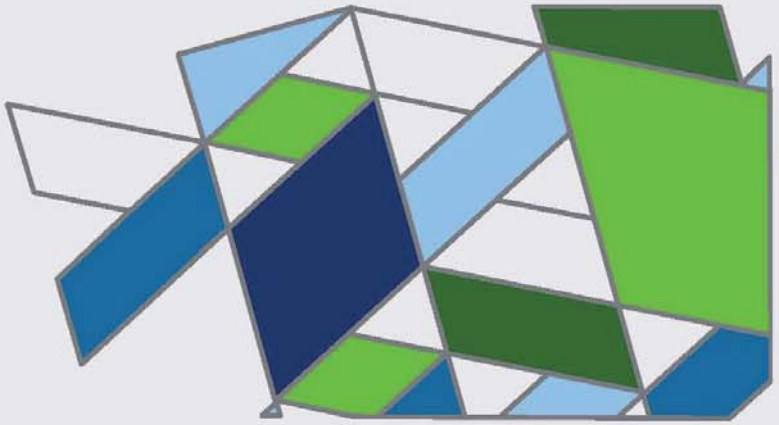
MBMV

2014



Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen

Editoren

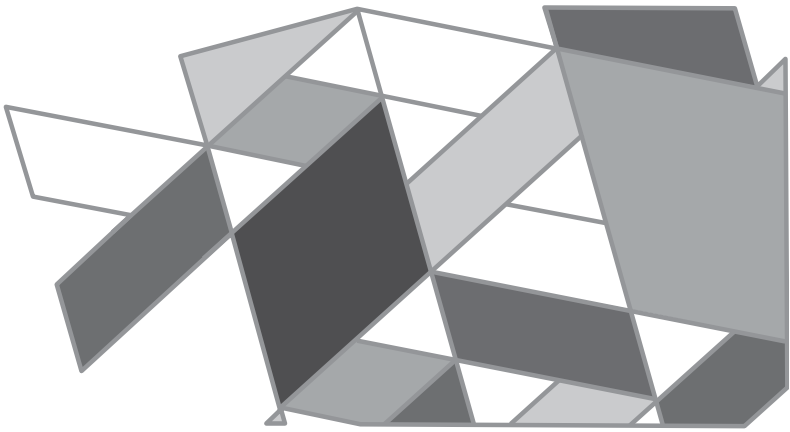
Jürgen Ruf
Dirk Allmendinger
Matteo Michel

Cuvillier Verlag Göttingen
Internationaler wissenschaftlicher Fachverlag

MBMV 2014

MBMV

2014



Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen

Editoren

Jürgen Ruf
Dirk Allmendinger
Matteo Michel

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2014

978-3-95404-637-9

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2014
Gedruckt auf umweltfreundlichem, säurefreiem Papier aus nachhaltiger Forstwirtschaft.

978-3-95404-637-9

Vorwort

Der Workshop "*Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*" (MBMV) ist bereits der siebzehnte gemeinsame Workshop der Fachgruppen 3 und 4 der Kooperationsgemeinschaft "Rechnergestützter Schaltungs- und Systementwurf (RSS)" der Gesellschaft für Informatik (GI), der Informationstechnischen Gesellschaft im VDE (ITG) und der Gesellschaft für Mikroelektronik, Mikro- und Feinmechanik (GMM). In diesem Jahr findet der Workshop erstmals im IBM Forschungs- und Entwicklungslabor in Böblingen statt.

Die MBMV ist ein Forum um Trends, neuste Ergebnisse und aktuelle Probleme auf dem Gebiet der Methoden zur Modellierung und Verifikation sowie der Beschreibungssprachen digitaler, analoger und Mixed-Signal-Schaltungen zu diskutieren. Auch Aspekte des Entwurfs und Tests von hardwarenaher eingebetteter Software werden im Rahmen dieses Workshops beleuchtet.

Die Veranstaltung dient ebenso zur Vertiefung von Kontakten zwischen Universitäten, Forschungseinrichtungen und der Industrie und soll somit den gegenseitigen Austausch von Ideen und Problemstellungen fördern. Zudem bietet der Workshop jungen Forschern eine Plattform um Ihre Ideen einem größeren Publikum zu präsentieren.

Die Beiträge des diesjährigen Workshops können den folgenden Schwerpunkten zugeordnet werden: formale und semiformale Spezifikation und Modellierung, Entwurfsmethodik, Codesign (analog/digital, Hardware/Software), formale Verifikation (Äquivalenz- und Eigenschaftsbeweise), Verifikation nichtfunktionaler Eigenschaften, simulationsbasierte Verifikation, sowie Verifikation und Validierung hardwarenaher Software.

Das Programmkomitee hat aus 29 eingereichten Beiträgen ein interessantes Programm zusammengestellt, das aus 17 langen Beiträgen und 7 Posterpräsentationen besteht. Das Programm wird durch zwei eingeladene Sprecher bereichert, die zu den Themen „From Physics to Business Value – A Broad View to IT System Environment" und „Silicon Technology Outlook" berichten.

Unser Dank gilt den Autoren für die Einreichung, die fristgerechte Überarbeitung und die Präsentation ihrer Arbeiten beim Workshop. Für die sorgfältige Begutachtung und konstruktive Kritik gebührt den Mitgliedern des Programmkomitees unser besonderer Dank. Wir bedanken uns bei den eingeladenen Sprechern *Martin Schmatz,* Manager Systems Department, IBM Research, Rüschlikon, Schweiz und *Dieter Wendel,* Distinguished Engineer, IBM Research & Development, Böblingen. Zudem sind wir dem Verein zur Förderung der Informatik (FIT) e.V. der Universität Kaiserslautern insbesondere dessen Vorsitzenden Prof. Markus Nebel zu Dank für die Übernahme der Konferenzabrechnung verpflichtet. Desweiteren gilt unser Dank unserem lokalen Management im Forschungslabor, das die Durchführung dieses Workshops in den IBM Räumlichkeiten und die Nutzung der lokalen Infrastruktur im Labor der IBM erst ermöglicht hat. Insbesondere danken wir Marco Eibach für seine Unterstützung bei der Organisation, Sibylle Schäfer für ihre tatkräftige Begleitung des Workshops sowie Michael Kiess für die Bereitstellung des MBMV Webauftritts.

Wir wünschen allen Teilnehmern einen interessanten und bereichernden Workshop sowie die Möglichkeit für den Austausch von Ideen und die Knüpfung neuer Kontakte.

Böblingen, im Januar 2014 Jürgen Ruf, Dirk Allmendinger und Matteo Michel

Inhaltsverzeichnis

Programmkomitee **11**

Manuel Gesell, Felipe Bichued, and Klaus Schneider

Using Different Representations of Synchronous Systems in SAL **13**

Karsten Scheibler, Bernd Becker

Implication Graph Compression inside the SMT Solver iSAT3 ..... **25**

Christian Appold

A New Approach to Use Partial Results During Image Computation in BDD Based Symbolic Model Checking **37**

Bastian Koppelmann, Markus Becker und Wolfgang Müller

Portierung der TriCore-Architektur auf QEMU **49**

Hanno Eichelberger, Patrick Heckeler, Jürgen Ruf, Stefan Huster, Sebastian Burg, Thomas Kropf, Wolfgang Rosenstiel, Thomas Greiner

Erkennen von Speicherverletzungen im Testbetrieb von eingebetteter Software **61**

Mohamed Ammar Ben Khadra, Yu Bai, Klaus Schneider

Synthesis of Distributed Synchronous Specifications to SysteMoC **71**

Kristin Krüger, Carna Radojicic, Christoph Grimm

Semi-Symbolische Analyse eines ΣΔ-Modulators **83**

Maarten Boersma, Ulrike Schmidt, Markus Kaltenbach

Automatic detection of sticky clock gating functions **93**

Roberto Urban, Kai Lehninger, Maximilian Heyne, Mario Schölzel, H.T. Vierhaus

Vergleich der Beschreibung und Simulation einer Befehlssatzarchitektur in LISA und CoMet **101**

Christoph Kuznik, Bertrand Defo und Wolfgang Müller

Semi-automatische Generierung von Überdeckungsmetriken mittels methodischer Verifikationsplan-Verarbeitung 113

Stefan Huster, Merdin Macic, Sebastian Burg, Hanno Eichelberger, Patrick Heckeler, Jürgen Ruf, Thomas Kropf and Wolfgang Rosenstiel

Increasing Software Reliability by Integrating Formal Verification and Robustness Testing 125

Rafal Baranowski, Michael A. Kochte, Hans-Joachim Wunderlich

Verifikation Rekonfigurierbarer Scan-Netze 137

Christian Bartsch, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, Wolfgang Kunz

Efficient SAT/Simulation-based model generation for low-level embedded software .. 147

Konrad Möller, Martin Kumm, Björn Barschtipan, Peter Zipf

Dynamically Reconfigurable Constant Multiplication on FPGAs 159

Martin Kumm, Peter Zipf

Efficient High Speed Compression Trees on Xilinx FPGAs 171

Robert Fischbach, Michael Dittrich, Andy Heinig

Effizienter Design Rule Check von 3D Systemaufbauten mit einer hierarchischen XML-basierten Modellierungssprache 183

Sebastian Burg, Patrick Heckeler, Stefan Huster, Hanno Eichelberger, Jörg Behrend, Jürgen Ruf, Thomas Kropf, Oliver Bringmann

LoCEG: Local Preprocessing in SAT-Solving through Counter-Example Generation 193

Niels Thole, Görschwin Fey

Equivalence Checking on System Level using Stepwise Induction 197

Aljoscha Windhorst, Hoang M. Le, Daniel Große, Rolf Drechsler

Funktionale Abdeckungsanalyse für C-Programme 201

Vladimir Kolchuzhin, Jan Mehner, Milind Shende, Erik Markert, Ulrich Heinkel, Christian Wagner, Thomas Gessner

System Level Modeling of Piezoresistive Effect of Carbon Nanotubes for Sensor Application 205

S. Stieber, Johann-P. Wolff, Ch. Haubelt, Rainer Dorsch

Hybride Prototypisierung eines Sensorsubsystems 209

Mathias Soeken, Max Nitze, Rolf Drechsler

Formale Methoden für Alle (Erweiterte Zusammenfassung) 213

Heinz Riener, Oliver Keszocze, Rolf Drechsler, Görschwin Fey

A Logic for Cardinality Constraints (Extended Abstract) 217

Programmkomitee

- Bernd Becker, *Universität Freiburg*
- Jens Brandt, *Robert Bosch GmbH*
- Oliver Bringmann, *Universität Tübingen*
- Manfred Dietrich, *Fraunhofer IIS*
- Gero Dittmann, *IBM Forschung*
- Rolf Drechsler, *Universität Bremen*
- Martin Freibothe, *Intel Mobile Communications GmbH*
- Michael Glaß, *Universität Erlangen-Nürnberg*
- Carsten Gremzow, *Bergische Universität Wuppertal*
- Christoph Grimm, *Techn. Universität Kaiserslautern*
- Christian Haubelt, *Universität Rostock*
- Ulrich Heinkel, *Techn. Universität Chemnitz*
- Jörg Henkel, *Universität Karlsruhe*
- Christoph Jäschke, *IBM Deutschland Forschung und Entwicklung*
- Uwe Knöchel, *Fraunhofer IIS / EAS Dresden*
- Thomas Kropf, *Bosch*
- Wolfgang Kunz, *Techn. Universität Kaiserslautern*
- Gunther Lehmann, *Infineon, München*
- Paul Molitor, *Universität Halle*
- Wolfgang Müller, *Universität Paderborn*
- Peter Oehler, *Continental A. S., Frankfurt / M.*
- Frank Oppenheimer, *Offis*
- Jürgen Ruf, *IBM Deutschland Forschung und Entwicklung*
- Klaus Schneider, *Techn. Universität Kaiserslautern*
- Christoph Scholl, *Universität Freiburg*
- Jens Schönherr, *HTW Dresden*
- Martin Speitel, *Fraunhofer IIS, Erlangen*
- Dominik Stoffel, *Techn. Universität Kaiserslautern*
- Jürgen Teich, *Universität Erlangen*
- Klaus Waldschmidt, *Universität Frankfurt / M.*
- Markus Wedler, *Synopsys*
- Robert Wille, *Universität Bremen*
- Reimund Wittmann, *IP GEN, Bochum*

Using Different Representations of Synchronous Systems in SAL

Manuel Gesell, Felipe Bichued, and Klaus Schneider
TU Kaiserslautern
gesell@cs.uni-kl.de
bichued@rhrk.uni-kl.de
schneider@cs.uni-kl.de

Abstract

In general, synchronous systems can be represented as a set of so-called synchronous guarded actions (SGAs) that consist of a trigger condition and an atomic action. Whenever the trigger condition holds, i.e., the guarded action is enabled, then the action is immediately executed. While the synchronous semantics demands that *all enabled* actions have to be executed concurrently within the same variable environment, it is possible for certain sets of guarded actions to deviate from the synchronous execution scheme without changing the behavior. This is important to make use of tools like SRI's Symbolic Analysis Laboratory (SAL) that work with invariants and guarded actions, but only a subset of the enabled actions are chosen for execution. If the particular choice of the enabled guarded actions for execution is not determined, we may consider different choices that might influence the resource requirements needed for formal verification. In this paper, we therefore investigate how three possible representations influence the runtime and memory requirements of automatic verification runs of SRI's SAL.

1. Introduction

The synchronous model of computation [Hal93, BCE+03] has proved to be very convenient for the development of reactive embedded systems. In particular, this is the case for systems consisting of application-specific hardware *and* software since compilers can generate hardware as well as software from the same synchronous programs. In general, a synchronous system performs its execution in discrete reaction steps that are considered as clock ticks of a global clock. In each reaction, *all* input values are read, and depending on these values, *all* output values are determined in addition to the change of the internal state of the system.

Synchronous languages like Esterel [BG92] and Quartz [Sch09] offer the explicit notion of reaction steps and many convenient statements for the design of reactive systems. The explicit notion of (logical) time also requires different kinds of assignments like immediate assignments that assign values in zero time and delayed assignments that assign values with a delay of one time unit.

For safety-critical applications, an important advantage of synchronous languages is the availability of a precisely defined formal semantics. This allows one to translate programs to state transition systems and to use formal verification techniques. Model checking procedures [GV08] are already established and are even integrated within the compilers, e.g., to check for instantaneous loop bodies, to guarantee the absence of write conflicts and runtime errors, to solve causality problems, and many other issues that might appear during compilation. However, it is well-known that the particular system representation is a crucial point for a formal verification task. For example, even a bad variable ordering in a BDD-based model checker may lead to an exponential blow-up in run time or memory usage while a better ordering could work efficiently.

In this paper, we therefore explore the possibilities of representing a synchronous system in SRI's Symbolic Analysis Laboratory (SAL), and evaluate their effect on the performance of SAL's model checker. To this end, we start in all cases with synchronous guarded actions as a general system representation. Since SAL only supports interleaved guarded actions, we already presented in [GS13] a possible translation of *synchronous guarded actions (*SGAs*)* to SAL's *interleaved guarded actions* (*IGAs*) to demonstrate that SAL can also be used to verify synchronous systems. This paper aims at comparing that system representation against alternatives with respect to the performance of the later model checking. Therefore, the approach presented in [GS13] (GC) will be compared with two others: One based on SAL's synchronous composition of modules (SC) and another one based on a translation to equation systems (ES). Therefore, we implemented the tool *aif2sal* that is capable of generating these representations from an *SGAs* description of a Quartz program (see Figure 1).

The three transformations GC, SC, and ES are based on different paradigms that lead to different computations of a macro step in SAL. The ES representation describes the behavior of all *SGAs* in a single transition step by describing them as invariants. Therefore, no guarded action is required and an equation system must be solved in each reaction step. The SC transformation models the behavior of each variable by a single module containing all guarded actions writing this variable. The synchronous model of computation assures that only a single guarded action defines the value of a variable in a reaction step. Hence, in each reaction step all modules execute a single guarded action synchronously to define the behavior. This obliges SAL to resolve the data-dependencies between the modules. The GC approach describes the behavior in a single module by *IGAs*. This requires the explicit modeling of the data-dependencies as described in [GS13]. Unlike the other approaches, the behavior of a single reaction step of the original system requires several transition steps in GC.

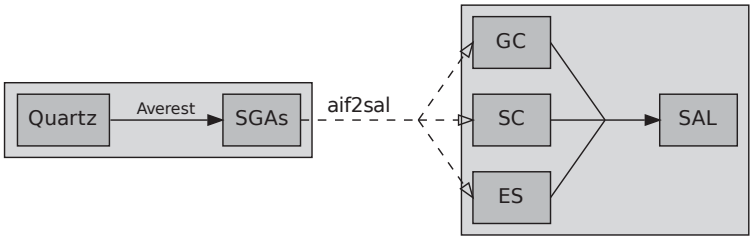


Figure 1: Averest/SAL linkage

The paper is organized as follows: the next section describes some preliminaries like the synchronous model of computation, the Averest tool-kit and SRI's SAL tool. Then, Section 3 presents related work we found in the literature. The main part of the paper is presented in Section 4, where

the above three representations are described. In Section 5, we present experimental results to compare the considered transformations and argue about the most advanced representation. Finally, we conclude the paper and discuss future work.

2. Preliminaries

In this section, we describe the synchronous model of computation, the Averest system and SAL.

2.1. Synchronous Models of Computation

The execution of synchronous languages [Hal93, BCE+03] is divided into a discrete sequence of reaction steps that are also called macro steps. Within each macro step, the system reads all inputs and instantaneously generates all outputs together with the next internal state depending on the current internal state and the read inputs. To compute the outputs and the next internal state, macro steps are divided into finitely many micro steps. Micro steps are atomic actions of the programs like assignments to variables. Since all micro steps of a macro step are executed in the same variable environment given by their reaction/macro step, all variables have unique values in each macro step. It may be the case that the trigger condition of an action depends on a variable that is modified by the action itself. Such cyclic dependencies are considered in the causality analysis for synchronous programs that checks whether for all inputs, the outputs can be determined in an order that respects the data dependencies.

2.2. Averest

Averest[1] is a framework for specification, implementation and verification of reactive systems and is developed by our group. The programs are written using the synchronous programming language Quartz and are translated to the Averest Intermediate Format (AIF), which is a representation of the program's behavior in terms of *SGAs*. An *SGA* $\langle \gamma \Rightarrow \alpha \rangle$ contains a boolean guard γ and an atomic assignment α that is executed whenever γ holds. Assignments can be either immediate $\langle \gamma \Rightarrow \texttt{x=}\tau \rangle$ or delayed $\langle \gamma \Rightarrow \textbf{next}\texttt{(x)=}\tau \rangle$. Both evaluate the right-hand side τ in the current step. While the immediate assignment transfers the obtained value already in the current step to the left-hand side `x`, the delayed assignments transfer the value only in the next macro step to `x`.

A simple module called ABRO can be seen on the left-hand side of Figure 2. This module has three inputs (indicated by **?**) `a`, `b` and `r`, and one output `o` (indicated by **!**). The program waits for the input events `a` and `b` and immediately emits output `o` as soon as the last one of `a` and `b` occurred. This behavior can be restarted with the reset input `r`.

The program contains two specifications `s1` and `s2`, where the first asserts that either `a` or *b* holds whenever *o* is emitted. Property `s2` asserts that `o` does not hold in successive points of time.

The ABRO module is compiled to the *SGAs* shown on the right-hand side of Figure 2. As can be seen, the guarded actions are separated into control flow and data flow. Control flow actions are assignments to control flow locations and data flow actions are assignments to local and output variables. Apart from the control flow locations `wa`, `wb` and `wr`, a new location `w0` (often called *boot-location*) is added by the compiler to serve as initial state.

[1] http://www.averest.org

```
module ABRO(event ?a,?b,?r,!o) {
   loop
      abort {
         wa: await(a);
         ||
         wb: await(b);
         emit(o);
         wr: await(r);
      } when(r);
} satisfies {
   s1 : assert A G (o ⇒ a ∨ b);
   s2 : assert A G (o ⇒ X ¬o);
}
```

```
system ABRO :
interface :
    a, b, r : input event bool
          o : output event bool
locals :
    w0, wa, wb, wr : label bool
synchronous guarded actions :
    control flow:
        True ⇒next(w0)=True
        ¬w0 ⇒next(wa)=True
        ¬w0 ⇒next(wb)=True
        ¬r∧wa∧¬a∨r∧(wr∨wa∨wb)⇒next(wa)=True
        ¬r∧wb∧¬b∨r∧(wr∨wa∨wb)⇒next(wb)=True
        ¬r∧(wr∨a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)
          ⇒next(wr)=True
    data flow:
        ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)⇒o=True
specifications:
    s1: A G o → a ∨ b
    s2: A G o → X ¬o
```

Figure 2: ABRO Example

2.3. SAL

SRI's Symbolic Analysis Laboratory[2] is a framework intended for performing abstraction, program analysis and model checking, and it provides an intermediate language which will be the target of our translation process. A typical SAL system is represented by a context containing a set of modules and assertions. Each module declares a distinct set of inputs, outputs, local and global variables, as well as definitions (invariants) and transitions. Variables can be either current (X) or next variables (X'), where assignments to current variables take place in the current state, and to next variables in the following state. SAL allows to compose modules either synchronously ($||$) or asynchronously ($[]$). In synchronously composed modules, a transition from each module is executed simultaneously. With asynchronous composition however, an enabled transition from exactly one module is executed non-deterministically. Transitions can be written as an equation or as guarded commands. The equational format defines the trajectory of single variables, while the guarded commands define single transitions in the system. A guarded command of SAL contains, contrary to *SGAs*, a set of assignments and is enabled when its guard evaluates to true. Furthermore, SAL non-deterministically picks one of the enabled guarded commands and updates the next-state variables accordingly. The structure and behavior is equivalent to *IGAs* described in [GS13]. A system without enabled guarded commands leads to a deadlock. A large set of tools such as symbolic/bounded model checkers, simulators and others, can then be used for analysis and verification.

3. Related Work

The idea of transforming *SGAs* to other models of computation was already done before: an automatic translation to SystemC by generating a dynamic schedule for modules in order to preserve the semantics was presented in [BGS10]; and in [BBS11] *SGAs* are translated to asynchronous DPNs.

[2] http://sal.csl.sri.com/

There exists also a similar approach [SB08, BS08] that refines the translation of *SGAs* to transition systems at the level of micro steps. The intention of [SB08, BS08] was to use these transition systems at the micro step level to perform causality analysis by means of theorem proving and bounded model checking. We follow similar ideas in the GC transformation, but work at a different level of abstraction, and furthermore, our approach allows us to improve the system representation by handling e.g. the default reaction in a different way.

The differences between *SGAs* and Hoare's parallel commands [Hoa78] are that *SGAs* do not have a disjoint set of variables and they communicate over shared variables (broadcast).

In contrast to Dijkstra's guarded commands [Dij75], our *IGAs* have only a single *repetitive construct* consisting of the entire set of *IGAs*. Hence, our GC translation targets a subset of Dijkstra's guarded commands. This is justified since we do not wish to use the guarded actions as primary input language, and use them rather as intermediate representation.

The Z2SAL project [DNS06, DNS11] connects Z to SAL by defining a transformation to SAL's input language. Additionally, we compare different approaches of representing Quartz in SAL. One of our transformations uses the built-in synchronous-composition operator of SAL similar to [PSSD00] where dynamic constructs were used to embed the behavior of multi-threaded Java programs in SAL.

4. Different Representations of Synchronous Systems

In this section, we present three different approaches of describing synchronous systems in SAL's input language. To ease the translation process, we have developed a tool called *aif2sal*, which is capable of converting AIF to the three different representations in SAL.

4.1. Guarded Commands GC

Guarded commands in SAL are interpreted as interleaved guarded actions (*IGAs*), meaning that in each transition step an enabled guard action is non-deterministically chosen to define the step's behavior. This is very different from the way that *SGAs* work and thus many problems have to be solved when representing a synchronous program as *IGAs*. These problems are described in detail in [GS13] and will be summarized in the following.

By executing the guarded actions in an interleaved fashion, a data dependency problem appears because of the non-deterministic choice of a single action. The *IGAs* now can execute in arbitrary orders, including those where a value is not yet present for a specific variable. Furthermore, the temporal behavior of a synchronous program is violated due to the increased number of steps it takes to complete a macro step.

$$\begin{array}{lcl|lcl} \gamma_1 & \Rightarrow & \mathtt{x} = \tau_1 & \delta_1 & \Rightarrow & \mathbf{next}(\mathtt{x}) = \upsilon_1 \\ & \vdots & & & \vdots & \\ \gamma_n & \Rightarrow & \mathtt{x} = \tau_n & \delta_m & \Rightarrow & \mathbf{next}(\mathtt{x}) = \upsilon_m \end{array}$$

Figure 3: Guarded Actions for variable `x`

The key to solve these problems is to modify the guards such that the data dependencies are explicitly stated. Hence, the solution of [GS13] represents each *SGA* containing an immediate assignment by separate *IGAs*, and all *SGAs* containing a delayed assignment are composed to a single

IGA called *conclusion*. Additionally, we need to introduce for all variables written by immediate assignments a new variable, the valid flag $\mathtt{x}_v$, that determines the validity of the value contained in the variable and is used to deactivate all *IGAs* writing to the corresponding variable x once a value is determined in the current macro step. Hence, the synchronous guarded actions for the variable x in Figure 3 are converted into the following *IGAs*:

$$
\begin{array}{lcl}
\neg \mathtt{x_v} \wedge \left(\bigwedge_{v \in \mathsf{read}(\gamma_1 \Rightarrow \mathtt{x}=\tau_1)} v_\mathrm{v} \right) \wedge \gamma_1 & \Rightarrow & \left[\begin{array}{ll} \mathtt{x} & = \tau_1 \\ \mathtt{x_v} & = \mathsf{true} \end{array} \right] \\
\vdots & & \\
\neg \mathtt{x_v} \wedge \left(\bigwedge_{v \in \mathsf{read}(\gamma_n \Rightarrow \mathtt{x}=\tau_n)} v_\mathrm{v} \right) \wedge \gamma_n & \Rightarrow & \left[\begin{array}{ll} \mathtt{x} & = \tau_n \\ \mathtt{x_v} & = \mathsf{true} \end{array} \right] \\
\neg \mathtt{x_v} \wedge \left(\bigwedge_{i=1\ldots n} \neg \gamma_i \right) & \Rightarrow & \left[\begin{array}{ll} \mathtt{x_v} & = \mathsf{true} \end{array} \right]
\end{array}
\quad \Bigg| \quad
\bigwedge_{v \in \mathcal{V}} v_\mathrm{v} \Rightarrow
\left[\begin{array}{l}
\vdots \\
x = \begin{cases} v_1, & \text{if } \delta_1 \\ \vdots & \\ v_m, & \text{if } \delta_m \\ \text{defaultAct (x)}, & \text{else} \end{cases} \\
x_\mathrm{v} = \bigvee_{i=1\ldots m} \delta_i \\
\vdots
\end{array} \right]
$$

Whenever we determined a valid value for x by enabling $\mathtt{x}_v$, all *IGAs* writing x are automatically deactivated by the term $\neg \mathtt{x_v}$. The following term ensures that all values required to evaluate the *IGA* are valid. The original behavior is still encoded in γ_i and the assignment $\mathtt{x} = \tau_i$. Additionally, the assignment $\mathtt{x_v} = \mathsf{true}$ indicates the validity of x for the current macro step. In case none of the guards hold, no assignment for that variable takes place in the current macro step and thus it must contain the default value (e.g. the value of the previous step) – which also means that $\mathtt{x}_v$ must be enabled. Therefore, the default value of the variable should be already contained in the variable x. This is ensured by the *conclusion* (right-hand side) that executes for all variables the delayed assignments. There, the default value must be assigned to all variables not written by a delayed assignment. This is possible, because the value of a variable x is not used unless $\mathtt{x_v}$ holds. Additionally, the *conclusion* initiates the execution of the next macro step by resetting the valid flags of all variables (not written by delayed assignments).

