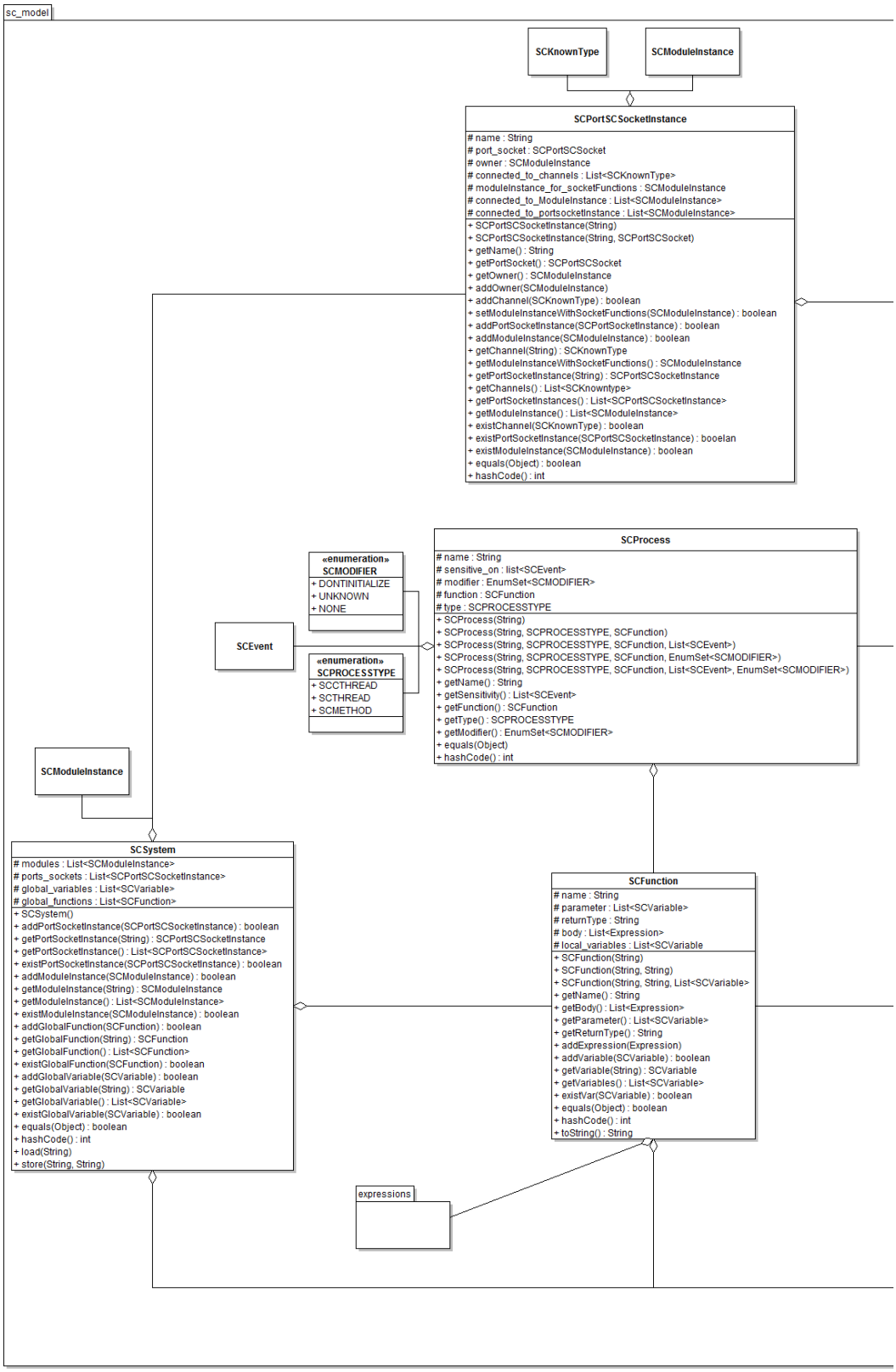
Grundsätzlich speichern wir bei der Transformation des abstrakten Syntaxbaumes jede Information, die dieser zu den unterstützen SystemC-Konstrukten enthält. Dementsprechend haben wir auch die Zwischenrepräsentation angelegt. Somit gehen uns keine Informationen verloren.

