Fachgebiet Programmierung Eingebetteter Systeme
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

Analyse und Transformation dynamischer Speicherverwaltung in Hardware/Software Co-Designs

Masterarbeit

zur Erlangung des Grades
Master of Science (M. Sc.)

im Studiengang Informatik

Rolf Schröder
Matr.-Nr. 342126

5. November 2013

Betreuer:

Prof. Dr. rer. nat. Sabine Glesner
Dipl.-Inform. Marcel Pockrandt

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 5. November 2013

Danksagung

Ich möchte mich bei allen Freunden und meiner Familie für die Unterstützung im letzten halben Jahr bedanken. Euer beständiges Nachfragen und Eure Aufmunterungen haben mir sehr geholfen.

Ein besonderer Dank gebührt meinem Betreuer Marcel Pockrandt, ohne den diese Arbeit nicht möglich gewesen wäre.

Zusammenfassung

Eingebettete Systeme operieren häufig in sicherheitskritischen Umgebungen. Ihr korrektes Verhalten ist extrem wichtig, da Fehler schwerwiegende Folgen nach sich ziehen können. Gleichzeitig stehen ihnen naturgemäß nur begrenzte Ressourcen zur Verfügung, um ihre Aufgabe zu erfüllen. Unter diesen Umständen muss der Qualitätssicherung im Entwicklungsprozess eines solchen Systems besonders hohe Aufmerksamkeit geschenkt werden. Tests, so ausführlich sie auch sein mögen, reichen nicht aus, um die Korrektheit des Systems zu beweisen. Deshalb sind formale Verifikationstechniken nötig.

Die Modellierungssprache SystemC wird in der Industrie häufig verwendet, um Hardware/Software Co-Designs zu implementieren. SystemC basiert auf C++, einer Programmiersprache, welche es dem Designer erlaubt, flexibel mit Speicher umzugehen. Dazu gehört insbesondere auch die dynamische Allokation von Speicherbereichen.

In dieser Arbeit stellen wir unseren Ansatz zur Modellierung und Verifikation des dynamischen Speicherverhaltens von SystemC-Modellen vor. Der Ansatz ermöglicht es, ein gegebenes SystemC-Design in ein Netzwerk von UPPAAL Timed Automata zu transformieren. Dieses Netzwerk erweitern wir um ein Speichermodell, um SystemC-Speicheroperationen in UPPAAL abbilden zu können. Unser Modell erlaubt es im Besonderen auch dynamisches Speichermanagement zu simulieren. Das UPPAAL-Modell ist strukturell dem SystemC-Modell nachempfunden, sodass man die Ergebnisse der Verifikation leicht nachvollziehen kann.

Das entstandene UPPAAL-Modell kann zudem mit UPPAAL Modelchecker verifiziert werden. Die interne Speicherrepräsentation ermöglicht es, Eigenschaften zu definieren, die den korrekten Umgang mit dem Speicher abbilden (z. B. Zugriff nur auf zuvor reservierte Speicherzellen). Genügt das Modell einer Eigenschaft nicht, kann mit Hilfe des von UPPAAL generierten Gegenbeispiels nachvollzogen werden, wo im originären SystemC-Modell der Fehler liegt.

Unser Ansatz erweitert die im Fachgebiet Programmierung Eingebetteter Systeme der TU Berlin entwickelte Transformation von SystemC nach UPPAAL Timed Automata. Wir haben das bereits vorhandene Werkzeug STATE modifiziert, um die Transformation inklusive Speichermodell zu automatisieren. Wir konnten unseren Ansatz erfolgreich an einer Fallstudie testen. Die Ergebnisse zeigen, dass unser Ansatz die Anwendbarkeit im Vergleich zu vorherigen Ansätzen erweitert und die Verifikation von zusätzlichen speicherbezogenen Eigenschaften erlaubt.

In Zukunft planen wir einige Optimierungen, um den von uns induzierten Overhead weiter zu reduzieren, und die Unterstützung weiterer SystemC-Sprachelemente.

Abstract

Embedded Systems often operate in safety critical environments. Their correct functioning is of great importance because errors may have serious consequences. Moreover, do these systems dispose only limited resources to fulfill their tasks. Given these constraints, quality assurance during the design process must be a great concern. Tests and simulations, as extensive as they may be, do not guarantee the correctness /correct functioning of the system and therefore formal verification techniques are needed.

SystemC is a modeling language for hardware/software co-designs commonly adopted in the industry. The language is based on C++, a programming language which allows for flexible memory management. This includes notably dynamic memory allocation.

In this work, we present an approach to model and verify static and dynamic memory operations used in SystemC designs. We are able to transform a given SystemC model in a network of UPPAAL Timed Automata. This network is extended by a sound memory model which can be used to represent memory operations. The resulting UPPAAL model is structurally similar to the original design in order to easily follow the verification results.

We used UPPAAL's internal model checker to verify the model. Our memory representation enables us to define and verify memory related properties. We can therefore prove correct memory manipulation. If the SystemC model does not satisfy any of theses properties, UPPAAL provides a corresponding counterexample which can be used to track back the property violation in the original source code.

Our approach extends the transformation from SystemC to UPPAAL developed at the Software Engineering for Embedded Systems Group at the Technische Universität Berlin. We modified the existing pipeline to insert our new memory model in the automatic transformation process. We successfully evaluated the soundness of our approach using a case study. Our results show that our model supports a larger set of SystemC models compared to previous approaches. We were also able to verify more memory related properties.

For future work, we plane to optimize our model further to reduce the overhead introduced by our approach. We also aim to support a larger subset of the SystemC language.

Inhaltsverzeichnis

1 Einleitung

Eingebettete Systeme operieren häufig in sicherheitskritischen Umgebungen. Ihr korrektes Verhalten ist extrem wichtig, da Fehler schwerwiegende Folgen nach sich ziehen können. Gleichzeitig stehen ihnen naturgemäß nur begrenzte Ressourcen zur Verfügung, um ihre Aufgabe zu erfüllen. Unter diesen Umständen muss der Qualitätssicherung im Entwicklungsprozess eines solchen Systems besonders hohe Aufmerksamkeit geschenkt werden. Tests, so ausführlich sie auch sein mögen, reichen nicht aus, um die Korrektheit des Systems zu beweisen.

1.1 Problemstellung

Speicher ist eine wichtige Ressource in eingebetteten Systemen. Fehler in der Verwendung von Speicher sind häufig nur schwer detektierbar, können aber gleichzeitig verheerende Konsequenzen haben. Eine besondere Herausforderung liegt zusätzlich darin, dass trotz des technologischen Fortschritts die Größe der Hardwarekomponente Speicher stark limitiert ist. Um dennoch mit einer großen Menge Daten arbeiten zu können, ist es nötig, flexibel mit Speicher umzugehen. Dazu gehört nicht nur der Gebrauch vom Pointern, sondern auch die dynamische (De-)Allokation von Speicherbereichen. Beide Techniken sind fehlerträchtig, weil Programmierfehler selten im Vorfeld erkannt werden und während der Laufzeit schnell zu Systemabstürzen führen können. Demzufolge muss bei der Verifizierung eines eingebetteten Systems besonderes Augenmerk auf die Korrektheit jeglicher Art von Speichermanipulation gelegt werden.

Viele Verifikationstechniken besitzen allerdings kein oder nur ein rudimentäres Speichermodell. Somit lassen sich Systeme, die Speichermanipulationen durchführen, nur schlecht überprüfen. Grundsätzlich wäre es möglich, mittels Abstraktions- und Transformationstechniken die fraglichen Operationen zu entfernen. Die manuelle Transformation ist aber sehr fehleranfällig. Selbst wenn dieser Prozess automatisiert wird, ist es nur in Ausnahmefällen möglich die Semantik des Systems dabei nicht zu verändern. Außerdem stellt dieses Vorgehen die gesamte Verifikation in Frage, da sie ohne die Untersuchung des Speicherverhaltens wesentlich an Aussagekraft verliert.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist die Entwicklung eines Modells, das den Speicher eines Hardware/Software Co-Design abbildet und die Verifikation bestimmter Eigenschaften des Speicherverhaltens zulässt. Das Modell muss eindeutig definiert sein und eine automatisierte Überführung erlauben. Weiterhin muss die formale Verifikation performant und damit praxistauglich sein.

Wir konzentrieren uns auf Systeme, die in SystemC (s. Abschnitt 2.1) modelliert werden. Folglich beschränkt sich unser Modell nur auf die Speicherrepräsentation eines solchen Designs. Die Entwicklung eines Modells, das das originäre C++-Speichermodell – das beispielsweise auch Codesegmente enthält – nachempfindet, ist unnötig. Unser Ansatz soll die wichtigsten Speicheroperationen, die in einem SystemC-Design Verwendung finden, erfassen können und deren Überprüfung ermöglichen.

Dabei stellen wir folgende Anforderungen an unser Konzept:

- **Eindeutigkeit**
  Das Speichermodell muss eindeutig definiert sein und die Semantik des SystemC-Designs erhalten.
- **Lesbarkeit**
  Das Modell sollte möglichst einfach zu verstehen sein. Bei einem Fehler kann so leichter nachvollzogen werden, in welchem Kontext eine Bedingung verletzt wurde. Die vereinfachte Generierung von Gegenbeispielen ist ein weiterer Vorteil. Eine Anlehnung an das C++-Speichermodell bietet sich daher an, weil es dem Designer bekannt ist und die Übersetzung intuitiver gestaltet.
- **Ausdrucksmächtigkeit**
  Unser Modell muss die wichtigsten Speicheroperationen unterstützen, um auf realistische Szenarien anwendbar zu sein. Dazu gehört der Gebrauch von Pointern und Referenzen, und die statische bzw. dynamische Speicherallokation.
- **Overhead**
  Wir benutzen zum Verifizieren von SystemC-Designs einen Modelchecker. Modelchecking ist eine mächtige und etablierte Technik, um Eigenschaften eines modellierten Systems sicher zu stellen. Allerdings besteht immer die Gefahr der Zustandsexplosion, die eine Verifikation extrem langwierig oder gar unmöglich macht. Deshalb darf das Speichermodell selbst nicht zu viel Overhead erzeugen oder muss jeweils für einen konkreten Anwendungsfall optimierbar sein.

Wir benutzen UPPAAL (s. Abschnitt 2.4) als Modelchecker. Eine Übersetzung eines SystemC-Designs in ein UPPAAL-Modell, die das soeben charakterisierte Speichermodell integriert, ermöglicht die automatisierte Verifikation des Speicherverhaltens einer Vielzahl von Programmen.

1.3 Ansatz

In [21, 22, 30] wurde bereits ein Ansatz vorgestellt, um ein SystemC-Design in ein UPPAAL-Modell zu übersetzen. Allerdings werden starke Einschränkungen an das zu verifizierende System gemacht. Unter anderem werden keine Pointer unterstützt. Durch Einführung eines geeigneten Speichermodells konnte diese Restriktion in [27] gelockert werden, allerdings nur für statische Allokation. Der Heap, in dem sich zur Laufzeit erstellte und gelöschte Objekte befinden, kann nach wie vor nicht abgebildet werden.

Wir erweitern den Ansatz um Speicherreservierung und -freigabe während der Laufzeit zu modellieren. Unser Konzept modelliert die – normalerweise vom Betriebssystem bereitgestellte – Funktionalität, die Auskunft über die Belegung bestimmter Speicheradressen gibt. Dies bedeutet, dass unser Speichermodell und die zugehörigen Übersetzungsregeln es ermöglichen, während der Verifikation Speicherbelegung zu simulieren. Wir repräsentieren den Speicher eines SystemC-Modells als eine Menge von typisierten Arrays, wobei es genau ein Array pro Datentyp gibt. Ein Variablenzugriff im SystemC-Modell wird übersetzt in einen Elementzugriff im entsprechenden Datentyparray (vgl. [27]). Zusätzlich führen wir zu jedem Datentyparray ein zugehöriges, gleich langes Statusarray ein, das Auskunft über die Belegung der Elemente gibt. Somit sind wir in der Lage, während der Verifikation Speicher zu „belegen“ bzw. zu „löschen“. Da uns das Statusarray es ferner ermöglicht zwischen verschiedenen Belegungen zu differenzieren, können wir den Stack und den Heap getrennt von einander darstellen und trotzdem dasselbe Datentyparray nutzen. Dieses Modell ist einfach zu verstehen und trotzdem effizient.

Die Größe von Datentypen in UPPAAL muss statisch zu ermitteln sein und kann nicht zur Laufzeit geändert werden. Aus diesem Grund ist es notwendig, den maximalen Speicherverbrauch des Systems abzuschätzen und die Größe der Datentyp- und Statusarrays so zu wählen, dass die Allokation immer gelingt. Wir verwenden hierfür eine selbst entwickelte Analysetechnik.

1.4 Motivation

Eingebettete Systeme umgeben uns im Alltag auf Schritt und Tritt. In Waschmaschinen, Mikrowellen, Telephonen und Unterhaltungselektronik erleichtern sie unser Leben oder machen es ganz einfach angenehmer. Sie verrichten ihre Dienste aber auch in sicherheitskritischen Umgebungen wie in der Medizintechnik oder Transportmitteln. Hier ist ein korrektes Funktionieren von herausragender Bedeutung, da im Fehlerfall sogar Lebensgefahr für die Nutzer eines solchen Systems besteht. Dementsprechend wichtig ist die Qualitätssicherung der Produkte. Testen kann dazu verwendet werden, bestimmte Fehler zu erkennen. Allerdings ist Testen allein nicht ausreichend um die Korrektheit eines Systems zu beweisen. Formale Verifikationstechniken können genutzt werden, um die Abwesenheit von Fehlern formal zu beweisen und hierdurch sicherzustellen, dass das System korrekt funktioniert.

Ein eingebettetes System ist häufig stark an seine spezifische Aufgabe angepasst. Es besteht meist aus Hardware- und Softwarekomponenten um Schnelligkeit mit Flexibilität zu kombinieren. SystemC ist eine Modellierungssprache, mit der ein solcher integrierter Entwurf beschrieben werden kann. Da die Sprache in der Industrie weit verbreitet ist, wäre eine Verifikation von in SystemC modellierten eingebetteten Systemen sehr willkommen. Die Sprache beschreibt ein Hardware/Software Co-Design allerdings in einer informellen Semantik. Formale Verifikationstechniken benötigen jedoch eine formale Semantik. Deswegen lassen sich die hier üblichen Beweistechniken nicht direkt auf SystemC anwenden. Diesem Problem kann man entgehen, in dem man das zu verifizierende System in eine Sprache überführt, die eine formale Semantik besitzt. Bestehende Ansätze zur Transformation eines SystemC-Designs sind noch nicht ausgereift, weil sie meistens nur eine Teilmenge der Sprache abbilden. Insbesondere gibt es häufig nur eine begrenzte oder gar keine Unterstützung für Operationen, die den Speicher manipulieren. Dies gilt auch für den bereits erwähnten Ansatz [27], der SystemC-Designs in UPPAAL Timed Automata übersetzt. Zwar werden einige Speicheroperationen unterstützt, allerdings beschränkt sich der Ansatz nur auf statische Allokation.

Speicher wird immer preiswerter und eingebettete Systeme haben immer mehr davon zur Verfügung. Deshalb werden Produkte, die flexibel mit Speicherplatz umgehen können, in Zukunft immer häufiger Anwendung finden. Die Entwicklung eines Modells, dass eine automatische Transformation beliebiger solcher SystemC-Designs und eine formale Verifikation ermöglicht, birgt viele Vorteile. Designfehler können schneller gefunden und die Produkte abgesichert werden. Der Entwicklungsprozess wird zudem günstiger und kürzer.