The SAL GC-representation of our running example has the structure shown in Figure 4. One can see that only a single valid flag (for the variable o) is required, because all other variables are written by delayed assignments. Additionally, the specifications were adapted to cover the changed temporal behavior. All newly introduced immediate states have in common that not all variables have a valid value and so the adapted specification only requires that the original specification is satisfied in states where all variables contain valid values.

4.2. Synchronous Composition SC

Another idea is to exploit SAL's synchronous composition primitive and divide the program into a set of synchronous modules. To that end, each variable, with the exception of inputs, will be represented as an independent module, and these modules will be then composed synchronously to provide the overall system behavior. It is worth noting that the semantics of the synchronous composition closely matches that of the synchronous model of computation. Since every variable has a unique value in each reaction step determined by a single *SGA* every module will execute exactly one transition.

In contrast to the GC transformation, the SC transformation is just a syntactic rewrite of the original program, in the sense that no guarded action will be modified. In this approach, data

```
ABROGC: MODULE =
BEGIN
  INPUT a,b,r : BOOLEAN
  OUTPUT o_v, o : BOOLEAN
  LOCAL w0,wa,wb,wr,o_v : BOOLEAN
  INITIALIZATION [w0 = wa = wb = wr = o = o_v = FALSE]
  TRANSITION [
    [] ¬o_v ∧ ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa) ⟶
        o_v' = TRUE ;
        o' =  TRUE ;
    [] ¬o_v∧(r∨¬(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)) ⟶
        o_v' = TRUE ;
    [] o_v ⟶
        wr' =  ¬r∧(wr∨(a∧wa∧b∧wb)∨(b∧wb∧¬wa)∨(a∧wa∧¬wb));
        wb' =  ¬r∧wb∧¬b∨r∨¬w0;
        wa' =  ¬r∧wa∧¬a∨r∨¬w0;
        w0' =  TRUE ;
        o' =  FALSE ;
        o_v' = FALSE ;
  ]
END;
s1 : THEOREM ABROGC ⊢ AG[¬o_v U (o ⇒ a ∨ b)];
s2 : THEOREM ABROGC ⊢ AG[¬o_v U (o ⇒ X¬o)];
```

Figure 4: GC Representation

dependencies are resolved internally by SAL. The synchronous composition combines all definitions, initializations and transitions of the composed modules, taking care that the combination is still casually correct. In case inconsistencies in the conjunction of the transitions are found, proof obligations are generated, but this problem does not apply here because the Averest compiler rules out causally incorrect programs.

The translation to SC consists of separating *SGAs* by their written variable into individual modules as depicted in Figure 5a (for variable *o*). Each module will have every other variable that is read by the *SGAs* as input and a single output being the writable variable itself. All *SGAs* for each variable are then collected, and used to properly initialize the module and to describe its transitions as guarded commands.

```
oMod : MODULE =
BEGIN
  INPUT  a, b, r, wa, wb : BOOLEAN
  OUTPUT o : BOOLEAN
  INITIALIZATION
    [o=¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)]
  TRANSITION
    [  ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)
          ⟶ o' = TRUE;
    [] ELSE ⟶ o' = FALSE; ]
END
```

(a) SC: Single module

```
 w0Mod : MODULE = ...
 waMod : MODULE = ...
 wbMod : MODULE = ...
 wrMod : MODULE = ...
  oMod : MODULE = ...
ABROSC : MODULE = w0Mod
                  || waMod
                  || wbMod
                  || wrMod
                  ||  oMod;
s1 : THEOREM ABROSC ⊢ AG(o ⇒ a ∨ b);
s2 : THEOREM ABROSC ⊢ AG(o ⇒ X¬o);
```

(b) SC: Composition

Figure 5: Single Module and Synchronous Composition

In Figure 5b, we see how such a synchronous composition might look like. We simply compose all writable variables (`w0`, `wa`, `wb`, `wr` and `o`) into a single module. It is important to note that a composed module will be deadlocked whenever at least one of the modules is deadlocked, hence we

introduce an **ELSE** guard to guarantee that there is always a transition to be taken. Moreover, the **ELSE** guard will assign the default value to the variable, which is effectively the default reaction.

4.3. Equation System ES

This transformation converts the *SGAs* into equations (one per variable). It is usually not trivial to generate such an equation system, and a corresponding transformation is already implemented in the Averest system. The translation of *SGAs* to equations will generate exactly one equation for each output, local and location variable. Furthermore, an additional carrier variable must be added for each variable to which an immediate and delayed assignment is made. The carriers will simply hold the value until the next point of time.

The execution of an equation system under the synchronous model of computation is then as follows: in every macro step new input variables are read and all of the equations are evaluated with regards to the newly read values. The resulting right-hand side of each equation is then assigned to its respective variable.

This can be easily done in SAL by using definitions instead of guarded commands. Definitions in SAL are of the form $\langle X = \texttt{EXPR} \rangle$ for the current state or $\langle X' = \texttt{EXPR} \rangle$ for the next state. In contrast to guarded commands, which are picked individually, all definitions are evaluated in every state and the resulting value for the expression EXPR is assigned to the variable. Once we have the

```
ABROES : MODULE =
BEGIN
  INPUT a, b, r : BOOLEAN
  OUTPUT o : BOOLEAN
  LOCAL wa, wb, wr, w0 : BOOLEAN
  INITIALIZATION [w0 = wa = wb = wr = FALSE]
  DEFINITION o = ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa);
  TRANSITION
    w0' = TRUE;
    wa' = ¬r∧wa∧¬a∨r∨¬w0;
    wb' = ¬r∧wb∧¬b∨r∨¬w0;
    wr' = ¬r∧(wr∨(a∧wa∧b∧wb)∨(b∧wb∧¬wa)∨(a∧wa∧¬wb));
END;
s1 : THEOREM ABROES ⊢ AG(o ⇒ a ∨ b);
s2 : THEOREM ABROES ⊢ AG(o ⇒ X¬o);
```

Figure 6: ES: Single module

SGAs given as equations, the translation to SAL is straightforward. We will have a single module containing the original inputs and outputs and all other variables as local variables, as seen on Figure 6. SAL supports in the **DEFINITION** section only assignments to variables of the current state, hence a distinction between **INITIALIZATION/TRANSITION** and **DEFINITION** is necessary. All equations defining a next-state variable must be initialized in the **INITIALIZATION** section and described in the **TRANSITION** section. Hence, the immediate assignments will be represented as an invariant in the **DEFINITION** section and will be evaluated in every state, including the initial one. The **INITIALIZATION** section will be evaluated in the initial state and contains the initialization of variables written by delayed assignments like the location variables and the

carriers[3] to their default values. The **TRANSITION** section contains equations for evaluating the next-state.

Note that problems like the default reaction were already handled during the translation to an equation system by simply adding an extra branch to every equation, assigning the variable's default value whenever none of the previous conditions hold.

5. Experimental Results

The presented transformations were used to verify and benchmark the experiments using SAL's symbolic model checker (sal-smc). All experiments[4] were performed on a Intel® Core™ i5-3470 CPU @ 3.20GHz using Ubuntu 13.04.

In the following, we briefly describe each example, in terms of what they do, number of variables in the original Quartz program, number of *SGAs* after compilation, and the number and kind of properties that were verified for each. This gives an idea on the complexity of each example:

P	#SGA	#GC	#SC	#ES	GC	SC	ES
ABRO	7	4	6(12)	5	0.11	0.06	0.05
ABROM[M=10]	23	2	14(36)	13	0.74	1.13	0.50
ABROM[M=13]	29	2	17(45)	16	4.27	7.92	3.27
AuntAgatha	2	4	3(4)	3	0.12	0.07	0.09
VendingMachine	23	23	12(32)	11	1.14	0.15	0.07
LightControl	36	25	12(47)	11	1.79	0.44	0.40
MinePumpController	42	41	22(61)	21	7.60	0.22	0.09
RSFlipFlop	7	2	5(11)	8	53.51	1.18	1.18
MemoryController	41	28	17(87)	31	407.95	42.93	3.42
IslandTrafficControl	83	62	35(109)	36	504.64	62.40	1.94

Figure 7: Size of the Representations and Execution Times (in sec) of SAL

ABROM is a larger version of the ABRO example which waits for M events in parallel instead of just two. It contains **22** *SGAs* for $M = 10$ or **28** for $M = 13$, **2** inputs, **1** output and **3** safety properties.

AuntAgatha is an implementation of an old puzzle where the reader has to find who killed Aunt Agatha in Dreadsburry Mansion based on simple boolean statements. The problem is represented by **2** *SGAs*, **21** inputs, **1** output and **3** boolean properties.

VendingMachine is a vending machine controller that dispenses gum in reaction to the insertion of nickels and dimes, which is described by **2** *SGAs*, **2** inputs, **2** outputs, and **3** safety properties.

LightControl models the light control system of a room with regards to to its occupancy. Its functions include switching the light on/off, dimmer control and notification of alarms. The implementation contains **36** *SGAs*, **22** inputs, **12** outputs, and **10** safety specifications.

[3] Carriers are present only when the program utilizes immediate and delayed assignments for a single variable, which is not the case for ABRO.

[4] All examples a publicly available under `http://www.Averest.org/examples`.

MinePumpController starts or stops the pump of a mine according to alerts issued by the carbon dioxide and methane monitors, as well as the water level. It contains **40** *SGAs*, **27** inputs, **30** outputs, and **7** safety specifications.

RSFlipFlop describes a RS-Flipflop with NOR-gates of equal delay, modeled as a single macro step. It contains **7** *SGAs*, **2** inputs, **2** outputs, and **8** specifications (three safety and five co-Büchi).

MemoryController models a memory controller providing mutual exclusion by maintaining region locks for addresses. The implementation contains **41** *SGAs*, **5** inputs, **12** outputs, and **8** safety specifications.

IslandTrafficControl: An island is connected via a tunnel with the mainland. Inside the tunnel is a single lane so that cars can either travel from the mainland to the island or vice versa, which is signaled by traffic lights on both ends of the tunnel. It is represented by **75** *SGAs*, **15** inputs, **32** outputs, and **13** specifications (eleven safety and two Büchi) modeled in **5** modules.

The table in Figure 7 roughly measures the size of the original program regarding the number of *SGAs* (#SGA), as well as the size of each representation in terms of the number of guarded commands (#GC) for GC, number of modules and guarded commands (#SC[5]) for SC, and the number of equations (#ES) for ES. Interestingly, ABROM contains only 2 guarded commands in GC, while having about 20 *SGAs* in the original Quartz program. This happens because ABROM only features delayed assignments, which according to the transformation described in 4.1, are combined into a single guarded command called *conclusion*. This also explains the particularly good performance for GC on verifying it (Figure 7). The number of equations used in ES corresponds with the number of modules used for SC. For each variable an equation is contained in ES and SC contains besides the module for the synchronous composition for each variable a module. In case a carrier variable has to be introduced for ES, they will differ.

P	GC		SC		ES	
	#V	#N	#V	#N	#V	#N
ABRO	34	786	30	616	14	183
ABROM[M=10]	80	2289	98	3855	52	1221
ABROM[M=13]	98	3339	122	3381	64	2065
AuntAgatha	100	1434	40	130	48	180
VendingMachine	130	10855	138	4496	22	384
LightControl	98	7108	114	6908	44	837
MinePumpController	126	98949	136	8500	36	636
RSFlipFlop	36	1883	38	929	24	1010
MemoryController	158	931422	188	92815	98	48134
IslandTrafficControl	144	773001	212	58217	58	30401

Figure 8: BDD size in terms of the number of variables (#V) and number of nodes (#N)

As for the actual performance of each representation, Figure 7 shows that ES is generally faster than GC or SC. In the worst case (IslandTrafficControl), it was more than 250 times faster than an equivalent program in the GC representation and roughly 32 times faster than SC. Not surprisingly, the complexity of the properties can also increase the verification time in certain cases, as with the

[5]The number inside parenthesis is the sum of the number of guarded commands in the context.

GC representation for RSFlipFlop, which regardless of being a fairly minimal Quartz program, contains complex assertions concerning the stability of the circuit. To further corroborate that the ES representation is indeed the best representation, we measured the size of the BDD with respect to the number of variables (#V) and the number of nodes (#N) for each representation. As seen on Figure 8, the size of the BDD for the ES transformation was usually smaller than its counterparts, which certainly relates strongly to the times measured in Figure 7.

6. Conclusions

We considered three different ways to represent synchronous systems in SAL's transition language, and evaluated them concerning the performance of model checking. The chosen representations differ in the number of guarded actions that are chosen for execution in every reaction step. As a result, we can clearly say that the ES transformation, which represents the system as definitions (equations) in SAL, is in general the best choice.

References

[BBS11] Baudisch, D., J. Brandt, and K. Schneider: *Translating synchronous systems to data-flow process networks*. In Yeo, S. S., B. Vaidya, and G.A. Papadopoulos (editors): *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 354–361, Gwangju, Korea, 2011. IEEE Computer Society.

[BCE+03] Benveniste, A., P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone: *The synchronous languages twelve years later*. Proceedings of the IEEE, 91(1):64–83, 2003.

[BG92] Berry, G. and G. Gonthier: *The Esterel synchronous programming language: Design, semantics, implementation*. Science of Computer Programming, 19(2):87–152, 1992.

[BGS10] Brandt, J., M. Gemünde, and K. Schneider: *From synchronous guarded actions to SystemC*. In Dietrich, M. (editor): *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.

[BS08] Brandt, J. and K. Schneider: *Formal reasoning about causality analysis*. In Mohamed, O. Ait, C. Muñoz, and S. Tahar (editors): *Theorem Proving in Higher Order Logics (TPHOL)*, volume 5170 of *LNCS*, pages 118–133, Montréal, Québec, Canada, 2008. Springer.

[Dij75] Dijkstra, E.W.: *Guarded commands, nondeterminacy and formal derivation of programs*. Communications of the ACM (CACM), 18(8):453–457, 1975.

[DNS06] Derrick, J., S. North, and T. Simons: *Issues in implementing a model checker for Z*. In Liu, Z. and J. He (editors): *Formal Methods and Software Engineering (FMSE)*, volume 4260 of *LNCS*, pages 678–696, Macao, China, 2006. Springer.

[DNS11] Derrick, J., S. North, and A.J.H. Simons: *Z2SAL: a translation-based model checker for Z*. Formal Aspects of Computing, 23(1):43–71, January 2011.

[GS13] Gesell, M. and K. Schneider: *Translating synchronous guarded actions to interleaved guarded actions*. In *Formal Methods and Models for Codesign (MEMOCODE)*. IEEE Computer Society, 2013.

[GV08] Grumberg, O. and H. Veith (editors): *25 Years of Model Checking – History, Achievements, Perspectives*, volume 5000 of *LNCS*. Springer, 2008.

[Hal93] Halbwachs, N.: *Synchronous programming of reactive systems*. Kluwer, 1993.

[Hoa78] Hoare, C.A.R.: *Communicating sequential processes*. Communications of the ACM (CACM), 21(8):666–677, 1978.

[PSSD00] Park, D.Y.W., U. Stern, J.U. Skakkebaek, and D.L. Dill: *Java model checking*. In *Automated Software Engineering (ASE)*, pages 253–256, Grenoble, France, 2000. IEEE Computer Society.

[SB08] Schneider, K. and J. Brandt: *Performing causality analysis by bounded model checking*. In *Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi'an, China, 2008. IEEE Computer Society.

[Sch09] Schneider, K.: *The synchronous programming language Quartz*. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

Implication Graph Compression inside the SMT Solver iSAT3*

Karsten Scheibler
Albert Ludwigs Universität
Freiburg
scheibler@informatik.uni-freiburg.de

Bernd Becker
Albert Ludwigs Universität
Freiburg
becker@informatik.uni-freiburg.de

Abstract

The iSAT algorithm aims at solving boolean combinations of linear and non-linear arithmetic constraint formulas (including transcendental functions), and thus is suitable to verify safety properties of systems consisting of both, linear and non-linear behaviour. The iSAT algorithm tightly integrates interval constraint propagation into the conflict-driven clause-learning framework. During the solving process, this may result in a huge implication graph. This paper presents a method to compress the implication graph on-the-fly. Experiments demonstrate that this method is able to reduce the overall memory footprint up to an order of magnitude.

1. Introduction

Computer-aided verification techniques have been intensively studied over the past decade and are becoming increasingly accepted by the industry. Applications range from hardware verification of digital circuits to the analysis of embedded systems. Nowadays, embedded systems are a hybrid of digital components and additional analog parts (e.g. a digital processor controlling a robotic arm and reacting to analog sensor inputs). An increasing number of applications in particular in the verification area are applying SAT Modulo Theory solvers for finding errors or proving the absence of errors in such systems. When modeling hybrid systems, boolean combinations of non-linear arithmetic functions (including transcendental functions like $\sin$, $\cos$, and $\exp$) arise naturally. That is one main reason the iSAT algorithm has been invented in [FHT+07, Her11]. This algorithm can be used for determining the satisfiability of formulas containing arbitrary boolean combinations of linear and non-linear arithmetic constraints.

*This work has been partially supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (DFG, SFB/TR 14 AVACS, http://www.avacs.org/) and by the Cluster of Excellence BrainLinks-BrainTools (DFG, grant number EXC 1086, http://www.brainlinks-braintools.uni-freiburg.de/)

The iSAT algorithm extends the conflict-driven clause-learning (CDCL) framework – successfully used in propositional SAT solving – by seamlessly integrating interval constraint propagation (ICP) into it. Therefore, deductions made by ICP are also recorded in the implication graph used by CDCL to allow efficient backtracking and conflict clause creation. Depending on the input formula ICP may produce long deduction sequences resulting in a huge implication graph. In this paper we present an approach to reduce the size of the implication graph considerably and thus decrease the overall memory footprint.

The paper is structured as follows. After introducing the iSAT algorithm in Section 2, we give a motivating example in Section 3, before describing our method to compress the implication graph in Section 4. In Section 5 we discuss the experimental results and conclude with a summary in Section 6.

2. Preliminaries

2.1. CDCL, Resolution, SMT

Given a propositional formula and asking the question if there exists an assignment to its variables rendering the formula `true` is also known as the *satisfiability problem*. Programs for solving this kind of problem are called SAT solvers. A very simple approach to tackle the satisfiability problem for a given formula is to try every possible assignment to its variables systematically. For a formula containing n variables this would require to try up to 2^n different variable assignments. With $n > 100$ this method won't give an answer in reasonable time.

Fifty years ago DPLL [DLL62] was proposed. This method exploits that every boolean formula can be rewritten to an equi-satisfiable *conjunctive normal form* (CNF) using the Tseitin-transformation [Tse68]. A CNF consists of a conjunction of clauses with each clause being a disjunction of literals and a literal being a boolean variable or its negation. To satisfy a formula in CNF every clause in the formula has to be satisfied. As a consequence, if an assignment to a subset of the variables already results in an unsatisfied clause (a so-called *conflict*) every assignment containing this subset is known to not satisfy the formula. This results in a huge reduction of variable assignments to be tested.

Additionally *boolean constraint propagation* (BCP) is used to detect implied assignments. Everytime a clause with n literals contains $n - 1$ literals being already assigned to `false`, the remaining literal has to be `true` in order to retain a chance to satisfy the formula. Such a clause implying a literal is called an *implication clause*. Moreover, today's SAT solvers add conflict clauses to the formula to prune the search space even further – so-called *conflict-driven clause learning* (CDCL) [SS96]. To allow backtracking and to ease the creation of a conflict clause an *implication graph* is used. It stores all currently assigned literals and keeps a link to the implication clause in case a literal was implied.

For the creation of conflict clauses *resolution* [Rob65] is used. To resolve two clauses C_1 and C_2 regarding a variable v both clauses have to contain v – but in opposite polarities. For example the clauses $C_1 = (a \vee \neg b \vee c)$ and $C_2 = (\neg d \vee e \vee \neg c)$ could be resolved regarding c yielding the resolvend $R = (a \vee \neg b \vee \neg d \vee e)$. The resolvend is always satisfied if the two originating clauses are satisfied. When adding a further conjunct to a CNF one may only exclude solutions but may not add new ones, therefore $C_1 \wedge C_2$ is equivalent to $C_1 \wedge C_2 \wedge R$.

SAT Modulo Theory (SMT) lifts the CDCL working principle to a higher level. In SMT every literal may represent a theory atom, e.g. $(x + y < 10)$. The SAT solver now works on the boolean abstraction of the underlying problem and assigns `true` or `false` to the theory atoms. If the SAT solver finds a satisfying assignment for the boolean abstraction, a theory solver is used to check if the conjunction of theory atoms satisfying the clauses is indeed satisfiable. If this is not the case the boolean abstraction is refined with a conflict clause which forbids the conflicting theory atoms. This scheme is also abbreviated as DPLL(T) or CDCL(T) – with T being the theory used.

2.2. The iSAT algorithm, iSAT3

The iSAT algorithm [FHT+07, Her11] uses *interval constraint propagation* (ICP, see e.g. [BG06]) to check the consistency of theory atoms – but there is no straight separation between the theory and SAT solver part. Instead ICP is tightly integrated into the CDCL framework. This deep integration has the advantage of sharing the common core of the search algorithms between the propositional and the theory-related part of the solver. So strictly speaking the iSAT algorithm goes beyond CDCL(ICP).

Theory atoms in the iSAT algorithm may contain linear and non-linear arithmetic involving transcendental functions. Examples for theory atoms are: $(x^2 + y^2 = z^2)$, $(|v - w| < \min(v, w))$ or $(\sqrt[3]{x} + \sin y < e^z)$. The iSAT algorithm supports boolean, integer- and real-valued variables Additionally every variable is bounded.

Before solving a given formula each theory atom is decomposed by a Tseitin-like transformation into *simple bounds* and unit clauses containing *primitive constraints*. During this process fresh auxiliary variables are introduced. A simple bound imposes a lower or an upper bound to a variable. Such a bound could be strict, e.g. $(x < 4)$ or non-strict, e.g. $(x \leq 4)$. Since each bound is represented as a floating point number, we use outward rounding (this means we round down for lower bounds and round up for upper bounds), to get a safe interval enclosure. Having a fixed set of primitive constraints makes it easier to apply ICP later on. A primitive constraint contains besides an unary or binary operator up to three associated variables. The following primitive constraints are supported:

unary primitive constraints	$h = -x$	
	$h = \lvert x \rvert$	
	$h = \sin x$	
	$h = \cos x$	
	$h = \exp x$	
	$h = x^n$	($n \geq 0$ integer constant)
	$h = \sqrt[n]{x}$	($n \geq 1$ integer constant)
binary primitive constraints	$h = x + y$	
	$h = x - y$	
	$h = x \cdot y$	
	$h = \min(x, y)$	
	$h = \max(x, y)$	

During the search process a so-called contractor is used to narrow the intervals occuring in each primitive constraint. Let's illustrate this with a small example. Assume the primitive constraint $(x = y + z)$ is given together with the following intervals for the contained variables:

$x \in [1,9], y \in [1,3]$ and $z \in [4,10]$. To contract the interval of x, the intervals of y and z are added. This yields the interval $[5,13]$, which is then intersected with the original interval of x and results in $[5,9]$ as the new interval for x. To contract the interval of y the primitive constraint is redirected to $y = x - z$. The resulting interval does not provide a stronger lower or upper bound, so the interval for y stays unchanged. After redirecting the primitive constraint to $z = x - y$ the interval for z is contracted to $[4,8]$. Contractors for other primitive constraints work in a similar way.

The three basic elements of the CDCL framework – (1) propagate, (2) resolve conflicts, (3) decide – are also present in the iSAT algorithm, but are extended for the operation on integer- and real-valued intervals in addition to boolean variables. Deciding an integer- or real-valued variable corresponds to splitting its interval and selecting the lower or upper half. Furthermore, in the propagation phase ICP is executed in addition to BCP. During ICP it is checked if every primitive constraint is still consistent with the current interval valuation of the variables. An interval valuation $\rho : \mathit{Var} \to \mathbb{I}_{\mathbb{R}}$ is a mapping from a set of variables *Var* to a set of intervals $\mathbb{I}_{\mathbb{R}}$. A unit clause containing a primitive constraint c is *inconsistent* under an interval valuation ρ, iff no values in the intervals $\rho(x)$ of the variables x in c can satisfy c, i.e.