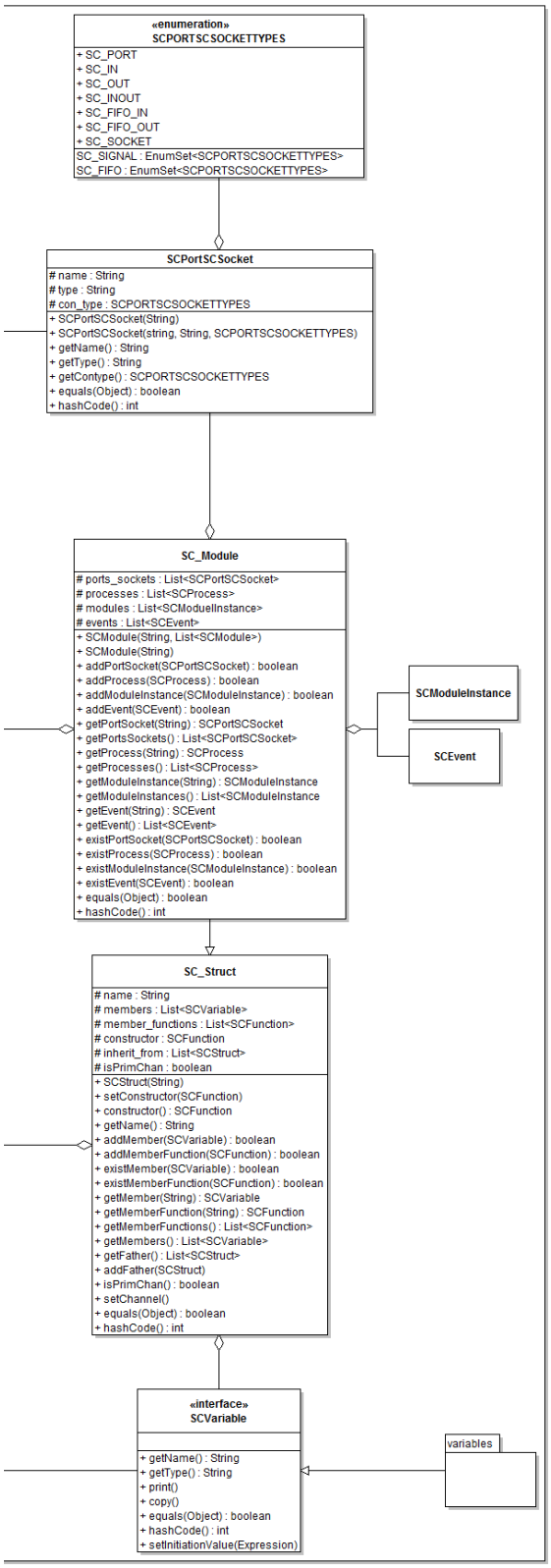
In Zukunft wird die Zwischenrepräsentation dahingehend weiterverwendet, dass Überführungen in verschiedenste Verifikationswerkzeuge entworfen werden. Das sind beispielsweise SMT-Solver wie UCLID oder Modelchecker wie BLAST.
Eine Überführung von SystemC-Modellen in UPPAAL-Timed-Automata wird auf Basis von SysCIR erstellt und wird STATE ersetzen, welches den Schritt der vollständigen Zwischenrepräsentation bisher auslässt. 20