1.5 Aufbau der Arbeit

Der Rest dieser Arbeit gliedert sich wie folgt: Das nächste Kapitel erläutert SystemC und UPPAAL Timed Automata genauer. Weiterhin wird die bereits vorhandene Übersetzung eines SystemC-Designs nach UPPAAL erklärt. Es folgt eine nähere Untersuchung aller von uns unterstützten speicherspezifischen Sprachkonstrukte für SystemC. Kapitel 3 stellt verwandte Arbeiten vor. In Kapitel 4 erläutern wir unser Speichermodell und legen dar, wie wir dieses bei der Übersetzung integrieren. Das fünfte Kapitel verdeutlicht, wie wir unseren Ansatz nutzen, um speicherbezogene Eigenschaften zu verifizieren. In Kapitel 6 erklären wir, wie die Automatisierung der Transformation konkret aussieht. Es folgt Kapitel 7, in dem wir unseren Ansatz an einer Fallstudie evaluieren. Zum Schluss fassen wir unsere Ergebnisse noch zusammen und geben einen Ausblick über zukünftige Forschungsthemen.

2 Grundlagen

In diesem Kapitel geben wir einen kurzen Überblick über die von uns verwendeten Technologien und Konzepte. Wir erläutern die Modellierungssprache SystemC und ihre wichtigsten Speicheroperationen. Danach stellen wir kurz Modelchecking als Verfahren vor. Weiterhin besprechen wir das Konzept der *Timed Automata* und das Werkzeug UPPAAL. Es folgt eine Behandlung der bereits vorhandenen Transformation von SystemC nach UPPAAL Timed Automata [21, 22, 30, 27].

2.1 SystemC

SystemC ist eine Modellierungs- und Simulationssprache mit der digitale Systeme beschrieben werden können. Dabei werden sowohl die Hardware- als auch die Softwarekomponenten in SystemC definiert. Das entstandene Modell kann durch eine bereitgestellte Laufzeitumgebung simuliert werden, um dessen Echtzeitverhalten zu beobachten.

Die Sprache ist aktuell im IEEE Standard 1666-2011 [24] informell definiert. Es gibt eine frei erhältliche Implementierung, welche von der Accellera Systems Initiative entwickelt wird. Diese ist als C++-Klassenbibliothek realisiert. Ein ANSI-konformer C++-Compiler kann ein gegebenes Design mit Hilfe der Bibliothek übersetzen. Das Ergebnis ist ein ausführbares Programm, welches die Laufzeitumgebung (den SystemC-Kernel) enthält und somit die Simulation unmittelbar ermöglicht.

2.1.1 Struktur von SystemC-Modellen

In SystemC werden die Hardware- und Softwarekompenenten eines Systems in Modulen strukturiert. Ein Modul hat einen Zustand (interne Variablen) und enthält *Ports*, um mit anderen Modulen zu kommunizieren. Zwei oder mehrere Module sind über jeweils einen Port miteinander verbunden. Ein *Interface* gibt vor, wie über einen solchen Port kommuniziert wird. Dafür wird eine Menge von abstrakten Methoden definiert. Ein *Channel* ist eine konkrete Implementierung eines Interfaces. Der Channel wird an die entsprechenden Ports gebunden und ermöglicht somit die Kommunikation zwischen Modulen. Abbildung 1 veranschaulicht den Informationsaustausch von Modulen in SystemC. Durch den Ansatz wird die Kommunikation (über Ports) sauber von den Berechnungen (innerhalb der Module) getrennt. Dies ermöglicht den einfachen Austausch einzelner Komponenten (z. B. eines Bus-

systems), ohne dafür die Struktur des gesamten Systems ändern zu müssen. Im Designprozess kann so auch mühelos evaluiert werden, ob eine Komponente eher als Hard- oder als Software realisiert werden sollte.

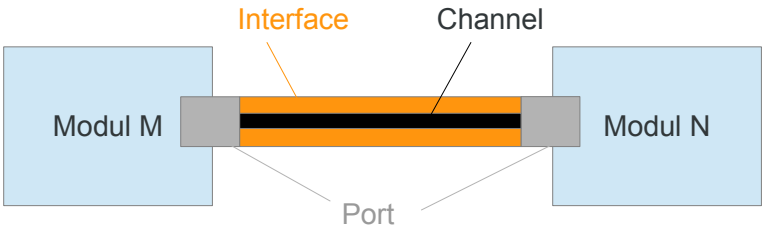


Abbildung 1: Modulkommunikation in SystemC

2.1.2 Verhalten von SystemC-Modellen

Module enthalten außerdem Prozesse, die während der Simulation nebenläufig ausgeführt werden. Der im SystemC-Kernel enthaltene Scheduler simuliert diese Nebenläufigkeit, führt die einzelnen Prozesse jedoch sequentiell aus. Ein Prozess wird in einer C++-Methode gekapselt. Man unterscheidet zwischen Methoden- und Thread-Prozessen. Ein Methoden-Prozess kann nicht unterbrochen werden und wird daher bei jedem Aufruf vollständig ausgeführt. Im Gegensatz dazu kann ein Thread-Prozess die Kontrolle wieder an den den Scheduler abgeben. Dies geschieht immer dann, wenn der Prozess auf ein bestimmtes *Event* wartet. Ein Event kann Zeitverlauf sein oder Kommunikation über einen Port. Prozesse können selber auch aktiv Events auslösen.

2.1.3 Ausführungssemantik

Der Scheduler kontrolliert Simulationszeit, steuert den Informationsaustausch und koordiniert die Prozessausführung. Um die Gleichzeitigkeit der Aktivitäten abzubilden, führt SystemC – vergleichbar mit VHDL[1] – sogenannte Deltazyklen ein. Ein Deltazyklus besteht aus einer *Evaluation*- und einer *Update*-Phase. Während der ersten Phase wird jeder rechenbereite Prozess solange ausgeführt, bis er blockiert (er erwartet Eingabedaten oder Zeitverlauf). Dabei werden Änderungen an Ports nicht direkt übertragen sondern solange vorgehalten, bis die zweite Phase beginnt. In dieser Phase werden (neue) Daten über die Kommunikationskanäle ausgetauscht, wodurch Prozesse wieder rechenbereit werden können. Beide Phasen wechseln sich so lange ab, bis alle Prozesse blockiert sind. Erst dann lässt

[1] Very High Speed Integrated Circuit Hardware Description Language, eine Hardwarebeschreibungssprache um elektronische Systeme textuell darzustellen

der Scheduler Simulationszeit vergehen und startet den nächsten Deltazyklus. Das Konzept eignet sich hervorragend um gleichzeitig stattfindende Ereignisse sequentiell abzubilden: In keiner der beiden Phasen vergeht Simulationszeit, sodass die Prozesse entweder alle „gleichzeitig“ rechnen oder Daten senden bzw. empfangen. Die Reihenfolge, in der Prozesse in der *Evaluation*-Phase ausgeführt werden, ist nicht definiert. Abbildung 2 veranschaulicht die beschriebenen Phasen. Die Simulation kann in beliebigen Zeiteinheiten durchgeführt werden und der Scheduler ist ebenfalls in der Lage, Zeitintervalle zu überspringen, wenn in diesen nichts passiert.

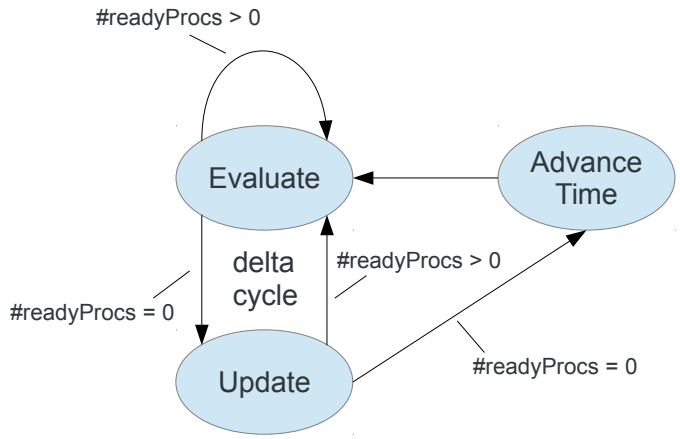


Abbildung 2: Deltazyklen und Zeitverlauf in SystemC

2.2 Speichermanipulationen in SystemC

In diesem Abschnitt werden kurz die wichtigsten speicherspezifischen Operationen in SystemC erläutert. Da SystemC die Programmiersprache C++ erweitert, erbt sie die entsprechende Funktionalität. Hierzu gehören im Wesentlichen das Verweisen (Referenzieren) auf bestimmte Speicherbereiche, der Zugriff auf referenzierte Objekte (Dereferenzieren) und das dynamische – d.h. während der Laufzeit – Erstellen und Entfernen von Objekten. Unter „Objekt“ verstehen wir im Folgenden einen manipulierbaren Speicherbereich.

2.2.1 Pointer und Referenzen

Um Objekte im Speicher zu adressieren, nutzt man einen *Pointer* (Zeiger). Ein „Objekt“ kann hierbei ein Datum (eine Variable) sein oder eine Funktion. Im Folgenden betrachten wir nur Datenobjekte und Pointer auf diese. Zu jedem Datentyp `T` gibt es einen entsprechenden Datentyp `T*`, der die Adresse eines Objekts dieses Typs speichert. Mit dem Konstrukt `T* ptr` wird ein Pointer deklariert, der

auf einen Speicherbereich vom Typ `T` verweist. Ein Pointer selbst belegt unabhängig von seinem Typ bei modernen Betriebssystemen immer einen gleich großen Speicherplatz, da der Wert des Pointers der Adresse des referenzierten Objekts entspricht. Ein Pointer kann während der Laufzeit auf verschiedene Adressen zeigen, sofern er nicht konstant ist. Pointer werden häufig bei Funktionsaufrufen genutzt, um Objektkopien zu vermeiden und steigern somit die Performanz von Programmen. Ferner werden sie für dynamische Speicheroperationen benötigt.

Der unäre Adressoperator `&` verbindet einen Pointer mit einem Objekt: `T* ptr = &object_of_type_T`. Um mit dem referenzierten Objekt selbst zu arbeiten, muss ein Pointer dereferenziert werden. Dies geschieht mit dem Derefenzierungsoperator `*` und ist analog zur Refenzierungsoperation: `T obj = *ptr_to_type_T`.

Wird ein Struct über einen Pointer adressiert und ein Feld dieses Structs soll ausgelesen werden, so wird der binäre Zugriffsoperator `->` genutzt. Der linke Operand ist dabei der Pointer und der rechte das Feld des vom Pointer referenzierten Structs. Ein typischer Ausdruck in diesem Zusammenhang ist die Zuweisung `ptr_to_struct->member_of_struct = value`. Das ist äquivalent zu `(*ptr_to_struct).member_of_struct = value`.

Pointer sind eine notorische Fehlerquelle. Da ein Pointer auf beliebige Adressen im Speicher zeigen kann, ist es durchaus möglich, dass ein Programm auf Bereiche zugreifen will, für die es keine Zugriffsrechte besitzt. In einem solchen Fall reagiert das Betriebssystem auf die Zugriffsverletzung mit einem sogenannten *Segmenentation Fault*, was meistens dazu führt, dass das Programm abstürzt. Das Problem tritt auch auf, wenn mit einem Pointer gearbeitet werden soll, dem niemals eine Adresse zugewiesen worden ist. Das Benutzen eines *Nullpointers* führt ebenfalls häufig zum Absturz.

Die oben genannten Speicheroperationen wurden bereits in C [26] eingeführt. C++ [36] erweitert die Möglichkeiten des Programmierers mit dem Konzept der Referenzen. Eine Referenz ist ein Alias für ein bereits vorhandenes Objekt. Sie ist kein Objekt im eigentlichen Sinne und besitzt weder eine eigene Adresse noch einen eigenen Wert. Der Compiler ersetzt im Programm alle Referenzen durch die Adresse des referenzierten Objekts. Der Ausdruck `T& ref = obj_of_type_T` deklariert die Referenz `ref` und initialisiert sie als Alias auf das Objekt `obj_of_type_T`. Eine Referenz in C++ entspricht im Gebrauch einem konstanten Pointer in C. Sie ist immer bindend und kann im Gegensatz zu Pointern nicht manipuliert werden.

In C++ nutzt man häufig *call-by-reference* um Objektkopien bei Funktionsaufrufen zu vermeiden. Wird ein Parameter als Referenz übergeben, belegt er keinen weiteren Speicherplatz und alle Operationen auf ihm werden tatsächlich auf dem

beim Funktionsaufruf angegebenen Objekt ausgeführt. Die zugehörige Funktionsdeklaration lautet `return_type func_name(T& param)`.

2.2.2 Arrays

Ein Array ist eine Datenstruktur, die eine Menge von Elementen des gleichen Typs enthält. Es belegt einen zusammenhängenden Bereich im Speicher, dessen Größe `size_of_one_element * array_length` entspricht. In C/C++ wird eine Arrayvariable als ein konstanter Pointer auf das erste Element des Arrays realisiert. Das bedeutet, dass jede Operation, die auf einem Array ausgeführt wird, auch durch einen Pointer realisiert werden kann. Dementsprechend ist der folgende Code-Ausschnitt korrekt:

```
1 int arr[] = {7, 23, 42};
2 int i = *arr; // arr == &arr[0] ~> i == 7
3 int *ptr = arr; // ptr == arr == &arr[0]
4 int j = ptr[1]; // j == 23
5 int k = *(arr + 2); // k == 42
6 int l = *(ptr + 2); // l == 42
```

Listing 1: Pointer und Arrays sind äquivalent

Eine typische Fehlerquelle bei Arrays ist der Zugriff mit Hilfe eines Index, der außerhalb der Grenzen des Arrays liegt. Greift man mit dem Index auf Speicherzellen vor oder nach dem Array zu (Pointerarithmetik erlaubt dies), so ist das Verhalten des Programms unvorhersehbar. Abhängig von den nachfolgenden Anweisungen kann das Programm sofort, zu einem späteren Zeitpunkt oder gar nicht abstürzen. Der Fehler ist in jedem Fall schwer zu lokalisieren, da das Verhalten je nach Speicherbelegung variiert.

2.2.3 Dynamische Objekte

Die Operatoren `new` und `delete` dienen dazu, während der Laufzeit ein Objekt zu erstellen bzw. zu löschen. Ein dynamisch erzeugtes Objekt befindet sich im Heap und nicht im Stack. Der Ausdruck `new T` erzeugt ein Objekt vom Typ `T` und ruft die Standardinitialisierung des Objekts auf. Mit `new T(init_values)` hingegen können dem Konstruktor Argumente übergeben werden. Der `new`-Operator gibt einen Pointer zurück, der die Adresse des erzeugten Objekts enthält. Der Ausdruck `delete ptr_to_dyn_obj` löscht ein Objekt im Heap. Handelt es sich um ein dynamisch alloziertes Array, so muss `delete [] dyn_arr` genutzt werden, um alle Speicherzellen korrekt freizugeben. Diese Operation darf nur auf

wirklich existierenden Objekten ausgeführt werden. Listing 2 zeigt eine mögliche Verwendung der Operatoren.

```
1  int* b; // three int pointer variables on the stack
2  int* c;
3  int* d;
4
5  b = new int(42); // create object on heap, call specific ctor
6  delete b; // delete from heap
7
8  c = new int; // create int object on the heap
9  *c = 23; // assign value to the chunk on the heap
10 delete(c); // delete from heap
11 delete(c); // double free attempt: throws a runtime error!
12
13 d = new int[4]; // create an int array on the heap
14 delete [] d; // delete whole array from heap
```

Listing 2: Die Operatoren `new` und `delete`

In C++ stehen ebenfalls die von C bekannten Operatoren `malloc` und `free` zur Verfügung. Sie reservieren Speicher im Heap bzw. geben ihn wieder frei. Von ihrer Benutzung ist in C++ allerdings abzuraten, weil sie mehrere Nachteile in sich bergen. Zum einen wird – im Gegensatz zu `new`/`delete` – der Konstruktor bzw. Dekonstruktor des Objektes nicht aufgerufen. Dies kann dazu führen, dass notwendige Ressourcen nicht reserviert bzw. freigegeben werden (*memory leak*). Zum anderen muss bei `malloc` eine Typumwandlung vorgenommen (die Funktion gibt `void*` zurück) und die zu reservierende Größe des Speicherbereichs explizit angegeben werden. Beides ist umständlich und nicht gut lesbar. Der `new`-Operator nimmt dem Programmierer diese Arbeiten ab.