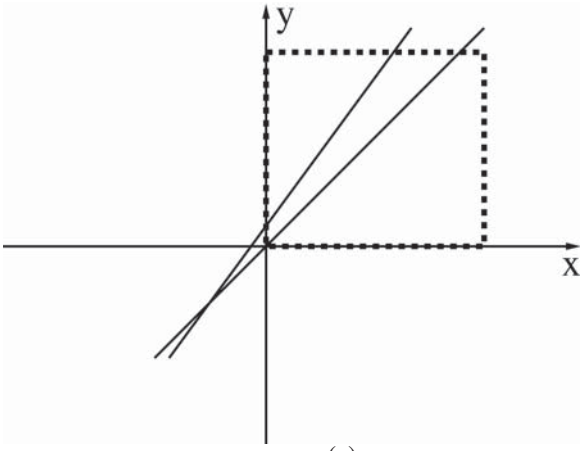
$$
\begin{array}{lcll}
\neg\exists v \in \rho(x) & : & v \sim r \quad \text{if } c = (x \sim r), & (r \in \mathbb{Q}) \\
\neg\exists v \in \rho(x), \neg\exists v' \in \rho(\circ y) & : & v \sim v' \quad \text{if } c = (x = \circ y), & (\circ \in \{-, \mathrm{abs}, \sin, \cos, \exp, \sqrt[n]{\cdot}, \cdot^n\}) \\
\neg\exists v \in \rho(x), \neg\exists v' \in \rho(y \circ z) & : & v \sim v' \quad \text{if } c = (x = y \circ z) & (\circ \in \{+, -, *\})
\end{array}
$$

where $\sim \in \{<, \leq, \geq, >\}$. Otherwise c is *consistent* under ρ.

If the primitive constraint is consistent and a new bound was deduced it is checked if the newly deduced bound has a negligible progress compared to the existing bound. If this is the case the new bound is ignored to prevent infinite propagation sequences and thus to guarantee termination. Furthermore, the iSAT algorithm may terminate with an inconclusive answer, because in general equations like $x = y \cdot z$ can only be satisfied by point intervals. However, reaching such point intervals by ICP cannot be guaranteed for real-valued variables. In such cases iSAT will return a so called *candidate solution*.

As mentioned, the iSAT algorithm uses interval splitting to decide integer- and real-valued variables. To enforce termination of the algorithm, interval splits are only performed if the considered interval is larger the so-called *minimal splitting width*. Additionally, every newly deduced (non-conflicting) bound has to be larger or equal to the *minimum progress* – otherwise the deduced bound is discarded. These two parameters may be specified by the user.

In this paper we focus on the third implementation of the iSAT algorithm – which we name iSAT3 [SKB13]. All three implementations (with HySAT [HEFT08] and iSAT [EKKT08] being the first two) share the same major core principles of tightly integrating ICP into the CDCL framework. However, while the core solvers of HySAT and iSAT operate on simple bounds, the core of iSAT3 uses literals and additionally utilizes an *abstract syntax graph* (ASG) for more advanced formula preprocessing. At first glance it looks like a minor design decision to choose between simple bounds and literals – because every simple bound can be seen as a literal and every literal can be expressed as a simple bound – but as the results in [SKB13] show it is beneficial to decide for literals, because it makes it easier to apply techniques known from propositional SAT solving like conflict clause minimization [SB09], assumptions [ES03] and further techniques [PD07, CHS09].

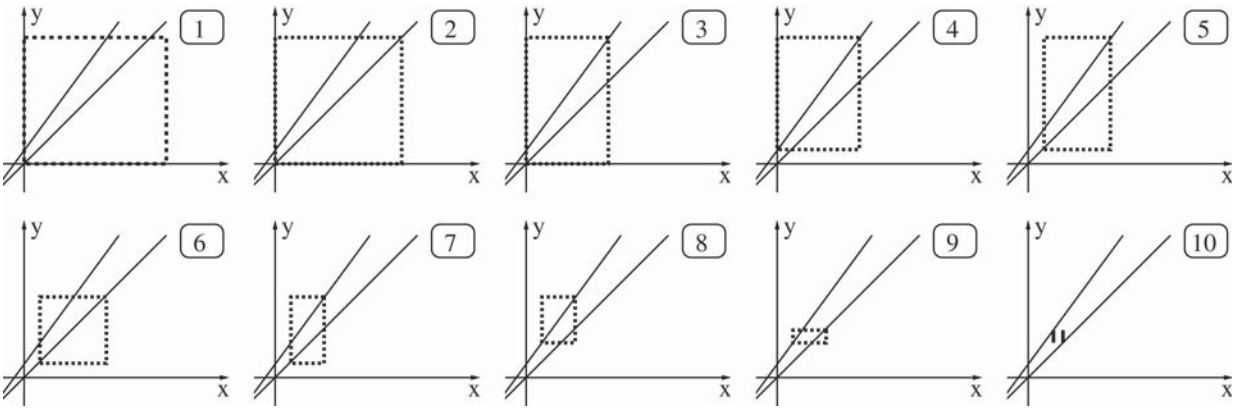


(a)

```
DECL
float [0,1000000] x,y;

EXPR
y = 2.00001*x + 0.25;
y = 2*x;
```

(b)



(c)

Figure 1: A small benchmark with two linear theory atoms listed in (b). There is no intersection within the given initial intervals $x, y \in [0, 1000000]$ as illustrated in (a). ICP will continuously shrink the intervals (like shown in (c) 1-9) until it finally deduces contradicting bounds for a variable in (c) 10. The pictures in (a) and (c) are only a rough illustration of what is happening, because for this example ICP needs in fact millions of deductions until the conflict is discovered.

3. Motivating example

We illustrate the inner workings of iSAT3 with the help of a small example shown in Figure 1(b). To keep it simple, we omitted any boolean structure or further theory atoms. Within the benchmark the variables x and y are declared – both within the initial interval $[0, 1000000]$. Furthermore, two linear theory atoms are specified. Every theory atom describes a line in the $\mathbb{R}^2$ space. These two lines are very close, but they have no intersection inside the initial intervals as illustrated in Figure 1(a). So this benchmark is unsatisfiable. With gaussian elimination this could be easily determined. ICP is also able to detect the unsatisfiability, but it needs a long deduction sequence to do so – exactly this is the purpose of this artificial example. Now one could get the impression that ICP is a weak procedure if it needs that many deductions. However, the beauty of ICP lies in its simple elegance and the ability to handle linear, non-linear and transcendental functions all within the same framework – while gaussian elimination is tailored for handling only quadratic systems of linear equations.

1	$(y$	$=$	$2.00001 * x + 0.25)$			$\wedge$	$(y$	$=$	$2 * x)$		
2	$(0$	$=$	$x + (-\frac{100000}{200001} * y) + (\frac{100000}{200001} * 0.25))$			$\wedge$	$(0$	$=$	$x + (-0.5 * y))$		
3	$(-\frac{25000}{200001}$	$=$	$x + (-\frac{100000}{200001} * y))$			$\wedge$	$(0$	$=$	$x + (-0.5 * y))$		
4a	$(t2 = x + t1) \wedge (t1 = c1 * y) \wedge$						$(t4 = x + t3) \wedge (t3 = c2 * y) \wedge$				
4b	$(c1 \geq \lfloor -\frac{100000}{200001} \rfloor) \wedge (c1 \leq \lceil -\frac{100000}{200001} \rceil) \wedge$					$\bigwedge$	$(c2 \geq -0.5) \wedge (c2 \leq -0.5) \wedge$				
4c	$(t2 \geq \lfloor -\frac{25000}{200001} \rfloor) \wedge (t2 \leq \lceil -\frac{25000}{200001} \rceil)$						$(t4 \geq 0) \wedge (t4 \leq 0)$				
5a	$(t2 = x + t1) \wedge (t1 = c1 * y) \wedge (t4 = x + t3) \wedge (t3 = c2 * y) \wedge$										
5b	$(c1 \geq \lfloor -\frac{100000}{200001} \rfloor) \wedge (c1 \leq \lceil -\frac{100000}{200001} \rceil) \wedge (c2 \geq -0.5) \wedge (c2 \leq -0.5) \wedge$										
5c	$(\neg b1 \vee (t2 \geq \lfloor -\frac{25000}{200001} \rfloor)) \wedge (\neg b1 \vee (t2 \leq \lceil -\frac{25000}{200001} \rceil)) \wedge (b1 \vee \neg(t2 \geq \lfloor -\frac{25000}{200001} \rfloor) \vee \neg(t2 \leq \lceil -\frac{25000}{200001} \rceil)) \wedge$										
5d	$(\neg b2 \vee (t4 \geq 0)) \wedge (\neg b2 \vee (t4 \leq 0)) \wedge (b2 \vee \neg(t4 \geq 0) \vee \neg(t4 \leq 0)) \wedge$										
5e	$(\neg b3 \vee b1) \wedge (\neg b3 \vee b2) \wedge (b3 \vee \neg b1 \vee \neg b2) \wedge (b3)$										

Figure 2: The translation of the original theory atoms to CNF.

										deduced	current
1	$t4_{lb}$	$=$	$x_{lb} + t3_{lb}$	$=$	0	$+$	-500000	$=$	-500000	-500000	0
2	$t4_{ub}$	$=$	$x_{ub} + t3_{ub}$	$=$	1000000	$+$	0	$=$	1000000	1000000	500000
3	x_{lb}	$=$	$t4_{lb} - t3_{ub}$	$=$	-500000	$-$	0	$=$	-500000	-500000	0
4	x_{ub}	$=$	$t4_{ub} - t3_{lb}$	$=$	0	$-$	-500000	$=$	500000	1000000	500000
5	$t3_{lb}$	$=$	$t4_{lb} - x_{ub}$	$=$	-500000	$-$	0	$=$	-500000	-500000	-500000
6	$t3_{ub}$	$=$	$t4_{ub} - x_{lb}$	$=$	1000000	$-$	1000000	$=$	0	0	0

Figure 3: ICP for the primitive constraint $(t4 = x + t3)$. Here ICP was able to deduce a new stronger upper bound for x in line 4.

Before solving a benchmark, iSAT3 normalizes it and converts it into a CNF. With the help of the auxiliary variables $t1, t2, t3, t4, c1, c2, b1, b2$ and $b3$, the theory atoms are decomposed into primitive constraints and simple bounds. Figure 2 summarizes what happens during formula preprocessing and Tseitin-transformation. Lines 1-3 show how the theory atoms are rewritten. Lines 5a-5e list the complete CNF – but for sake of simplicity we will use the CNF shown in lines 4a-4c, because unit propagation will directly assign $b1 = b2 = b3 =$`true`. The conjuncts with a darker gray background are primitive constraints, the ones with a lighter gray background are simple bounds. A simple bound containing a number like $\lfloor 0.1 \rfloor$ represents the largest floating pointing number smaller or equal to 0.1 – analogously does $\lceil 0.1 \rceil$ stand for the smallest floating point number larger or equal to 0.1 As a last step we add the initial bounds as unit clauses to the formula – if there is no bound information contained already for the associated variable. This results in the following formula:

$$(t2 = x + t1) \wedge (t1 = c1 * y) \wedge (t4 = x + t3) \wedge (t3 = c2 * y) \wedge (c1 \geq \lfloor -\frac{100000}{200001} \rfloor) \wedge (c1 \leq \lceil -\frac{100000}{200001} \rceil) \wedge$$
$$(c2 \geq -0.5) \wedge (c2 \leq -0.5) \wedge (t2 \geq \lfloor -\frac{25000}{200001} \rfloor) \wedge (t2 \leq \lceil -\frac{25000}{200001} \rceil) \wedge (t4 \geq 0) \wedge (t4 \leq 0) \wedge (x \geq 0) \wedge (x \leq 1000000) \wedge$$
$$(y \geq 0) \wedge (y \leq 1000000) \wedge (t1 \geq \lfloor -1000000 \frac{100000}{200001} \rfloor) \wedge (t1 \leq 0) \wedge (t3 \geq -500000) \wedge (t3 \leq 0)$$

Assume this formula is now passed to the solver core of iSAT3. Since every simple bound is interpreted as a literal, BCP will process all unit clauses containing the initial bounds and marks every variable for which a new bound was set. Then ICP is executed and looks at all primitive constraints containing a variable marked during the BCP stage. Regarding our example assume x

was marked first and ICP looks at the primitive constraint $(t4 = x + t3)$. Based on the current lower and upper bounds of $t4, x$ and $t3$ ICP will now try to deduce stronger lower and upper bounds for all three variables. As Figure 3 shows, only the value for x_{ub} deduced in line 4 is an improvement compared to the current x_{ub}. A new literal is created to represent this simple bound. Every assignment to a literal results in a new entry in the implication graph. While a decision has no associated implication clause, each deduced literal needs to have a clause implying it. Regarding x_{ub}, the reasons for this deduction are the values of $t4_{ub}$ and $t3_{lb}$. This results in the implication shown in Figure 4 line 1 – lines 2-4 show how it is possible to reformulate this implication as a clause. Later on, such clauses will contain the newly deduced bound as first literal as can be seen in Figure 5. ICP then continues to deduce new values for $x, y, t1$ and $t3$. Figure 1(c) illustrates this. From a geometrical point of view ICP constructs wrapping boxes around each theory atom and calculates the intersection of those boxes. This is done as long as new non-conflicting bounds can be deduced. Figure 5 shows a short excerpt of the deductions made by ICP in iSAT3 for this example. As the numbers in Figure 5 suggest, the intervals of x and y are slowly shrinking. In fact ICP will need millions of deductions to finally discover the conflict. With millions of entries one can imagine that the implication graph will occupy a huge amount of memory. This problem will now be addressed by the method we present in the next section.

1	$((t4 \leq 0)$	$\wedge$	$(t3 \geq -500000))$	$\Rightarrow$	$(x \leq 500000)$
2	$\neg((t4 \leq 0)$	$\wedge$	$(t3 \geq -500000))$	$\vee$	$(x \leq 500000)$
3	$\neg(t4 \leq 0)$	$\vee$	$\neg(t3 \geq -500000)$	$\vee$	$(x \leq 500000)$
4	$(t4 > 0)$	$\vee$	$(t3 < -500000)$	$\vee$	$(x \leq 500000)$

Figure 4: Creation of the implication clause for the deduced new upper bound for x.

4. Implication Graph Compression

As Figure 5 shows newly deduced bounds (e.g. the new upper bound for x deduced in line 2) themselves occur as reasons later on (for example in line 4). This is due to the fact that for all ICP operations the current bounds are used – resulting in the associated literals to occur in the implication clauses. The basic idea to compress the implication graph is to throw away all *intermediate bounds* and to adapt the implication clauses of the remaining bounds accordingly. In this context an intermediate bound is a bound which is only needed to deduce further stronger bounds. Regarding our motivating example almost all bounds are intermediate bounds. In fact for this example it would be enough to create an implication clause containing the initial bounds as reasons and implying a contradicting bound for one variable.

The question is now how the implication clauses should be adapted. For a given implication clause we need to replace all contained intermediate bounds with their own reasons – as long as all reasons do not contain intermediate bounds themselves. Strictly speaking this is exactly what resolution does. Regarding Figure 5 line 4, we would resolve with the clause in line 2 regarding the literal $(x \leq 499997.37501312\ldots)$ and its negation $(x > 499997.37501312\ldots)$, resulting in the new implication clause:

$$(t3 \geq -499997.37501312\ldots) \vee (t4 < 0) \vee (t2 > -0.12499937\ldots) \vee (t1 < -499997.50001249\ldots)$$

This clause states that $(t3 \geq -499997.37501312\ldots)$ can be deduced because of bounds which are no intermediate bounds. As there is no other clause using $(x \leq 499997.37501312\ldots)$ as a reason,

1	$(x \leq 500000)$	$\vee$	$(t4 > 0)$	$\vee$	$(t3 < -500000)$
2	$(x \leq 499997.37501312\ldots)$	$\vee$	$(t2 > -0.12499937\ldots)$	$\vee$	$(t1 < -499997.50001249\ldots)$
3	$(t1 \leq -0.12499937\ldots)$	$\vee$	$(t2 > -0.12499937\ldots)$	$\vee$	$(x < 0)$
4	$(t3 \geq -499997.37501312\ldots)$	$\vee$	$(t4 < 0)$	$\vee$	$(x > 499997.37501312\ldots)$
5	$(y \geq 0.24999999\ldots)$	$\vee$	$(t1 > -0.12499937\ldots)$		
6	$(t3 \leq -0.12499999\ldots)$	$\vee$	$(y < 0.24999999\ldots)$		
7	$(y \leq 999994.75002625\ldots)$	$\vee$	$(t3 < -499997.37501312\ldots)$		
8	$(x \geq 0.12499999\ldots)$	$\vee$	$(t4 < 0)$	$\vee$	$(t3 > -0.12499999\ldots)$
9	$(t1 \geq -499994.87503874\ldots)$	$\vee$	$(y > 999994.75002625\ldots)$		
10	$(x \leq 499994.75003937\ldots)$	$\vee$	$(t2 > -0.12499937\ldots)$	$\vee$	$(t1 < -499994.87503874\ldots)$
11	$(t1 \leq -0.24999937\ldots)$	$\vee$	$(t2 > -0.12499937\ldots)$	$\vee$	$(x < 0.12499999\ldots)$
12	$(y \geq 0.50000124\ldots)$	$\vee$	$(t1 > -0.24999937\ldots)$		
13	$(t3 \geq -499994.75003937\ldots)$	$\vee$	$(t4 < 0)$	$\vee$	$(x > 499994.75003937\ldots)$
14	$(t3 \leq -0.25000062\ldots)$	$\vee$	$(y < 0.50000124\ldots)$		
15	$(y \leq 999989.50007874\ldots)$	$\vee$	$(t3 < -499994.75003937\ldots)$		
16	$(t1 \geq -499992.25007812\ldots)$	$\vee$	$(y > 999989.50007874\ldots)$		
17	$(x \geq 0.25000062\ldots)$	$\vee$	$(t4 < 0)$	$\vee$	$(t3 > -0.25000062\ldots)$
18	$(x \leq 499992.12507874\ldots)$	$\vee$	$(t2 > -0.12499937\ldots)$	$\vee$	$(t1 < -499992.25007812\ldots)$
19	$(t1 \leq -0.37500000\ldots)$	$\vee$	$(t2 > -0.12499937\ldots)$	$\vee$	$(x < 0.25000062\ldots)$
20	$(t3 \geq -499992.12507874\ldots)$	$\vee$	$(t4 < 0)$	$\vee$	$(x > 499992.12507874\ldots)$
...			...		

Figure 5: An excerpt of the deductions done by ICP regarding the benchmark from Figure 1(b). Floating point numbers with ... are truncated for better readability

the bound itself is now obsolete and can be removed from the implication graph. Of course the new bound for $t3$ occurs itself as a reason for another implication (Figure 5 line 7). Here again, we do resolution and resolve regarding $(t3 \geq -499997.37501312\ldots)$ and its negation. We get an implication clause with no intermediate bounds and because $(t3 \geq -499997.37501312\ldots)$ is not used in another place we can remove it.

So the idea is to do resolution between the implication clauses – but instead of doing this as a post-processing step after the implication graph already exists, we want to do this on-the-fly and therefore prevent the growth of implication graph right from the start. To do so, we delay the addition of newly deduced bounds to the implication graph – instead we store them temporarily together with their implication clauses. The pseudocode of the algorithm is shown in Figure 6. In line 2 ICP is executed and returns an implication clause. In this clause the first literal is always the implied literal. A new clause is created in line 3 and used in lines 4-15 to resolve intermediate bounds. In line 7 the associated variable of a simple bound is determined, e.g. x for the simple bound $(x \leq 500000)$. In line 9 it is checked if a variable has already a temporary implication clause for its lower or upper bound. If yes, we can do resolution between this temporary implication clause and the current implication clause in line 11. Here we exploit that the implied literal is always stored in the first position of an implication clause. Then in line 20 and 21 we store the new bound together with its implication clause. These values are written to the implication graph in line 25 and 28 if (1) BCP is able to deduce implied literals inbetween, (2) or no further deductions are possible in the current decision level.

If the sheer amount of the intermediate bounds is the problem, then it could seem tempting to favor CDCL(ICP) over the tight integration of ICP into CDCL – because in such a scenario the implication graph would never contain intermediate bounds. On the other hand if we would move ICP into a separate theory solver which is only used for consistency checks of the theory atoms

```
1  on_the_fly_compression() {
2      clause = do_icp();
3      c = new clause;
4      append(c, clause[0]);
5      for (i = 1; i < length(clause); i++) {
6          literal = clause[i];
7          variable = get_associated_variable(literal);
8          is_lb = is_lower_bound(literal);
9          if (tmp_clauses[variable, !is_lb] != NULL) {
10             t = tmp_clauses[variable, !is_lb];
11             for (j = 1; j < length(t); j++) append(c, t[j]);
12         } else {
13             append(c, clause[i]);
14         }
15     }
16     clause = c;
17     variable = get_associated_variable(clause[0]);
18     is_lb = is_lower_bound(clause[0]);
19     variables[is_lb] = variables[is_lb] ∪ variable
20     tmp_bounds[variable, is_lb] = clause[0];
21     tmp_clauses[variable, is_lb] = clause;
22 }
23 update_impl_graph() {
24     foreach (v in variables[0]) {
25         append_to_impl_graph(tmp_bounds[v, 0], tmp_clauses[v, 0]);
26     }
27     foreach (v in variables[1]) {
28         append_to_impl_graph(tmp_bounds[v, 1], tmp_clauses[v, 1]);
29     }
29     clear(variables, tmp_bounds, tmp_clauses);
31 }
```

Figure 6: Pseudocode of the two core functions needed for implication graph compression. The function `on_the_fly_compression()` resolves the implication clauses on-the-fly and stores the current bounds. If those bounds and the implication clauses have to be stored in the implication graph the function `update_impl_graph()` will be used.

satisfying a clause, we would lose the ability to trigger boolean propagations as early as possible. With our approach we get both, (1) we are still able to quickly alternate between ICP and BCP and (2) we have a compact implication graph.

If we now apply the algorithm of Figure 6 to the benchmark from Figure 1(b) the overall memory footprint drops from 620 MB down to 6.5 MB. This is not surprising, because the benchmark was designed to trigger very long ICP sequences. Therefore, the next section will answer the more interesting question, how the approach will perform on a set of standard benchmarks.

5. Experimental Results

Table 1 shows the experimental results over a set of standard benchmarks for the iSAT algorithm. We selected those benchmarks with a considerable amount of ICP to measure the effectiveness of our method. In the first column the names of the benchmarks are listed. Columns 2 and 3 contain

Benchmark	without IGC		with IGC		Change	
	Runtime (s)	Memory (MB)	Runtime (s)	Memory (MB)	Runtime	Memory
minigolf	15.05	526.77	7.43	59.02	-50.63%	-88.80%
train_system1	82.18	1063.75	117.86	486.09	+43.42%	-54.30%
train_system2	15.28	967.59	10.46	66.45	-31.54%	-93.13%
train_system3	825.46	7824.00	818.55	1489.50	-0.84%	-80.96%
train_system3*	-	MEMOUT	2539.81	1858.54	-	-
train_distance_ctrl1	8.11	578.41	10.79	89.62	+33.05%	-84.51%
train_distance_ctrl2	13.83	486.88	21.16	115.16	+53.00%	-76.35%
train_distance_ctrl3	15.81	521.56	25.82	198.25	+63.31%	-61.99%
etcs_train_system	200.08	2364.00	194.75	361.73	-2.66%	-84.70%
renault_clio	1.54	154.52	2.44	208.34	+58.44%	+34.83%
collision_avoidance1	73.69	670.46	43.61	16.73	-40.82%	-97.50%
collision_avoidance2	51.51	1004.32	37.01	16.32	-28.15%	-98.38%
hysat_6_bus	407.15	696.92	157.08	247.80	-61.42%	-64.44%
bouncing_ball_euler	2.36	72.58	3.40	59.97	+44.07%	-17.37%
arbiter	58.38	105.62	105.79	169.95	+81.21%	+60.91%
$\sum$	1770.43	17037.38	1556.15	3584.93	-12.10%	-78.96%

Table 1: Comparison between iSAT3 with and without implication graph compression. The runtime (User time) and memory consumption (Maximum resident set size) was measured with `time -v`.

the runtime and memory consumption for iSAT3 without implication graph compression, while Columns 4 and 5 show the values for iSAT3 with implication graph compression. In Columns 6 and 7 the change of runtime and memory consumption is recorded.

The results show that our method is able to reduce the memory consumption considerably – for some benchmarks even by more than an order of magnitude. As expected the runtime keeps roughly unchanged summed up over all benchmarks. Some benchmarks are solved faster, while others need more runtime. This is due to the fact that the decision heuristics is influenced indirectly. The solver core of iSAT3 uses the Variable State Independent Decaying Sum (VSIDS) heuristics when a decision has to be made. VSIDS calculates a so-called activity for every variable. This activity depends on how often a variable is seen during the conflict analysis – due to the compression of the implication graph this is likely to be different. This leads to other activity values and may therefore guide the solver to traverse the search space in a different way – leading to changed runtimes for individual benchmarks.

As stated in Section 2 the user may specify the two parameters minimal splitting width and minimum progress to influence the solving process. Usually smaller values reduce the number of candidate solutions and increase the number of conclusive answers, but at the cost of more deductions and decisions to be made – which is likely to increase runtime and the size of the implication graph. If we compress the implication graph in such situations, this may allow completing the solving process instead of hitting a MEMOUT inbetween. The benchmark train_system3 is listed twice in Table 1 and is an example for such a case. In the second run marked with $\star$ we used a minimal splitting width of 0.0001 and a minimum progress of 0.00001 – this is a hundredth of the values used for all other measurements. The benchmark is a bounded model checking problem. With a minimum splitting width of 0.01 and a minimum progress of 0.001 the unsatisfiability of depth 64 could not be proven, while this was possible with the smaller values for those two parameters. Without implication graph compression the solver was unable to complete the benchmark and ran into a MEMOUT, while with implication graph compression the solver was able to finish.

6. Conclusion

We presented a method to compress the implication graph of iSAT3. The method exploits the fact that ICP may produce long sequences of slowly shrinking intervals leading to a huge implication graph occupying a large amount of memory. Our approach addresses this problem by doing on-the-fly resolution between implication clauses. This considerably reduces the overall memory consumption – in some cases even more than an order of magnitude and may therefore allow to complete the solving process instead of hitting a MEMOUT inbetween.

References

[BG06] F. Benhamou and L. Granvilliers. Continuous and Interval Constraints. In *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, pages 571–603. 2006.