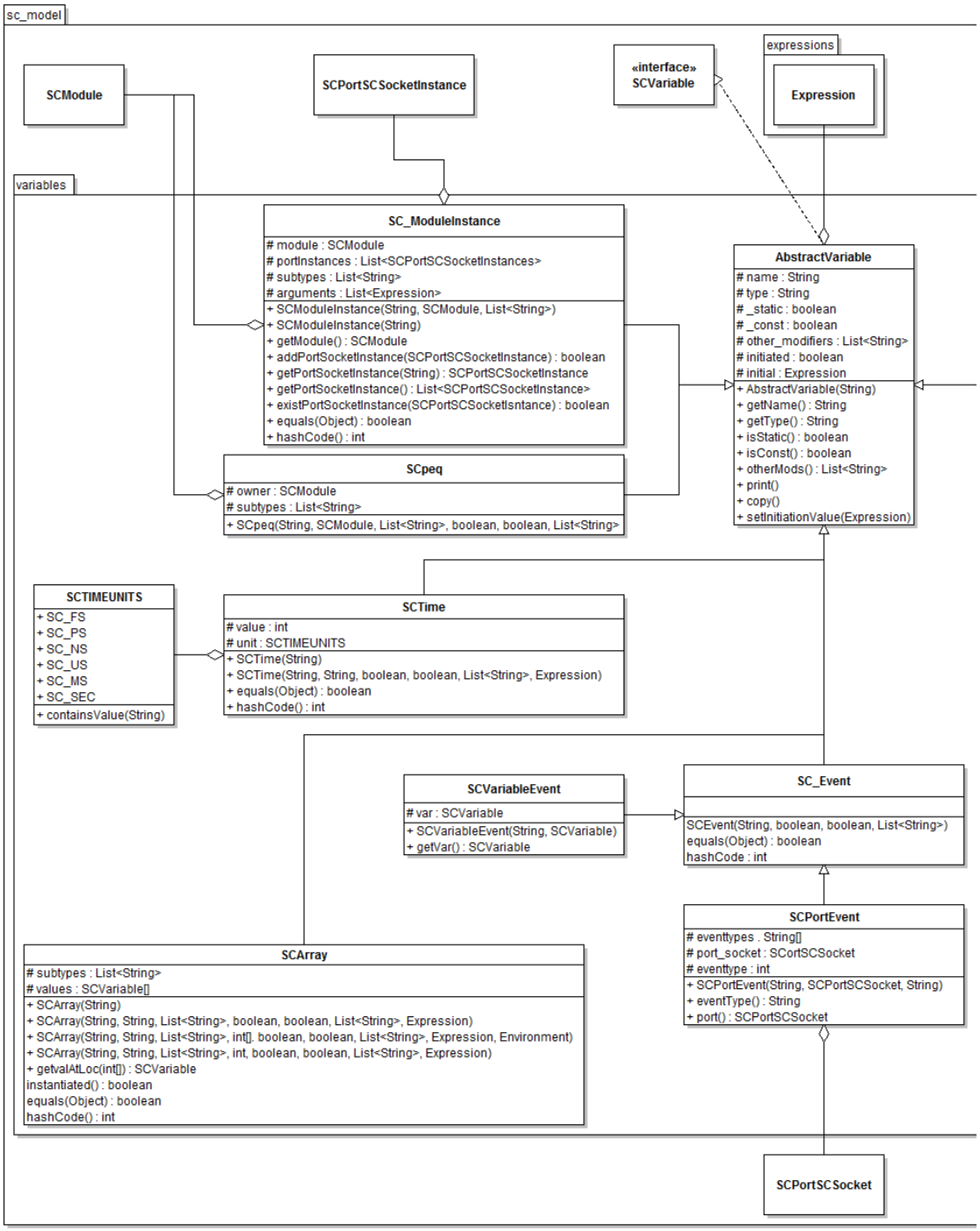
7 Anhang A

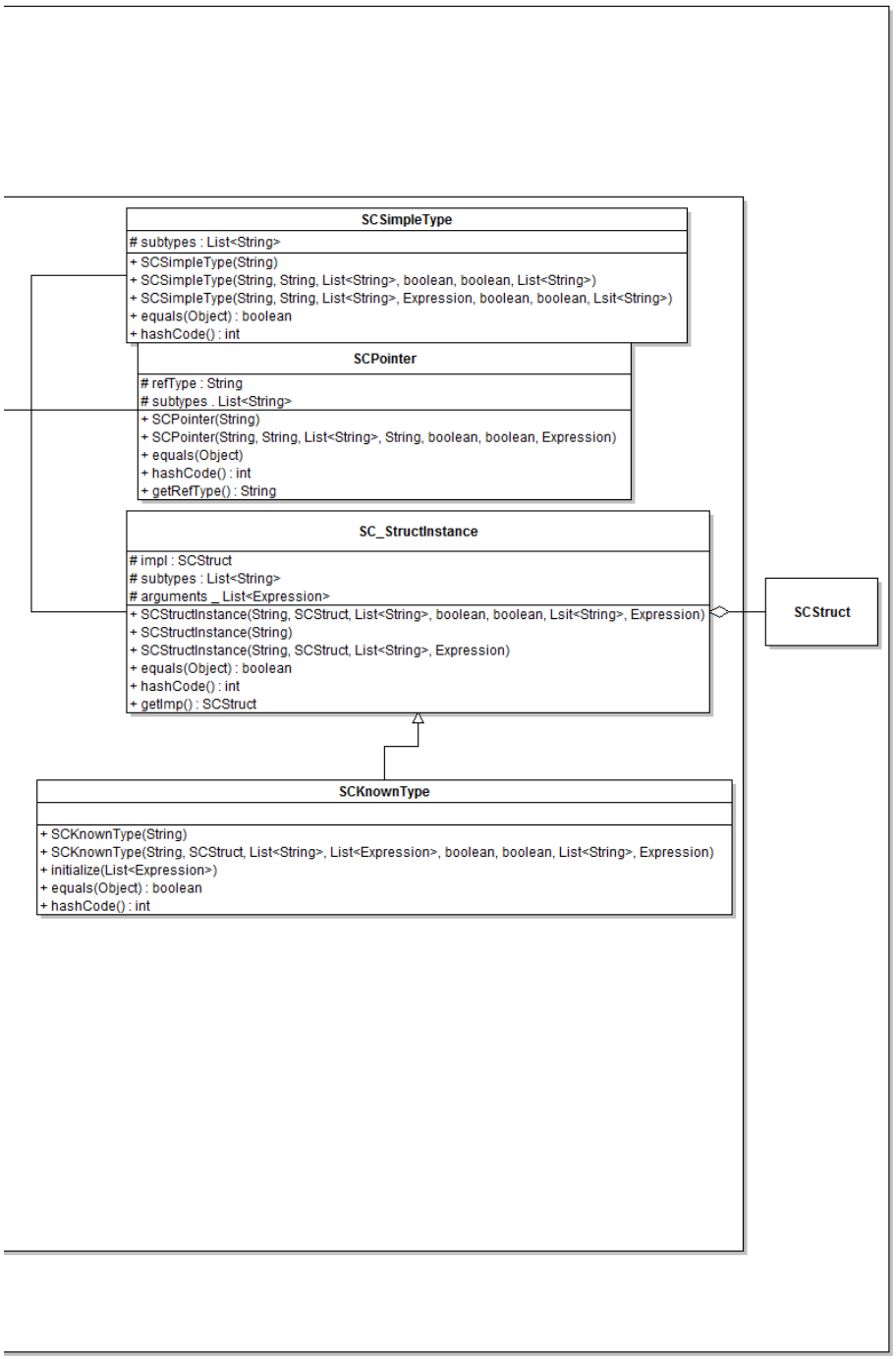

sc_model
SCPortSCSocketInstance
SCPortSCSocket
SCSystem
SCProcess
SCModule
SCFunction
SCStruct
«interface»
SCVariable
expressions
«abstract»
Expression
ConstantExpression
BinaryExpression
...