2.3 Modelchecking

Modelchecking [10, 13] ist ein Verfahren, um zu verifizieren, ob ein gegebenes Modell M einer Eigenschaft P genügt. M ist dabei als Transitionssystem gegeben und muss zunächst vom eigentlich zu verifizierenden System S abgeleitet werden. P wird je nach Bedarf als logische Formel, regulärer Ausdruck oder Automat ausgedrückt, um eine Invariante oder eine Zustandseigenschaft anzugeben. Ein Modelchecker exploriert alle möglichen Pfade in M und prüft, ob P in jedem Zustand wahr ist.

Modelchecking ist *path sensitive*: Während der Verifizierung ist der aktuelle Pfad – und damit die Abfolge der Zustände – immer bekannt. Wird P verletzt, so kann

genau nachvollzogen werden, in welchen Kontext dies geschah. Das macht das Verfahren besonders interessant, um Fehler in Computerprogrammen zu finden (vgl. [7, 14]). Hinzu kommt, dass in nebenläufigen Systemen alle ebenfalls möglichen Verschränkungen in Betracht gezogen werden. Dadurch kann der Modelchecker Fehler identifizieren, die nur sehr schwer durch traditionelle Qualitätssicherung gefunden werden.

Ein bekannter Nachteil des Verfahren ist das Problem der Zustandsexplosion, das durch den kombinatorischen Aufwand bei der Verifizierung besteht. Dies kann dazu führen, dass der Rechner an die von der Hardware bestimmten Grenzen stößt und das Modell nicht verifizierbar ist. Weiterhin ist es nicht trivial, das Modell M aus dem System S zu generieren. In der Praxis wird S häufig durch M über- oder unterapproximiert, sodass das Ergebnis der Verifizierung von M nicht unbedingt auf S übertragbar ist.

2.4 UPPAAL Timed Automata

UPPAAL ist ein Werkzeug zur Modellierung, Simulation und Verifikation von Echtzeitsystemen. Ein UPPAAL-Modell besteht aus einem Netzwerk von *Timed Automata*. Die Verifikation geschieht durch einen integrierten Modelchecker. UPPAAL wird von den Universitäten Uppsala (Schweden) und Aalborg (Dänemark) gemeinsam entwickelt.

2.4.1 Timed Automata

Timed Automata [1, 25] sind endliche Automaten, die um eine zeitliche Dimension erweitert wurden. Dazu werden Uhren und Zeitbedingungen (*clock constraints*) eingeführt. Mit Hilfe von Timed Automata kann zeitabhängiges Verhalten gut modelliert werden. Der Zustand eines Timed Automaton entspricht dem Zustand (der *location*) des zugrundeliegenden, endlichen Automaten und dem aktuellen Zustand aller Uhren. Alle Uhren im System verlaufen synchron. Die Uhrzeit (einer bestimmten Uhr) ist ein reeller Wert, der ausgelesen werden kann, um beispielsweise die Transition zwischen Locations zu bedingen. Solch eine Bedingung an einer Kante des Automaten nennt man *guard*.

Timed Safety Automata [20] erweitern die ursprünglichen Timed Automata um lokale Invarianten. Mit diesen kann für eine Location definiert werden, wie lange sich der Automat in dieser aufhalten darf. Im Folgenden konzentrieren wir uns auf Timed Safety Automata und bezeichnen sie der Einfachheit halber als Timed Automata (TA).

Um ein komplexes System zu modellieren schaltet man eine Menge von Timed Automata zu einem Netzwerk zusammen. Dies ermöglicht die Beschreibung gleichzeitig stattfindender Berechnungen in einem Gesamtsystem. Die Automaten werden mittels spezieller Kommunikationskanäle verbunden. Ein TA sendet auf einem Kanal ein Signal, welches von einem anderen TA empfangen wird. Sind die Transitionen zweier TA über einen Kanal verbunden, so dürfen die Automaten nur gleichzeitig über diese Transitionen zur folgenden Location wechseln. Durch diese sogenannte Synchronisation können Abhängigkeiten im System sehr gut repräsentiert werden.

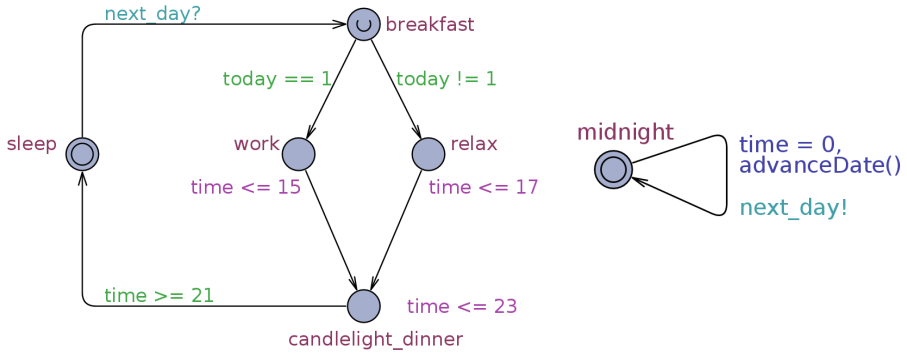
2.4.2 Besonderheiten von UPPAAL Timed Automata

UPPAAL erweitert das Konzept der Timed Automata in mehreren Punkten. Unter anderem führt UPPAAL parametrisierbare Templates ein, aus denen dann konkrete Automaten instanziiert und zu einem Netzwerk verbunden werden können. Die Templates besitzen lokale Variablen. Die TA kommunizieren über die bereits erwähnten Kanäle mit einander; sie können allerdings sowohl *binary* (1:1) als auch *broadcast channels* (1:n) nutzen. Innerhalb des Automaten besteht Zugriff auf lokale und globale Variablen. UPPAALs Beschreibungssprache führt bestimmte Datentypen ein: (bounded) Integer, Boolean, Structs und Arrays stehen für die Modellierung zur Verfügung. Variablen können mit einer C-artigen Sprache manipuliert werden. Außerdem ist es möglich, globale Methoden zu definieren, die auf diesen Variablen operieren. Diese bezeichnen wir als „native Methoden“. In UPPAAL kann eine Location als *urgent* (dringend) oder *committed* (verpflichtend) deklariert werden. Das bedeutet, dass an dieser Stelle keine Zeit vergehen darf. Committed Locations haben zusätzlich eine höhere Priorität und müssen deshalb bevorzugt werden.

Zur Veranschaulichung zeigt Abbildung 3 (S. 14) ein Modell mit zwei Timed Automata. Guards sind grün, Signalübertragungen türkis, Invarianten violett und Variablenzuweisungen bzw. Methodenaufrufe blau hinterlegt. Listing 3 zeigt die diesem System zugehörigen, globalen Variablen und Methoden.

Das Beispiel zeigt den Alltag eines Lebemanns: er geht nur montags (`today == 1`) arbeiten und ruht sich den Rest der Woche aus. Täglich geht er abends Essen. Er geht zwar nicht vor 21 Uhr schlafen (`time >= 21`), verlässt jedoch spätestens 23 Uhr das Restaurant (`time <= 23`). Er erfährt über den Kanal `next_day`, dass ein neuer Tag angebrochen ist. Der zweite TA, der Kalender, beginnt den neuen Tag, in dem er das genannte Signal schickt (`next_day!`), die Uhrzeit zurücksetzt (`time = 0`) und den neuen Wochentag berechnet (Aufruf der nativen Methode `advanceDate()`).

Da der Lebemann und der Kalender beide über den binären Kanal `next_day` verbunden sind, können sie nur gleichzeitig an den entsprechenden Transitionen schalten. D.h. der Lebemann kann nur von der Location `sleep` zur Location `breakfast` wechseln, wenn der Kalender auch schaltet (da er nur eine Location besitzt, bleibt der Kalender aber nach der Transition in derselben).



(a) ein Lebemann (b) der Kalender

Abbildung 3: UPPAAL-Beispielmodell

```
1  /* global declarations */
2  chan next_day; // binary communication channel
3  clock time; // clock variable
4  int[1,7] today = 7; // weekday, starts on sunday
5
6  /* global native methods */
7  void advanceDate() {
8    if (today == 7)
9      today = 1;
10   else
11     today++;
12 }
```

Listing 3: Globale Variablen und Methoden des Lebemann-Beispiels

2.4.3 Verifikation mit UPPAAL

UPPAAL besitzt einen eingebauten Modelchecker, der dazu verwendet werden kann, UPPAAL-Modelle automatisiert zu verifizieren. Die zu verifizierenden Eigenschaften werden als sogenannte Pfadformeln vom Benutzer vor der Verifikation beschrieben.

Tabelle 1: Pfadformeln in UPPAAL

Name	Pfadformel	Bedeutung
reachability	`E<>`ϕ	*Possibly*: Ausgehend vom Ursprungszustand gibt es einen Pfad, sodass ϕ schließlich gilt.
safety	`A[]`ϕ	*Invariantly*: ϕ gilt auf allen Pfaden.
	`E[]`ϕ	*Potentially always*: Es existiert ein Pfad, auf dem ϕ immer gilt.
liveness	`A<>`ϕ	*Eventually*: Auf jedem Pfad gilt ϕ irgendwann.
	ψ `-->` ϕ	*Leads to*: Wenn ψ gilt, dann gilt auch ϕ irgendwann.

UPPAALs Pfadformeln müssen in einer formalen, maschinenlesbaren Sprache, einer vereinfachten TCTL (*Time Computation Tree Logic*), definiert werden. Eine Pfadformel ist die Kombination aus einem Pfadquantor und einer Zustandsformel. Die vereinfachte TCTL erlaubt keine Verschachtelung von Pfadquantoren. Pfadquantoren beziehen sich nicht auf einen bestimmten Zustand, sondern auf Bäume von Zuständen (etwa *entlang aller Pfade gilt x* oder *es existiert ein Pfad, auf dem x gilt*). Zustandsformeln wiederum erlauben es, einfache Bedingungen wie *var == 42* aufzustellen oder aber zu überprüfen, ob sich ein Prozess (lies: instanzierte TA) in einer bestimmten Location aufhält. Einfache Zustandsformeln lassen sich mittels booleschen Operatoren zu komplexeren zusammensetzen. Eine Zustandsformel darf keine Auswirkungen auf das System haben (*side-effect free*).

UPPAAL unterstützt im Wesentlichen drei verschiedene Pfadformeln, die sich je nach Bedarf verfeinern lassen. Tabelle 1 erläutert diese näher. Die Variablen ϕ und ψ stehen für beliebige Zustandsformeln. Die Quantoren `E` und `A` stehen für die aus der Prädikatenlogik bekannten Existenzquantor $\exists$ (*exists*) bzw. Allquantor $\forall$ (*always*).

Zusätzlich zu den normalen Zustandsformeln gibt es noch eine besondere „Formel“ zur Erkennung von Deadlocks. Ein UPPAAL-Modell befindet sich in einem Deadlock, wenn keiner der TA schalten kann, unabhängig davon wieviel Zeit vergeht. Bei einem Echtzeitsystem ist es durchaus möglich, dass es niemals in einen Deadlock geraten darf, weswegen diese zusätzlich Abfrage sehr anwendungsorientiert ist. Mit `'A[] not deadlock'` lässt sich verifizieren, dass das System sich niemals in einem Deadlock befindet.

Möchte man im Lebemann-Beispiel untersuchen, ob dieser auch wirklich nur montags arbeitet, so lässt man die Eigenschaft `'A[] not (bonvivant.work && today != 1)'` verifizieren. Der Modelchecker prüft, ob es tatsächlich unmöglich ist (`A[] not`), dass der Lebemann arbeitet (`bonvivant.work`), aber der

Wochentag kein Montag ist (`today != 1`). Gleichermaßen überprüft die Eigenschaft `'E<> (bv.candlelight_dinner && time == 24)'`, ob die Möglichkeit besteht, dass unser Protagonist sein Abendprogramm manchmal etwas länger ausdehnt. Die erste Eigenschaft wird vom System erfüllt, die zweite nicht.

Eine nähere Beschreibung zu UPPAAL und weitere Beispiele findet man unter [2] und [15].

2.5 Transformation von SystemC nach UPPAAL

Es gibt bereits einen mehrfach verbesserten Ansatz, um ein SystemC-Modell nach UPPAAL Timed Automata zu übersetzen [21, 22, 27, 30, 31]. Die Transformation ermöglicht es, viele SystemC-Sprachelemente (Zuweisungen, Funktionsaufrufe, Pointer, Events, Sensitivitäten und den Wait-Notify-Mechanismus) in ein UPPAAL-Modell zu integrieren. An diesem lässt sich das ursprüngliche SystemC-Design noch sehr gut nachvollziehen. Dies erleichtert das Debuggen erheblich, denn man kann Fehler, die der Modelchecker bei der Verifikation gefunden hat, schnell nachvollziehen. Mit dem Werkzeug STATE wurde die Übersetzung automatisiert und erfolgreich an einem SystemC-Design aus der Industrie getestet. Im Folgenden gehen wir zunächst auf die Darstellung von SystemC-Konstrukten in UPPAAL ein. Anschließend erläutern wir das bisher in dieser Transformation eingesetzte Speichermodell.

2.5.1 SystemC-Designs in UPPAAL

Ein UPPAAL-Modell besteht aus einer Menge von TA-Templates, globalen Variablen und Methoden und konkreten Instanzen der Templates, die an globale Variablen gebunden werden können. Die Grundidee des Ansatzes zur Übersetzung besteht darin, jede Methode des SystemC-Designs in ein eigenes TA-Template zu überführen. Aus diesem Template werden dann so oft Instanzen erzeugt, wie es Prozesse gibt, die diese Methode benutzen. Dies ist deshalb nötig, weil ein Modul (in welchem die Methode definiert wurde) mehrere Prozesse kapseln kann. Diese könnten sich unter Umständen gleichzeitig an verschiedenen Stellen in derselben Methode befinden. Folglich benötigt man zwei semantisch identische TA, um die Gleichzeitigkeit der Ausführung abzubilden. Zudem gibt es spezielle Templates für den Scheduler, Events und primitive Kanäle.

Abbildung 4 zeigt die verschiedenen Kommunikationsmöglichkeiten der TA-Instanzen. Diese Kanäle sind als globale Variablen im UPPAAL-Model definiert. Der Scheduler nutzt `activate` um Prozesse zu starten. Diese wiederum geben die Kontrolle mit `deactivate` wieder ab. Mit den Kanälen `wait/notify` können

Prozesse mit Events interagieren. Die anderen Kanäle dienen dazu, Deltazyklen und das Voranschreiten der Simulationszeit abzubilden.