[CHS09] Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache Conscious Data Structures for Boolean Satisfiability Solvers. *JSAT*, 6(1-3):99–120, 2009.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[EKKT08] Andreas Eggers, Natalia Kalinnik, Stefan Kupferschmid, and Tino Teige. Challenges in Constraint-Based Analysis of Hybrid Systems. In Angelo Oddi, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 5655 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2008.

[ES03] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, S. Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling, and Computation*, 1(3-4):209–236, 2007.

[HEFT08] Christian Herde, Andreas Eggers, Martin Fränzle, and Tino Teige. Analysis of Hybrid Systems Using HySAT. In *ICONS*, pages 196–201. IEEE Computer Society, 2008.

[Her11] Christian Herde. *Efficient solving of large arithmetic constraint systems with complex Boolean structure: proof engines for the analysis of hybrid discrete-continuous systems*. PhD thesis, 2011.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[Rob65] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[SB09] Niklas Sörensson and Armin Biere. Minimizing Learned Clauses. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[SKB13] Karsten Scheibler, Stefan Kupferschmid, and Bernd Becker. Recent Improvements in the SMT Solver iSAT. In Christian Haubelt and Dirk Timmermann, editors, *MBMV*, pages 231–241. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2013.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[Tse68] Grigori S. Tseitin. On the complexity of derivations in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logics*. 1968.

A New Approach to Use Partial Results During Image Computation in BDD Based Symbolic Model Checking

Christian Appold
Chair of Computer Science V, University of Würzburg, Germany
appold@informatik.uni-wuerzburg.de

Abstract

One efficient technique for model checking is BDD based symbolic model checking. This technique has been used prosperously in many industrial verification projects. In symbolic model checking the repeated image computations with sets of states and the transition relation of a system are crucial. These computations dictate the runtime and memory requirements of verification runs. In this paper, we present a new algorithm for image computation in BDD based symbolic model checking. The algorithm uses a new method to reuse already computed partial results in recursive BDD operations. As our experimental results show, this algorithm can lead to significant runtime and memory improvements.

1. Introduction

Due to the shift towards multi-core CPUs, the presence of concurrent software is steadily increasing. Such software consists of several parallel threads, which are executed asynchronously and interleaved. However, concurrent software tends to be very error-prone, bugs are often subtle and hard to detect. Therefore, to enable the usage of concurrent software in safety-critical areas, reliable techniques to verify its correct operation are mandatory. One successful formal verification technique for concurrent systems is temporal logic model checking [CE82, QS82]. There, desired properties of a system are formalized by a temporal logic (like CTL [BAMP81] or LTL [Pnu81]) and the state-space of the system is investigated exhaustively to validate the desired properties. A very efficient model checking technique is symbolic model checking [McM93] based on Binary Decision Diagrams (BDDs) [Bry86]. But despite some improvements, BDD based model checking can still be very memory and time consuming. One main reason are the repeated image computations between sets of states and the transition relation of a system. To diminish the memory requirements when storing transition relations, partitioned transition relations have been suggested [BCL91]. There, a transition relation is split into several pieces that often can be represented by a small BDD. However, the repeated image computations can still be very memory and time consuming. In this paper we present new approaches to improve image computation with BDDs. First, we present an algorithm that considers completely computed partial results during image computations in a novel way. Through simultaneous traversal of BDDs with partial results the algorithm avoids the necessity of a subsequent computation of the union of BDDs with partial

results and the result of pending computations. Additionally, we suggest different procedures to use our new algorithm for image computation.
The rest of this paper is organized as follows. First, we introduce BDD based symbolic state-space generation (see Section 2) and outline related work (see Section 3). Thereafter, we describe our image computation algorithm with the new handling of partial results in Section 4. Afterwards, several approaches to integrate our new algorithm in forward reachability analysis are introduced in Section 5. Experimental results which exemplify the efficiency of our new procedures can be found in Section 6. The paper closes with a conclusion and an outlook to future work.

2. BDD based State-Space Generation

Binary decision diagrams (BDDs) [Bry86] are used in symbolic model checking to store sets of states as well as the transition relation of a system storing their characteristic functions as a BDD. If a BDD with N Boolean variables is used to encode sets of system states, a BDD with $2 \cdot N$ Boolean variables is necessary to encode the transition relation of the system: *N* variables are required for the *from*-state and also *N* variables for the *target*-state of a transition. We consider only interleaved variable ordering in this work, because variable ordering where every pair of corresponding *from*- and *target*-state variables is next to each other in a BDD is often the most efficient variable ordering in terms of nodes required to store the transition relation. More information about the representation of finite state systems with BDDs can be found in [CGP00]. This paper targets on forward reachability analysis, i.e. the image computations are forward images. There, the computation of a set of successor states $S'(\vec{x})$ of a set of states $S(\vec{x})$ according to a transition relation $R(\vec{x}, \vec{x}')$ can be described by $S'(\vec{x}) = (\exists(\vec{x})(R(\vec{x}, \vec{x}') \wedge S(\vec{x})))\{\vec{x}' \leftarrow \vec{x}\}$, which is often called *relational product* [RV01, BK08]. In the relational product R, S, and S' are BDDs that represent the characteristic functions of the corresponding binary encoded sets. Thereby, $\vec{x} = (x_1, ..., x_N)$ is a vector with the Boolean from-state variables and $\vec{x}' = (x'_1, ..., x'_N)$ a vector with the Boolean target-state variables. The operation $\exists(\vec{x})$ is the existential quantification with regard to the variables in $\vec{x}$ and the operation $\{\vec{x}' \leftarrow \vec{x}\}$ renames the target-state variables to their corresponding from-state variables.

3. Related Work

An efficient algorithm AndExist to improve the AND operation in relational product computations with BDDs was developed in [McM93]. There, existential quantification of the from-state variables is applied to results of subproblems of the AND as soon as possible. Hence, the computation of the result BDD of an AND between a BDD for a set of states and a BDD for a transition relation before existential quantification of the from-state variables can be avoided. In [BK08] an algorithm that additionally uses an immediate renaming of the target-state variables to the from-state variables to compute the relational product in one BDD traversal has been presented. But we couldn't find any prior work with experimental results to compare the performance of the separate and the combined computation of the relational product. Therefore, we compare those approaches in this paper and present experimental results for partitioned transitions relations and an algorithm similar to the algorithm in [BK08]. Further optimizations of image computation with partitioned transition relations (e.g. [BCL91]) have been suggested. Much work has been done in optimizing the use of disjunctively partitioned transition relations for model checking of

asynchronous systems. The authors of [WYIG07] and [TCP08] presented approaches to derive optimal partitions for partitioned transition relations. Furthermore, they investigated strategies to improve the order in which the partitions are applied. In [CMS06], the computational differences of the standard breadth-first symbolic state-space generation algorithm, with and without chaining, and the saturation algorithm are investigated. Nevertheless, we couldn't find an approach which uses partial results similar to our new algorithm from section 4.

4. Relational Product Computation with our New Handling of Partial Results

In this section we present a new algorithm for relational product computation with BDDs for interleaved variable ordering. The algorithm uses a new approach to combine previously computed partial results with the result of pending computations. This is achieved by traversing BDDs with already computed partial results simultaneously in recursion steps for pending computations until a terminal case of the algorithm is reached. If a terminal case has been reached, the current partial result is united with the result of the terminal case. Therewith, the explicit combination of completely computed partial results and the associated traversals of BDDs at recursion steps can be avoided. For a given BDD with a partial result $Partial(\vec{x})$, the algorithm computes the characteristic function of $S'(\vec{x}) = (\exists(\vec{x})(R(\vec{x},\vec{x}') \wedge S(\vec{x})))\{\vec{x}' \leftarrow \vec{x}\} \vee Partial(\vec{x})$.
Our new algorithm for the combined computation of the relational product is shown in Algorithm 1. The algorithm implements our new handling of partial results and performs additionally an immediate existential quantification and renaming of the target-state variables to their corresponding from-state variables. As parameter values the algorithm gets a BDD for a transition relation R, a BDD for a set of states S, and a BDD with already completely computed partial results *Partial* (see line 1). First, it is investigated if a terminal case of the recursive computation has been reached (see lines 2 and 3). The terminal cases and their corresponding return values can be found in Algorithm 2. A terminal case occurs if the root vertex of the BDD with the transition relation R or the root vertex of the BDD with the set of states S is a terminal vertex for the value '0' (see lines 2 and 3 in Algorithm 2). In the usual algorithm, the corresponding return value is the terminal vertex for value '0'. Here, additionally the current BDD with the partial result has to be united with the result of a recursion step. Hence, the return value of our algorithm is the BDD *Partial*. Due to existential quantification and the renaming to the from-state variables in the relational product, the values of target-state variables which the transition relation permits determine the values of the corresponding from-state variables of successor states. If the BDD for the set of states S isn't the BDD for the terminal vertex with value '0', there exists at least one assignment of the input variables of this BDD that leads to the terminal vertex for the value '1'. In both terminal cases in line 4 of Algorithm 2, no vertices for target-state variables and thus no restrictions for the possible values of those variables exist in the BDD for the transition relation. Consequently, all possible variable assignments are allowed for the target-state variables and at minimum the one combination of the current state variables that leads in both BDDs to a terminal vertex for value '1'. Therewith, the BDD for the terminal vertex '1' is the return value of such a recursion step. No terminal case exists in our algorithm for a recursion step where the current root vertex of the BDD R is no terminal vertex and in which a terminal vertex for value '1' is the root vertex of the BDD S. In such a recursion step the return value of the algorithm can not be determined without considering further vertices of the current BDD R. Therefore, by further recursive calls our algorithm here continues to traverse the BDD R until the return value of the succeeding recursion steps can be decided. Calls of our

algorithm where the BDDs R and S are the same, but for which the BDD with the partial results differs can lead to different return values. For this reason, the root vertex of the BDD with the partial results is an additional parameter of computed table accesses (see lines 4 and 5 in Algorithm 1). The BDD *Partial* possesses only vertices for from-state variables, because it contains only

Algorithm 1: Relational product computation with our new consideration of partial results.

```
1  RelProductPartial(BDD R, BDD S, BDD Partial)
2  if IsTerminalCase(R, S) then
3      return TerminalCase(R, S, Partial);
4  if ComputedTableHasEntry(RelProdPartial, R, S, Partial) then
5      return ComputedTable(RelProdPartial, R, S, Partial);
6  targetVarPartial = TargetVariable(Partial);
7  w = FromVariable(Partial); v = TopVariable(R, S, targetVarPartial);
8  if isFromStateVariable(v) then
9      R_0 = RelProductPartial(R_{v=0}, S_{v=0}, Partial);
10     R_1 = RelProductPartial(R_{v=1}, S_{v=1}, R_0); Result = R_1;
11 else
12     R_0 = RelProductPartial(R_{v=0}, S_{v=0}, Partial_{w=0});
13     R_1 = RelProductPartial(R_{v=1}, S_{v=1}, Partial_{w=1});
14     Result = FindOrAddUniqueTable(GetFromStateVar(v), R_0, R_1);
15 InsertComputedTable(RelProductPartial, R, S, Partial, Result);
16 return Result;
```

completely computed partial results for which the immediate renaming to the from-state variables has already been applied. Hence, the from-state vertices in this BDD and the target-state vertices in the transition relation determine the values permitted for from-state variables in the result of the relational product. For this reason, they have to be considered together in our new algorithm and the corresponding target-state variable to the variable of the current root vertex of the BDD with the partial results is computed in line 6. Subsequently, this target-state variable is taken into account to determine the top-variable of a recursion step (see line 7). If the current top-variable is a from-state variable the BDD *Partial*, and the result of the first recursive call of the algorithm R_0 respectively, is considered as completely computed partial result in the recursive calls of the algorithm in lines 9 and 10. This is correct, because the variables of the root vertices of those BDDs equal the current from-state top-variable or appear later in the variable ordering. Thus, as the vertices for from-state variables of previously computed partial results have to be evaluated with the associated target-state variables, those vertices have to be evaluated subsequently in the recursive traversal to unite the corresponding BDDs with the result of pending computations. For a target-state variable v as top-variable the cofactors of the BDD *Partial* have to be computed with respect to the from-state variable w to the target-state top-variable, because the BDD *Partial* only contains from-state variables that have to be considered as their corresponding target-state variables (see lines 12 and 13). Afterwards, to implement the immediate renaming at recursion steps with target-state variables as top-variable, the from-state variable to the target-state variable is computed using the function *GetFromStateVar()*. Then, the result of the recursion step is generated as

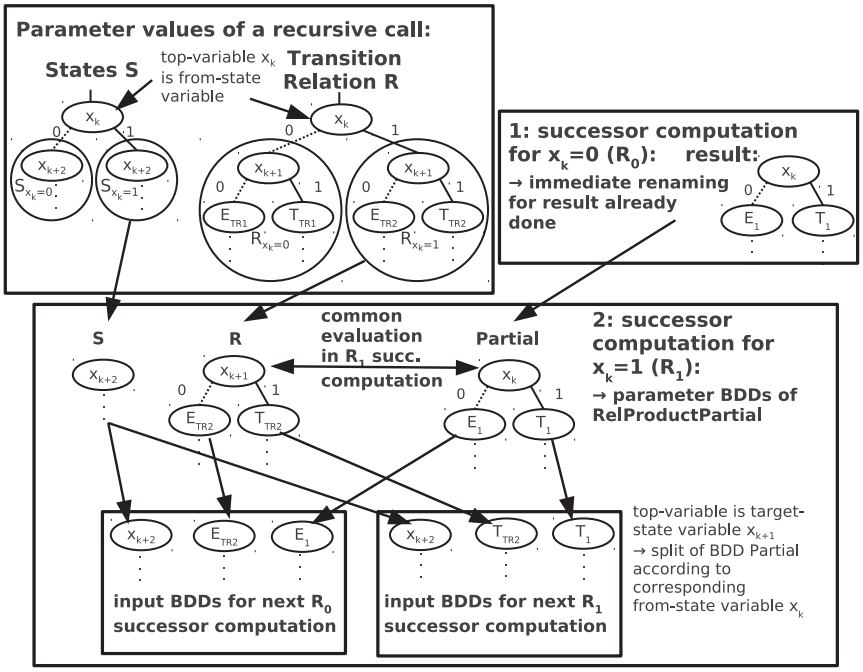


Figure 1: Example relational product computation with our new algorithm.

in the usual algorithm to compute the relational product (see line 14). The verification experiments for the combined computation of the relational product in section 6 without our new approach to consider partial results but with immediate existential quantification and immediate renaming of the target-state variables to their corresponding from-state variables have been done with a slightly modified implementation of Algorithm 1. There, the impact of the BDDs with partial results and their effects e.g. on the terminal cases, the computed table accesses and the top-variable selection has been removed. The result is a an algorithm that is similar to the algorithm from [BK08].

Algorithm 2: Terminal cases and their corresponding return values.

```
1 TerminalCase(BDD R, BDD S, BDD Partial)
2 if IsTerminalVertexZero(R) || IsTerminalVertexZero(S) then
3     return Partial;
4 if IsTerminalVertexOne(R) || R==S then
5     return BDDTerminalVertexOne;
```

Figure 1 illustrates the principle of operation of our new algorithm. There, above as parameter values of a recursive call of our algorithm an extract of a BDD for a set of states and an extract of a BDD for a transition relation is shown. For the BDD with the partial result we assume the empty set for this call of the algorithm. On the right side, Figure 1 shows the result of the successor computation for the value '0' of the current top-variable x_k. It is computed with the subtrees of the BDD for the states and the transition relation that correspond to value '0' of variable x_k ($R_{x_k=0}$ and $S_{x_k=0}$) in the BDDs of the parameter values of the first call of the algorithm. Due to the immediate renaming, the top-variable x_k of the result BDD is already a from-state variable. Below, the parameter values for the computation of the subtree for value '1' of variable x_k can

be seen. The parameter value *Partial* which has been the empty set at the preceding recursion step for top-variable x_k, is now the result BDD of the successor computation for the value '0' of variable x_k (R_0). As mentioned before, the vertices of those BDD have to be evaluated with the target-state vertices of the BDD of the transition relation. The next recursive calls are done with the corresponding successors for an assignment of the value '0', and '1' respectively, to the new top-variable x_{k+1}.
Although, the new approach to consider already computed partial results is presented for forward image computation here, the approach and the corresponding algorithm can be also adapted for usage for backward image computation. This would enable its usage for backward reachability analysis, where backward images are computed repeatedly.

5. Image Computation using our New Algorithm

In this section we present opportunities to use the new algorithm from section 4 for reachability analysis with partitioned transition relations. Except of the approach denoted ***BDD P. Part All*** in Algorithm 3, they can also be used for image computations with a single monolithic transition relation. Beneath approaches for exclusive usage, we introduce with Algorithm 3 an approach that dynamically tries to select the most performant procedure. Thereby, Algorithm 3 is specified for one complete forward image computation with image computations for all partitions of a partitioned transition relation. The three different methods employed there for image computations are highlighted bold.
In line 4 the function $Image(R_i, S_k)$ executes our combined computation of the relational product with immediate existential quantification and immediate shift to the from-state variables (denoted ***BDD P. Imm***) as sketched at the end of section 4. The parameter S_k represents the set of states that has to be explored. The first procedure using our new algorithm (denoted ***BDD P. Part Image***) can be found in line 8. Here, the parameter value for already computed partial results is intialized with the empty set in the first call of the recursive algorithm. Instead of the empty set, in the second approach with our new algorithm (denoted ***BDD P. Part All***, see line 11) the set of states S_{k+1} explored in previous image computations within the current complete exploration are the parameter value for already completely computed partial results. Therewith, the union of the set of states S_{k+1} and the states explored in the subsequent image computation is done simultaneously within the computation of the relational product and their explicit union is avoided. Algorithm 3 tries to choose the most efficient image computation approach dynamically comparing the execution times of previous image computations. The currently used image computation procedure is specified in the variable *algType* and for each approach the runtime of the last image computation is saved in the array *last*. To determine the runtime of successor computations the difference of the CPU time before and after the successor computation is computed (see line 13). Changes in the properties of the involved BDDs during reachablity analysis can lead to enormous changes of the runtime of an image computation approach. This is considered in Algorithm 3 by decreasing older previously saved runtimes of image computations. Therefore, we use a variable *counter* that counts the number of subsequent image computations within an image computation approach. In our verification experiments we decremented the last execution times of the approaches that haven't been used at current all 50 image computations. Additionally, before comparing the execution time of the currently used approach with the stored last execution times of the other procedures, their execution time is decreased by multiplication with 0.8. Therewith, the significance of execution times that

have been saved some time ago is reduced. In our example approach we have statically chosen plausible values to decrement older runtimes. An improvement that could lead to further runtime improvements would be the dynamic adaption of these parameters during reachability analysis. When choosing the multiplicative factors of the algorithm, beneath a lower relevance of older time measurements it has to be considered that changing the image computation approach too often and spuriously can also lead to a runtime increase.

Algorithm 3: Combination of three image computation approaches (BDD P. ALG COMB).

```
1  foreach i ∈ Partitions do
2      start = clock(); counter++;
3      if algType == 0 then
4          S_{k+1} = S_{k+1} ∪ Image(R_i, S_k); // Approach: BDD P. Imm
5          end = clock(); IndexOne=1; IndexTwo=2;
6      else
7          if algType == 1 then
8              S_{k+1} = S_{k+1} ∪ ImageNew(R_i, S_k, ∅); // Approach: BDD P. Part Image
9              end = clock(); IndexOne=0; IndexTwo=2;
10         else
11             S_{k+1} = ImageNew(R_i, S_k, S_{k+1}); // Approach: BDD P. Part All
12             end = clock(); IndexOne=0; IndexTwo=1;
13     last[algType] = end - start;
14     if counter > 50 then
15         last[IndexOne] = last[IndexOne] · 0.9; last[IndexTwo] = last[IndexTwo] · 0.9;
16         counter = 0;
17     if (last[algType] ≥ 0.8 · last[IndexOne] && (last[IndexOne] ≤ last[IndexTwo]) then
18         algType = IndexOne; counter = 0;
19     else if last[algType] ≥ 0.8 · last[IndexTwo] then
20         algType = IndexTwo; counter = 0;
```

6. Experimental Results

In this section we present the results of our verification experiments. All testcases describe asynchronous systems with replicated components. A system state of such an asynchronous system consists of the values of global shared variables and the local variables of each component. The verification experiments run on just a single core of an Intel Core i7 CPU with 3.4 GHz and 16 GB main memory. As model checker we used the symbolic model checker Sviss [WBE08]. For our verification experiments we have chosen the variable ordering concatenated, because it is efficient for asynchronous systems. In this variable ordering, first the from- and target-state bits for the global states appear. Thereafter, the local state bits of the components follow one after another. Thereby, the local state bits of a component are arranged consecutively in the variable ordering. The experimental results can be found in Table 1 and Table 2. We present experimental

	BDD M. Imm		BDD P. Stand.		BDD P. Imm	
Problem	BDD Nodes	Time	BDD Nodes	Time	BDD Nodes	Time
MCSLock 22	2,286,675	25m 54s	6,480,412	15h 1m	3,632,515	1h 17m
MCSLock 25	3,548,956	1h 6m	-	$> 24h$	5,547,264	3h 23m
MCSLock 30	6,515,089	6h 38m	-	$> 24h$	9,834,609	14h 14m
MCSLock 33	9,040,525	18h 43m	-	$> 24h$	-	$> 24h$
MMLoc 4	166,995	3s	273,802	32s	184,717	5s
MMLoc 5	2,957,444	10m 49s	4,716,311	28m 47s	3,012,077	9m 1s
MMLoc 6	60,441,077	9h 8m	-	$> 24h$	61,266,848	8h 45m
CCP 7	1,390,036	2m 15s	2,818,917	24m 17s	1,700,234	1m 58s
CCP 9	16,551,428	1h 24m	33,964,449	13h 43m	20,233,051	55m 47s
CCP 11	-	$> 24h$	-	$> 24h$	228,267,388	22h 13m
DP 21	53,484,687	7m 46s	5,891,614	2h 18m	3,706,116	12m 45s
DP 23	mem ov	-	15,614,780	9h 29m	9,803,484	44m 26s
DP 28	mem ov	-	-	$> 24h$	110,577,764	18h 24m
Pet 4	98,669,142	1m 0s	332,576	4s	332,576	0s
Pet 6	mem ov	-	11,146,490	1h 51m	8,380,456	12m 42s
Pet 7	mem ov	-	-	$> 24h$	101,714,419	7h 53m

Table 1: Verification results for monolithic and partitioned transition relations

results for a monolithic BDD as transition relation and also for partitioned transition relations with BDDs. For the experiments denoted with *Imm*, the algorithm for the combined computation of the relational product that has been sketched at the end of section 4 has been used. Thereby, for the experimental results presented in the column *BDD M. Imm* a single monolithic BDD has been used to store the transition relation. All other verification experiments have been done with partitioned transition relations with BDDs. The experiments denoted with *BDD P. Stand.* in Table 1 use the standard separate computation of the operations of the relational product without immediate existential quantification and immediate renaming. The other abbreviations in the tables equal the abbreviations used in section 5.

In the tables, we first present experimental results of verification experiments with MCSLock, a modified variant of the list-based queuing algorithm from [MCS91]. The next benchmark is a mutual exclusion algorithm. Beneath the local variables of the components, there is a global variable that points to components and controls access to the critical section. Also experimental results for the CCP cache coherence protocol (see e.g. [PRZ01]) and the dining philosophers problem (see [BA06], abbreviated DP in the Tables) are presented. The last testcase is the Peterson Mutual Exclusion Protocol [Pet81]. In all tables the number of components is denoted in the column *Problem* after the name of the verification benchmark. *BDD Nodes* is the largest number of live BDD nodes that appeared during a verification experiment. *Time* is the runtime, where *s*, *m*, and *h* are abbreviations for seconds, minutes, and hours. As our experimental results show, the algorithm with the combined computation of the relational product (*BDD P. Imm*) leads to large runtime and memory improvements for all testcases in comparison with the standard approach (*BDD P. Stand*). Further runtime improvements can be achieved for three of the five testcases with the new usage of partial results and the approach *BDD P. Part Image* (up to 20 percent). Those runtime improvements can be further enlarged using the procedure *BDD P. Part All*. Therewith, we achieve runtime

	BDD P. Part IMAGE		BDD P. Part ALL		BDD P. ALG COMB.	
Problem	BDD Nodes	Time	BDD Nodes	Time	BDD Nodes	Time
MCSLock 22	2,887,889	1h 4m	3,004,177	1h 2m	3,004,177	41m 5s
MCSLock 25	4,399,943	2h 46m	4,588,360	4h 23m	5,314,560	1h 48m
MCSLock 30	7,856,491	11h 45m	8,195,103	15h 58m	8,484,061	6h 9m
MCSLock 33	-	$> 24h$	-	$> 24h$	12,309,537	13h 3m
MMLoc 4	184,701	6s	175,516	6s	183,296	6s
MMLoc 5	3,012,059	9m 10s	2,778,239	10m 50s	3,012,059	9m 10s
MMLoc 6	61,266,828	9h 29m	56,543,512	18h 3m	61,266,828	9h 5m
CCP 7	1,700,234	1m 48s	1,428,542	1m 35s	1,700,234	1m 32s
CCP 9	20,233,051	47m 51s	16,851,950	38m 28s	19,020,049	36m 14s
CCP 11	228,267,388	18h 35m	189,939,928	13h 55m	228,267,388	13h 38m
DP 21	3,706,116	10m 23s	3,008,783	6m 28s	3,410,698	6m 34s
DP 23	9,803,484	35m 39s	7,954,722	23m 46s	9,041,428	22m 28s
DP 28	110,577,764	14h 51m	89,688,585	8h 15m	101,819,125	8h 33m
Pet 4	332,576	0s	332,576	0s	332,576	0s
Pet 6	8,380,456	14m 44s	8,101,647	13m 36s	8,380,456	13m 11s
Pet 7	101,714,419	9h 18m	94,715,793	8h 52m	101,714,419	8h 23m

Table 2: Verification results for partitioned transition relations

improvements up to 38 percent for the CCP testcase, and up to 56 percent for the DP testcase in comparison with the approach *BDD P. Imm*. Also the memory requirements are reduced up to 19 percent for those testcases. For the experiments with the MCSLock testcase we observe runtime improvements with the procedure *BDD P. Part Image* and a worse runtime with the approach *BDD P. Part All*. Nevertheless, when using our dynamic image computation approach from Algorithm 3, big runtime improvements can be observed (up to 57 percent) in comparison to the usual combined relational product computation. The reason for this performance improvements probably are varying properties of the BDDs involved in relational product computations during state-space traversal. Presumably, for distinct areas of the state-space different image computation approaches provide the best runtime. The dynamic algorithm is able to adapt the image computation approach to changing conditions during state-space traversal. Therewith, for each testcase the shortest runtime or runtimes close to the shortest runtime among the presented image computation approaches are achieved. For a monolithic transition relation the use of our new approaches can also can lead to considerable runtime improvements. The exclusive usage of our new approach to consider partial results leads for some testcases to an runtime increase. We assume this is due to a lower cache hit rate that isn't compensated by the positive effects of the new partial result combination strategy. For a cache hit with our new approach, additionally the BDD with the partial results has to be the same.