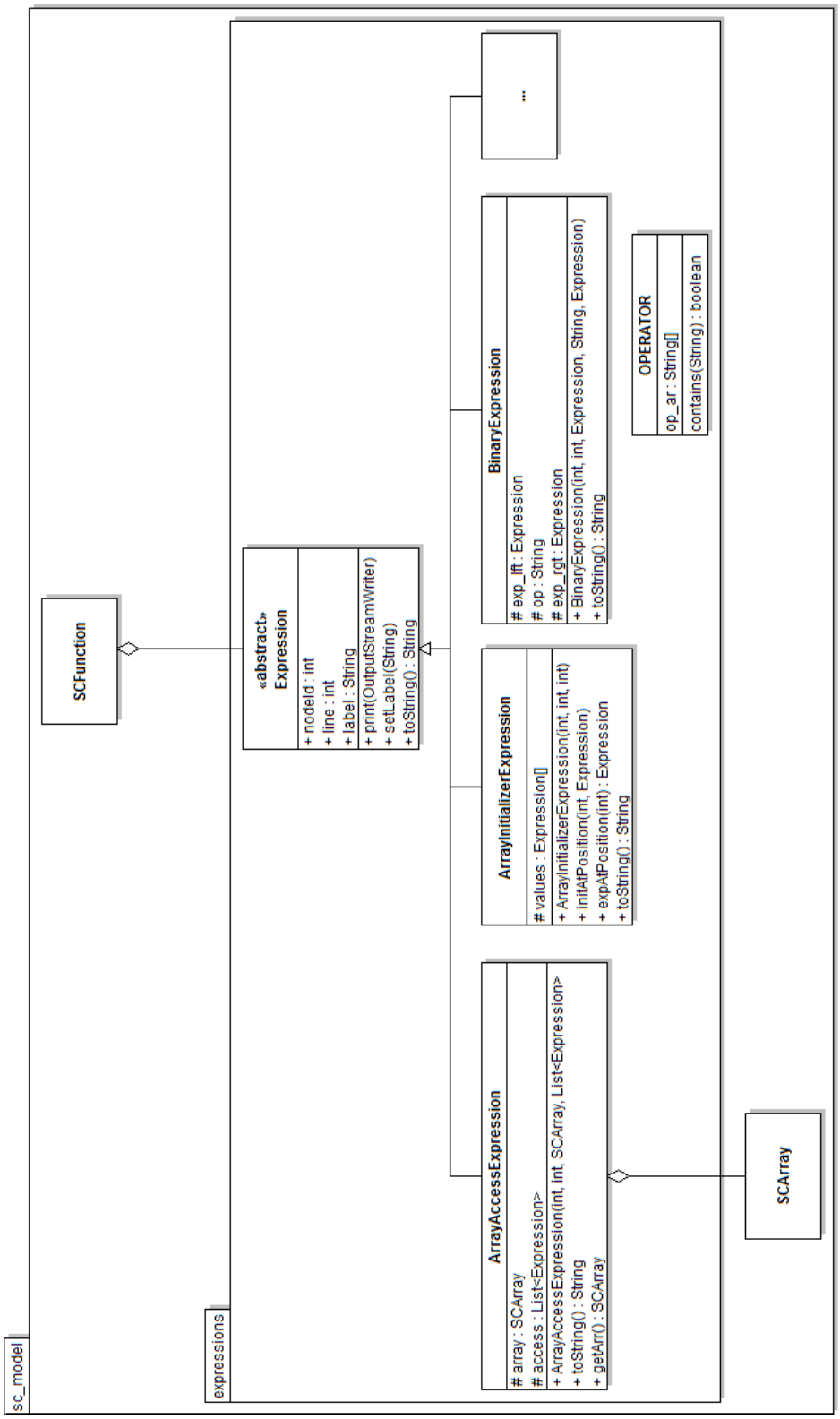

variables
SCEvent
SCPortEvent
SCVariableEvent
SCModuleInstance
SCSimpleType
«abstract»
AbstractVariable
SCTime
SCStructInstance
SCpeq
SCPointer
SCArray
SCKnownType