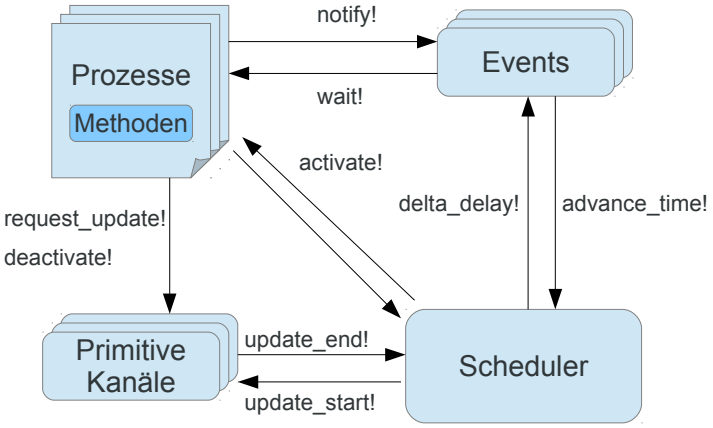
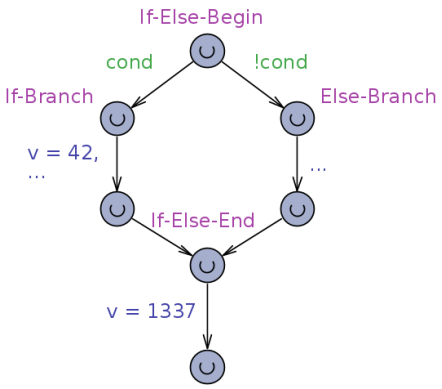


Abbildung 4: SytemC-Modell in UPPAAL Timed Automata (vgl. [27] S. 10)

Im Allgemeinen kann sequentieller C++/SystemC-Code direkt als sogenannte Update-Anweisung an die Transitionen im UPPAAL-Model geschrieben werden. Lediglich Funktionsaufrufe oder Kontrollstrukturen wie If-Else oder Schleifen müssen gesondert behandelt werden. Abbildung 5 veranschaulicht beispielhaft die Transformation einer If-Else-Kontrollstruktur in ein äquivalentes Automatenkonstrukt. Die If-Bedingung wird als Guard an die jeweilige Transition geschrieben, die Then- bzw. Else-Blöcke erhalten eigene Pfade im Automaten und die Variablenzuweisung wird in eine Update-Anweisung übersetzt.

```
if (cond) {
  v = 42;
  ...
} else {
  ...
}
v = 1337;
```

(a) SystemC



(b) UPPAAL

Abbildung 5: Eine If-Else-Verzweigung in SystemC und UPPAAL

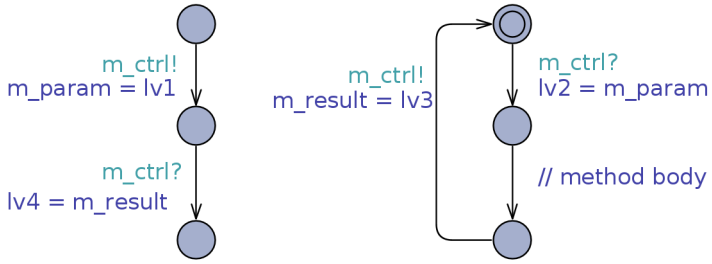


Abbildung 6: Methodenaufruf und Rücksprung in UPPAAL

Wird im ursprünglichen SystemC-Design eine Methode aufgerufen, so muss im UPPAAL-Modell von einer TA-Instanz in die nächste (und wieder zurück) gesprungen werden. Dafür wird für jede Methode ein Kommunikationskanal `ctrl` erstellt. Der Caller sendet ein Signal auf diesem Kanal und versetzt das TA-Template des Callees damit in die Lage, seinen Initialzustand zu verlassen. Am Ende der aufgerufenen Methode sendet nun wiederum der Callee ein Signal über `ctrl` und erlaubt dem Caller so, nach dem Funktionsaufruf weiter zu schalten. Weiterhin müssen die Argumente des Funktionsaufrufs übertragen werden. Dies geschieht mit Hilfe von globalen „Transportvariablen“, da diese die einzige Möglichkeit in UPPAAL darstellen, mit deren Hilfe TA-Templates Daten austauschen können. Dieses Konzept wird auf Abbildung 6 noch einmal deutlich. Der Caller (links) ruft den Callee (rechts) auf. Der Kanal `m_ctrl` steuert den (Rück-)Sprung. Die Transportvariablen `m_param` und `m_result` dienen dem Datenaustausch. `lv*` sind jeweils beliebige, lokale Variablen.

Näheres zu diesen Grundlagen findet man in [15] ab S. 24.

2.5.2 Speicherrepräsentation

Der Ansatz [27] realisiert Speicher in Form von typisierten Arrays. Dies entspricht nicht dem SystemC-Speichermodell, in dem Daten unterschiedlichen Typs nebeneinander residieren können. Diese Repräsentation ist allerdings notwendig, weil UPPAAL keine polymorphen Datentypen unterstützt. Demzufolge kann der Speicher nicht als monolithische Datenstruktur dargestellt werden. Für jeden im SystemC-Design vorkommenen Datentypen (ausgenommen SystemC-Spezifika wie beispielsweise Signale) wird während der Übersetzung das entsprechende Datentyparray erstellt. Jede Variable wird als Element im Datentyparray repräsentiert. D.h. überall dort, wo sie Verwendung findet, wird im UPPAAL-Modell der Wert des entsprechenden Elements ausgelesen. Pointer auf Variablen eines Typs entsprechen dem Index des zugehörigen Datentyparrays.

Abbildung 7 stellt das Speichermodell dem originären Modell von SystemC/C++-Speicher gegenüber. Links im Bild sieht man den normalen Funktionsstack, wie er in SystemC vorkommt. Rechts befindet sich die zugehörige UPPAAL-Übersetzung mit den beiden Datentyparrays für die Typen `int` und `data`. Sämtliche Variablen im SystemC-Modell werden zu Integern transformiert und dienen als Pointer auf die eigentlichen Werte, welche als Elemente im entsprechenden Datentyparray liegen. Deshalb zeigen im entstandenen Modell sowohl der Integer `i` als auch der Integerpointer `p` auf das gleiche Feld im Datentyparray. Der Unterschied wird in ihrer Benutzung deutlich: Anstelle von `i` wird beispielsweise an Transitionen `intMem[i]` benutzt, bei `p` ist das nicht der Fall.

Das Speichermodell ist nicht in der Lage dynamisch allozierten Speicher zu repräsentieren, denn es wird nicht zwischen Stack und Heap unterschieden. Jeglicher genutzter Speicher muss im Vorfeld bekannt sein und die Zuweisung von Speicherbereichen geschieht nicht während, sondern vor der Verifikation durch den Modelchecker. In SystemC/C++ werden statische Variablen erst bei ihrer Deklaration alloziert und dynamische Objekte erst beim Aufruf der entsprechenden Funktionen.

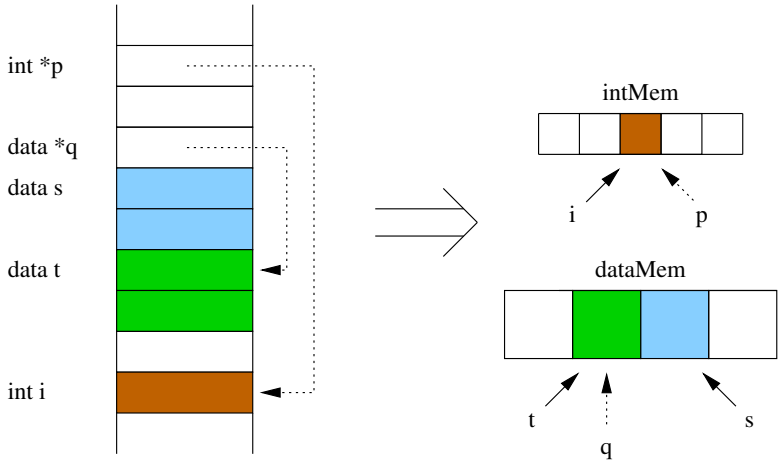


Abbildung 7: Speicher in SystemC (links) und UPPAAL Timed Automata (rechts); vgl. [27] S. 20.

3 Verwandte Arbeiten

In diesem Kapitel gehen wir auf verwandte Arbeiten ein, die sich mit der Formalisierung der SystemC-Semantik beschäftigen. Wir diskutieren ebenfalls Ansätze zur Modellierung des Speichers in C/C++ und dessen Verifizierung.

3.1 Formale Semantiken für SystemC

Verschiedene Ansätze versuchen, die inhärente SystemC-Semantik zu formalisieren. Ruf et al. [33] nutzen *Abstract State Machines* (ASMs) um eine formale SystemC-Semantik zu spezifizieren. Dabei wird die Interaktion des SystemC-Schedulers mit den Prozessen (Methoden und Threads) im System genau definiert. Dies erlaubt zwar die internen Abläufe während der Simulation abzubilden, nicht jedoch ein konkretes SystemC-Design zu verifizieren. Ähnlich wird in [18, 19] ein SystemC-Design in eine *Finite State Machine* (FSM) transformiert. Der entstandene Automat eignet sich theoretisch zum Modelchecking, bildet aber die Struktur des SystemC-Designs nicht mehr ab. In [16] wird ein weiterer Ansatz vorgestellt, um SystemC-Modelle formal zu verifizieren: Zeitliche Bedingungen werden in Hardware-Komponenten übersetzt und dem Design hinzugefügt. Dadurch kann das System sowohl virtuell (durch die Simulation mit SystemC) als auch real (durch Synthetisieren des Modells) verifiziert werden. In vergleichbarer Weise wird in [17] *Bounded Model Checking* (BMC) genutzt, um zeitliche Eigenschaften der Hard- und Software jeweils separat zu verifizieren. Beide Ansätze können nicht mit dynamischer Sensitivität bzw. mit Zeitgrenzen umgehen. Salem [34] beschreibt eine denotationelle Semantik für SystemC, jedoch nur für die synchrone Teilmenge der Sprache. Traulsen et al. [37] überführen ein SystemC/TLM-Modell nach Promella/Spin [23]. Diese ermöglicht es ihnen ebenfalls, Eigenschaften des Modells (Deadlockfreiheit und eigens gesetzte Assertions) zu verifizieren. Ihre Übersetzung ist nicht automatisiert und unterstützt nur TLM zur Kommunikation (d. h. keine nativen SystemC-Channels). Gleiches gilt für die in [29] vorgestellte Idee, um ein SystemC/TLM-Design zu formalisieren. Die Autoren transformieren das Modell in mehrere *Communation State Machines*. Das Verfahren modelliert einzelne SystemC-Module und den Scheduler als deterministische bzw. nichtdeterministische endliche Automaten.

Cimatti et al. [8, 9] übersetzen ein gegebenes SystemC-Design in ein sequentielles C-Programm, um es anschließend durch einen Modelchecker prüfen zu lassen. Das Scheduling ist dabei entweder direkt im sequentiellen Programm eingebettet oder

Teil des Modelchecking-Algorithmus. Dieser Ansatz unterstützt bis jetzt weder Structs noch Arrays noch Pointer.

Zhang et al. [38] stellen ebenfalls ein formales Modell, die *SystemC Waiting-State Automata*, vor, welches mit Hilfe von Modelchecking-Techniken verifiziert werden soll. Die Transformation in Automaten geschieht auf Delta-Zyklus-Level. Komplexe Prozessinteraktionen sowie Speicherverwaltung werden nicht berücksichtigt.

Keines der genannten Konzepte zur Formalisierung der Semantik sieht ein Speichermodell vor, daher können Pointer oder dynamische Speicherverwaltung nicht modelliert werden. Mit Ausnahme des in Abschnitt 2.5 besprochenen Ansatzes sind Kroening und Blanc [28, 3] unseres Wissens nach die Einzigen, die Speicher in SystemC-Designs nutzen können. Die Autoren transformieren ein gegebenes Design automatisch in eine Menge von Kripke-Strukturen, deren Labels ([atomare] Aussagen) die Berechnungen und deren Transitionen die Thread-Synchronisation modellieren. Eine Pointeranalyse erlaubt es ihnen für jeden Pointer zu bestimmen, auf welche Objekte er wann zeigen könnte. Durch diese Informationen kann die Derefenzierungsoperation aufgeschlüsselt werden. Spezielle Prädikate (an den Zuständen) bilden die Lebensspanne dynamisch erzeugter Objekte ab. Der Ansatz abstrahiert die Hardware und unterstützt weder Simulationszeit noch Interprozesskommunikation (via Channel oder Sockets).

3.2 Speichermodelle für C/C++

Cohen et al. [11, 12] etablieren ein typisiertes Speichermodell über dem (untypisierten) C-Speichermodell, in dem sie mittels zusätzlichen Invarianten sicherstellen, dass verschiedene Pointer nicht genutzt werden, um überlappende Objekte zu referenzieren. Sie unterstützen Pointerarithmetik und Zugriffe auf beliebige Speicherbereiche, aber keine dynamische Speicherallokation. Mit Hilfe dieses Ansatzes konnte der Microsoft Hypervisor verifiziert werden.

In [35] präsentieren Sinz et al. einen Ansatz, um mittels Bounded Model Checking C-Programme zu verifizieren. Sie modellieren den Speicher als Bytearray und nutzen einen *memory modification graph* um die einzelnen Zustände (die Speicherbelegung) während der Programmausführung verfolgen zu können. Die Autoren waren in der Lage, typische Fehler bei Speicheroperationen (*invalid free/double free*, *buffer overflow*, *memory leaks*) zu finden.

Chen et al. [6] nutzen je nach Bedarf zwei verschiedene Speichermodelle. Auf höherer Ebene, auf der die Korrektheit der Algorithmen bewiesen werden soll, benötigt man nur die Operationen *load* und *store*. Der Speicher kann also als Zuordnung von Adressen zu Werten realisiert werden (*mapPage*). Um den physischen Speicher genauer abbilden zu können, nutzen die Autoren ein anderes Speichermodell:

Jede Zelle im Speicher wird als Tupel repräsentiert, welches unter anderem ihre Adresse, ihren Wert, ihren Zustand (belegt/frei) und den Typ des Wertes enthält. Im Tupel gibt es ebenfalls einen Boolean, welcher Auskunft darüber gibt, ob es sich bei dieser Zelle um die *erste* Speicheradresse eines zusammengehörigen Blocks handelt (da beispielsweise ein `int` aus 4 Bytes besteht). Anhand dieses Indikators lässt sich prüfen, ob ein Pointer auf eine valide Adresse zugreift. Durch die anderen Informationen können wie in den vorherigen Ansätzen auch wieder ungültige Speicheroperationen wie das Auslesen nicht allozierter Zellen oder der Zugriff auf einen Wert falschen Typs erkannt werden.

Chens zweites Speichermodell erweitert den Heap von Reynolds [32]. Dessen Speichermodell basiert auf *separation logic* (einer Erweiterung der Hoare-Logik). Dabei wird der Heap in *heaplets* unterteilt, welche es ermöglichen, Invarianten über lokalen (d.h. nur einem Prozess zugänglichen) Datenstrukturen zu verifizieren. Mit Hilfe von Inferenzregeln können Aliasing-Bedingungen – ein Datum wird mehrmals referenziert – beschrieben werden.

Reynolds veranschaulicht seinen Ansatz an einer einfachen, imperativen Sprache. Alle anderen Ansätze konzentrieren sich auf C und unterstützen keine C++- bzw. SystemC-Spezifika. Wir benötigen ein Speichermodell für SystemC, welches es uns ermöglicht die wichtigsten Speicheroperationen in C++ abzubilden. Dazu nutzen wir das Burstall-Bornat-Modell [5, 4], welches einen Heap pro Datentyp vorsieht. Wir modellieren die Heaps durch Arrays und überwachen ebenfalls die Speicherbelegung der einzelnen Zellen.

4 Formales Speichermodell für SystemC

In diesem Kapitel erläutern wir zunächst die Anforderungen, die wir an ein zu transformierendes SystemC-Modell stellen. Anschließend formulieren wir die Grundlagen unseres Speichermodells, welches auf [27] basiert. Wir zeigen danach auf, wie wir SystemC-Sprachelemente übersetzen, um im resultierenden UPPAAL-Modell den Speicher repräsentieren zu können. Insbesondere gehen wir dabei darauf ein, wie es uns gelingt, dynamische Speicherallokation zu modellieren.

4.1 Anforderungen

SystemC basiert auf C++ und somit stehen alle in C++ vorhandenen Konstrukte zur Verfügung. UPPAAL ist weniger ausdrucksmächtig und kann daher nicht jedes SystemC-Design komplett modellieren. Daher stellen wir folgende Anforderungen an ein zu verifizierendes SystemC-Modell:

- Typumwandlung (*type casting*) findet nicht statt.
- `int`, `bool` und `structs` bzw. Arrays über diesen sind die einzigen Datentypen.
- Es gibt keine Funktionspointer.
- Rekursion wird nicht eingesetzt.
- Variablennamen sind immer eindeutig (d.h. sie werden in einem inneren Scope nicht überdeckt) und Methoden werden nicht überladen.
- Der Speicher wird nicht direkt adressiert.
  (`const int* io_ptr = 0x123abc`)
- Pointerarithmetik wird nur in Zusammenhang mit Arrays genutzt.
- Felder von `structs` werden nicht direkt adressiert.

Einige dieser Einschränkungen lassen sich durch einfache Maßnahmen beheben (Umbenennung, Refactoring, automatisiertes Preprocessing). Weitere Restriktionen betreffen Konstrukte, die in SystemC kaum verwendet werden (direkter Speicherzugriff, Funktionspointer). Insofern beeinträchtigen diese Anforderungen die Anwendbarkeit unseres Ansatzes nur wenig.

4.2 Speicherrepräsentation

Um die automatische Übersetzung eines SystemC-Designs nach UPPAAL Timed Automata zu ermöglichen, benötigen wir zunächst ein formales Speichermodell.