7. Conclusion and Outlook

In this paper we presented experimental results for the combined computation of the relational product and a new approach to consider partial results in relational product computation. Our approach often leads to large runtime and memory reductions. Also, we presented a procedure

which dynamically tries to select the image computation approach with the least runtime during state-space traversal. In the future, we intent to improve our procedure to change the image computation method dynamically with machine learning approaches. Additionally, we will examine the combined usage of our new partial result combination strategy and the usual procedure within one relational product computation.

References

[BA06] Ben-Ari, M.: *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BAMP81] Ben-Ari, M., Z. Manna, and A. Pnueli: *The temporal logic of branching time*. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176. ACM, 1981.

[BCL91] Burch, J. R., E. M. Clarke, and D. E. Long: *Symbolic model checking with partitioned transition relations*. pages 49–58. North-Holland, 1991.

[BK08] Baier, Christel and Joost Pieter Katoen: *Principles of Model Checking*. The MIT Press, 2008.

[Bry86] Bryant, R. E.: *Graph-based algorithms for boolean function manipulation*. IEEE Transactions on Computers, 35:677–691, 1986.

[CE82] Clarke, E. M. and E. A. Emerson: *Design and synthesis of synchronization skeletons using branching-time temporal logic*. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CGP00] Clarke, E. M., O. Grumberg, and D. A. Peled: *Model checking*. MIT Press, 2000.

[CMS06] Ciardo, G., R. Marmorstein, and R. Siminiceanu: *The saturation algorithm for symbolic state-space exploration*. Int. J. Softw. Tools Technol. Transf., 8:4–25, Feb. 2006.

[McM93] McMillan, K. L.: *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[MCS91] Mellor-Crummey, J. M. and M. L. Scott: *Algorithms for scalable synchronization on shared-memory multiprocessors*. ACM Transactions on Computer Systems, 9, 1991.

[Pet81] Peterson, G. L.: *Myths about the mutual exclusion problem.* Inf. Process. Lett., 12(3):115–116, 1981.

[Pnu81] Pnueli, A.: *A temporal logic of concurrent programs*. Theoretical Computer Science, 13:45–60, 1981.

[PRZ01] Pnueli, A., S. Ruah, and L. Zuck: *Automatic deductive verification with invisible invariants*. pages 82–97. Springer, 2001.

[QS82] Queille, J. P. and J. Sifakis: *Specification and verification of concurrent systems in cesar*. In *DesProceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RV01] Robinson, Alan and Andrei Voronkov (editors): *Handbook of automated reasoning*. Elsevier Science Publishers B. V., 2001.

[TCP08] Thomas, D., S. Chakraborty, and P. Pandya: *Efficient guided symbolic reachability using reachability expressions*. Int. J. Softw. Tools Technol. Transf., 10:113–129, February 2008.

[WBE08] Wahl, T., N. Blanc, and A. Emerson: *Sviss: Symbolic verification of symmetric systems*. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[WYIG07] Wang, C., Z. Yang, F. Ivančić, and A. Gupta: *Disjunctive image computation for software verification*. ACM Trans. Des. Autom. Electron. Syst., 12, April 2007.

Portierung der TriCore-Architektur auf QEMU*

Bastian Koppelmann, Markus Becker und Wolfgang Müller
Fakultät für Elektrotechnik, Informatik und Mathematik
Universität Paderborn/C-LAB, Fürstenallee 11, 33102 Paderborn
{kbastian, beckerm, wolfgang}@c-lab.de

In diesem Artikel stellen wir die Portierung der TriCore-Architektur des Prozessorherstellers Infineon auf den quelloffenen Prozessor- und Systememulator QEMU vor. Dazu beschreiben wir die Modellierung der Befehlssatzarchitektur durch den dynamischen Binärübersetzer in QEMU und gehen auf TriCore-spezifische Details der QEMU-Portierung ein. Unsere aktuelle Implementierung umfasst eine Untermenge des TriCore-Befehlsatzes für die Emulation im User-Mode. Erste Ergebnisse der Portierung zeigen wir anhand von Testprogrammen in Assembler und C. Durch einen direkten Verlgeich mit dem von Infineon bereitgestellten TSIM-Simulator konnten wir die Korrektheit unserer Portierung validieren und dabei bereits eine um bis zu 886-fache Ausführungsbeschleunigung messen.

1 Einleitung

Der Entwicklungsaufwand für eingebettete HW/SW-Systeme wird zunehmend durch ihre SW-Anteile dominiert, die in den letzten Jahren überproportional angestiegen sind. Dies gilt insbesondere für Anwendungen in den Automobil- und Automatisierungsindustrien. Virtuelle Plattformen erhöhen die Produktivität in der Entwicklung SW-intensiver eingebetter Systeme, da sie die gemeinsame HW/SW-Entwicklung und -Integration bereits in einer frühen Phase des Entwurfs auf Basis virtueller Umgebungen ermöglichen. In solchen Umgebungen sind schnelle Prozessormodelle entscheidend, um die Effekte des für die Zielarchitektur optimierten Maschinencodes frühzeitig betrachten zu können. Konventionelle HW/SW-Cosimulation mit interpretierenden Befehlssatzsimulatoren verlangsamt die Simulation um mehrere Größenordnungen gegenüber nativ ausgeführter SW.

Fortgeschrittenere Ansätze zu der Prozessormodellierung stammen aus der Virtualisierung und der Systememulation. Da hier die funktionale Emulation der Prozessorarchitektur und deren Schnittstellen im Vordergrund stehen, können Details der Mikroarchitektur abstrahiert werden. SW-Emulation auf Basis von Just-in-Time-Kompilierung ermöglicht Ausführungszeiten von 1000-10000 MIPS auf aktueller PC-Hardware [15]. Der in QEMU [6, 13] realisierte dynamische Binärübersetzer hat sich dabei als besonders effizient erwiesen, sodass sich die Verlangsamung durch verschiedene Optimierungstechniken auf etwa eine Größenordnung beschränken lässt. In den vergangenen Jahre haben sich viele Forschungs- und Entwicklungsarbeiten mit der Anbindung von QEMU an Systementwurfsumgebungen zur Analyse eingebetter SW,

*Diese Arbeit wurde zum Teil durch das Bundesministerium für Bildung und Forschung (BMBF) im Rahmen der Projekte EffektiV (01S13022) und ARAMiS (01IS11035) gefördert.

beispielsweise in SystemC [1], beschäftigt. QEMU eignet sich hierfür aufgrund vollständiger Quelloffenheit besonders gut, wodurch es auch für Universitäten als Forschungsgegenstand interessant ist. So wurden bereits verschiedene QEMU-Erweiterungen zur Analyse prozessorspezifischer SW in Bezug auf Ausführungszeit, Energieverbrauch und Fehlereffekten entwickelt und untersucht. Derzeitig beschränken sich die meisten Untersuchungen jedoch auf ARM- und MIPS-Architekturen sowie auf einige spezielle Prozessormodelle der PowerPC-Architektur.

Die von Infineon entwickelte TriCore-Architektur [3] kommt vorwiegend in Automobilsteuergeräten zum Einsatz. Da dieser Bereich hochspezialisiert ist, existieren nur wenige kommerzielle oder gar frei verfügbare Simulatoren. Es besteht daher großes Interesse seitens Industrie und Forschung, die TriCore-Architektur auf schnellen und quelloffenen Emulatoren, wie QEMU, verfügbar zu machen. Dieser Artikel beschreibt unsere Arbeiten an einer QEMU-Unterstützung für die TriCore-Architektur und präsentiert bereits erste Ergebnisse. Dazu haben wir uns zunächst auf die Portierung einer geeigneten Untermenge des umfangreichen TriCore-Befehlssatzes konzentriert, sodass wir C-Programme mit dem GCC-Kompiler von HighTec [2] in Binärprogramme übersetzen können um sie als Anwendung in einer User-Mode-Emulation auf einem Linux-PC auszuführen. Zur Überprüfung der Korrektheit des Modells haben wir die Programmausführung in QEMU mit dem von Infineon bereitgestellten TSIM-Simulator [9] verglichen. Wir konnten anhand der gewählten Beispielprogramme zeigen, dass der Zustand des emulierten Prozessors in den Ausführungsschritten funktional korrekt abgebildet wird. Unsere Geschwindigkeitsvergleiche zeigen zudem, dass trotz Einschränkungen im Bezug auf die umgesetzten Instruktionen unserer aktuellen Portierung eine bis zu 886-fache Beschleunigung der Programmausführung gegenüber TSIM möglich ist.

Der restliche Aufbau des Artikels ist wie folgt. Abschnitt 2 gibt einen Überblick über die Emulationstechnologie von QEMU und Besonderheiten der TriCore-Architektur. In Abschnitt 3 beschreiben wir die Abbildung der TriCore-Architektur in QEMU. In Abschnitt 4 stellen wir erste experimentelle Ergebnisse zum Vergleich mit TSIM vor. Abschnitt 5 verweist auf andere Arbeiten zum Thema virtuelle Plattformen (für die TriCore-Architektur) und Kopplungsansätze mit SystemC. Abschnitt 6 fasst die vorgestellten Arbeiten zusammen und gibt einen kurzen Ausblick auf unsere laufenden und zukünftigen Arbeiten.

2 Grundlagen

2.1 QEMU

QEMU ist eine quelloffene SW zur PC-Virtualisierung, sowie Prozessor- und Systememulation [6, 13]. QEMU unterstützt viele Architekturen aus dem Bereich eingebetteter Systeme, beispielsweise ARM, PowerPC, SPARC, MIPS oder Microblaze als Guest, sowie x86, IA-64, ARM, PowerPC, MIPS oder Sparc als Host. Neben der HW-gestützten Virtualisierung bietet QEMU folgende Betriebsmodi für reine SW-Emulation: User-Mode- und Full-System-Emulation. Erstere ermöglicht eine effiziente Ausführungsumgebung für ein einzelnes Anwendungsprogramm im unprivilegierten Prozessormodus. Dabei werden die Betriebssystemaufrufe abgefangen und durch das darunter liegende Linux-Betriebssystem des Host-PCs behandelt. Die Full-System-Emulation hingegen ermöglicht die Emulation eines vollständigen Systems inklusive physikalischem Speicherbus und Peripherie, sodass ein kompletter SW-Stack der Zielarchitektur ausgeführt werden kann. Das Grundprinzip der SW-Emulation in QEMU basiert auf dynamischer Binärübersetzung (oder auch Just-in-Time-Kompilierung). Dadurch kann zur Ausführungszeit auf effiziente Weise die Befehlssatzarchitektur eines Zielsystems durch die eines Host-PCs emuliert werden. Im Gegensatz zu statischer Binärübersetzung wird dabei nur der

Code übersetzt, der durch den emulierten Programm-Counter erreicht wird. Dadurch kann der Aufwand für die Übersetzung nicht erreichter SW-Anteile vermieden werden. Zudem hat ein dynamischer Ansatz den Vorteil, dass auch selbsmodifizierender Code unterstützt wird und der Programmcode nicht im Voraus bekannt sein muss. So kann im Gegensatz zu einem Ahead-of-Time Ansatz auch Programmcode dynamisch – beispielsweise durch ein Betriebssystem – nachgeladen werden. Die Binärübersetzung unterscheidet sich von klassischen interpretierenden Befehlssatzsimulatoren insofern, als dass die Phasen *Fetch* und *Decode* für die Emulation pro Befehl nur einmalig durchlaufen werden. Darüber hinaus kann der Übersetzungsprozess auf der Ebene von Basisblöcken des Maschinencodes durchgeführt werden, d.h., lineare Befehlssequenzen die mit einem Sprungbefehl enden. Diese Basisblöcke werden als Ganzes übersetzt und in einem Puffer gehalten. Dadurch wird die vielfach redundante Übersetzung der Basisblöcke, beispielsweise im Fall von Schleifen und Subroutinen, vermieden. Der Emulationsaufwand beschränkt sich nach einer Anlaufphase im Wesentlichen auf das Auffinden und Anspringen der bereits übersetzten Basisblöcke sowie ggf. der Emulation von Peripherie, was die Ausführungsgeschwindigkeit der SW nah an der nativen Ausführung hält.

QEMU reduziert zudem den Portierungsaufwand für neue Plattformen durch eine indirekte Übersetzung des emulierten Befehlssatzes mittels eines generischen Zwischenformates (Microcode), durch das die Portierung neuer Zielarchitekturen unabhängig von der Architektur des Host-PCs ist. Die Blöcke des Microcodes werden zunächst optimiert bevor sie durch den sogenannten Tiny-Code-Generator (TCG) in nativ ausführbare Code-Blöcke für den Host-PC übersetzt und gepuffert werden. Diese können dann direkt aus dem Puffer über eine Hash-Tabelle mit der Einsprungsadresse als Schlüssel angesprungen werden. Um die Emulation weiter zu beschleunigen, kann die Ausführung der übersetzten Blöcke direkt miteinander verkettet werden, sodass häufiges Rückkehren in die Emulatorhauptschleife vermieden wird. HW-Interrupts können durch die emulierten Peripheriegeräte asynchron ausgelöst werden, was dazu führt, dass die aktuelle Verkettung der übersetzten Basisblöcke aufgelöst wird, damit nach dem gegenwärtigen Block zwecks Interrupt-Behandlung in die Hauptschleife zurückgekehrt wird.

2.2 TriCore-Architektur

Infineon TriCore ist eine 32-Bit-Befehlssatzarchitektur [8] die vorranig für die Anwendung im Automobilbereich entwickelt und optimiert wurde. Dabei vereint die TriCore-Architektur MCU-, RISC-, DSP-Eigenschaften in einem umfangreichen Befehlssatz. Das Befehlssatzformat unterstützt Instruktionswörter mit 16 und 32 Bits. Dabei können viele Befehle in einem Taktzyklus ausgeführt werden. Daten werden in Little-Endian-Reihenfolde im Speicher organisiert.

Eine Besonderheit der Architektur ist die automatische Verwaltung der Prozessorkontexte. Dazu werden Prozessorregister im Zuge eines Funktionsaufrufs durch eine Call-Instruktion in dafür vorgesehene *Context-Save-Areas* (CSA) gesichert. Eine weitere Besonderheit der Architektur ist die Möglichkeit Core-Special-Function-Register (CFSR) in den Addressraum einzublenden. Für den Zugriff auf die CFSR stellt der Befehlssatz Instruktionen bereit. Diese Eigenschaften werden bei der QEMU-Portierung berücksichtigt.

3 Modellierung der TriCore-Architektur in QEMU

Dieser Abschnitt beschreibt die notwendigen Schritte zur Integration der TriCore-Architektur in QEMU. Die Erweiterungen gliedern sich in die drei Abschnitte Prozessorzustandsmodellierung, Befehlssatzmodellierung und Speichermodellierung. Die folgende Beschreibung orientiert sich

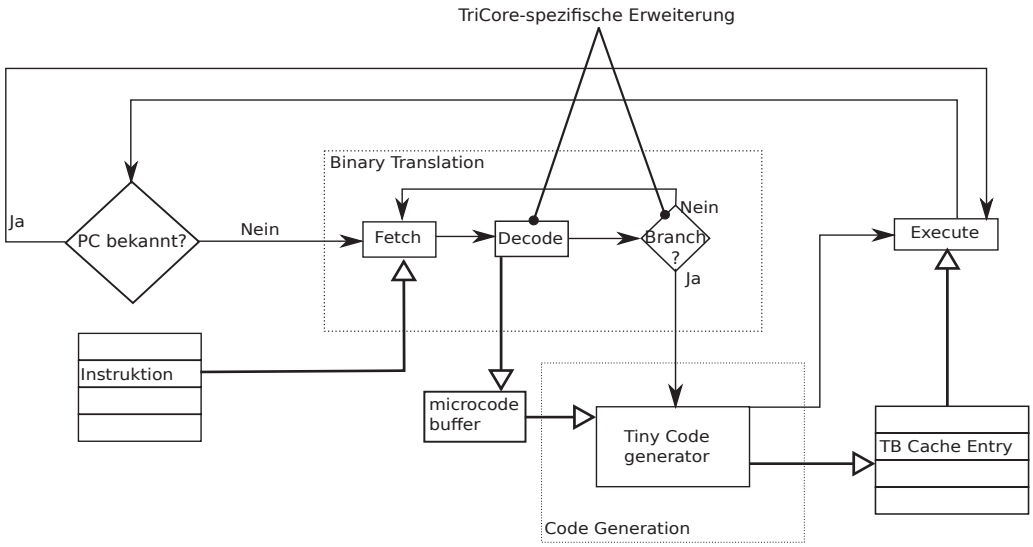


Abbildung 1: TriCore-Erweiterungen der Binärübersetzung in QEMU (adaptiert von [14])

am Ablauf der in Abbildung 1 dargestellten Phasen *Fetch*, *Decode* und *Execute* der Binärübersetzung in QEMU. Dabei gehen wir insbesondere auf die Umsetzung TriCore-spezifischer Architektureigenschaften ein, d.h., TriCore-Befehlssatz, Context-Save-Areas (CSA) und Core-Special-Function-Register (CFSR). Die realisierte Emulation der Befehlssatzarchitektur demonstrieren wir beispielhaft anhand des Ablaufs der Übersetzung eines Basisblocks in QEMU.

3.1 Prozessorzustandsmodellierung

Die Modellierung der Zielarchitektur in QEMU reduziert sich auf eine funktionale Schnittstelle des Prozessors. Wir konzentrieren uns zunächst auf eine Abbildung des Prozessors im nicht privilegierten User-Mode, d.h., wir verzichten bspw. auf die Abbildung von physikalischem Speicher und der Memory-Management-Unit (MMU). Das Programmiermodell umfasst 32 General-Purpose-Register (GPR), 3 Systemregister sowie maximal 64kB für Core-Special-Function-Register (CSFR).

Die GPR werden in 16 Daten- und 16 Adressregister eingeteilt. Per Konvention reserviert die Binärschnittstelle GP-Register für dedizierte Aufgaben, z.B. das Sichern der Rücksprungadresse (in A11) oder des Stack-Pointers (in A10). Die Systemregister umfassen das Programm-Status-Word (PSW), die Previous-Context-Information-and-Pointer-Register (PCXI) und den Program-Counter (PC). Die Register D8-D15, A10-A15, PSW und PCXI bilden den sogenannten *Upper-Context*. Diese Register werden automatisch bei einer Call-Instruktion in eine Context-Save-Area (CSA) gesichert. Abbildung 2 zeigt die Übersicht der Prozessorkontextregister. Den Prozessorzustand haben wir in analog zu anderen Architekturen in QEMU durch die Datenstruktur *CPUState* abgebildet, sodass der Prozessorregisterzugriff aus dem QEMU-Programmcode in Form eines einfachen Feldzugriffs erfolgen kann. Damit der Zugriff auch durch den vom Binärübersetzer erzeugten Microcode möglich ist, müssen die Felder des Prozessorzustandes als Variablen beim Tiny-Code-Generator (TCG) registriert werden.

Die CSFR bilden einen erweiterten 64KB-Registersatz, der in den virtuellen Adressraum der Anwendung eingeblendet werden kann. Dieser Registersatz enthält die 32 GPR, die 3 Systemregister, sowie spezielle Register für Aufgaben wie z.B. die Steuerung der MMU, I/O oder das Verwalten der CSAs. Mittels der Instruktionen *Move To/From Core Register* (MTCR/MFCR)

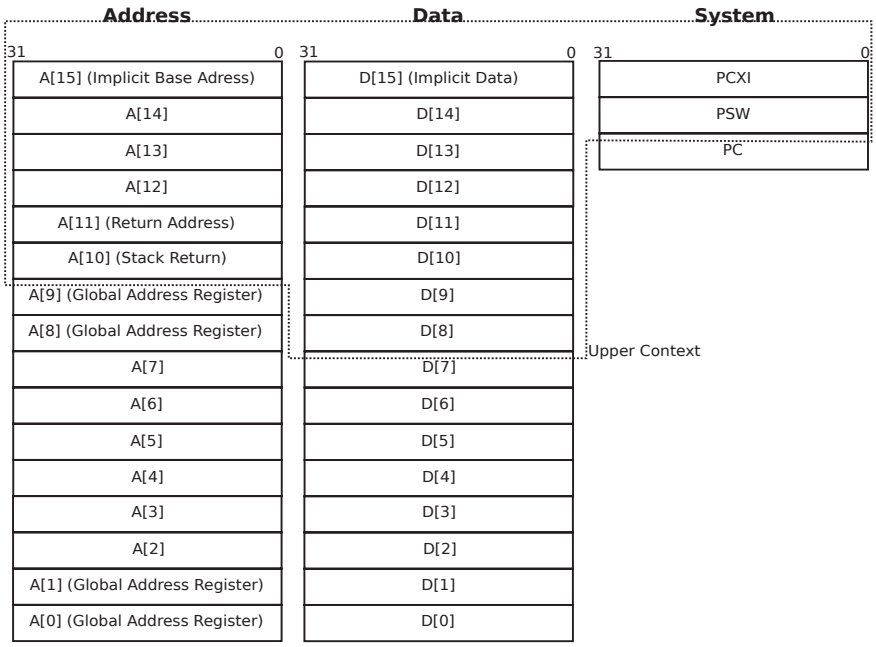


Abbildung 2: Register des TriCore-Prozessorkontextes [8]

können diese Register über einen Offset zur Basisadresse der CFSR-Region zugegriffen werden. Die CSFR wurden zunächst soweit implementiert, wie es für die Emulation des von uns gewählten Befehlssatzes im User-Mode notwendig war. Die User-Mode-Emulation in QEMU erlaubt es zudem nicht, den Zugriff auf HW-Register abzufangen. Im folgenden Abschnitt zeigen wir u.a., wie dieses Problem für die CFSR-Implementierung im Rahmen der Befehlssatzmodellierung gelöst wurde.

3.2 Befehlssatzmodellierung

Dieser Abschnitt beschreibt die Abbildung des TriCore-Befehlssatzes durch den Binärübersetzer in QEMU. Die dazu notwendigen Erweiterungen des Binärübersetzers betreffen die Phasen *Fetch* und *Decode* (siehe Abbildung 1). Hier findet die blockweise Dekodierung des Maschinencodes und Übersetzung in das Zwischenformat (Microcode) für den Tiny-Code-Generator statt.

Für die aktuelle Portierung haben wir uns auf eine Befehlssatzuntermenge aus 37 Instruktionen der TriCore-Architektur beschränkt, die es uns ermöglicht, kompilierte C-Programme in einer User-Mode-Emulation auszuführen. Der reine TriCore-Befehlssatz (ohne FPU und MMU) umfasst ca. 300 Instruktionstypen, wobei sich viele dieser Instruktionstypen ledglich durch Anzahl der Register bzw. spezielle Adressierungsmodi unterscheiden. Diese lassen sich in folgende Befehlsklassen unterteilen: *Arithmetik/Logik*, *Load/Store*, *Kontrollfluss*, *Registertransfer* und *System*. Im Folgenden gehen wir näher auf die einzelnen Befehlsklassen ein und erläutern Besonderheiten der Abbildung spezieller Instruktionen durch die QEMU-Portierung.

Arithmetik/Logik: Enthält Operationen, wie z.B. *add*, *addi* oder *and*, die direkt auf GP-Registern arbeiten, d.h., Operanden und Ergebnis werden aus GP-Registern gelesen bzw. in diese geschrieben. Diese können häufig durch einzelne äquivalente Microcode-Operationen umgesetzt werden, die durch TCG-Register auf den Prozessorkontext zugreifen.

Registertransfer: Enthält die Befehle zum Datenfluss zwischen Prozessorregistern, wie z.B. *mov*. Diese Instruktionen können i.d.R. wie die Arithmetik-Instruktionen auf eine äquivalenten Microcode-Operation abgebildet werden. Eine Ausnahme bildet der Befehl *Move To Core Register* (MTCR), welche den Inhalt eines GPR in ein CFSR kopiert. CSFR werden über eine 16-Bit-Konstante adressiert, die auf die Basisadresse der CFSR-Speicherregion addiert wird. Die Basisadresse wird dabei von der TriCore-Anwendung festgelegt. Um den Registerzugriff durch den Microcode zu gewährleisten, müssen daher alle unterstützten CSFRs als TCG-Register abgebildet werden. In der Phase *Decode* wird anhand der 16-Bit-Konstante das TCG-Register bestimmt welches dem CFSR entspricht und so der Datenfluss zwischen GPR und CFSR dynamisch hergestellt.

Load/Store: Enthält Befehle für den Datenfluss zwischen Prozessorregistern und Speicher bzw. HW-Registern. Diese lassen sich mit einer äquivalenten Microcode-Operation umsetzen, sofern diese direkte Adressierung verwenden. Für indirekte Adressierung muss ein temporäres TCG-Register allokiert werden, um das Ergebnis der Addition des Basisregisters und des Offsets zu speichern. Im Anschluss wird mit einer weiteren Microcode-Operation der Inhalt von der Speicherstelle in das temporäre TCG-Register gelesen bzw. geschrieben. Danach muss das temporäre TCG-Register wieder freigegeben werden.