sc_model
SCKnownType
SCModuleInstance
SCPortSCSocketInstance
name : String
port_socket : SCPortSCSocket
owner : SCModuleInstance
connected_to_channels : List<SCKnownType>
moduleInstance_for_socketFunctions : SCModuleInstance
connected_to_ModuleInstance : List<SCModuleInstance>
connected_to_portsocketInstance : List<SCModuleInstance>
+ SCPortSCSocketInstance(String)
+ SCPortSCSocketInstance(String, SCPortSCSocket)
+ getName() : String
+ getPortSocket() : SCPortSCSocket
+ getOwner() : SCModuleInstance
+ addOwner(SCModuleInstance)
+ addChannel(SCKnownType) : boolean
+ setModuleInstanceWithSocketFunctions(SCModuleInstance) : boolean
+ addPortSocketInstance(SCPortSocketInstance) : boolean
+ addModuleInstance(SCModuleInstance) : boolean
+ getChannel(String) : SCKnownType
+ getModuleInstanceWithSocketFunctions() : SCModuleInstance
+ getPortSocketInstance(String) : SCPortSCSocketInstance
+ getChannels() : List<SCKnowntype>
+ getPortSocketInstances() : List<SCPortSCSocketInstance>
+ getModuleInstance() : List<SCModuleInstance>
+ existChannel(SCKnownType) : boolean
+ existPortSocketInstance(SCPortSCSocketInstance) : booelan
+ existModuleInstance(SCModuleInstance) : boolean
+ equals(Object) : boolean
+ hashCode() : int
SCProcess
name : String
sensitive_on : list<SCEvent>
modifier : EnumSet<SCMODIFIER>
function : SCFunction
type : SCPROCESSTYPE
+ SCProcess(String)
+ SCProcess(String, SCPROCESSTYPE, SCFunction)
+ SCProcess(String, SCPROCESSTYPE, SCFunction, List<SCEvent>)
+ SCProcess(String, SCPROCESSTYPE, SCFunction, EnumSet<SCMODIFIER>)
+ SCProcess(String, SCPROCESSTYPE, SCFunction, List<SCEvent>, EnumSet<SCMODIFIER>)
+ getName() : String
+ getSensitivity() : List<SCEvent>
+ getFunction() : SCFunction
+ getType() : SCPROCESSTYPE
+ getModifier() : EnumSet<SCMODIFIER>
+ equals(Object)
+ hashCode() : int
«enumeration»
SCMODIFIER
+ DONTINITIALIZE
+ UNKNOWN
+ NONE
SCEvent
«enumeration»
SCPROCESSTYPE
+ SCCTHREAD
+ SCTHREAD
+ SCMETHOD
SCModuleInstance
SCSystem
modules : List<SCModuleInstance>
ports_sockets : List<SCPortSCSocketInstance>
global_variables : List<SCVariable>
global_functions : List<SCFunction>
+ SCSystem()
+ addPortSocketInstance(SCPortSCSocketInstance) : boolean
+ getPortSocketInstance(String) : SCPortSCSocketInstance
+ getPortSocketInstance() : List<SCPortSCSocketInstance>
+ existPortSocketInstance(SCPortSCSocketInstance) : boolean
+ addModuleInstance(SCModuleInstance) : boolean
+ getModuleInstance(String) : SCModuleInstance
+ getModuleInstance() : List<SCModuleInstance>
+ existModuleInstance(SCModuleInstance) : boolean
+ addGlobalFunction(SCFunction) : boolean
+ getGlobalFunction(String) : SCFunction
+ getGlobalFunction() : List<SCFunction>
+ existGlobalFunction(SCFunction) : boolean
+ addGlobalVariable(SCVariable) : boolean
+ getGlobalVariable(String) : SCVariable
+ getGlobalVariable() : List<SCVariable>
+ existGlobalVariable(SCVariable) : boolean
+ equals(Object) : boolean
+ hashCode() : int
+ load(String)
+ store(String, String)
SCFunction
name : String
parameter : List<SCVariable>
returnType : String
body : List<Expression>
local_variables : List<SCVariable
+ SCFunction(String)
+ SCFunction(String, String)
+ SCFunction(String, String, List<SCVariable>
+ getName() : String
+ getBody() : List<Expression>
+ getParameter() : List<SCVariable>
+ getReturnType() : String
+ addExpression(Expression)
+ addVariable(SCVariable) : boolean
+ getVariable(String) : SCVariable
+ getVariables() : List<SCVariable>
+ existVar(SCVariable) : boolean
+ equals(Object) : boolean
+ hashCode() : int
+ toString() : String
expressions


«enumeration»
SCPORTSCSOCKETTYPES
+ SC_PORT
+ SC_IN
+ SC_OUT
+ SC_INOUT
+ SC_FIFO_IN
+ SC_FIFO_OUT
+ SC_SOCKET
SC_SIGNAL : EnumSet<SCPORTSCSOCKETTYPES>
SC_FIFO : EnumSet<SCPORTSCSOCKETTYPES>
SCPortSCSocket
name : String
type : String
con_type : SCPORTSCSOCKETTYPES
+ SCPortSCSocket(String)
+ SCPortSCSocket(string, String, SCPORTSCSOCKETTYPES)
+ getName() : String
+ getType() : String
+ getContype() : SCPORTSCSOCKETTYPES
+ equals(Object) : boolean
+ hashCode() : int
SC_Module
ports_sockets : List<SCPortSCSocket>
processes : List<SCProcess>
modules : List<SCModuelInstance>
events : List<SCEvent>
+ SCModule(String, List<SCModule>)
+ SCModule(String)
+ addPortSocket(SCPortSCSocket) : boolean
+ addProcess(SCProcess) : boolean
+ addModuleInstance(SCModuleInstance) : boolean
+ addEvent(SCEvent) : boolean
+ getPortSocket(String) : SCPortSCSocket
+ getPortsSockets() : List<SCPortSCSocket>
+ getProcess(String) : SCProcess
+ getProcesses() : List<SCProcess>
+ getModuleInstance(String) : SCModuleInstance
+ getModuleInstances() : List<SCModuleInstance
+ getEvent(String) : SCEvent
+ getEvent() : List<SCEvent>
+ existPortSocket(SCPortSCSocket) : boolean
+ existProcess(SCProcess) : boolean
+ existModuleInstance(SCModuleInstance) : boolean
+ existEvent(SCEvent) : boolean
+ equals(Object) : boolean
+ hashCode() : int
SCModuleInstance
SCEvent
SC_Struct
name : String
members : List<SCVariable>
member_functions : List<SCFunction>
constructor : SCFunction
inherit_from : List<SCStruct>
isPrimChan : boolean
+ SCStruct(String)
+ setConstructor(SCFunction)
+ constructor() : SCFunction
+ getName() : String
+ addMember(SCVariable) : boolean
+ addMemberFunction(SCFunction) : boolean
+ existMember(SCVariable) : boolean
+ existMemberFunction(SCFunction) : boolean
+ getMember(String) : SCVariable
+ getMemberFunction(String) : SCFunction
+ getMemberFunctions() : List<SCFunction>
+ getMembers() : List<SCVariable>
+ getFather() : List<SCStruct>
+ addFather(SCStruct)
+ isPrimChan() : boolean
+ setChannel()
+ equals(Object) : boolean
+ hashCode() : int
«interface»
SCVariable
+ getName() : String
+ getType() : String
+ print()
+ copy()
+ equals(Object) : boolean
+ hashCode() : int
+ setInitiationValue(Expression)
variables