Die Verwendung von Arrays zur Modellierung von Speicher für die formale Verifikation ist ein gut untersuchter und weit verbreiteter Ansatz (vgl. [5, 4]). Klös [27] schlägt hierfür typisierte Heaps zur Modellierung vor. Pro Datentyp wurde ein sogenanntes „Datentyparray“ genutzt, welches alle Werte der Variablen dieses Typs enthält. Variablen selbst werden in Integer umgewandelt und repräsentieren den Index des Elements, welches ihren Wert enthält. Pointer werden ebenfalls so dargestellt. Allerdings wird für sie kein Speicherplatz im Datentyparray reserviert. Ein Pointer selbst ist nicht im Speichermodell repräsentiert. Wir erweitern diesen Ansatz und führen zusätzlich „Statusarrays“ ein, die es ermöglichen, während der Laufzeit neuen Speicher zu belegen.

4.2.1 Datentyparrays

Die Datentyparrays modellieren die Segmente des Speichers, die Daten enthalten. Der Speicher des Systems wird auf verschiedene Datentyparrays aufgeteilt, die jeweils Objekte desselben Typs enthalten. Klös [27] fordert, dass während des Programmablaufs Speicher weder reserviert noch freigegeben wird. Deshalb kann im Vorfeld statisch berechnet werden, wieviel Speicherplatz das Programm belegen wird und die Größe der Datentyparrays abgeleitet werden.

4.2.2 Statusarrays

Um dynamisch mit Speicher arbeiten zu können, muss die vom Betriebssystem bereitgestellte Funktionalität, welche Auskunft über die Belegung bestimmter Speicheradressen gibt, modelliert werden. Deshalb legen wir für jedes Datentyparray ein gleich langes „Statusarray“ an. Jedes Element dieses Statusarrays gibt an, ob das zugehörige Element im Datentyparray belegt ist oder nicht. Hierdurch kann zur Laufzeit Speicher „belegt“ werden: Zur Allokation wird ein freies Arrayelement lokalisiert, als belegt markiert und dessen Index wie zuvor als Speicheradresse genutzt. Die Deallokation verhält sich entsprechend gegensätzlich.

Eine einfache Unterscheidung zwischen freien und belegten Speicherzellen reicht jedoch nicht aus, um Arrays korrekt abzubilden. Die Länge eines Arrays wäre nicht am Statusarray abzulesen, da Speicherzellen „hinter“ dem Array auch belegt sein könnten. Weiterhin ist es möglich, dass die Größe des Arrays auch nicht zur Kompilierzeit bekannt ist (z.B. `int x = foo(); int buf[x];`). Die Länge

wird aber unter anderem benötigt, um ein Array wieder korrekt aus dem Speicher zu entfernen. Die Bestimmung der Größe von Arrays während der Laufzeit ist deshalb unabdingbar.

Unsere Lösung hierfür besteht darin, nicht nur zwischen „frei“ und „belegt“ zu unterscheiden, sondern auch die Belegungsart zu erfassen. Konkret bedeutet das, dass eine Speicherzelle im Statusarray verschiedene – intern durch Integer repräsentierte – Zustände annehmen kann. Diese geben Auskunft über die Art der Belegung:

- `MEM_FREE`: Adresse nicht belegt
- `MEM_OBJ`: Adresse durch ein einfaches Objekt belegt
- `MEM_ARR_HEAD`: Adresse von dem ersten Element eines Arrays belegt
- `MEM_ARR_TAIL`: Adresse von einem Folgeelement des Arrays belegt

Mit Hilfe dieser präziseren Unterscheidung ist das Freigeben eines Arrays nun sicher zu realisieren. Ein Head-Element und alle direkt folgenden Tail-Elemente repräsentieren das komplette Array.

Abbildung 8 veranschaulicht das Speichermodell anhand des Datentyps `int`. Das zugehörige Status- und Datentyparray nennen wir `intMemState` bzw. `intMem` in der Implementierung. Im Beispiel ist Platz für zwei Integervariablen im Speicher reserviert (`i1 = 7` und `i2 = 42`). Außerdem wurde ein Array angelegt und mit den Werten `[97, 98, 99]` belegt. Der Pointer `p` zeigt auf `i1`.

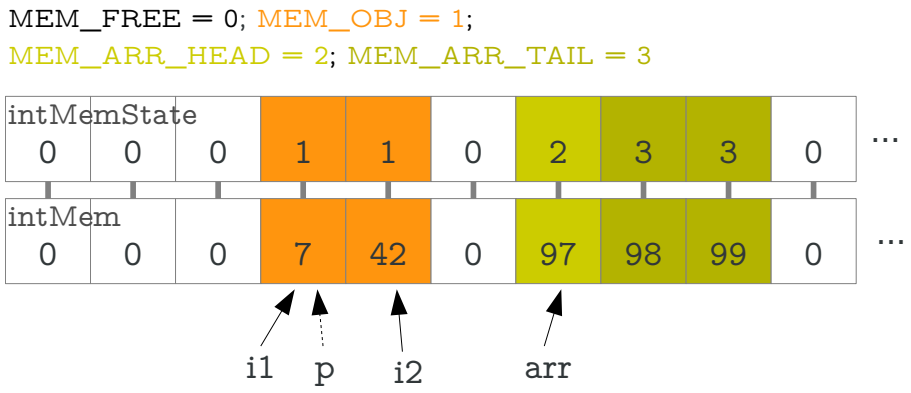


Abbildung 8: Mögliche Beispielspeicherbelegung für den Datentyp `int`

In der Abbildung ist ebenfalls zu sehen, dass sowohl Variablen als auch Pointer unabhängig von ihrem Typ als Integer modelliert werden, die auf ein Element im Status- bzw. Datentyparray zeigen. Diese Indizes bezeichnen wir als *Adresspointer*. Da zu jedem Datentyparray immer ein gleich langes Statusarray gehört, zeigen die Adresspointer immer auf zwei Elemente „gleichzeitig“. Im Folgenden wird daher je nach Situation davon gesprochen, dass ein Adresspointer auf ein bestimmtes Element im Status- oder Datentyparray zeigt.

Die Datentyp- und Statusarrays sind global im UPPAAL-Modell definiert. Für jeden Datentyp des SystemC-Modells gibt es die entsprechenden Arrays. Somit kann jeder TA zu jedem Zeitpunkt auf den Speicher „zugreifen“. Während der Übersetzung von SystemC nach UPPAAL Timed Automata ist nicht bekannt, wieviel Speicher das Programm belegen wird. UPPAAL unterstützt allerdings nur Arrays statischer Größe. Deshalb muss die benötigte Größe der Speicherarrays abgeschätzt werden. Unsere Herangehensweise hierfür erläutern wir in Kapitel 6.

4.2.3 Speicherreservierung

Ein C++-Programm kann Speicher entweder auf dem Stack oder auf dem Heap reservieren. Auf dem Stack werden unter anderem lokale Variablen gespeichert (statische Speicherallokation). Dies geschieht automatisch, sobald eine Variable deklariert wird. Sie werden am Ende ihres Gültigkeitsbereichs (Scopes) – ebenfalls automatisch – gelöscht. Der Heap steht für die dynamische Speicherreservierung zu Verfügung. Hierfür muss explizit Speicherplatz angefordert werden (`new`). Die Daten werden solange vorbehalten, bis das Programm sie wieder freigibt (`delete`).

Wir erweitern unser Speichermodell, um die Unterscheidung zwischen Stack und Heap zu erlauben. Dies erreichen wir dadurch, dass wir die drei Status, die Speicherbelegung repräsentieren (einfache Variable, Array-Head und Array-Tail), verfeinern. Wir verdoppeln ihre Anzahl, um festzuhalten, wo der Speicher belegt wurde. Wird ein einfaches Objekt deklariert, so setzen wir im Statusarray nicht mehr den Status `MEM_OBJ`, sondern `MEM_STATIC_OBJ` bzw. `MEM_DYNAMIC_OBJ`. Gleiches gilt für Array-Head/-Tail.

Desweiteren sorgen wir dafür, dass statische Variablen immer so weit wie möglich am Anfang und dynamische Daten am Ende des Datentyparrays residieren. Das hat den Vorteil, dass die Fragmentierung des Speichers vermindert wird: Der Funktionsstack wächst und schrumpft im vorderen Bereich des Datentyparrays, wohingegen der Heap unabhängig davon weiter hinten residiert. Im übrigen kann Fragmentierung nur beim Reservieren und Freigeben von Arrays entstehen, da alle anderen Variablen – aufgrund der Trennung nach Datentypen – jeweils nur eine Speicherzelle belegen. Die (flexible) Einteilung zwischen Stack und Heap im Datentyparray erleichtert es dem Designer zudem, Veränderungen im Speicher nachzuvollziehen.

Um Speicher im Modell zu reservieren, muss ein freies Element im Statusarray gefunden werden. Die lineare Suche beginnt am Anfang (Ende) bei statischer (dynamischer) Allokation. Zur Allokation eines Arrays, muss ein entsprechend großer, zusammenhängender Bereich gesucht werden. Die Speicherplatzbelegung wird im Statusarray dokumentiert. Wir implementieren die Suche als native UPPAAL-Methode, die die Adresse der allozierten Speicherzelle zurückgibt. Falls das Ob-

jekt initialisiert werden soll, kann der aus der Suche resultierende Index umgehend genutzt werden, um im Datentyparray den entsprechenden Wert zu schreiben.

Listing 4 zeigt beispielhaft die wichtigsten Methoden zum Erstellen von statischen Integer-Objekten in Pseudo-Code. Dynamische Objekte werden analog erstellt, mit dem Unterschied, dass die Suche einer freien Position am Ende des Statusarrays beginnt und der Typ der Belegung im Statusarray entsprechend angepasst wird. Wir generieren diese Methoden für jeden Datentyp automatisch und fügen sie dem UPPAAL-Modell hinzu.

```
1  // allocate an integer statically
2  int allocStatic_int() {
3    int pos = FindFreePosStatic_int();
4    if (pos == -1) {
5      // handle error
6    } else {
7      markAsUsedStatic_int(pos);
8    }
9    return pos;
10 }
11
12 // allocate and init an integer statically
13 int allocAndInitStatic_int(int value) {
14   int pos = allocStatic_int();
15   setValue_int(pos, value);
16   return pos;
17 }
18
19 // allocate an integer array statically
20 int allocStaticArr_int(int length) {
21   int pos = FindFreePosStaticArr_int(length);
22   if (pos == -1) {
23     // handle error
24   } else {
25     markAsUsedStaticArrHead_int(pos);
26     for(int i = pos + 1; i < (pos + length); i++) {
27       markAsUsedStaticArrTail_int(i);
28     }
29   }
30   return pos;
31 }
```

Listing 4: Native UPPAAL-Methoden zur statischen Speicherbelegung (abstrahiert)

4.2.4 Speicherfreigabe

Um ein Objekt aus dem Speicher zu löschen, ist es nötig, das zugehörige Element im Statusarray als frei zu markieren. Im Hinblick auf eine möglichst effiziente Verifikation setzen wir den Wert der Variable – gespeichert im Datentyparray – wieder auf den Standardwert des jeweiligen Typs zurück. Dadurch wird der Zustandsraum verkleinert. Das Zurücksetzen ist nicht kohärent mit SystemC/C++, wo Speicherzellen in ihrem aktuellen Zustand belassen werden. Dies ist allerdings akzeptabel, da der Zugriff auf eine freigegebene Speicheradresse ohnehin einen Fehler darstellt. Bei der Verifikation ist ein solches Problem leicht am Status der Speicherzelle abzulesen.

Die Speicherfreigabe realisieren wir ebenfalls durch native UPPAAL-Methoden. Die Methode zum Freigeben eines Arrays löscht das erste Arrayelement (gegeben durch seinen Adresspointer) und alle folgenden Elemente, deren Status sie als zugehörige Arrayelemente auszeichnen (`MEM_ARR_TAIL`). Beim Freigeben wird nun deutlich, warum es nötig ist zwischen Head und Tail des Arrays zu unterscheiden: Theoretisch könnten nämlich zwei Arrays direkt nebeneinander im Speicher residieren. Ohne die Differenzierung wäre nicht ersichtlich, an welcher Position das erste Array aufhört bzw. das zweite beginnt. Abbildung 9 (S. 29) veranschaulicht das Problem.

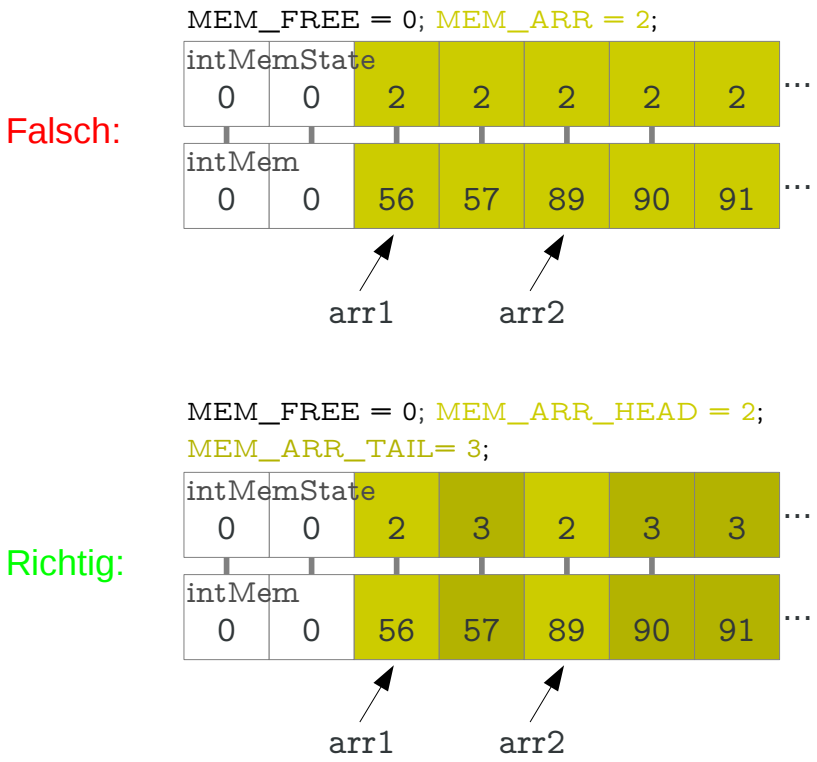


Abbildung 9: Verschiedene Statusbelegungen für Arrays (ohne Unterscheidung zwischen Stack/Heap)

Listing 5 zeigt die wichtigsten Methoden zum Löschen von Integer-Objekten in Pseudo-Code. Ob die Länge eines Arrays vorher bekannt war oder nicht ist in unserem Modell unerheblich, wie man an der Methode `deleteStaticArr_int()` beispielhaft sehen kann.

```
1  // delete an integer from stack
2  void deleteStatic_int(int pos) {
3    // check pos for errors & handle if needed
4    // ...
5    markAsUnused_int(pos);
6    // optimize for UPPAAL model checker
7    setDefaulValue_int(pos);
8  }
9
10 // delete int array from stack
11 void deleteStaticArr_int(int pos) {
12   deleteStatic_int(pos);
13   int i = pos + 1;
```

```
14    while(elemBelongsToArr_int(i)) {
15      deleteStatic_int(i);
16      i++;
17    }
18  }
19
20
21  // still an array?
22  boolean elemBelongsToArr_int(int pos) {
23    return (intMemState[pos] == MEM_STATIC_ARR_TAIL
24            or intMemState[pos] == MEM_DYN_ARR_TAIL);
25  }
```

Listing 5: Methoden zur Speicherfreigabe

Die genannten Methoden werden ebenfalls im UPPAAL-Modell für jeden Datentyp generiert. Sie können also von jedem TA genutzt werden, um Speicher freizugeben.

4.3 Transformation nach UPPAAL

Nachdem wir im vorherigen Abschnitt unser Speichermodell und die Zugriffsmöglichkeiten erläutert haben, zeigen wir nun, wie wir diese in die Transformation eines SystemC-Designs nach UPPAAL Timed Automata integrieren.