System: Enthält Befehle zur Beeinflussung der Operationsmodi des Prozessors. Unsere Implementierung beschränkt sich derzeit auf die Instruktion *debug*. Diese löst im Falle des aktivierten Debug-Modus' ein Debug-Ereignis aus. Ist der Debug-Modus nicht aktiviert, so wird die Operation wie eine *nop* behandelt. QEMU sieht in der User-Mode-Emulation keine kontrollierte Beendigung vor. Stattdessen wird die Emulation indirekt durch das ausgeführte Anwendungsprogramm beendet, welches dazu einen *exit*-Betriebssystemausruf ausführen muss. Dies veranlasst wiederum das Betriebssystem den QEMU-Prozess unmittelbar zu beenden. Wir benutzen die Debug-Operation als Hilfsmittel zur kontrollierten Beendigung der Programmausführung, da unsere Implementierung zum einen derzeit noch nicht die Weiterleitung von Betriebssystemaufrufen an das Linux des Host-PCs unterstützt und zum anderen wir die Kontrolle über den Programmabbruch für Auswertungen der Programmausführung benötigen. Dazu muss das Rückkehren vom gegenwärtig ausgeführten Basisblock in die QEMU-Hauptschleife zwecks Behandlung von Ausnahmen (Exceptions) ermöglicht werden. QEMU bietet mit den Helper-Funktionen einen Mechanismus, der es erlaubt, C-Funktionen aus einem Übersetzten Basisblock an zuspringen. Dazu muss während der Phase *Decode* ein Aufruf in den Microcode des übersetzten Basisblocks generiert werden. Auf diese Weise können wir in die Ausnahmebehandlung springen und mit einer *Exception-ID* den QEMU-Prozess beenden, nachdem gesammelte Daten ausgewertet und gespeichert wurden.

Kontrollfluss: Enthält Befehle zur Herbeiführung einer bedingten oder unbedingten Verzweigung der sonst linearen Programmabarbeitung. Diese markieren immer das Ende eines Basisblocks und damit auch das Ende einer Binärübersetzungsphase (siehe Abbildung 1). Wir unterscheiden dabei *Jump-*/Branch-, *Loop-* und *Call-*/*Return*-Instruktionen. Erstere lassen sich analog zu anderen Architekturen umsetzen. So können wir die *Jump*-Instruktionen durch eine *Move*-Operation im Microcode umsetzen, die das emulierte PC-Register mit der Adresse des Sprungziels lädt. Bei *Branch*-Instruktionen muss zudem die Auswertung der Sprungbedingung im Microcode generiert werden und für ein negatives Ergebnis die Ausführung der für den Sprung verantwortlichen Operation durch das Anspringen einer anschließend eingefügten Sprungmarke im Microcode ausgelassen werden. Der TCG stellt dazu die Möglichkeit, Sprungmarken zunächst zu allokieren um sie später zu setzen, sodass die i.d.R. zuerst eingefügte *brcond*-Operation ein noch nicht gesetztes Sprungziel im Microcode adressieren kann. Die

16-Bit Opcode Formats

	15-14	13-12	11-10	09-08	07-06	05-04	03-02	01-00
SB	disp8				op1			
SBC	const4		disp4		op1			
SC	const8				op1			
SR	op2		s1d		op1			
SRC	const4		s1/d		op1			
SRO	s2		off4		op1			
SRR	s2		s1/d		op1			
SSR	S2		S1		op1			

32-Bit Opcode Formats

	31-30	29-28	27-26	25-24	23-22	21-20	19-18	17-16	15-14	13-12	11-10	9-8	7-6	5-0
ABS	off18 [9:6]		op2	off18 [13:10]		off18 [5:0]			off18 [17:14]		s1/d		op1	
B	disp24[15:0]								disp24[23:16]				op1	
BOL	op2	disp15							const4		s1		op1	
BRN	op2	disp15							n[3:0]		s1		n[4]	op1
BRR	op2	disp15							s2		s1		op1	
RC	d		op2			const9					s1		op1	
RLC	d		const16								s1		op1	
RR	d		op2				-	n	s2		s1		op1	
RRPW	d		pos		op2	width			s2		s1		op1	
SYS	-		op2			-					s1/d		op1	

Abbildung 3: Instruktionsformate der 37 gewählten Befehle

TriCore-Architektur ermöglicht eine effiziente Implementierung von Schleifendurchläufen mit der *Loop*-Instruktion. Dazu wird in einer einzelnen Instruktion das Dekrementieren eines Zählerregisters, dessen Überprüfung auf den Wert Null und – im Fall eines Zählerwertes größer Null – der Rücksprung zum Schleifenkopf zusammengefasst. Dies kann durch eine Erweiterung der Branch-Instruktionen mittels einer zusätzlichen *subi*-Operation im Microcode realisiert werden.

Eine Besonderheit der TriCore-Portierung stellt die Realisierung der *Call*- und *Return*-Instruktionen dar. Im Grunde handelt es sich dabei um unbedingte Sprünge (also Jump-Instruktionen). Allerdings muss z.B. im Fall von *Call* zusätzlich die Rücksprungadresse sowie der aktuelle Kontext gesichert werden. Ersteres kann mit einer zusätzlichen *Move*-Operation im Microcode umgesetzt werden, d.h., Sicherung des aktuellen PC-Registers erhöht um die Wortlänge der aktuellen Instruktion in das Register A11. Das Sichern des Kontextes ist aufwändiger und lässt sich aus Effizienzgründen besser durch eine Helper-Funktion umsetzen, welche in Abschnitt 3.4 genauer beschrieben wird. Die Realisierung der *Return*-Instruktion erfolgt analog dazu.

Abbildung 3 zeigt eine Übersicht der Instruktionsformate der von uns gegenwärtig realisierten Befehlssatzmenge aus 37 TriCore-Instruktionen. Die gewählte Menge genügt zur Ausführung kompilierter C-Programme in der User-Mode-Emulation.

3.3 Speichermodellierung

Wir haben uns zunächst auf die Implementierung und Validierung der TriCore-Befehlssatzarchitektur in QEMU anhand eines Anwendungsprogramms im User-Mode konzentriert. In der User-Mode-Emulation beschränkt sich die Emulation des Speichers auf den virtuellen Adress-

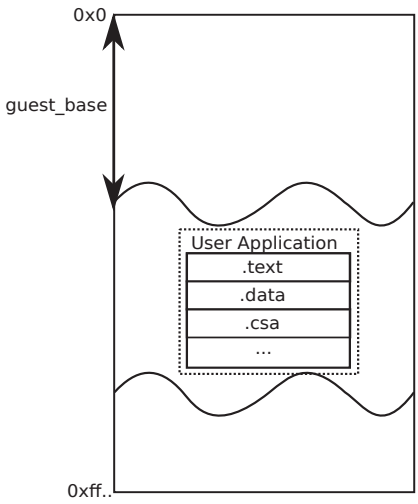


Abbildung 4: User-Mode-Emulation

bereich eines Anwendungsprogramm unter Linux. Die Speicherverwaltung wird in diesem Fall von dem Betriebssystem des Host-PCs übernommen, welches QEMU ausführt. Das Abfangen von Speicherzugriffen auf HW-Register zwecks Peripherie-Emulation entfällt daher zunächst.

Abbildung 4 zeigt die Realisierung der Speicheremulation im User-Mode. QEMU bildet den Adressraum der auszuführenden TriCore-Anwendung in den Adressraum des ausführenden QEMU-Prozesses. In diesen Speicherbereich werden alle Sektionen (Text/Code, Daten usw.) der Anwendung geladen. Alle Speicherbereiche einer Anwendung müssen als Sektionen der geladenen ELF-Datei definiert sein, damit sie durch den Binärübersetzer adressiert werden können. Das betrifft z.B. auch den Speicherplatz für die Context-Save-Areas (CSA) der für die Unterstützung der *Call*- und *Return*-Instruktionen benötigt wird. Um adressierbaren Speicher für die CSAs zu reservieren haben wir die *.data*-Sektion entsprechend vergrößert, sodass die CSAs hinter der eigentlichen Daten-Sektion abgelegt werden können (siehe Abbildung 4).

Die Konvertierung zwischen den Adressräumen (TriCore zu Host-PC oder umgekehrt) wird durch einfache Addition bzw. Subtraktion eines Offsets (*guest_base*) realisiert. QEMU stellt zwei Makros zur Adresskonvertierung: *g2h(<guest-addr>)* und *h2g(<host-addr>)*. Konvertierungen der Byte-Reihenfolge von Datentypen entfällt im Fall einer x86-Hosts, da auch die TriCore-Architektur mit Little-Endian-Kodierung arbeitet.

3.4 Binärübersetzung eines Basisblocks

Wir betrachten die indirekte Binärübersetzung eines einfachen Basisblocks bestehend aus drei Tri-Core-Instruktionen (*mov*, *addi* und *call*) anhand von Tabelle 1. Die Spalten repräsentieren den Basisblock im TriCore-Befehlssatz (links), im TCG-Microcode (Mitte) und im x86-Befehlssatz für den Host-PC (rechts). Die Übersetzung eines Basisblocks orientiert sich am Ablaufdiagramm in Abbildung 1. Eine Übersetzungsphase wird durch den Aufruf der Funktion *gen_intermediate_code()* gestartet, wenn das emulierte PC-Register auf den Anfang eines Basisblocks zeigt, dieser aber noch nicht im QEMU-Puffer vorhanden ist. Die Microcode-Generierung für einen Basisblock startet bzw. endet mit einem Prolog und Epilog. Dazwischen werden binäre TriCore-Instruktionsworte der Reihe nach geladen und dekodiert, bis eine Kontrollfluss-Operation erreicht wird (siehe Abbildung 1). Die Dekodierung der TriCore-Befehle geschieht in einer Routine in der Typ und Format der Instruktion durch die Anwendung von Bitmasken bestimmt und die entsprechenden Operanden extrahiert werden. Anhand

TriCore-Assemblercode	TCG-Microcode	x86-Assemblercode
`mov d0,5`	`movi_i32 d0,0x5`	`mov 0x5,ebx` `mov ebx,0x70(ebp)`

`addi d0,d0,5000`	`movi_i32 tmp0,0x1388`
	`add_i32 d0,d0,tmp0`
		`mov 0x70(ebp),ebx`
		`add 0x1388,ebx`
		`mov ebx,0x70(ebp)`

`call a000000a`	`movi_i32 tmp0,0xa0000008`
	`movi_i32 tmp1,0x2`
	`movi_i32 tmp2,0x6007277c`
	`call tmp2,0x0,0,env,tmp0,tmp1`
	`movi_i32 PC,0xa000000a`
		`mov ebp,(esp)`
		`mov 0xa0000008,ebx`
		`mov ebx,0x4(esp)`
		`mov 0x2,ebx`
		`mov ebx,0x8(esp)`
		`call 0x6007277c`
		`mov 0xa000000a,ebx`
		`mov ebx,0x8(ebp)`

	`exit_tb 0x0`	`xor eax,eax` `jmp 0x621dfeb4`

Tabelle 1: Indirekte Übersetzung eines TriCore-Basisblocks in der realisierten QEMU-Portierung

dieser Daten werden funktional äquivalente Microcode-Operationen in den Microcode-Puffer geschrieben. Im einfachsten Fall kann die TriCore-Instruktion durch eine entsprechende Microcode-Operation abgebildet werden (siehe *mov*-Operation). Die *addi*-Instruktion benötigt bereits zwei Microcode-Operationen, die über ein temporäres TCG-Register (*tmp0*) verknüpft sind.

Die Sicherung des Kontextes für die *call*-Instruktion realisieren wir durch den Aufruf einer Helperfunktion aus dem Microcode. Die Helperfunktion benötigt als Parameter die Rücksprungadresse sowie die Länge des aktuellen Instruktionswortes. Diese werden mittels *movi* in zwei temporäre TCG-Register *tmp0* und *tmp1* geladen. Ein weiteres temporäres TCG-Regiser (*tmp2*) wird mit der Adresse der Helperfunktion geladen. Für den Aufruf der Helperfunktion wird eine *call*-Operation mit den Parametern *tmp0*, *tmp1* und *tmp2* in den Microcode-Basisblock generiert. Die Helperfunktion prüft zunächst, ob ein freier CSA-Frame zur Sicherung des aktuellen Kontextes vorhanden ist. Das Kopieren der Kontextregister in die CSA-Region aus der Helperfunktion heraus erfordert das Konvertieren der Zieladressen mittels des *g2h*-Makros. Abschließend müssen noch die verketteten Listen zur Verwaltung freier und belegter CSAs aktualisiert werden.

Die Call-Instruktion beendet die Übersetzungsphase für den aktuellen Basisblock, was im Microcode durch eine *exit_tb*-Operation markiert wird. QEMU unterstützt das direkte Verketten übersetzter Basisblöcke im Epilog um unnötiges Rückkehren in die Emulatorhauptschleife zu reduzieren. Direkte Basisblockverkettung wird in unserer Portierung aktuell nicht unterstützt, sodass hier noch Optimierungspotential für die Geschwindigkeit besteht. Am Ende einer Übersetzungsphase wird die Generierung des nativen Basisblocks für die Architektur des Host-PCs (hier x86) durch den TCG angestoßen.

4 Experimentelle Ergebnisse

Dieser Abschnitt beschreibt die Bewertung unserer Portierung bzgl. ihrer funktionalen Korrektheit und der Ausführungsgeschwindigkeit im Vergleich zu Infineons TSIM anhand von Testprogrammen.

4.1 Testprogramme

Wir haben sowohl Assembler- als auch C-Programme untersucht. Die funktionale Korrektheit von einzelnen Instruktionen überprüfen wir anhand von kurzen Assembler-Routinen. Für den Geschwindigkeitsvergleich haben wir bekannte Algorithmen in C, beispielsweise die rekursive Berechnung der Fibonacci-Folge, untersucht. Für diese Programme mussten wir einen Startup-Code in Assembler schreiben, der zunächst den Stack sowie die CSAs initialisiert, bevor die eigentliche Main-Funktion des Programms aufgerufen wird. Die C-Programme haben wir mit der TriCore-Portierung des GCC-Compilers der Firma *HighTec* übersetzt [2].

4.2 Vergleich mit TSIM

Funktionaler Vergleich

Zur Bewertung der funktionalen Korrektheit der TriCore-Abbildung in QEMU prüfen wir den Zustand des emulierten Prozessors vor bzw. nach der Ausführung einer Instruktion. Dazu vergleichen wir die QEMU-Portierung mit dem TSIM von Infineon, den wir als Referenzmodell betrachten. Wir gehen davon aus, dass unsere Abbildung korrekt ist, wenn die Registerinhalte in allen Ausführungsschritten übereinstimmen. Der TSIM ermöglicht nach jedem Instruktionsschritt die Ausgabe veränderter Register. Wir haben die QEMU-Portierung erweitert um den Vergleich auf Basis eines gemeinsamen Ausgabeformats zu ermöglichen. Dazu haben wir QEMU um einen Vergleichsmodus erweitert, in dem wir den Aufruf einer eigens implementierten Helperfunktion nach jeder TriCore-Instruktion in den Microcode generieren. Diese Helperfunktion gibt die Registeränderungen im TSIM-Format zwecks direktem Vergleich aus. Für die von uns gewählten Testprogramme konnten wir zeigen, dass alle Prozessorzustände in korrekter Folge in QEMU abgebildet werden.

Geschwindigkeitsvergleich

Für den Geschwindigkeitsvergleich haben wir zwei C-Testprogramme ausgewählt: *Add* und *Fibonacci*. Das *Add*-Programm enthält eine Schleife, die k-mal iteriert wird und in jeder Schleifeniteration vier Additionen durchführt. Das Fibonacci Programm berechnet rekursiv das n-te Element der Fibonacci-Folge. Über die Parameter n und k haben wir die Ausführungszeiten der Testprogramme skaliert. Die Zeitmessungen haben wir mit *GNU-Time* durchgeführt. Abbildung 5 zeigt die logarithmisch skalierten Ausführungszeiten der Testprogramme auf einem Notebook mit Intel Core2Duo-Prozessor jeweils für QEMU und TSIM. Sowohl QEMU als auch TSIM beanspruchen lediglich einen Prozessorkern des Host-PCs. Das Fibonacci-Programm wird bei $n \leq 14$ schneller im TSIM ausgeführt, da bei derart kurzen Ausführungszeiten der Aufwand durch die Binärübersetzung stark ins Gewicht fällt. Bei $n \geq 15$ kehrt sich das Verhältnis um, da die QEMU-Ausführungdauer nur geringfügig ansteigt im Gegensatz zu TSIM. Auf lange Sicht zeigen beide Ansätze einen nahezu proportionalen Geschwindigkeitsunterschied (siehe *Add*-Programm). Wir messen einen Geschwindigkeitsvorteil von QEMU um einen Faktor von bis zu 886 bei 300 MIPS auf der gegebenen Hardware. Wir erwarten, dass dieser Vorteil durch die Unterstützung direkter Blockverkettung noch erheblich gesteigert werden kann.

5 Existierende Arbeiten

Infineon verweist für die Modellierung und Simulation von TriCore-Prozessoren [3] auf Altium Crossview Pro Simulator, Lauterbach TRACE32-Sim, QTronic Silver und Synopsys CoMET/-

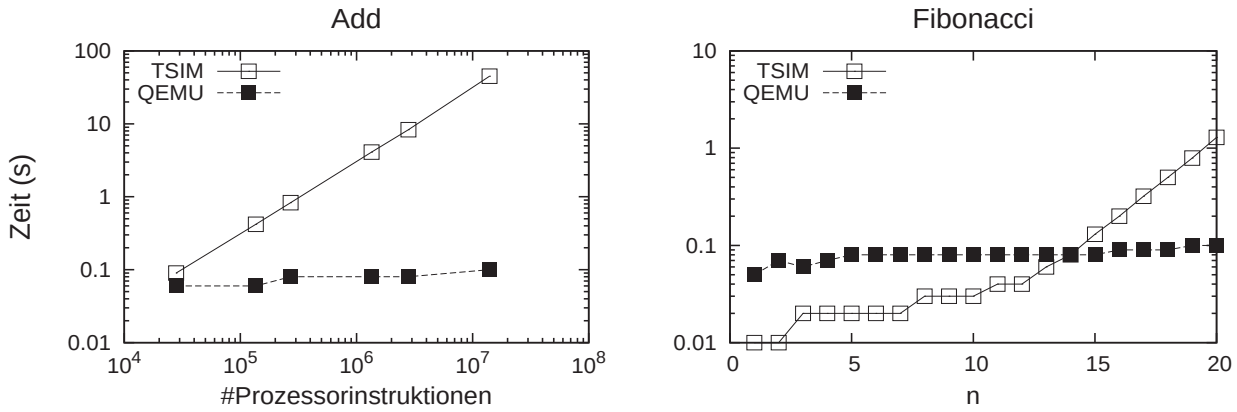


Abbildung 5: TriCore-Simulationsgeschwindigkeit mit TSIM und QEMU

METeor. Diese Lösungen sind allesamt kommerziell und demnach nicht quelloffen. Einige Lösungen integrieren den TSIM-Simulators von Infineon. QTronic gibt eine Simulationsgeschwindigkeit von 50 MIPS auf einem gewöhnlichen PC an, was deutlich unterhalb der möglichen Geschwindigkeiten für moderne JiT-basierte Simulationsumgebungen liegt.

Uns ist zu diesem Zeitpunkt nicht bekannt, dass bereits öffentliche Portierungen der TriCore-Architektur für QEMU existieren. Das gilt sowohl für den Hauptentwicklungspfad als auch öffentliche Entwicklungszweige. Neben QEMU existieren kommerzielle und eingeschränkt offene Simulationsumgebungen bzw. Emulatoren, die auf einer vergleichbaren Technologie beruhen. Hier ist insbesondere Open Virtual Platforms (OVP) zu nennen, welches auf dem von Imperas entwickelten Just-in-Time-Codemorphing beruht und Simulationsgeschwindigkeiten von 1000-7000 MIPS auf einem Intel i7 mit 3,4 GHz erzielt [4]. Der Simulationskern von OVP sowie einige der Prozessormodelle sind nicht quelloffen und müssen lizensiert werden. OVP bietet allerdings offene Schnittstellen zur Prozessormodellierung und Kopplung mit SystemC. OVP bietet keine freien TriCore-Prozessormodelle. Uns ist auch nicht bekannt, dass TriCore-Modelle für OVP von Drittanbietern existieren. Aufgrund der genannten Einschränkungen von OVP bzgl. der Quelloffenheit und des Lizenzmodells haben wir uns für eine Portierung auf QEMU entschieden.

Wenngleich QEMU keine nativen Schnittstellen für die Systemmodellierung und Simulation in SystemC anbietet, so sind in den vergangenen Jahren verschiedene Kopplungsansätze im Rahmen universitärer Untersuchungen entwickelt worden. Monton et al. haben eine Kopplung für Gerätetreiberentwicklung in QEMU auf Basis von HW-Beschreibungen in SystemC vorgeschlagen [17]. Eine Bibliothek zur Kopplung des QEMU-Systememulators mit TLM2.0-Busmodellen wurde im Rahmen eines GreenSoCs-Projektes veröffentlicht [5]. Gligor et al. haben TLM2.0-Wrappermodule für den QEMU-Systememulator entwickelt und auf dieser Basis verschiedene Cache-Modellierungen zur Abschätzung von Performanz- und Energieverbrauch in Multicore-Architekturen untersucht [14]. An der Universität Paderborn wurden TLM2.0-Kopplungen des QEMU-Systememulators [16] sowie Kopplungen des User-Mode-Emulators mit abstrakten Middleware-, RTOS- und HAL-Modellen in SystemC entwickelt und untersucht [10, 12, 11]. Diese Kopplungsansätze basieren i.d.R. auf speziellen Versionen von QEMU und sind demnach nicht aufwärtskompatibel. Mit TLMu existiert eine ebenfalls frei verfügbare Wrapper-Bibliothek mit TLM2.0-Schnittstelle, die auch aktuelle QEMU-Versionen unterstützt [7]. TLMu unterstützt derzeit nur die ARM-, MIPS- und CRIS-Architekturen.

6 Zusammenfassung

In diesem Artikel haben wir unsere Arbeiten zur Portierung der TriCore-Architektur des Prozessorherstellers Infineon für den schnellen und quelloffenen Prozessor- und Systememulator QEMU vorgestellt. Erste Ergebnisse konnten wir anhand der Ausführung einiger Testprogramme durch die User-Mode-Emulation mit QEMU erzielen. Dabei konnten wir im direkten Vergleich mit dem TSIM-Simulator von Infineon zeigen, dass unsere Programme korrekt ausgeführt werden und – trotz einiger Einschränkungen unserer gegenwärtigen QEMU-Portierung – bereits eine Geschwindigkeitssteigerung um bis zu Faktor 886 erzielt werden kann. Unsere zukünftigen Arbeiten werden sich mit der Vervollständigung des Befehlssatzes und der Unterstützung der Systememulation beschäftigen. Des weiteren beabsichtigen wir die Integration unserer TriCore-Portierung in den Hauptentwicklungspfad von QEMU sowie die Anbindung an SystemC-basierte Systementwurfsumgebungen zwecks Integration und Analyse TriCore-spezifischer SW auf Basis virtueller Systemprototypen.

Literatur

[1] Accellera Website: http://www.accellera.org/.

[2] HighTec Website. http://www.hightec-rt.com/.

[3] Infineon Website. http://www.infineon.com/.

[4] Open Virtual Platforms Website. http://www.ovpworld.org/.

[5] QEMU-SystemC Website. http://forge.greensocs.com/de/Projects/QEMUSystemC.

[6] QEMU Website. http://www.qemu.org/.

[7] Transaction Level eMulator Website. http://edgarigl.github.io/tlmu/.

[8] TriCore User's Manual V 1.3.8, Januar, 2008.

[9] TriCore Instruction Set Simulator (ISS) User Guide V 1.0, September, 2006.

[10] M. Becker, G. Di Guglielmo, F. Fummi, W. Mueller, G. Pravadelli, and T. Xie. RTOS-Aware Refinement for TLM2.0-based HW/SW Designs. In *DATE '10*, 2010.

[11] M. Becker, U. Kiffmeier, and W. Mueller. HeroeS: Virtual Platform Driven Integration of Heterogeneous Software Components for Multi-Core Real-Time Architectures. In *ISORC 2013*, 2013.

[12] M. Becker, H. Zabel, and W. Mueller. A Mixed Level Simulation Environment for Stepwise RTOS Refinement. In *DIPES'10*, 2010.

[13] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC'05*, 2005.

[14] M. Gligor, N. Fournel, and F. Pétrot. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *CODES+ISSS*, 2009.

[15] Imperas Software. Open Virtual Platforms (OVP). http://www.ovpworld.org/.

[16] F. Mischkalla, D. He, and W. Mueller. Closing the Gap Between UML-Based Modeling, Simulation and Synthesis of Combined HW/SW Systems. In *DATE'10*, 2010.

[17] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. Mixed SW/SystemC SoC Emulation Framework. In *ISIE'07*, 2007.

Erkennen von Speicherverletzungen im Testbetrieb von eingebetteter Software

Hanno Eichelberger*, Patrick Heckeler,
Jürgen Ruf, Stefan Huster, Sebastian Burg,
Thomas Kropf, Wolfgang Rosenstiel
Eberhard Karls Universität Tübingen
{nachname}
@informatik.uni-tuebingen.de

Thomas Greiner
Hochschule Pforzheim
thomas.greiner@hs-pforzheim.de

Zusammenfassung

Die letzten circa 20% an Bugs in eingebetteter Software sind aufwändig zu beheben und werden oft erst im Testbetrieb im Endsystem erkannt. Diese können durch gängige Verifikationstechniken vorab schwer detektiert werden, da sie von externen Systemen sowie Sensordaten in Verbindung mit Benutzereingaben abhängen. Treten Fehler im Testbetrieb auf, so ist es schwer, diese im Labor zu rekonstruieren und zu debuggen. Zur Optimierung des Debugging-Prozesses werden in unserem Ansatz die externen Eingaben der Software während des Testbetriebs aufgezeichnet. Diese Aufzeichnungen repräsentieren reale Sensordaten in Verbindung mit Benutzereingaben. Die Eingabesequenzen, welche zu einem Fehlerverhalten des Systems führen, werden gespeichert und im Labor wiedergegeben. Während der Wiedergabe können die Zwischenzustände der Software analysiert werden, um die Ursache des Fehlerverhaltens festzustellen. Im Fokus dieser Arbeit liegt die Detektion von Speicherverletzungen, welche häufig die Ursache für Ausnahmezustände in eingebetter Software sind. Zur Erkennung von Speicherverletzungen wird bei unserem Ansatz während der Wiedergabe der Eingabesequenzen im Labor ein Speicheranalysewerkzeug eingesetzt. Dieses erkennt falsche Zugriffe auf Speicherbereiche sowie die fehlende Deallokation von Speicher. Der Entwickler kann, basierend auf einem Bericht des Werkzeugs, fehlerhafte Implementierungen korrigieren, anschließend testen und dadurch das Fehlerverhalten des Systems effizient beheben. Durch die vorgestellte Methodik ist eine effiziente Analyse und ein effizientes Debugging von Speicherproblemen bei eingebetteten Systemen möglich, welche im ersten Testbetrieb aufgetreten sind.