sc_model
SCModule
SCPortSCSocketInstance
«interface»
SCVariable
expressions
Expression
variables
SC_ModuleInstance
module : SCModule
portInstances : List<SCPortSCSocketInstances>
subtypes : List<String>
arguments : List<Expression>
+ SCModuleInstance(String, SCModule, List<String>)
+ SCModuleInstance(String)
+ getModule() : SCModule
+ addPortSocketInstance(SCPortSCSocketInstance) : boolean
+ getPortSocketInstance(String) : SCPortSCSocketInstance
+ getPortSocketInstance() : List<SCPortSCSocketInstance>
+ existPortSocketInstance(SCPortSCSocketIsntance) : boolean
+ equals(Object) : boolean
+ hashCode() : int
AbstractVariable
name : String
type : String
_static : boolean
_const : boolean
other_modifiers : List<String>
initiated : boolean
initial : Expression
+ AbstractVariable(String)
+ getName() : String
+ getType() : String
+ isStatic() : boolean
+ isConst() : boolean
+ otherMods() : List<String>
+ print()
+ copy()
+ setInitiationValue(Expression)
SCpeq
owner : SCModule
subtypes : List<String>
+ SCpeq(String, SCModule, List<String>, boolean, boolean, List<String>
SCTIMEUNITS
+ SC_FS
+ SC_PS
+ SC_NS
+ SC_US
+ SC_MS
+ SC_SEC
+ containsValue(String)
SCTime
value : int
unit : SCTIMEUNITS
+ SCTime(String)
+ SCTime(String, String, boolean, boolean, List<String>, Expression)
+ equals(Object) : boolean
+ hashCode() : int
SCVariableEvent
var : SCVariable
+ SCVariableEvent(String, SCVariable)
+ getVar() : SCVariable
SC_Event
SCEvent(String, boolean, boolean, List<String>)
equals(Object) : boolean
hashCode : int
SCPortEvent
eventtypes . String[]
port_socket : SCortSCSocket
eventtype : int
+ SCPortEvent(String, SCPortSCSocket, String)
+ eventType() : String
+ port() : SCPortSCSocket
SCArray
subtypes : List<String>
values : SCVariable[]
+ SCArray(String)
+ SCArray(String, String, List<String>, boolean, boolean, List<String>, Expression)
+ SCArray(String, String, List<String>, int[], boolean, boolean, List<String>, Expression, Environment)
+ SCArray(String, String, List<String>, int, boolean, boolean, List<String>, Expression)
+ getvalAtLoc(int[]) : SCVariable
instantiated() : boolean
equals(Object) : boolean
hashCode() : int
SCPortSCSocket