4.3.1 Übersetzungsregeln

Im Abschnitt 4.2 haben wir beschrieben, wie Speicher und Speicheroperationen in UPPAAL dargestellt werden können. Allerdings müssen alle SystemC-Ausdrücke bei der Transformation entsprechend angepasst werden, um das Speichermodell auch zu nutzen. Dies geschieht durch eine Menge von Übersetzungsregeln, die vorgeben, wie ein bestimmter Ausdruck in SystemC transformiert werden soll. Die Regeln sind nur für nicht weiter zerlegbare SystemC-Ausdrücke definiert und lassen sich ohne weiteres in einen automatisches Übersetzungwerkzeug einbauen. Jeder komplexe Ausdruck in SystemC wird rekursiv solange zerlegt, bis eine Regel angewandt werden kann. Die Regeln sind in Tabelle 2 aufgelistet und im Folgenden näher erläutert.

In der Tabelle stehen V, P, A für beliebige Datentypen. E und F entsprechen beliebigen Ausdrücken in SystemC, auf die die Übersetzungsregeln ebenfalls angewandt werden. [V|P|A]Mem repräsentieren die Datentyparrays des zugehörigen Typs. Beispielhaft steht v für eine normale Variable vom Typ V, p für einen Pointer auf ein Objekt vom Typ P und a für ein Array des Typs A.

Tabelle 2: Übersetzungsregeln

Nr.	SystemC		UPPAAL
	Deklarationen		
1	V v;	⇒	int v = allocStatic_V();
2	P* p;	⇒	int p = -1;
3	A a[E];	⇒	int a = allocStaticArr_A(E);
	Deklarationen mit Initialisierung		
4	V v = E;	⇒	int v = allocStatic_V(); VMem[v] = E;
5	P* p = E;	⇒	int p = E;
6	P* p = new P;	⇒	int p = allocDynamic_P();
7	P* p = new P(E);	⇒	int p = allocDynamic_P(); PMem[p] = E;
8	A a[] = {v_0, …,v_{n-1}};	⇒	int a = allocStaticArr_A(n); AMem[a+0] = v_0; …; AMem[a+(n-1)] = v_{n-1};
9	A* a = new A[E];	⇒	int a = allocDynamicArr_A(E);
	Zugriff und Dereferenzierung		
10	v	⇒	VMem[v]
11	&v	⇒	v
12	p	⇒	p
13	*p	⇒	PMem[p]
14	a[E]	⇒	AMem[a+E]
15	&a[E]	⇒	a+E
16	v.member	⇒	VMem[v].member
17	p→member	⇒	PMem[p].member
18	v.ptr→member	⇒	PMem[VMem[v].ptr].member
19	a[E].member	⇒	AMem[a+E].member
20	a[E].ptr→member	⇒	PMem[AMem[a+E].ptr].member
21	NULL	⇒	-1
	Zuweisungen		
22	v = E;	⇒	TMem[v] = E;
23	p = new P;	⇒	p = allocDynamic_P();
24	p = new P(E);	⇒	p = allocDynamic_P(); PMem[p] = E;
25	p = new P[E];	⇒	p = allocArrDynamic_P(E);
26	a[E] = F;	⇒	AMem[a+E] = F;
	Speicherfreigabe		
27	delete p	⇒	deleteDynamic_P(p);

weitere Regeln auf der nächsten Seite

Tabelle 2 – weitere Regeln

Nr.	SystemC		UPPAAL
28	delete [] a;	⇒	deleteDynamicArr_A(a);
29	/* am Ende eines Scopes */	⇒	deleteStatic_V(v_0); ...; deleteStatic_V(v_n);

Deklarationen Wir modellieren in unserem Modell eine Variable als Adresspointer auf ein Element des zugehörigen Datentyparrays. Folglich muss bei der Deklaration einer Variable der für sie nötige Speicherplatz reserviert und ein Adresspointer angelegt werden. Dafür nutzen wir die in Listing 4 (S. 27) eingeführten Methoden. Regel 1 gibt an, wie die Deklaration einer Variable übersetzt wird. Im UPPAAL-Modell entspricht der Adresspointer einem Integer, denn er zeigt auf ein Element im Datentyparray. Der Aufruf der Funktionen `allocStatic_T()` belegt den Speicher im Statusarray und gibt die Adresse (den Index) des Elements zurück.

Pointer werden nicht als „wirkliche“ Variablen, d.h. Variablen, die selbst Speicherplatz belegen, modelliert. Trotzdem muss ein entsprechender Adresspointer (als lokale Variable des TA) angelegt werden, wie in Regel 2 beschrieben. Pointer werden in unserer Darstellung mit -1 (s.a. Regel 21) initialisiert, also einer ungültigen Adresse. Hier weichen wir von SystemC/C++ ab, wo nicht initialisierte Pointer auf eine beliebige Adresse zeigen. Wird diese ausgelesen, so hat der Programmierer definitiv einen Fehler gemacht, was das Setzen einer ungültigen Adresse rechtfertigt. Alternativ kann einem Pointer während der Deklaration direkt ein Wert (d.h. eine Adresse) zugewiesen werden wie in Regel 5 ersichtlich. Arrays werden wie in C/C++ auch als Pointer dargestellt. Allerdings wird bei der Deklaration eines statischen Arrays sofort Speicherplatz belegt, sodass wir die entsprechende Methode zur Speicherbelegung aufrufen und dem Adresspointer direkt eine gültige Adresse zuweisen (Regel 3).

Regel 4 ist eine Erweiterung von Regel 1. Nachdem die Variable im Speicher reserviert wurde, wird ihre Adresse genutzt, um ihr einen Wert zu zuweisen. Gleiches gilt für ein Array mit Initialisierung (Regel 8).

Die Regeln 6, 7 und 9 zeigen, wie einfache Variablen und Arrays auf dem Heap alloziert werden. Wie bereits erwähnt, ist der Unterschied zum Stack die Art und Weise, auf welche nach freiem Speicher gesucht wird und welche Status den Speicherzellen zugewiesen wird.

Zugriff und Dereferenzierung An Regel 10 erkennt man, wie Variablen nach der Überführung in unser Speichermodell benutzt werden. Überall dort, wo der Wert der Variable ausgelesen werden soll, wird der zugehörige Adresspointer genutzt, um den Wert aus dem Datentyparray zu lesen. Die Adresse einer Variable entspricht ihrem Adresspointer. Regel 11 verdeutlicht dies. Ein Pointer wiederum ist gleichzusetzen mit seinem Adresspointer (Regel 12), da er selbst nicht im Speicher liegt. Wird der Pointer im SystemC-Code dereferenziert, so wird auf das Datentyparray zugegriffen (Regel 13).

Beim Zugriff auf ein bestimmtes Arrayelement wird der Adresspointers des Arrays als Offset genutzt, um das entsprechende Datum auszulesen. Gleiches gilt, wenn die Adresse eines Arrayelements benötigt wird (Regel 14 & 15).

Die Felder eines Structs besitzen keine eigenen Adresspointer, weil wir annehmen, dass sie niemals referenziert werden. Somit wird der Zugriff auf ein Feld übersetzt, indem mittels Adresspointer das Struct im Datentyparray gesucht und dann dessen Feld ausgelesen wird (Regel 16). Da in unserem Modell sowohl Variablen als auch Pointer zu Adresspointern werden, ist es unwichtig, wie auf ein Feld zugegriffen wird: Die Operatoren −> und . werden identisch behandelt. Daher ähneln sich die Regeln 16 und 17. Die Regeln 18 - 20 und 22 - 26 leiten sich alle aus den vorherigen her und ergeben sich bei der Übersetzung aufgrund der erwähnten Rekursion automatisch. Sie dienen lediglich zur Veranschaulichung.

Speicherfreigabe Die Regeln 27 und 28 zeigen, wie die in Listing 5 (S. 29) gezeigten Methoden genutzt werden, um einen Speicherbereich freizugeben.

Wir haben in Abschnitt 4.2.3 erwähnt, dass lokale Variablen am Ende ihres Gültigkeitsbereichs gelöscht werden. Dies ist in den meisten Fällen das Ende der Funktion, allerdings nicht immer. Beispielsweise steht eine Variable, die innerhalb einer Schleife definiert wurde, nach dem Austritt aus der Schleife nicht mehr zur Verfügung. Insofern müssen lokale Variablen nicht erst am Ende einer Funktion gelöscht werden, sondern am Ende ihres Scopes. Um sicherzustellen, dass der Stack korrekt bereinigt wird, löschen wir am Ende jedes Scopes alle Variablen, die im aktuellen Scope deklariert wurden (kohärent zu SystemC/C++). Deshalb rufen wir die native `deleteStatic_T()`-Methode für jede dieser Variablen auf. Sie ist nicht zu verwechseln mit dem C++-Operator `delete`, mit welcher Speicher im Heap freigeben wird (vgl. Listing 5, S. 29). Regel 29 ist keine Übersetzungsvorschrift im eigentlichen Sinne, gehört aber zu einer korrekten Transformation dazu.

4.3.2 Transformation von `sc_main()`

Die Methode `sc_main()` ist der Eintrittspunkt für die Simulation in SystemC. In ihr werden alle Top-Level-Modulinstanzen deklariert und die Simulation gestartet. Da Modulinstanzen nicht dynamisch erstellt werden dürfen, kann mittels statischer Code-Analyse untersucht werden, wieviele Modul-Instanzen im System vorhanden sind. Bei der Instanziierung eines Moduls werden alle Member des Moduls alloziert und der Konstruktor aufgerufen. Top-Level-Module können ihrerseits wieder Module enthalten, die dann ebenfalls instanziiert werden.

Die Simulation wird erst gestartet, wenn all Modulinstanzen generiert wurden. Um dieses Verhalten in UPPAAL abzubilden, erstellen wir einen speziellen TA, der sich nur um die Instanziierung aller Module kümmert. Dieses *SCMain-Template* führt zwei Schritte aus. Zuerst wird im Speichermodell für alle Member der Modulinstanzen, die im Speichermodell repräsentiert sind (d.h. beispielsweise nicht für Ports), Speicher alloziert. Das hat zur Folge, dass die ersten Elemente in den Datentyparrays meistens von Instanzmembern belegt sind. Der „Stack“ ist folglich zum Start der Simulation schon teilweise belegt.

Im zweiten Schritt werden – ausgehend von den Top-Level-Modulinstanzen – alle Konstruktoren aufgerufen. Die Konstruktoren sind als native Methoden in UPPAAL implementiert. Da im Konstruktor auf Member zugegriffen werden kann, wird dafür gesorgt, dass in solch einem Fall das entsprechende Element im Datentyparray genutzt wird. Die Zweiteilung erleichtert die Implementierung, da sauber zwischen Allokation (der Member) und Zuweisung (durch den Konstruktor) unterschieden wird. In SystemC kann dies zwar beides gleichzeitig geschehen, aber die Semantik wird durch unser Vorgehen nicht verändert, da beide Schritte direkt nacheinander ausgeführt werden.

Die letzte Transition des SCMain-Templates ist über einen binären Kanal mit der ersten Transition des Scheduler-Templates synchronisiert. Der Scheduler darf erst schalten, wenn alle Module instanziiert wurden. Dies ist analog zu `sc_main()`, in der erst nach der Moduldeklaration die Simulation durch den Aufruf der Methode `sc_start()` gestartet wird.

5 Formale Verifikation speicherbezogener Eigenschaften

Während der Benutzung verschiedener Speicheroperationen kann es immer wieder zu schwerwiegenden Fehlern kommen. Diese führen zu ungewolltem Verhalten (wenn Daten falsch ausgelesen werden) oder sogar zu seinem Absturz. Im Allgemeinen sind solche Fehler schwer zu reproduzieren, da sie sehr kontextabhängig sind. Außerdem hat man bei einem Programmabsturz praktisch keine Möglichkeit, den Ablauf im Nachhinein noch nachzuvollziehen.

Mit der Verifikation speicherbezogener Eigenschaften verfolgen wir deshalb mehrere Ziele. Sie muss automatisierbar sein, um mögliche Fehlerquellen auszuschließen und auch auf komplexere Systeme anwendbar zu sein. Alle Randfälle müssen betrachtet werden und beim Auftreten eines Fehlers muss ein entsprechendes Gegenbeispiel generiert werden können.

Im Folgenden beschreiben wir, welche Fehler auftreten und wie wir diese automatisiert erkennen können.

5.1 Speicherzugriff

Wird mittels Pointer auf den Speicher zugegriffen, muss sichergestellt sein, dass der Pointer auf eine valide Adresse zeigt. Zeigt der Pointer auf `NULL` oder ist die angegebene Adresse nicht reserviert, führt der Zugriff meistens zu einem Programmabsturz. Um sicherzustellen, dass dies nicht der Fall ist, setzen wir an jede Transition, in der auf den Speicher zugegriffen wird, einen Guard, der den Status der entsprechenden Speicherzelle prüft. Ist diese nicht belegt, kann der TA die Location nicht wechseln und das System läuft in einen Deadlock. Um das Modell auf einen solchen Fehler hin zu verifizieren, generieren wir die Eigenschaft `'E<> deadlock && (TA1.predefinedLocationX || TA2.predefinedLocationY || ...)'`. Damit stellen wir sicher, dass sich das System nur dann in einem Deadlock befindet, wenn einer der TA nicht schalten konnte. In diesem Fall soll auf eine invalide Speicheradresse zugegriffen werden und der Designer kann anhand des von UPPAAL generierten Gegenbeispiels (dem Pfad, der zum aktuellen Deadlock führte) nachvollziehen, wie dies geschah. Abbildung 10 zeigt beispielhaft einen solchen Deadlock-Guard.

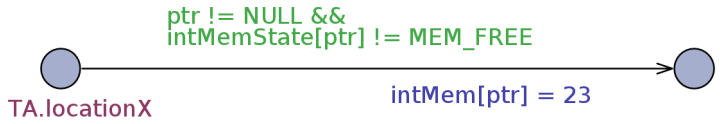


Abbildung 10: Automatisch generierter Guard für die Anweisung `*ptr = 23;`

5.2 Speicherfreigabe

Explizite Speicherfreigabe ist ebenfalls fehleranfällig. Hier können wir drei verschiedene Probleme erkennen.

Eine typische Fehlerquelle ist die Verwechslung der Ausdrücke `delete ptr` und `delete [] ptr`. Die erste Anweisung löscht ein einfaches Objekt im Heap, auf welches `ptr` zeigt, die zweite ein referenziertes Array. Werden beide Ausdrücke verwechselt, wird der Speicher nicht richtig bereinigt. Zum Zeitpunkt der Kompilierung ist nicht feststellbar, worauf der Pointer zeigt und der Compiler kann deshalb keine Warnung ausgeben. Um diesen Fehler zu erkennen, nutzen wir das global definierte Flag `MEM_ERR_INVALID_DELETE`. Beim Aufruf der Methoden `deleteDynamic_T()` und `deleteDynamicArr_T()` wird geprüft, ob sich laut zugehörigem Statusarray an der angegebenen Position ein einfaches Objekt bzw. ein Array befindet. Je nachdem welche Methode mit welcher Speicheradresse aufgerufen wird, kann das Flag gesetzt werden. Die zugehörige UPPAAL-Eigenschaft lautet `'A[] not MEM_ERR_INVALID_DELETE'` (d.h. der Fehler tritt niemals auf).

An diesem Punkt prüfen wir ebenfalls, ob eine mehrfache Deallokation (*double free*) auftritt. Dies geschieht, wenn der `delete`-Operator für eine Adresse aufgerufen, die bereits freigegeben wurde. Das führt häufig zu Programmabstürzen (*segmentation fault*) und/oder stellt eine Sicherheitslücke da. Ist die zu löschende Adresse bereits freigegeben, setzen wir das ebenfalls globale Flag `MEM_ERR_DOUBLE_FREE` und verifizieren entsprechend mit `'A[] not MEM_ERR_DOUBLE_FREE'`.

Als dritte, mögliche Fehlerquelle erkennen wir ebenfalls den Aufruf der `delete`-Operation auf eine Variable, die sich im Stack befindet. Der Operator `delete` darf nur auf Objekte aufgerufen werden, die vorher durch `new` im Heap erstellt wurden. Versucht das Programm, ein Objekt im Stack explizit zu löschen, hat dies fast immer Systemabstürze zur Folge. In solch einem Fall setzen wir das Flag `MEM_ERR_DELETE_ON_STATIC_VAR` und verifizieren wie üblich.