1. Einleitung

Heutzutage werden in der Industrie eingebettete Systeme in verschiedenen Bereichen verwendet. Für ein Unternehmen sind Fehler in der Software von eingebetteten Systemen teuer. Je später Fehler gefunden werden, desto aufwändiger wird es diese zu beheben, da die Software und die

*Diese Arbeit wurde mit Mitteln des Ministeriums für Wissenschaft, Forschung und Kunst Baden-Württemberg im Rahmen des kooperativen Promotionskollegs EAES gefördert.

umgebenen Systeme durch die ständige Weiterentwicklung immer komplexer werden. Die letzten ca. 20% der Fehler sind besonders aufwändig zu detektieren. Diese entstehen z.B. durch Speicherverletzungen in Software. Ein großer Markt wird von Infotainment-Systemen in Autos erschlossen, auf welche sich dieser Beitrag bezieht. Diese Systeme werden mit x86-Architekturen (z.B. Intel Atom) ausgeführt und verarbeiten geringe IO-Bandbreiten (GPS- und Benutzereingaben).
Gängige Verifikationsverfahren wie die statische Code-Analyse, Model Checking oder auch modellbasiertes Testen können häufig nicht alle Fehler in einem integrierten System feststellen. Hier ist eine modellbasierte Repräsentation des Systems oft zu komplex oder Eingabebereiche können nicht vollständig abgedeckt oder simuliert werden. Daher werden bestimmte Fehler nicht vor dem ersten Testbetrieb des Prototyps entdeckt. Wenn Fehler im Testbetrieb im integrierten Endsystem entdeckt werden, müssen diese zum Debuggen im Labor rekonstruiert werden. Eine manuelle Rekonstruktion und Behebung des Fehlers basierend auf einem Fehlerbericht ist meist sehr aufwändig. Häufig müssen verschiedene Eingabesequenzen manuell getestet werden bis ein Fehler gefunden wird. Dies verursacht hohe Entwicklungskosten. In dem Fall, dass eine lange fehlerverursachende Eingabesequenz rekonstruiert werden konnte, ist es trotzdem nötig, den Ursprung des Fehlers zu detektieren. Viele Fehler basieren auf komplexen Zusammenhängen im Quelltext, welche schwer zu erfassen und zeitaufwändig zu analysieren sind. Die manuelle Suche nach der Ursache eines Fehler ist meist ineffizient. Zur Sicherstellen, dass der Fehler behoben wurde, Zusätzlich muss nach der Modifikation des Quelltextes die fehlerhafte Eingabesequenz erneut beobachtbar getestet werden können, um sicherzustellen, dass der Fehler behoben ist.
Zur Optimierung dieses Prozesses zeichnet unsere Methode die externen Eingaben in eine eingebettete Software während des Testbetriebs auf. Eingabesequenzen, die zu einem Fehler führen, werden gespeichert, können im Labor wiedergegeben und somit in einer Debug-Umgebung untersucht werden. Während der Wiedergabe werden dynamische Analyseverfahren eingesetzt, um Fehlerursachen zu detektieren. Der Fokus dieses Beitrags liegt auf der Detektion von Speicherverletzungen, die Ursache für das Fehlerverhalten einer Software sind. Durch die automatisierte Rekonstruktion und Analyse von Fehlern kann die Entwicklungszeit reduziert werden.

2. Ziele

In folgender Liste werden die Ziele dieses Beitrages zusammengefasst:

- Vereinfachung der Erkennung von verbliebenen Fehlern in integrierten Endsystemen.
- Automatische Rekonstruktion der Fehler im Labor durch Aufzeichnung der externen Eingaben der Software während des Testbetriebs und Nachbildung der Eingaben in einem Testfall.
- Beschleunigung der Fehlerbehebung durch die Lokalisierung der Fehlerursache mit dynamischen Analyseverfahren, mit Fokus auf der Erkennung von Speicherverletzungen während der Wiedergabe durch ein dynamisches Speicheranalysewerkzeug.
- Sicherstellung der Fehlerbehebung durch Wiedergabe der Fehlersequenz als Testfall.
- Verbesserung der Beobachtbarkeit der Programmvisualisierung während der Wiedergabe.

Durch die Umsetzung dieser Ziele kann der Debugging-Prozess von Speicherproblemen in eingebetteter Software, welche im ersten Testbetrieb aufgetreten sind, automatisiert und beschleunigt werden.

3. Stand der Technik

In diesem Abschnitt werden vier Ansätze untersucht, welche die Aufzeichnung von Eingaben in eine Software (*Record*) und die Wiedergabe dieser im Labor (*Replay*) für dynamische Analysen (*offline-Analysen*) betrachten. Hierbei werden Ausführungen einer Software im Betrieb aufgezeichnet und später im Labor wiedergegeben. Während der Wiedergabe können aufwändige Analysen durchgeführt werden. Dies hat den Vorteil, dass die Ausführung im Betrieb nur durch das Recording gestört wird (z.B. durch Laufzeiteinbußen - *Performance Overhead*), allerdings nicht durch die weit aufwändigeren Analysen.
Der *WiDS Checker* [LLPZ07] ist ein Record/Replay Werkzeug für verteilte Systeme, welche mit der *WiDS API* implementiert sind. Die aufgezeichneten Ausführungen werden auf einem Simulator wiedergegeben. Die Aufzeichnung findet über die WiDS API statt. Hierbei werden alle eingehenden Events an einem Netzwerkknoten aufgezeichnet und auf einer Host Plattform auf einem Simulator reproduziert. Während des Replays werden bei jedem eingehenden Event durch den Benutzer spezifizierte Prädikate bzw. *Assertions* überprüft. Verletzungen von Prädikaten können Hinweise auf Fehlerursachen sein. Durch den WiDS Checker werden nur von extern empfangene Events und kein internes Verhalten aufgezeichnet. Dadurch wurde der Recording Performance Overhead bei Messungen der Entwickler bei maximal 2 % gehalten (Overhead ist trotzdem abhängig vom Workload). Allerdings müssen die Assertions manuell vom Benutzer spezifiziert werden. Diese werden nur direkt nach jedem eingehenden Event geprüft.
Im Ansatz von Narayanasamy et al. [NWT+07] wird die Ausführung von Multi-Threaded Software in eine Log-Datei aufgezeichnet. Hierfür wird das Record-Replay-Werkzeug *iDNA* [BCdJ+06] verwendet. Dieses instrumentiert dynamisch die geladenen Instruktionen und zeichnet diese auf. Der gemessene Performance Overhead liegt bei einer sechsfachen Ausführungszeit. Zusätzlich werden durchschnittlich je Instruktion 0.8 Bit Speicher im Log benötigt. Dadurch können bei langen Aufzeichnungen die Logs sehr groß werden (96 MB pro aufgezeichnete Milliarde an Instruktionen). Während der Wiedergabe der Aufzeichnungen werden dynamische offline-Analysen angewendet. Als Fehlertyp werden in diesem Artikel *Data Races* betrachtet. Die aufgezeichneten Ausführungen werden mit verschiedenen parallelen Ablauffolgen (*Thread-Schedules*) ausgeführt, um kritische parallele Datenzugriffe von ungefährlichen zu unterscheiden und somit *false positives* bei der Analyse zu vermeiden. Die Vorteile liegen bei diesem Ansatz bei einer deterministischen Rekonstruktion der Ausführung. Allerdings hat das Instruktions-basierte Recording einen sehr hohen Performance und Memory Overhead. Im Bereich von eingebetteten Systemen kann dies die normale Ausführung während des Testbetriebs stören.
Jalangi [SKBG13] ist ein Framework für das deterministische Record/Replay für JavaScript-Programme. Dieses wird in einer Browser-Umgebung ausgeführt. Zur Aufzeichnung der Ausführung werden benutzer-definierte Komponenten instrumentiert. Dadurch werden die Eingaben komponenten-bezogen aufgezeichnet und wiedergegeben. Die Aufzeichnung findet in JavaScript mittels *Shadow Values* statt. Beispielsweise wird die Integer-Variable *v* in ein Array abgeändert, bei welchem *v[i]* die Belegung von *v* im *i*-ten Lauf repräsentiert. Parameter von Methodenaufrufen werden ebenfalls aufgezeichnet. Der Performance Overhead wurde von den Autoren bei Aufzeichnungen von Ausführungen auf 26x gemessen. In Jalangi wurden verschiedene dynamische offline-Analyseverfahren implementiert: *Concolic Testing* (symbolische Ausführung und Generierung von nicht-getesteten Ausführungspfaden), Ermittlung der Quellen von *null* und *undefined* Zuweisungen, *Taint Analyses* (Informationsflussananalysen) sowie Zugriffsüberwachung auf Objekte (An-

zahl von Zugriffen). Die Umsetzung der Aufzeichnung sowie der dynamischen Analyse mit *Shadow Values* ist für JavaScript-Implementierungen gut geeignet. Allerdings muss hierbei der Quelltext sehr umfassend instrumentiert werden. Dies bietet Möglichkeiten für neue Fehlerquellen. Die Verwendung von *Pointer* erschwert die Umsetzung des Verfahrens in C/C++.
Burtsev et al. [Bur13] erweitern die Virtualisierungsplattform *Xen* [Fou13] mit Record/Replay-Funktionen. Hierfür wurde in Xen eine Zwischenschicht eingebaut, welche die wichtigsten Gerätetreiber aufzeichnet. Bei Benchmark-Messungen mit einem Linux-Gast-System wurde ein Performance Overhead von 5%-30% bei hohen Netzwerkaktivitäten und Speicheraktivitäten gemessen. Zusätzlich wurde in Xen ein Analyse Interface integriert, welches Zugriffe auf das System für dynamische offline-Analysen bietet. Die Record/Replay Engine wurde in zwei Fallstudien für dynamische offline Performanzanalysen sowie Exploitanalysen (durch Zugriff auf das low-level Systemverhalten) getestet. Der Ansatz bietet den Vorteil, über die virtuelle Maschine auf die Gerätetreiber zugreifen zu können, um Eingaben zeitgenau aufzuzeichnen und wiederzugeben. Der Performance Overhead scheint sehr gering, zusätzlich muss aber der Overhead für Xen hinzugerechnet werden. Viele eingebettete Systeme sind allerdings für Virtualisierung nicht ausgelegt.
Tabelle 1 stellt eine Übersicht über den Stand der Technik dar. Die untersuchten Record/Replay-Ansätze mit der Anwendung für offline-Analysen werden nicht für eingebettete Systeme eingesetzt, sondern hauptsächlich für Server und Desktop-Anwendungen. Außer in Jalangi für JavaScript [SKBG13] wird in all den anderen betrachteten Ansätzen die Instrumentierung in der Middleware umgesetzt. Bei [Bur13] werden virtuellen Maschinen verwendet und bei [NWT+07] findet das Recording auf Instruktionsebene mit einem hohen Performance Overhead statt. Beim WiDS Checker [LLPZ07] werden nur eingehende Events geloggt. Dieses Konzept ist nur von der Event-Frequenz abhängig und wird auch in unserem Ansatz verwendet. Unser Ansatz loggt Events in den Speicher und bei verfügbaren Ressourcen auf eine Festplatte. Der WiDS Checker ist allerdings nur für die WiDS API anwendbar. Wir instrumentieren die Software basierend auf einer Spezifikation und sind daher plattformunabhängig. Unser Ansatz fokussiert auf der Analyse von Speicherverletzungen, welche in keinem der recherchierten Ansätze untersucht wurde.

Tabelle 1: Stand der Technik Übersicht

Feature	Our Method	[Bur13]	[SKBG13]	[NWT+07]	[LLPZ07]
Plattform					
Host	X	Server	X	X	Simulator
Embedded	X				
Language	C/C++	C/C++	Javascript	C/C++	C++/Python
Instrum.					
Program	X		X		
Middleware		VM		iDNA	WiDS API
Record granularity	User-defined events	Drivers input	Instruction	Instruction	Event
Resources for record	Trace copy to file	5%-30% Slowdown	26X Slowdown	0.8 bit / instr. 6X Slowdown	2% Slowdown
Analysis	Memory Usage	Performance, Exploits	Taint, Concolic, Null origins	Data Races	Assertions

4. Methode

Unsere Methode wird komplementär zu vorhandenen Methoden wie modellbasiertem Testen [EHH+13, HSK+13] und statischer Code-Analyse [HHR+13] betrachtet. Sie wird in der letzten Phase des Softwareentwicklungsprozesses, dem Systemtest, angewandt. Der Ansatz (Abbildung 1) beinhaltet die Aufzeichnung der Eingaben der eingebetteten Software im Testbetrieb sowie die automatisierte Analyse von rekonstruierten Ausführungssequenzen. Im Schritt *A.* wird die Software für die Aufzeichnung der Eingaben instrumentiert. Der Schritt *B.* zeichnet die Eingaben in die Software während des Betriebes auf (Sensor- sowie Benutzereingaben), bis ein fehlerhaftes Verhalten durch die Testingenieure observiert wird. Die Aufzeichnungen werden im Labor in einen *Tester* transformiert (Schritt *C.*), welcher die eingebettete Software mit den selben Eingaben triggert wie im Originallauf im Testbetrieb. Im Schritt *D.* wird während der Wiedergabe der Eingaben durch den Tester die Software durch ein dynamisches Analysewerkzeug überwacht. Die Speicherüberwachung wird durch das Werkzeug *Valgrind* [NS07] durchgeführt. Dieses erkennt falsche Speicherzugriffe sowie die fehlende Deallokation von Speicher. Eine durch Valgrind erkannte Speicherverletzung kann einen Hinweis bieten, warum der observierte Fehler im Testbetrieb aufgetreten ist. Die weiteren einzelnen Schritte werden im folgenden beschrieben

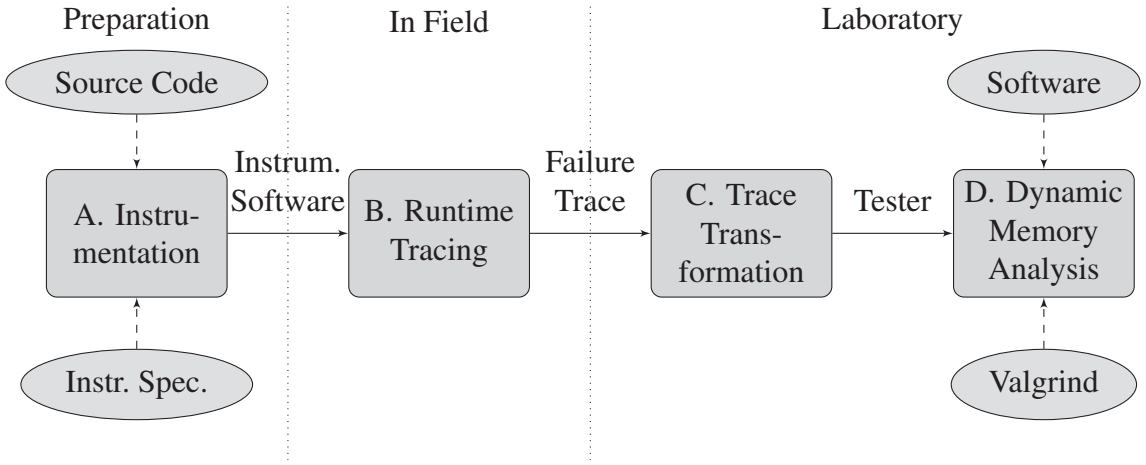


Abbildung 1: Übersicht Methode

A. Instrumentation: Zum Aufzeichnen der Eingaben der Software muss der Quelltext der Software instrumentiert werden. Der Entwickler kann durch eine Instrumentierungssprache definieren, an welchen Stellen im Source Code Aufzeichnungs-Code hinzugefügt werden soll. Hiermit kann die Aufzeichnung flexibel auf die jeweilige Software angepasst werden. Anstatt beispielsweise in einem Navigations-Display jede Berührung des Benutzers mit hoher Frequenz aufzuzeichnen, können nur einzelne zustandsüberführende Klicks gespeichert werden. Die Aufzeichnung für ein Event umfasst: den Typ, die notwendigen Parameter sowie eine genaue Systemzeit, um das Zeitverhalten rekonstruieren zu können. Zur Instrumentierung des Quelltextes wird der ANSI-C/C++ Parser *Clang* [Lat13] eingesetzt.

B. Runtime Tracing: Die Testingenieure führen die Software im Testbetriebs des Endsystems (z.B. ein Navigationssystem bei einer Erlkönigfahrt) aus. Während der Ausführung werden die eingehenden Eingaben erst in den Speicher aufgezeichnet und bei freien Ressourcen oder über-

schrittenen Speicherlimits in eine Datei auf eine externe Festplatte geschrieben. Im Falle von Fehlerbehandlungen, z.B. ein *Segmentation Fault*, werden die Daten in einer Fehlerbehandlungsroutine vom Speicher auf die Festplatte übertragen. Zur Handhabung von *Resets* muss die Aufzeichnung direkt in die Datei geschrieben werden. Hierfür muss die Aufzeichnung vorab hierfür konfiguriert werden. Aufzeichnungen, welche ein Fehlverhalten verursachen werden von den Testingenieuren als *Failure Traces* manuell gesichert.

C. Trace Transformation: Im Labor wird durch ein Testgenerator ein Tester generiert. Dieser Tester liest den Failure Trace und triggert dieselben Eingaben wie im Originallauf. Hierfür sendet der Tester die Events in der gleichen zeitlichen Abfolge mit Hilfe von Timer-Callback-Routinen an die Software. Hat der Tester eine Eingabe aufgerufen, so wird der relative zeitliche Abstand zur nächsten aufgezeichneten Eingabe errechnet und der Timer neu auf diese Dauer (in Millisekunden) gesetzt. Beim nächsten Aufruf des Timers wird die nächste aufgezeichnete Eingabe aufgerufen. Der Tester muss hierfür in den Source Code integriert werden. Für die Wiedergabe wird die Hardware der Zielplattform verwendet, um das gleiche operative und beobachtbare Verhalten wie beim Originallauf zu erhalten.

D. Dynamic Memory Analysis: Während der Wiedergabe der Aufzeichnungen wird das dynamische Speicheranalysewerkzeug Valgrind eingesetzt. Dieses kann verschiedene Speicherverletzungen und Speicherlecks detektieren. In dieser Arbeit werden drei Typen von Speicherverletzungen betrachtet: Nutzung von nicht-initialisierten Variablen, Zugriffe auf freigegebenen Speicher sowie die fehlende Deallokation von Speicher. Für alle erkannten Fehlertypen schreibt Valgrind einen Bericht mit Angaben der Quelltextzeilen, in welchen die jeweilige Verletzung aufgetreten ist. Der im Testbetrieb observierte Fehler kann durch die Korrektur der Speicherverletzungen behoben werden. Durch eine wiederholte Wiedergabe des Testers kann der Entwickler feststellen, ob der observierte Fehler tatsächlich behoben wurde.

Die Anwendung der dynamischen Analyse findet hier offline im Labor anstatt während des Testbetriebes statt. Dadurch können im Testbetrieb Seiteneffekte durch *Valgrind* (z.B. Performance Overhead) verhindert werden. Zusätzlich ist durch die Erstellung eines Testers das wiederholte Testen der Eingabesequenz möglich.

5. Experimente

Der Entwurf und die Entwicklung dieser Methode hat Mitte 2013 begonnen. Dieser Abschnitt beschreibt erste Implementierungen und Versuche. Als Fallstudie wurde die *GENIVI In-Vehicle Infotainment* Plattform *MeeGo IVI* ausgewählt. GENIVI [GEN13] ist ein Konsortium aus wichtigen Partnern aus der Automobilindustrie (BMW, Bosch, Hyundai, GM usw.). Dieses spezifiziert Standards und bietet Referenzimplementierungen für In-Vehicle Infotainment Systeme. MeeGo IVI ist eine Linux-basierte Open Source Betriebssystem-Implementierung von GENIVI. Dieses integriert das Navigationssystem *Navit* [Sch13], welches für erste Versuche von uns verwendet wurde. Navit ist für unseren Ansatz sehr interessant, da es, wie folgend beschrieben, schwer mit gängigen Verifikationstechniken verifizierbar ist. Es besitzt eine komplexe *Plugin*-Struktur mit einem dynamischen *Callback-Management*. Dynamisch weitergeleitete Callback-Aufrufe sind durch eine statische Code-Analyse nur schwer nachvollziehbar. Zusätzlich ist der Eingaberaum von GPS in

der Kombination mit Benutzereingaben sehr groß und Routenführungen lassen sich nur schwer mit modellbasierten Verfahren testen. In unseren ersten Versuchen wird der Quelltext von Navit mit Logging-Code-Fragmenten manuell instrumentiert. Dabei werden Methoden im Quelltext ausgewählt, welche Mausklicks und GPS-Eingaben verarbeiten. Für Mausklicks werden folgende Informationen gespeichert: die relative Zeit zum vorherigen Event in Millisekunden, ID für Mausklick-Events, die Position auf dem Bildschirm, die Maustaste sowie die Information, ob die Taste gedrückt ist. Für GPS-Eingaben wird Folgendes geschrieben: die relative Zeit zum vorherigen Event, ID für GPS Event, Breitengrad, Längengrad, Richtung des Autos (in Grad) sowie die Geschwindigkeit. Diese Code-Fragmente umfassen wenige Zeilen an Quelltext. Die Aufzeichnungen werden bei diesen Versuchen direkt in eine Datei geschrieben (wir planen später eine Zwischenspeicherung in den Speicher zu verwenden). Für die Wiedergabe wird eine Tester-Komponente in den Navit Quelltext integriert, welche einen gespeicherten Trace wiedergibt. Hierbei wird die Trace-Datei Zeile für Zeile gelesen. Die nächsten Zeilen werden beim Aufruf eines *Callback-Timers*, welcher in Navit integriert ist, gelesen. Der Callback-Timer kann auf Millisekunden genau parametrisiert werden. Als Eingaben für die Timer werden die gespeicherten relativen Zeiten in den Logs verwendet. Zur Ausführung der dynamischen Analyse während der Wiedergabe wird das modifizierte Navit mit dem Tester sowie mit dem Speicheranalysewerkzeug Valgrind gestartet. Während der Wiedergabe wird Valgrind angewendet. Dieses detektiert während der Laufzeit falsche Zugriffe auf Speicherbereiche und gibt am Ende der Ausführung einen Bericht über angelegten aber nicht freigegebenen Speicher aus. Es wurden Versuche mit drei Kategorien von Speicherverletzungen unternommen, welche im Folgenden beschrieben sind.

Nicht-initialisierte Variablen: Die GPS-Daten werden bei Navit durch externe Module aus dem GPS-Empfänger geladen, vom *Vehicle*-Plugin verarbeitet und an das *Navit*-Plugin über Callback-Routinen gesendet. Wir nehmen bei diesem Beispiel an, dass bei manchen GPS-Sensoren fehlerhafte GPS-Daten berechnet werden (z.B. außerhalb des Wertebereichs Breitengrad: -90° bis 90° und Längengrad:-180° bis 180°) wenn nicht genügend Satelliten in Reichweite sind. Wir haben hierfür in der Verarbeitungsroutine der GPS-Daten im *Navit*-Plugin, der *navit_vehicle_update*-Methode, die folgende fehlerverursachende Fallunterscheidung hinzugefügt. Diese prüft, ob die GPS-Koordinaten im Wertebereich sind.

```
void navit_vehicle_update(float lng, float lat, ...){
 if (lng>180||lng<-180||lat>90||lat<-90){
    lng=vehicle->coord->lng;
    lat=vehicle->coord->lat;
 }
 // Process lng and lat
}
```

Ist eine der Koordinaten außerhalb des Wertebereichs, so werden die Eingaben verworfen und die letzte gespeicherte GPS-Koordinate *coord* erneut als GPS-Eingabe verwendet. Diese wird global je Fahrzeug gespeichert. Die Koordinate *coord* wird im Quelltext bei jedem GPS Event im *Vehicle*-Plugin gesetzt. Daher ist die Variable beim ersten GPS Event nicht initialisiert. In einen aufgezeichneten Trace haben wir die erste GPS-Koordinate außerhalb des Wertebereiches injiziert. Diese wird durch die Fallunterscheidung verworfen und die Variable *coord* als GPS-Koordinate verwendet. Sie

wurde allerdings nicht bei den letzten GPS Event-Verarbeitungen gesetzt und beinhaltet dadurch nicht-initialisierten Speicher. Durch den Anwender kann ein Fehlverhalten des Navigationssystems erkannt werden, da die Karte kurz auf eine willkürliche Koordinate springt und anschließend korrekt weiterläuft. Bei Anwendung von Valgrind wird erkannt, dass die Variable *coord* im ersten Lauf nicht initialisiert ist, aber in der Verarbeitung von GPS-Koordinaten verwendet (z.B. bei der Visualisierung der aktuellen Position) wird. Basierend auf diesem Bericht kann der Entwickler den Fehler beheben. Der beschriebene Fehler ist nur schwer mit statischer Code-Analyse verifizierbar, da *coord* als Pointer in einem Plugin gesetzt (*Vehicle*) und in einem anderen (*Navit*) gelesen wird. Die gegenseitigen Aufrufzusammenhänge sind nicht nachvollziehbar, da diese über die dynamischen Callback-Routinen stattfinden. Mit modellbasiertem Testen könnte der Fehlerfall basierend auf der Modellierung der Fallunterscheidungen nachgebildet werden [EHH+13, HSK+13], allerdings kann es im Softwareentwicklungsprozess zu Differenzen zwischen Spezifikation und Implementierung kommen. Im Testbetrieb ist der Fehler leicht feststellbar, wenn der GPS-Sensor beim Start des Navigationssystems bei der Empfangssuche fehlerhaft reagiert.