SCSimpleType
subtypes : List<String>
+ SCSimpleType(String)
+ SCSimpleType(String, String, List<String>, boolean, boolean, List<String>)
+ SCSimpleType(String, String, List<String>, Expression, boolean, boolean, Lsit<String>)
+ equals(Object) : boolean
+ hashCode() : int
SCPointer
refType : String
subtypes . List<String>
+ SCPointer(String)
+ SCPointer(String, String, List<String>, String, boolean, boolean, Expression)
+ equals(Object)
+ hashCode() : int
+ getRefType() : String
SC_StructInstance
impl : SCStruct
subtypes : List<String>
arguments _ List<Expression>
+ SCStructInstance(String, SCStruct, List<String>, boolean, boolean, Lsit<String>, Expression)
+ SCStructInstance(String)
+ SCStructInstance(String, SCStruct, List<String>, Expression)
+ equals(Object) : boolean
+ hashCode() : int
+ getImp() : SCStruct
SCStruct
SCKnownType
+ SCKnownType(String)
+ SCKnownType(String, SCStruct, List<String>, List<Expression>, boolean, boolean, List<String>, Expression)
+ initialize(List<Expression>)
+ equals(Object) : boolean
+ hashCode() : int


sc_model
expressions
SCFunction
«abstract»
Expression
+ nodeId : int
+ line : int
+ label : String
+ print(OutputStreamWriter)
+ setLabel(String)
+ toString() : String
...
BinaryExpression
exp_lft : Expression
op : String
exp_rgt : Expression
+ BinaryExpression(int, int, Expression, String, Expression)
+ toString() : String
OPERATOR
op_ar : String[]
contains(String) : boolean
ArrayInitializerExpression
values : Expression[]
+ ArrayInitializerExpression(int, int, int)
+ initAtPosition(int, Expression)
+ expAtPosition(int) : Expression
+ toString() : String
ArrayAccessExpression
array : SCArray
access : List<Expression>
+ ArrayAccessExpression(int, int, SCArray, List<Expression>
+ toString() : String
+ getArr() : SCArray
SCArray

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Literatur

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, LNCS 3185, pages 200–236. Springer, 2004.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Workshop on Verification and Control of Hybrid Systems*, LNCS 1066, pages 232–243. Springer, October 1995.

[FZI] FZI Research Center for Information Technology. KaSCPar - Karlsruhe SystemC Parser Suite.

[GKD06] Daniel Große, Ulrich Kühne, and Rolf Drechsler. HW/SW Co-Verification of Embedded Systems using Bounded Model Checking. In *Great Lakes Symposium on VLSI*, pages 43–48. ACM Press, 2006.

[HFG08] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model Checking SystemC Designs Using Timed Automata. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 131–136. ACM press, 2008.

[HMT06] Ali Habibi, Haja Moinudeen, and Sofiene Tahar. Generating Finite State Machines from SystemC. In *Design, Automation and Test in Europe*, pages 76–81. IEEE, 2006.

[HPG11] Paula Herber, Marcel Pockrandt, and Sabine Glesner. Transforming SystemC Transaction Level Models into UPPAAL Timed Automata. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 161 – 170. IEEE Computer Society, 2011.

[HT05] Ali Habibi and Sofiène Tahar. An Approach for the Verification of SystemC Designs Using AsmL. In *Automated Technology for Verification and Analysis*, LNCS 3707, pages 69–83. Springer, 2005.

[IEE05] IEEE Standards Association. IEEE Std. 1666–2005, Open SystemC Language Reference Manual, 2005.

[MRR03] Wolfgang Müller, Jürgen Ruf, and Wolfgang Rosenstiel. *SystemC: Methodologies and Applications*, chapter An ASM based SystemC Simulation Semantics, pages 97–126. Kluwer Academic Publishers, 2003.

[PHG11] Marcel Pockrandt, Paula Herber, and Sabine Glesner. Model Checking a SystemC/TLM Design of the AMBA AHB Protocol. In *IEEE/ACM*

Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia), pages 66 – 75. IEEE Computer Society, 2011.

[RHG+01] Jürgen Ruf, Dirk W. Hoffmann, Joachim Gerlach, Thomas Kropf, Wolfgang Rosenstiel, and Wolfgang Müller. The Simulation Semantics of SystemC. In *Design, Automation and Test in Europe*, pages 64–70. IEEE Press, 2001.

[Sal03] Ashraf Salem. Formal Semantics of Synchronous SystemC. In *Design, Automation and Test in Europe (DATE)*, pages 10376–10381. IEEE Computer Society, 2003.

[TCMM07] Claus Traulsen, Jerome Cornet, Matthieu Moy, and Florence Maraninchi:. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software (SPIN '07)*, LNCS 4595, pages 204–222, Berlin, 2007. Springer.

[ZVM07] Yu Zhang, Franck Vedrine, and Bruno Monsuez. SystemC Waiting-State Automata. In *First International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007)*, 2007.