5.3 Arrayzugriff

Ein typischer Fehler in SystemC/C++ ist der Zugriff auf Arrayelemente mittels eines ungültigen Index (z.B. `int arr[3]; arr[5] = 23;`). In den allermeisten Fällen befinden sich an dieser Stelle im Speicher reservierte Daten, sodass das Programm zunächst ungehindert auf die entsprechende Adresse zugreift. Diese führt allerdings zu unbeabsichtigtem Verhalten. Da wir in unserem Speichermodell erkennen, welche Speicherzellen zu einem Array gehören, lässt sich verifizieren, ob der Index außerhalb der Arraygrenzen liegt. Hierfür prüfen wir zunächst, ob der Arraypointer selbst auf ein Head-Element zeigt und dann, ob alle Elemente zwischen dem Head und dem angegebenen Index Tail-Elemente sind. Das Prüfen aller Zwischenelemente ist nötig, um sicherzugehen, dass nicht auf ein weiter hinten im Speicher befindliches Array zugegriffen wird. Diese Fehlererkennung haben wir als native UPPAAL-Methode implementiert und nutzen sie ebenfalls als Guard. Abbildung 11 und Listing 6 veranschaulichen die Vorgehensweise.

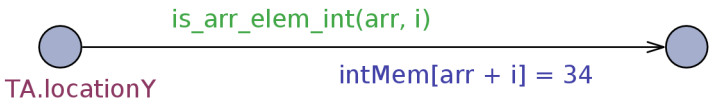


Abbildung 11: Automatisch generierter Guard für die Anweisung `arr[i] = 34;`

```
1  bool is_arr_elem_int(int head, int index) {
2    if (intMemState[head] != MEM_ARR_HEAD) {
3      return false;
4    }
5    if (index < 0) { // assuming arrays grow leftward
6      return false;
7    }
8    for (int i = head + 1; i <= head + index; i++) {
9      if (intMemState[i] != MEM_ARR_TAIL) {
10       return false;
11     }
12   }
13   return true;
14 }
```

Listing 6: Guard-Methode zum Prüfen eines Arrayelements für den Datentyp `int` (ohne Stack/Heap-Unterscheidung)

Die Generierung der beschriebenen Guards geschieht automatisch. Somit stellen wir sicher, dass kein Speicherzugriff übersehen wird.

Die vorgestellten Methoden zeigen, wie wir unsere Speicherrepräsentation nutzen können, um speicherbezogene Eigenschaften formal zu verifizieren. Dabei greifen wir auf intrinsische Eigenschaften des Speichermodells und UPPAALs eingebaute Funktionalitäten zurück. Die Methodik ist verständlich und lässt sich in Zukunft erweitern.

6 Implementierung

Um unseren Ansatz auf verschiedene SystemC-Modelle anwenden zu können, haben wir die bereits existierende Pipeline [27] angepasst. Abbildung 12 veranschaulicht grob alle Schritte, die benötigt werden, um ein gegebenes SystemC-Modell in UPPAAL verifizieren zu können. Weiße Kästchen repräsentieren jeweils Zwischenergebnisse und blaue Kästchen die Werkzeuge, die diese generieren oder nutzen. An den violett markierten Bereichen haben wir unsere Änderungen vorgenommen.

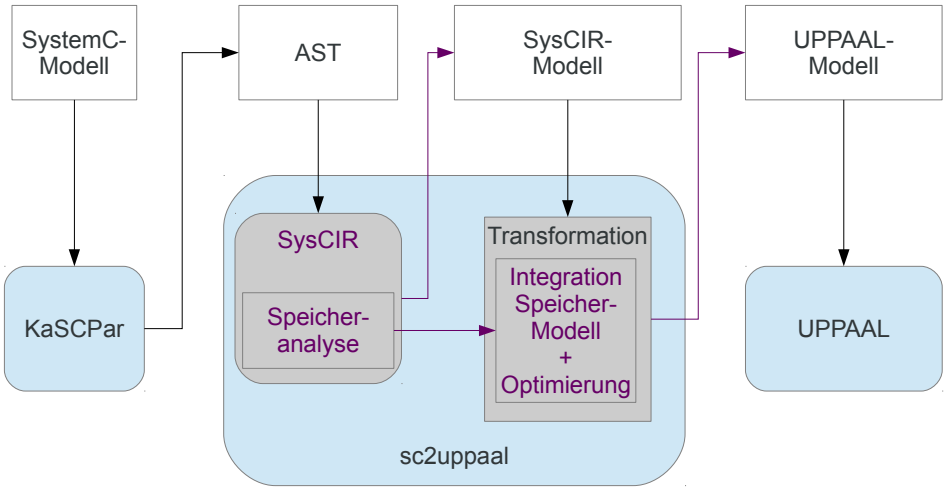


Abbildung 12: Die gesamte Pipeline, um ein SystemC-Modell in UPPAAL zu verifizieren.

Wir nutzen zunächst ein externes Werkzeug, den an der Universität Karlsruhe entwickelten *Karlsruhe SystemC Parser* (KaSCPar), um einen *Abstract Syntax Tree* (AST) vom zu untersuchenden SystemC-Design zu erstellen. Mit Hilfe des AST erstellen wir eine Zwischenrepräsentation (*SystemC Intermediate Representation*), welche das Modell in der Programmiersprache Java abbildet. Diese ermöglicht es uns unter anderem den statischen Speicherverbrauch zu analysieren. Ausgehend von der SysCIR generieren wir das endgültige UPPAAL-Modell. Beide Schritte sind im Werkzeug *sc2uppaal* gekapselt.

Die SysCIR wird am Lehrstuhl Programmierung Eingebetteter Systeme entwickelt. Hier waren verschiedene Anpassungen nötig, um etwa die C++-Operatoren `new` und `delete` abbilden zu können. Der Großteil des Implementierungaufwands lag in der Integration des Speichermodells in das UPPAAL-Modell. Dazu gehörte zum einen die Bereitstellung der nötigen Infrastruktur (die globalen Datentyparrays und andere Variablen, die Methoden zum Allozieren/Freigeben des Speichers, etc.),

zum anderen die automatische Transformation der SystemC-Ausdrücke nach den in Kapitel 4 beschriebenen Übersetzungsregeln.

Sowohl die SysCIR-Implementierung als auch sc2uppaal sind in Java implementiert, was sie plattformunabhängig macht. Die Integration neuer SystemC/C++-Ausdrücke in die SysCIR erforderte ca. 1200 Programmzeilen (LOC). Das Werkzeug sc2uppaal umfasst ungefähr 12000 LOC, wovon unsere Änderungen gut 2500 Zeilen ausmachen, die sich auf 20 Klassen verteilen.

Im Folgenden werden einige interessante Aspekte der Implementierung geschildert.

6.1 Analyse des Speicherverbrauchs

Um die Größe der Datentyparrays festzulegen, ist es erforderlich, den benötigten Speicheraufwand abzuschätzen. Ist das Array zu klein, tritt ein Fehler bei der Speicherreservierung auf. Ist es zu groß, erhöht dies die Größe jedes semantischen Zustands, was zu einem höheren Speicherverbrauch bei der Verifikation führt. Allerdings lässt sich der Gesamtverbrauch des statisch und dynamisch reservierten Speichers nur schwer erfassen.

Den statischen Speicherverbrauch können wir aber grob abschätzen. Eine Modulinstanz benötigt zunächst einmal so viel Speicher, wie ihre Member benötigen. Hinzu kommt der Verbrauch während der Ausführung der Modulmethoden. Im schlimmsten Fall liegen alle Methoden je einmal auf dem Stack (wir gehen davon aus, dass zwei oder mehr Methoden sich nicht gegenseitig aufrufen und erlauben auch keine Rekursion). Da ein Modul mehrere Prozesse kapselt und Prozesse sich gleichzeitig in den gleichen Methoden aufhalten können, muss der Verbrauch aller zeitverbrauchenden Methoden mit der Anzahl der Prozesse multipliziert werden. Nimmt man zu den Berechnungen des Modulspeicherverbrauchs noch den Verbrauch globaler Variablen und Funktionen hinzu, so erhält man eine akzeptable Abschätzung des statischen Speicherverbrauchs.

Der so berechnete Verbrauch wird in vielen Fällen überdimensioniert sein, bietet aber einen guten Ausgangspunkt für die Verifizierung. Außerdem kann es durchaus passieren, dass man den Verbrauch von Arrays nicht abschätzen kann. Ist die Größe des Arrays nicht konstant (etwa `int arr[x]`), so kann man dessen Speicherverbrauch nur umständlich durch statische Codeanalyse ermitteln. Eine im Vorfeld eher pessimistische Annahme über den statischen Speicherverbrauch, kann sich in diesem Fall durchaus bewahrheiten.

Hinzu kommt, dass der dynamische Speicherverbrauch noch schwieriger abzuschätzen ist. Zwar kann man aus einfachen Ausdrücken wie etwa `int* p = new int[3]` leicht den allozierten Speicher ablesen. Aber dieser könnte sich beispiels-

weise in einer Schleife befinden (und die zugehörigen `delete`-Aufrufe an anderer Stelle), sodass die Größe des Heaps extrem schwierig vorhersagbar ist.

Um das Problem der Speichergröße zu lösen, nutzen wir ein Script. Ausgehend von dem berechneten statischen Verbrauch zuzüglich der Auswertung des einfach zu berechnenden, dynamischen Verbrauchs, versucht das Script, das Modell durch UPPAAL verifizieren zu lassen. Kommt es zu einem Fehler bei der Speicherallokation, so wird das entsprechende Datentyparray um einen gewissen Prozentsatz vergrößert und die Verifikation noch einmal durchlaufen. Dies geschieht solange, bis keine Speicherfehler mehr auftreten oder zu viele Versuche gescheitert sind. Im zweiten Fall könnte die Ursache ein Speicherleck sein. Eine manuelle Untersuchung ist an diesem Punkt nötig. Wir konnten das Script für kleinere Beispielprogrammen erfolgreich einsetzen.

6.2 Optimierungen

Eines unserer Ziele ist ein möglichst geringer, durch das Speichermodell produzierter Overhead, um auch die Verifikation komplexerer System zu ermöglichen. In diesem Abschnitt gehen wir kurz auf Optimierungsstrategien ein, die wir dafür genutzt haben.

Wir nutzen UPPAALs Bounded Integer aus, um die Adresspointer so weit wie möglich einzuschränken. Tatsächlich kann ein Adresspointer nur Werte zwischen -1 (der ungültigen Adresse) und dem Index des letzten Elements des jeweiligen Datentyparrays annehmen. Deshalb erstellen wir in UPPAAL einen nativen Datentyp für die Adresspointer jedes im SystemC-Modell vorkommenen Typs. Wenn das Datentyparray des Typs `int` beispielsweise 100 Elemente umfasst, so definieren wir den Adresspointer für Integer als `typedef int[-1,99] int_addr_ptr` (d.h. eine Variable dieses Typs kann nur Werte zwischen -1 und 99 annehmen). Dies ermöglicht es UPPAAL Optimierungen vorzunehmen, die den Speicherverbrauch pro semantischem Zustand verringern.

Desweiteren repräsentieren wir nur diejenigen Variablen in unserem Speichermodell, für die es nötig ist. Eine Variable, deren Adresse niemals ausgelesen wird (mittels Referenzierung durch einen Pointer oder als Call-By-Reference-Parameter), wird nicht als Adresspointer, sondern als klassischer Integer- bzw. Boolean im UPPAAL-Modell realisiert. Durch diese Optimierung erreichen wir, dass die Status- und Datentyparrays von geringerer Größe sind und folglich der Speicherverbrauch des Modelcheckers verkleinert wird. Wir stützen uns bei der benötigten Analyse auf den in [27] auf S. 24 vorgestellten Algorithmus.

Wir nutzen ebenfalls die in [15] und [27] vorgestellten Optimierungen, um beispielsweise die Anzahl der Locations eines Timed Automaton zu reduzieren oder

die Werte aller Variablen – wann immer möglich – auf einen vordefinierten Standardwert zu setzen.

Eine weitere, mögliche Optimierung, die wir noch nicht implementiert haben, wurde bereits in [31] aufgezeigt: SystemC-Methoden, in denen keine Zeit vergeht, müssen nicht als Timed Automata modelliert werden, sondern können als native UPPAAL-Methoden das Modell verkleinern. Dies verringert die Anzahl der möglichen Zustände wesentlich, ist jedoch nur möglich, wenn innerhalb des SystemC-Moduls eine saubere Trennung zwischen Kommunikations- und Berechnungsmethoden vorgenommen wurde.

6.3 Qualitätssicherung

Um sicher zu gehen, dass die Übersetzungsregeln korrekt implementiert wurden, haben wir mehrere Maßnahmen getroffen. Dazu gehören unter anderem die manuelle Überprüfung jeder einzelnen Regel anhand eines entsprechenden Beispiels. Aus diesen Beispielen wurden über 30 Testfälle abgeleitet, die wir mittels des JUnit-Frameworks automatisiert haben. Die Größe der Testfälle variiert zwischen 10 und 200 Programmzeilen und umfasst ca. 2500 Zeilen insgesamt. In jedem Testfall wird das UPPAAL-Modell zunächst auf korrekte Syntax überprüft, um danach die eigentlichen Speicheroperationen zu verifizieren.

Die Testfälle umfassen unter anderem den Einsatz von Pointern als lokale Variablen, Parameter und Rückgabewerte, dynamische Speicherreservierung und -freigabe, Referenzen und Dereferenzierungen, Arrayzugriffe und hierarchische Modulkompositionen. Da wir teilweise auf einer bereits vorhandenen Implementierung aufgebaut haben, übernahmen wir auch vorhandene Testfälle, um typische SystemC/C++-Funktionalitäten (Kontrollstrukturen, Methodenaufrufe, Intermodulkommunikation, etc.) abzubilden und sicherzustellen, dass unsere Erweiterung die ursprüngliche Funktionalität nicht einschränkt.

7 Experimentelle Evaluierung

In diesem Kapitel präsentieren wir die von uns zur Evaluierung unseres Ansatzes durchgeführten Experimente. Dazu haben wir ein klassisches Producer/Consumer-Beispiel als Fallstudie genutzt. Wir vergleichen unsere Ergebnisse mit der Implementierung von Klös [27]. Die Experimente wurden auf einer 64-Bit Linuxmaschine mit 3-GHz-Prozessor (4-Kern) und 4Gb Arbeitsspeicher durchgeführt.

7.1 Producer/Consumer-Beispiel

Zur Evaluierung verwenden wir ein SystemC-Standardbeispiel. Dieses besteht aus zwei Modulen (*Producer* und *Consumer*), die über eine Fifo miteinander kommunizieren. Der Producer generiert je ein Produkt und schickt es an den Consumer. Dieser schnürt aus mehreren erhaltenen Produkten jeweils ein Paket. Unser Modell unterscheidet sich leicht von der bei SystemC mitgelieferten Beispielimplementierung, um dynamische Speicherverwaltung einbinden zu können.

Das SystemC-Modell umfasst insgesamt knapp 200 Zeilen. Bei der Transformation ensteht ein UPPAAL-Modell mit 8 TA-Templates, die die Funktionalitäten der Producer- und Consumermodule abbilden und 11 weiteren Templates, die für die Infrastruktur (Synchronisation, Fifo, Initialisierung, etc.) benötigt werden.

Um unseren Ansatz mit Klös [27] vergleichen zu können, haben wir das Beispiel in einem weiteren Schritt so modifiziert, dass keine dynamischen Speicheroperationen mehr vorkommen.

Die Listings 7 und 8 zeigen die beiden Module des Modells. In rot ist jeweils die statische Version der Programmzeilen dargestellt, die wir angepasst haben.

```
1  ////////////////////////////////////////////////////////////////
2  % dynamic version                          % static version
3  ////////////////////////////////////////////////////////////////
4  #define MIN_PRDCT_SIZE 1
5  #define MAX_PRDCT_SIZE 10
6
7  SC_MODULE(producer) {
8    sc_port<sc_clock> p_clk;
9    sc_fifo_out<int> fifo;
10   int curPrdctSize; // current product size
11
12   int* produce() {                          % int produce() {
13     sc_time t = sc_time(curPrdctSize * 100, SC_NS);
14     wait(t);
15     int* p = new int(curPrdctSize);         % int p = curPrdctSize;
16     curPrdctSize++;
17     if (curPrdctSize > MAX_PRDCT_SIZE) {
18       curPrdctSize = MIN_PRDCT_SIZE;
19     }
20     return p;
21   }
22   void dissassemble(int* product) {
23     delete product;                         % /* do nothing */
24   }
25   void main_method(void) {
26     int* product = NULL;                    % int product = -1;
27     while(true) {
28       product = produce();
29       fifo->write(*product);                % fifo->write(product);
30       dissassemble(product);                % dissassemble(&product);
31     }
32   }
33
34   SC_HAS_PROCESS(producer);
35   producer(sc_module_name name)
36   {
37     curPrdctSize = MIN_PRDCT_SIZE;
38     SC_THREAD(main_method);
39     sensitive << p_clk;
40   }
41 };
```

Listing 7: Die Methoden des Producer-Moduls (in rot die statische Version)

```
1  ////////////////////////////////////////////////////////////////
2  % dynamic version                          % static version
3  ////////////////////////////////////////////////////////////////
4  #define BUCKET_SIZE 3
5  SC_MODULE(consumer) {
6    sc_fifo_in<int> fifo;
7    int* bucket;                             % int bucket[BUCKET_SIZE];
8    int curNbrProducts; // current # of products
9
10   void new_bucket() {
11     if (bucket != NULL) {                  % /* do nothing */
12       delete [] bucket;                    % /* do nothing */
13     }                                      % /* do nothing */
14     bucket = new int[BUCKET_SIZE];         % /* do nothing */
15     for (int i = 0; i < BUCKET_SIZE; i++) {
16       bucket[i] = -1;
17     }
18     curNbrProducts = 0;
19   }
20   void consume(int &product) {
21     bucket[curNbrProducts] = product;
22     curNbrProducts++;
23     if (curNbrProducts == BUCKET_SIZE) {
24       new_bucket();
25     }
26   }
27   void main_method(void)
28   {
29     int product = 0;
30     bucket = NULL;                         % /* do nothing */
31     new_bucket();
32     while(true)
33     {
34       product = fifo->read();
35       consume(product);
36     }
37   }
38   SC_CTOR(consumer) {
39     SC_THREAD(main_method);
40   }
41 };
```

Listing 8: Die Methoden des Consumer-Moduls (in rot die statische Version)

Tabelle 3 (S. 46) zeigt die Ergebnisse der Experimente. Wir haben jeweils 1, 2 bzw. 3 Producer und Consumer instanziiert, die unabhängig von einander kommunizieren.

Kommuniziert nur ein Producer mit einem Consumer, induziert die dynamische Version bei der Überprüfung auf Deadlockfreiheit knapp 2% mehr Zustände im Vergleich zu Klös [27], unsere statische Version jedoch fast doppelt so viele. Dieser signifikante Unterschied lässt sich durch eine intrinsische Eigenschaft unseres Speichermodells erklären. Je nach dem, ob der Scheduler zuerst die MainMethod-Instanz des Producers oder die des Consumers startet, werden die Datentyparrays in unterschiedlicher Reihenfolge belegt. Da dies gleich zu Anfang geschieht, hat das eine annähernde Verdopplung des Zustandsraums zur Folge. In der dynamischen Variante, ist die Reihenfolge unwichtig, da der Producer erst Speicher belegt, nachdem Zeit vergangen ist (s. Listing 7, Z. 14f.) und somit immer erst der Consumer allozieren darf. Der Ansatz [27] weist im Speicher repräsentierten Variablen während der Transformation schon Speicheradressen zu, sodass während der Verifikation nicht mehr alloziert werden muss. Dadurch ändert sich die Belegung des Datentyparrays niemals. Dieser Ansatz hat allerdings den Nachteil, das Speicher nur statisch abgebildet werden kann. Außerdem kann nicht nachvollzogen werden, ob Variablen außerhalb ihres Gültigkeitsbereichs referenziert werden.

Tabelle 3: Ergebnisse des Producer/Consumer-Beispiels

	Klös [27] (statisch)	statisch	dynamisch
Übersetzungs-dauer (in Sek.)	0,61	1,22	1,23
	Verifikationsdauer / Anzahl der Zustände		
1 Producer / 1 Consumer			
Deadlockfreiheit	0,75s / 83.021	1,44s / 165.934	0,99s / 84.247
Pointercheck	0,46s / 83.021	1,83s / 188.398	0,99s / 94.173
2 Producer / 2 Consumer			
Deadlockfreiheit	4,25s /292K	108,68s /7.009K	9,67s /594K
Pointercheck	2,71s /292K	123,34s /8.087K	11,32s /647K
3 Producer / 3 Consumer			
Deadlockfreiheit	16,44s /849K	[1]	110,23s /5.164K
Pointercheck	10,62s /849K		127.23s /5.878K

Allgemein betrachtet gibt es n! verschiedene Speicherplatzbelegungen, wenn n TA zeitgleich (im Sinne von SystemC) den Speicher manipulieren können. Dies entspricht dem SystemC-Modell. Diese Eigenschaft wird in der Zustandsexplosion

[1] Kein Ergebnis nach ca. 2h Berechnung und 37Gb Speicherverbrauch (auf einer anderen Maschine).

bei der Verifikation der UPPAAL-Modelle mit zusätzlichen Instanzen (siehe Tabelle) offensichtlich. Hierbei ist zu bemerken, dass die große Zahl der semantischen Zustände (der Variablenbelegung) durch eine verhältnismäßig geringe Anzahl möglicher Kombination der Locations, in denen sich die einzelnen Instanzen befinden können, generiert wird. Dieser Overhead kann drastisch reduziert werden, in dem man den Scheduler deterministisch gestaltet. Das hat allerdings zur Folge, dass die Transformation nur noch der SystemC-Referenzimplementierung genügt, nicht jedoch dem Standard. Außerdem wäre ein grundlegender Umbau der STATE-Architektur [21, 22, 30, 27] hierfür nötig.

Mit der Eigenschaft Pointercheck in Tabelle 3 haben wir verifiziert, dass Pointer (inklusive Arrayzugriffe über Indizes) immer auf eine valide Adresse zeigen, wenn sie ausgelesen werden (wie in den Abschnitten 5.1 und 5.3 beschrieben). Im Vergleich zur Deadlockfreiheit benötigt unser Ansatz mehr Zustände. Das liegt daran, dass wir zum Verifizieren dieser Eigenschaften Guards in das UPPAAL-Modell einbauen, was zur Folge hat, das bestimmte Optimierungen (v.a. das Zusammenführen aufeinander folgender Transitionen) nicht mehr vorgenommen werden können.

Wir haben beispielhaft an der Version zwei Producer/zwei Consumer (2P2C) weitere speicherbezogene Eigenschaften verifiziert. Tabelle 4 zeigt die Ergebnisse der Verifikation der Eigenschaften *double free*, *wrong delete* (`delete` vs. `delete []`) und *static delete* (`delete`-Aufruf im Stack). Wir haben die 2P2C-Variante jeweils so modifiziert, dass der Fehler auftritt. In allen drei Fällen führte dies zum Programmabsturz. Die Verifikation des UPPAAL-Modells hat den Fehler immer gefunden und das generierte Gegenbeispiel verwies auf die von uns zuvor eingebaute Schwachstelle.

Tabelle 4: 2P2C-Variante mit Fehlern (Verifikationsdauer / Anzahl der Zustände)

Eigenschaft	ohne Fehler	mit Fehler
double free	6,02s / 594K	0,02s / 1160
wrong delete	6,0s / 594K	0,04s / 2348
static delete	5,75s / 594K	0,04s / 2144

Die Tabelle veranschaulicht, dass der komplette Zustandsraum nur bei einem korrekten Modell exploriert werden muss. Werden die Eigenschaften verletzt, findet der Modelchecker den Fehler sehr schnell.

7.2 Bewertung der Ergebnisse

Die Experimente zeigen, dass unser Modell es erlaubt, verschiedenste, speicherbezogene Eigenschaften zu verifizieren. Der induzierte Overhead ist stark abhängig

vom SystemC-Design. Allerdings dauert die Verifikation nur dann äußerst lange, wenn das Modell keine Fehler enthält. Der Zeitaufwand rechtfertigt sich durch die Sicherheit des formalen Beweises, dass die Eigenschaft nicht verletzt wird. Fehler findet der Modelchecker innerhalb kürzester Zeit. D.h. der Overhead beeinträchtigt die Generierung von Gegenbeispielen, die für die Anpassung des Designs nötig sind, nicht.

Die Übersetzungsdauer ist doppelt so lang wie in [27], aber nach wie vor gering in absoluten Zahlen. Im Vergleich zur Verifikationsdauer ist die Transformationszeit wesentlich kleiner und vermindert die Anwendbarkeit nicht.

8 Zusammenfassung und Ausblick

In dieser Arbeit haben wir ein formales Speichermodell für SystemC-Modelle vorgestellt, das es erlaubt statische und dynamische Speicheroperationen abzubilden und zu verifizieren. Wir haben weiterhin aufgezeigt, wie die Verifikation anhand von klar definierten Übersetzungsregeln automatisiert werden kann. Unseren Ansatz konnten wir erfolgreich evaluieren.

Das Speichermodell ist formal definiert und eindeutig. Es ist leicht verständlich und unterstützt alle wichtigen Speicheroperationen in SystemC. Diese von uns gesteckten Ziele haben wir erfüllt. Der entstandene Overhead ist im Vergleich zu Klös [27] höher, wird aber durch den deutlich erhöhten Funktionsumfang ausgeglichen.

Wir haben unseren Ansatz in das Framework STATE [21, 22, 30, 27] eingebaut und können SystemC-Modelle, die unseren Anforderungen genügen, automatisch nach UPPAAL übersetzen und verifizieren.

Zur Evaluierung unseres Ansatzes haben wir eine typische Fallstudie genutzt. Wir waren in der Lage, mehr speicherbezogene Eigenschaften zu verifizieren als vorherige Ansätze. Unsere Ergebnisse zeigen, dass unser Modell einen vergleichbaren Overhead produziert. Demgegenüber stehen jedoch die Unterstützung eines größeren SystemC-Sprachumfangs und die Möglichkeit, mehr speicherbezogene Eigenschaften verifizieren zu können.

8.1 Ausblick

Für die Zukunft sehen wir vor allem Potenzial in der Optimierung und Erweiterung unseres Ansatzes.

Der in der Evaluation zum Vorschein gekommene Overhead ist reduzierbar. Dazu gäbe es verschiedene Lösungsansätze. Zum einen könnte der Scheduler deterministisch entscheiden, welcher TA zuerst Zugriff auf den Speicher bekommt. Weiterhin wäre eine Analyse interessant, die es ermöglicht, festzustellen, ob zwei oder mehr TA sich überschneidende Speicherbereiche manipulieren. Ist dies nicht der Fall, könnte der Scheduler deterministisch über die Ausführungsreihenfolge entscheiden. Eine dritte Möglichkeit bestünde darin, den Stack teilweise aufzugeben, und bestimmte lokale Variablen immer im Speicher vorzuhalten (vergleichbar mit [27]). Allerdings müsste man einen Mechanismus finden, der sicherstellt, dass nicht auf Variablen außerhalb des Scopes zugegriffen wird (bzw. diese Absicht erkennbar bleibt).

Denkbar wäre auch ein Ansatz zur Defragmentierung des Speichers. Dadurch könnten die Größen der Datentyparrays unter Umständen verringert werden, was den Speicherbedarf der Verifikation vermindert. Allerdings muss hier sicher gestellt werden, dass der zusätzliche Mehraufwand den erhofften Nutzen nicht übersteigt.

Eine Erweiterung unseres Modells wäre die Unterstützung mehrdimensionaler Arrays. Dies könnte durch *Flattening* mit der entsprechenden Indexumwandlung realisiert werden. Die vollständige Integration anderer SystemC-Sprachelemente wie TLM werden es uns ebenfalls ermöglichen, unseren Ansatz an einer größeren Fallstudie zu evaluieren.

Anhang

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal 4.0. 2006.

[3] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A Tool for the Analysis of SystemC Models. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 467–470. Springer, 2008.

[4] Richard Bornat. Proving pointer programs in hoare logic. In Roland Backhouse and JoséNuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67727-7. doi: 10.1007/10722010_8. URL `http://dx.doi.org/10.1007/10722010_8`.

[5] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence*, 1972.

[6] Geng Chen, Lei Luo, and Lijie Wang. A precise memory model for operating system code verification. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1125–1132, 2011. doi: 10.1109/TrustCom.2011.153.

[7] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of c code. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, 2004.

[8] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying systemc: A software model checking approach. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 51–59, 2010.

[9] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Kratos: a software model checker for systemc. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 310–316, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL `http://dl.acm.org/citation.cfm?id=2032305.2032329`.

[10] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[11] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *IN CONF. THEOREM PROVING IN HIGHER ORDER LOGICS (TPHOLS), VOLUME 5674 OF LNCS*, 2009.

[12] Ernie Cohen, MichałMoskal, Stephan Tobies, and Wolfram Schulte. A precise yet efficient memory model for c. *Electronic Notes in Theoretical Computer Science*, 254:85–103, October 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.09.061. URL `http://dl.acm.org/citation.cfm?id=1630177.1630459`.

[13] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *2007 Future of Software Engineering*, FOSE '07, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.6. URL `http://dx.doi.org/10.1109/FOSE.2007.6`.

[14] Dawson Engler. Concur 2005 - concurrency theory. chapter Static analysis versus model checking for bug finding, pages 1–1. Springer-Verlag, London, UK, UK, 2005. ISBN 3-540-28309-9. doi: 10.1007/11539452_1. URL `http://dx.doi.org/10.1007/11539452_1`.

[15] Joachim Fellmut, Paula Herber, Sabine Glesner, and Stefan Jänichen. Automatische Übersetzung von systemc modellen in timed automata. Master's thesis, Technische Universität Berlin, 2008.

[16] Daniel Große and Rolf Drechsler. Checkers for SystemC designs. In *Formal Methods and Models for Codesign*, pages 171–178. IEEE Computer Society, 2004.

[17] 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. ISBN 1-59593-347-6.

[18] 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.

[19] 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.

[20] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.

[21] 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.

[22] 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.

[23] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first edition, 2003. ISBN 0-321-22862-6.

[24] IEEE Standards Associaton. IEEE Standard for Standard SystemC® Language Reference Manual (IEEE Std. 1666–2011, Revision of 1666-2005), 2011.

[25] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Morgan & Claypool Publishers, 2006.

[26] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. ISBN 0131103709.

[27] Verena Klös, Paula Herber, Marcel Pockrandt, and Sabine Glesner. Analysis and transformation of the memory structure of hw/sw co-designs. 2012.

[28] Daniel Kroening and Natasha Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *Proceedings of MEMOCODE 2005*, pages 101–110. IEEE, 2005.

[29] B. Niemann and C. Haubelt. Formalizing TLM with Communicating State Machines. *Forum on specification and Design Languages*, 2006.

[30] M. Pockrandt, P. Herber, and S. Glesner. Model checking a systemc/tlm design of the amba ahb protocol. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pages 66–75, 2011. doi: 10.1109/ESTIMedia.2011.6088527.

[31] Marcel Pockrandt, Paula Herber, Holger Gross, and Sabine Glesner. Optimized transformation and verification of systemc methods. 2012.

[32] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817.

[33] 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. ISBN 0-7695-0993-2.

[34] Ashraf Salem. Formal Semantics of Synchronous SystemC. In *Design, Automation and Test in Europe (DATE)*, pages 10376–10381. IEEE Computer Society, 2003. ISBN 0-7695-1870-2.

[35] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *Proceedings of the 5th international conference on Systems software verification*, SSV'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1929004.1929011`.

[36] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997. ISBN 0201889544.

[37] 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.

[38] 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.