Zugriffe auf freigegebenen Speicher: Navit gibt Navigationsassistenz-Hinweise auf der Kommandozeile oder per Sprachgenerierung aus. Wird beispielsweise in die falsche Richtung gefahren, so wird der Hinweis: *When possible, please turn direction* ausgegeben. Die Ausgabe wird noch ins Deutsche übersetzt. Der Befehl wird aus einer Datei gelesen und in den Speicher geladen. Wir haben einen Fehler injiziert, indem beim Laden des Befehltextes in den Speicher der allozierte Speicher gleich nach dem Laden wieder fehlerhaft freigegeben wird. Wird Navit ausgeführt und es wird in die falsche Richtung gefahren, so wird bei der Ausgabe des Hinweises auf den freigegebenen Speicher zugegriffen. Dadurch wird eine fehlerhafte Ausgabe durchgeführt. Erkannt wird der Fehler nur, wenn testweise mit einem Fahrzeug in die falsche Richtung gefahren wird. Wir haben eine solche Testfahrt aufgezeichnet. Bei der Wiedergabe erkennt Valgrind, dass es sich bei der Verarbeitung des Textes des Wendehinweises um freigegebenen Speicher handelt. Darauf basierend kann der Entwickler debuggen und die fehlerhafte Freigabe des Speichers entfernen. Auch bei diesem Fehlerfall findet ein Austausch des geladenen Textes als Pointer-Variable über mehrere Plugins statt, was eine statische Analyse erschwert. Die automatische modellbasierte Generierung eines Testfalls für das Fahren einer falschen Route ist nur schwer mit einem manuell implementierten Routengenerator lösbar.

Speicherlecks: Eine wichtige Funktion von Valgrind ist es, angelegten, aber nicht freigegebenen Speicher zu erkennen. Hierbei konnten bei der Anwendung von Valgrind bei der Wiedergabe einer Testfahrt verschiedene Speicherlecks in der bestehenden Software erkannt werden. Zusätzlich haben wir verschiedene Fehler injiziert, bei welchen Speicher angelegt, aber nicht gelöscht wird. Vor allem Speicherleck, welche von mehreren Plugins sowie verschiedenen Eingaben von GPS und den Benutzern abhängen, sind durch andere Verifikationstechniken schwer detektierbar.

Die drei Experimente zeigen, dass es anwendungsspezifische Fehler in einer Navigationssystemsoftware gibt, welche sich durch gängige Verifikationsverfahren aufgrund einer komplexen Plugin-Struktur und eines großen Eingaberaums nur schwer verifizieren lassen. Mit unserem Verfahren können diese Fehler im Testbetrieb leicht erkannt und effizient im Labor rekonstruiert werden. Die dynamische Speicheranalyse während der Wiedergabe gibt hilfreiche Hinweise zur Behebung der Ursachen von Fehlerverhalten der Software.

6. Zusammenfassung und Ausblick

Unser Ziel bei dieser Forschung ist die Aufzeichnung von Fehlern im ersten Testbetrieb und die effiziente Fehleranalyse bei der Wiedergabe im Labor. Dadurch können Fehler erkannt werden, welche mit gängigen Verifikationstechniken nur schwer feststellbar sind, da diese von externen Systemen, Benutzereingaben sowie Sensordaten abhängen. In diesem Beitrag wurden verschiedene bestehende Ansätze für Record/Replay mit einer dynamischen offline-Analyse vorgestellt. Keiner der untersuchten Ansätze war zugeschnitten für eingebettete Systeme oder Infotainment-Systeme. Die meisten verwenden performanzaufwändige Record/Replay-Verfahren oder sind nur für bestimmte APIs oder eine bestimmte Middleware implementiert. Keiner der Ansätze hat eine offline-Analyse durch ein dynamisches Speicheranalysewerkzeug gezeigt. Bei unserer vorgestellten Methode kann der Entwickler das Record/Replay während des Testbetriebs genau auf die Besonderheit der Software anpassen und nur relevante Events in eine Log-Datei speichern und später wiedergeben. Die aufgezeichneten Events können mithilfe eines Testers mit gleichem Timing (momentan Millisekunden-genau) wie im Originallauf mehrmals wiedergegeben werden. Während der Wiedergabe wird bei unserer Methode das Speicheranalysewerkzeug Valgrind angewendet. Dadurch können verschiedene Fehlertypen erkannt werden. Bei unseren Versuchen konnten wir Beispiele zeigen, bei welchen die fehlerhafte Nutzung von nicht-initialisiertem Speicher sowie freigegebenem Speicher erkannt wurde. Zusätzlich wurden Beispiele für Speicherlecks getestet. Bei den vorgestellten Beispielen handelt es sich um spezielle Ausführungsfälle, welche leicht im Testbetrieb, allerings nur schwer durch automatisiertes Testen oder automatisierte Verifikation erkannt werden können.
Das Verfahren bietet Potential für weitere Implementierungen und Messungen. Das Vorgehen für das Record/Replay muss weiter ausgebaut werden, um Originalläufe zeitgenau und deterministisch wiederzugeben. Hierfür müssen auch Thread Schedules betrachtet und reproduziert werden. Denkbar ist die Integration von Synchronisationspunkten zur exakten Einstreuung von aufgezeichneten Eingaben in die Eingabeverarbeitung. In weiteren Arbeiten möchten wir hierfür Messungen zeigen, inwieweit sich Originalläufe zu wiedergegebenen Läufen unterscheiden. Des Weiteren möchten wir verschiedene dynamische Analyseverfahren während der Wiedergabe testen. Sehr interessant sind hierbei Verfahren im Bereich der *Runtime Verification* [LS09], bei welchen das Verhalten der Software mit einer benutzerdefinierten Spezifikation verglichen wird. Die Idee hierfür wurde von uns bereits beim Phd Workshop der ICTSS 2013 präsentiert [EKGR13].

Literatur

[BCdJ+06] Bhansali, Sanjay, Wen Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka und Joe Chau: *Framework for instruction-level tracing and analysis of program executions*. In: *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, Seiten 154–163, New York, 2006.

[Bur13] Burtsev, Anton: *Deterministic System Analysis*. Dissertation, The University of Utah, 2013.

[EHH+13] Eichelberger, Hanno, Patrick Heckeler, Stefan Huster, Sebastian Burg, Jürgen Ruf, Thomas Kropf, Wolfgang Rosenstiel und Bastian Schlich: *Beschleunigte Robustheitstests für verhaltensbeschreibende Zustandsmaschinen*. In: *16. Workshop Methoden*

und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Seiten 161–170, Rostock, März 2013.

[EKGR13] Eichelberger, Hanno, Thomas Kropf, Thomas Greiner und Wolfgang Rosenstiel: *Runtime Verification Driven Debugging of Replayed Errors*. In: *Proceedings of PhD Workshop of the International Conference for Testing of Software and Systems (ICTSS)*, Istanbul, 2013.

[Fou13] Foundation, Linux: *Xen Virtualization Platform*. `http://www.xenproject.org/`, 2013. [Zugriff 18-Nov-2013].

[GEN13] GENIVI Alliance: *GENIVI Homepage*. `www.genivi.org`, 2013. [Zugriff 18-Nov-2013].

[HHR+13] Huster, Stefan, Patrick Heckeler, Jürgen Ruf, Sebstian Burg, Thomas Kropf und Wolfgang Rosenstiel: *A Software Testing Framework to Integrate Formal Verification Results*. In: *16. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Seiten 183–192, Rostock, März 2013.

[HSK+13] Heckeler, Patrick, Bastian Schlich, Thomas Kropf, Gabriel R. Cardoso, Hanno Eichelberger, Jürgen Ruf, Stefan Huster, Sebastian Burg und Wolfgang Rosenstiel: *Accelerated Robustness Testing of State-Based Components using Reverse Execution*. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, Band 2, Seiten 1188–1195, Coimbra, Portugal, März 2013. ACM.

[Lat13] Lattner, Chris: *clang: a C language family frontend for LLVM*. `http://clang.llvm.org/`, 2013. [Zugriff 18-Nov-2013].

[LLPZ07] Liu, Xuezheng, Wei Lin, Aimin Pan und Zheng Zhang: *WiDS checker: combating bugs in distributed systems*. In: *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, Seiten 19–19, Berkeley, 2007.

[LS09] Leucker, Martin und Christian Schallhart: *A brief account of runtime verification*. J. Log. Algebr. Program., 78(5):293–303, 2009.

[NS07] Nethercote, Nicholas und Julian Seward: *Valgrind: a framework for heavyweight dynamic binary instrumentation*. SIGPLAN Not., 42(6):89–100, 2007.

[NWT+07] Narayanasamy, Satish, Zhenghao Wang, Jordan Tigani, Andrew Edwards und Brad Calder: *Automatically classifying benign and harmful data races using replay analysis*. In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, Seiten 22–31, New York, 2007.

[Sch13] Schaller, Martin: *Navit Homepage*. `www.navit-project.org`, 2013. [Zugriff 18-Nov-2013].

[SKBG13] Sen, Koushik, Swaroop Kalasapur, Tasneem Brutch und Simon Gibbs: *Jalangi: a selective record-replay and dynamic analysis framework for JavaScript*. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, Seiten 488–498, New York, 2013.

Synthesis of Distributed Synchronous Specifications to SysteMoC

Mohamed Ammar Ben Khadra
TU Kaiserslautern
ammar@rhrk.uni-kl.de

Yu Bai
TU Kaiserslautern
bai@cs.uni-kl.de

Klaus Schneider
TU Kaiserslautern
schneider@cs.uni-kl.de

Abstract

Heterogeneous multi-core embedded systems are being increasingly used to meet performance requirements of modern applications. However, meeting the often stringent demands for correctness, resource utilization, and real-time response in a complex concurrent software is challenging. Model-based design is a widely accepted methodology to meet such requirements where a *functional* model of the system is developed independently from its *architectural* model. In that regard, functional models developed in synchronous languages have been successfully used for synthesis of hardware circuits and sequential software. However, significant work on distributed software synthesis from synchronous languages remains only theoretical. We discuss a semantics-preserving synthesis procedure for elastic networks (a common representation of distributed synchronous specifications) to SysteMoC which is an actor-oriented modeling library based on SystemC. Hence, we not only demonstrate a potential simulation platform that can be used to improve existing theory, but we also make a step towards a *formal* model-based design flow that combines synchronous and actor-oriented Models of Computation (MoC). Additionally, we discuss some experimental results based on our implementation of the synthesis procedure.

1. Introduction

Meeting the ever increasing demand for more processing power in many embedded systems is challenging. Compared to traditional software, concerns of embedded software designers go beyond developing concurrent software that can take advantage of the available hardware parallelism. Actually, the real challenge lies in meeting non-functional system requirements including correctness, resource utilization, and real-time response. To this end, model-based design is regarded by many as a promising approach to meet such stringent requirements where a functional model of the system is developed independently from its target architecture. That allows designers to focus on core functional issues without being distracted by architecture specific issues. That also enables re-using the same functional model across different target architectures. After validating the functional model, designers proceed with the synthesis phase where the design space is explored to find

a suitable software mapping and scheduling strategy. Additionally, hardware synthesis should be considered too for a comprehensive model-based design approach.

We identify some key properties that need to be satisfied in an ideal functional model. (P1) formal verifiability which is mandatory for safety-critical applications. However, it is also increasingly important for other applications due to software complexity. (P2) composability where the model can be analyzed and composed in a modular manner from existing models. (P3) analyzability for early estimation of non-functional system properties. (P4) expressiveness in a wide range of application domains including data-dominated as well as control-dominated applications. Additionally, hybrid system modeling is increasingly important with prevalence of cyber-physical systems. Finally, (P5) synthesizability such that the generated software is efficient, semantics-preserving, and satisfies the non-functional requirements of the system. Unfortunately, improving P5 mandates improving P3. In turn, that typically comes at the cost of reducing support for one or more of the other properties. For example, adhering to a restricted MoC such as static dataflow networks [LM87] improves P3 at the cost of reducing model expressiveness in P4. Those limitations have been acknowledged in projects like Ptolemy II [EJL+03] which focused on formal composition of multiple MoCs to achieve higher expressiveness.

Synchronous languages [BCE+03] have emerged since early eighties to address the specification issues of reactive systems. They are based on a simple MoC hypothesis of *perfect synchrony* where the execution of the system is divided into discrete reaction steps called *macro-steps*. In each macro-step, the system reads all inputs, executes a finite number of *micro-steps*, and finally produces outputs w.r.t. internal system state. All micro-steps are executed in the same variable environment i. e. a variable can take only one value in a macro-step. Micro-steps are assumed to consume no time, and time advances to the next macro-step after all micro-steps are finished. They offer key advantages over traditional programming languages including deterministic concurrency and explicit modeling of time. Time is essential for improved analyzability [Lee09] and it can be used for, e. g. , WCET analysis [LS03]. Moreover, they enable designers to associate different computation costs with different macro-steps through clock-refinement [GBS13]. Hence, early estimations of resource utilization can be conducted.

Simplicity of the synchronous hypothesis provides easier reasoning about system behavior. Additionally, the sound formal basis and lock-step composition of synchronous modules enable more efficient formal verification since asynchronous concurrent behaviors should not be considered. Hence, synchronous languages already satisfy properties P1 and P2 to a large extent and provide many interesting properties for P3. However, their main application domain was memory-bounded control-dominated applications. That is demonstrated with commercial successes for Esterel [Ber98] and Lustre [HCRP91] in efficient synthesis of real-time uni-processor software as well as hardware. In that respect, work on distributed software synthesis, also known as desynchronization, for multicore processors and distributed environments like automotive software, is still an active area of research.

All events in a synchronous model are synchronized and totally ordered w.r.t a global clock and communication takes zero-time. Obviously, that is an ideal assumption that cannot be efficiently realized in practice. Hence, many desynchronization approaches have been proposed in the literature to relax perfect synchrony while preserving the original semantics (see [Gir05] for a good survey). Among the proposed techniques are endo/isochronous distribution [BCL00] and efficient communication through polychrony [LTL03]. In that regard, there is a lack of suitable simulation tools that help in exercising and improving that body of theoretical knowledge.

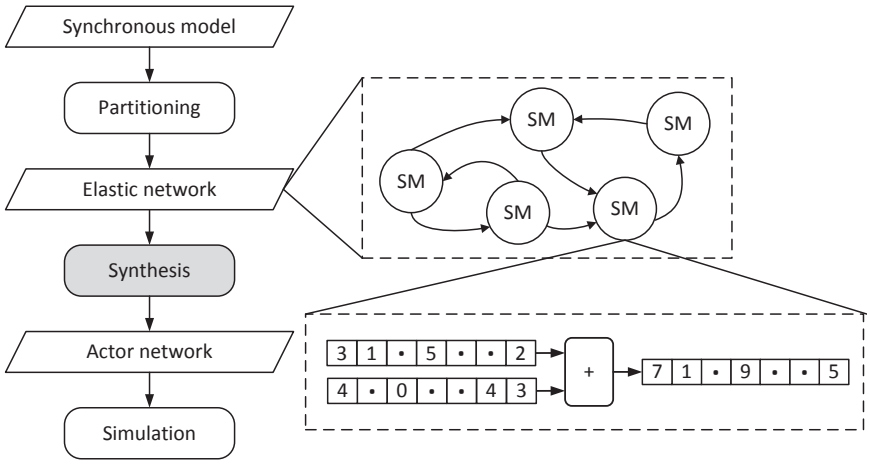


Figure 1: Our simulation flow where the considered synthesis procedure is highlighted. The communication model of synchronous modules (SM) using synchronous channels (SC) is emphasized.

We discuss in this paper a semantics-preserving synthesis procedure for elastic networks, an abstract distributed synchronous specification that consists of synchronous modules (SM) that communicate over synchronous channels (SC). Elastic networks are the first refinement step of perfect synchrony where computations are still synchronized while communication over SC takes time in the form of an integer number of clock cycles. The term has been adopted from elastic circuits [CCKT09] since they both share the same 'view' of the system. Hence, elastic network optimizations can be directly transferred to their hardware counterparts. We avoid the term synchronous data-flow since it is often confused with static dataflow networks [LM87]. Formally,

Definition 1. *A **synchronous module** is a tuple $(\mathcal{P}, \mathcal{L}, \mathcal{CF}, \mathcal{DF})$ where $\mathcal{P}$ is a set of input and output ports $\mathcal{P} = \mathcal{I} \cup \mathcal{O}$. $\mathcal{L}$ is the set of local variables which define the state of the module. $\mathcal{CF}$ and $\mathcal{DF}$ define the list of control-flow and data-flow guarded actions respectively.*

Definition 2. *A **synchronous channel** is a tuple $(sP, dP, \mathcal{D}, \mathcal{C}, \mathcal{T})$ where sP and dP are SM that are the source and destination port, respectively. $\mathcal{D}$ is minimum token delay. $\mathcal{C}$ is the maximum capacity of the channel. $\mathcal{T}$ is the supported token type which can be basic, e. g. integer, or aggregate.*

We used guarded actions to characterize SMs. Basically, a guarded actions consists of an action, usually an assignment operation, that is not executed until its associated boolean guard is satisfied. More on guarded actions has been provided in Section 2. An SC is a FIFO capable of holding tokens (valid data) and bubbles (empty). At each clock tick, a token can advance iff the item in the next FIFO cell is a bubble. That creates back-pressure on producing SM to stop if a consuming SM has stalled. $\mathcal{D}$ determines the minimum number of clock cycles needed by a token through an SC. That number may increase if a receiving SM stalls. Additionally, an SM cannot fire until there are tokens on **all** of its input ports and bubbles on **all** of its output ports (patience property). Upon firing, exactly one token is consumed/produced on every input/output port.

The entire simulation flow is depicted in Figure 1 where we obtain our elastic network representation from a partitioning stage that is beyond our scope. We focus in this work on the synthesis stage of elastic networks only. Our synthesis target is SysteMoC [FHT06], a SystemC based

```
module MAC(int ?a, ?b, !s) {
 int i, t, o;
 loop {
   w1: pause;
   t = i;
   o = a*t;
 }
 ||
 loop {
   w2: pause;
   i = a+b;
   if (t < 0) {
     w3: pause;
     s = o;
   } else {
     w4: pause;
     if (b>0)
       s = o + 1;
}}}
```

$$\begin{bmatrix}
\alpha_1: & start \vee w_1 & \Rightarrow & next(w1) = \mathsf{true} \\
\alpha_2: & start \vee w_3 \vee w_4 & \Rightarrow & next(w2) = \mathsf{true} \\
\alpha_3: & w_2 \wedge (t < 0) & \Rightarrow & next(w3) = \mathsf{true} \\
\alpha_4: & w_2 \wedge \neg(t < 0) & \Rightarrow & next(w4) = \mathsf{true} \\
 & & & \\
\beta_1: & w_1 & \Rightarrow & t = i \\
\beta_2: & w_1 & \Rightarrow & o = a \times t \\
\beta_3: & w_2 & \Rightarrow & i = a + b \\
\beta_4: & w_3 & \Rightarrow & s = o \\
\beta_5: & w_4 \wedge (b > 0) & \Rightarrow & s = o + 1
\end{bmatrix}$$

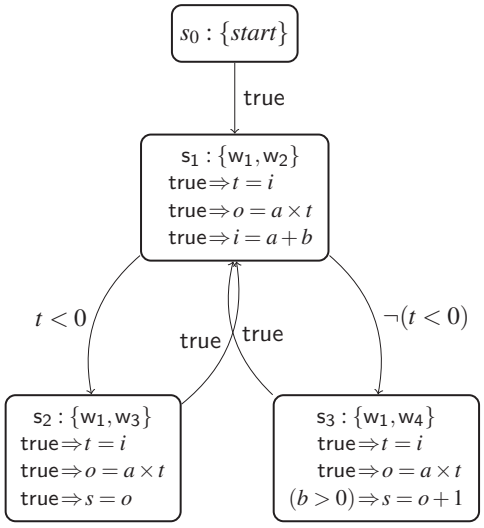


Figure 2: (left) Quartz module MAC, (middle) compiled guarded actions where $\mathcal{CF}$ (α_i) and $\mathcal{DF}$ (β_i), and (right) generated EFSM.

actor-oriented modeling library. SysteMoC provides both time-triggered and asynchronous data-triggered actor firing options. Hence, it enables gradual refinement of a synchronous model from perfect synchrony to a completely asynchronous actor network model. Our discussion of the synthesis procedure is organized as follows: First, we give in Section 2 the necessary background. Then, we discuss the details of synthesizing SMs and SCs in Section 3. Later, some experimental results based on our implementation are described in Section 4. Finally, we discuss some related work in Section 5.

2. Background

We give in this section the necessary background on synchronous guarded actions and later on actor-oriented modeling using SysteMoC.

2.1. Synchronous guarded actions

Figure 2 depicts an example synchronous module MAC written in the imperative synchronous language Quartz [Sch09] which is the specification language of our Averest framework[1]. Module MAC has input variables **`a`** and **`b`**, output variable **`s`**, and local variables **`i`**, **`t`**, **`o`**. Macro-steps are determined using **`pause`** statements which has been assigned *labels* to show the control-flow of the module. Basically, MAC consists of two infinite loops running in parallel. However, they are running in lock-steps synchronizing at each **`pause`**. Quartz syntax of MAC is shown only to provide an intuitive description of how a synchronous model would look like. Compiling MAC using Averest's Quartz compiler would yield a set of control-flow guarded actions (CGA) and data-flow guarded actions (DGA).

[1] Available at `http://www.averest.org`

A guarded action (GA) has the form $\langle \gamma \Rightarrow \alpha \rangle$, where guard γ is a boolean expression that needs to be satisfied in order for action α to be executed. We are mostly concerned with assignment actions. An *immediate assignment*, denoted by $\langle x = e \rangle$, assigns x to the evaluated result of expression e in the current macro-step. A *delayed assignment* denoted by $\langle \mathit{next}(x) = e \rangle$, assigns x to the value of e in the current macro-step. However, the assignment is executed in the next macro-step. Note that most synchronous languages can be compiled to guarded actions which makes our characterization of SM behavior in terms of set of CGAs ($\mathcal{CF}$) and set of DGAs ($\mathcal{DF}$) a common intermediate format. CGAs differ from DGAs in that their actions are assignments to control-flow *labels* whereas actions of DGAs assign values to module variables only.

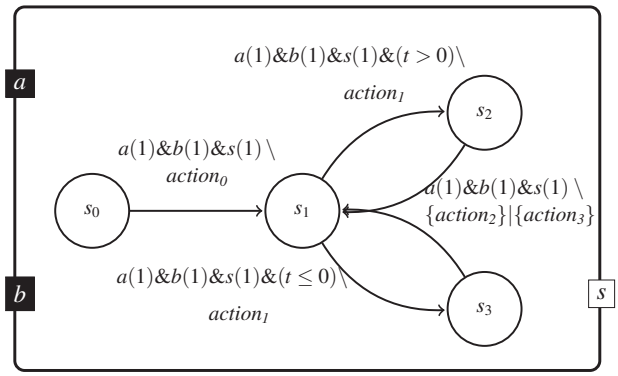
One can represent the behavior of synchronous system with a state machine that has only a single state with $\mathcal{DF}$ attached to it. At each clock tick, all guards γ of DGAs in $\mathcal{DF}$ are evaluated. Then, only the assignments α that have their guards γ evaluated to **`true`** will be executed. However, we are interested in a more efficient representation, in terms of computation, of the system by only evaluating DGAs that belong to the current synchronous reaction. To this end, we generate an Extended Finite State Machine (EFSM) from the given $\mathcal{CF}$ of the system. Then, each DGA is attached only to the state(s) where it could possibly be executed. An EFSM is formally defined in Definition 3. Figure 2 depicts $\mathcal{CF}$, $\mathcal{DF}$ and the generated EFSM of module MAC. The default starting state is s_0 where only label $start$ is set. Each state has been annotated with its control flow labels and its attached DGAs. Note that each state represents a synchronous reaction where all system variables should have a unique value. A *reaction to absence* should take place for variables that have not been assigned a value by DGAs of the current synchronous reaction. In that respect, variables in Quartz can be either of type *memorized* or type *event* depending on their required reaction to absence behavior. *Memorized* variables are assigned their same value in the previous reaction, whereas *event* variables take the default value for their data type e. g. **false** for booleans.

Definition 3. *An **Extended Finite State Machine (EFSM)** is a tuple $(S, s0, T, D)$, where S is a set of states, $s0 \in S$ is the initial state, and $T \subseteq (S \times G \times S)$ is a finite set of transition relations where G is the set of transition guards. D is a mapping $S \rightarrow D$, which assigns each state $s \in S$ a set of DGAs $D(s) \subseteq D$ which are executed in state s.*

2.2. Actor-oriented modeling in SysteMoC

SysteMoC is used to describe a network graph of communicating actors. Each actor has a set of input ports $\mathcal{I}$ and a set output ports $\mathcal{O}$. Ports have a supported token type e.g. double. An actor's input port should be connected to the output port of another actor that supports the same token type. The connection is a FIFO that has a configurable size. Actor's internal states are defined by a set of local variables $\mathcal{L}'$ that are readable and writable only inside the actor. The behavior of the actor is defined by a Firing-FSM (FFSM). Formally,

Definition 4. *An **actor network graph** is a directed bipartite graph (A, C, P, E) containing a set of actors A, a set of channels C, a channel parameter function $P : C \rightarrow N^{\infty} \times V^*$ which associates with each channel $c \in C$ its buffer size $n \in N^{\infty} = \{1, 2, 3, \ldots \infty\}$, and possibly also a non-empty sequence $v \in V^*$ of initial tokens, and finally a set of directed edges $E \subseteq (C \times A.\mathcal{I}) \cup (A.\mathcal{O} \times C)$. The edges are further constrained such that exactly one edge is incident to each actor port and the in-degree and out-degree of each channel in the graph is exactly one.*




```
class MAC : public smoc_actor {
    SC_HAS_PROCESS(MAC);
  smoc_firing_state s0, s1, s2, s3;
  // Other definitions omitted.
public:
  smoc_port_in<int> a, b; smoc_port_out<int> s;
  MAC(sc_module_name name)
    : // Constructor instantiations omitted.
  {
  // Only firing transitions of s1 are shown.
    s1=TILL(clk)>>(a(1)&&b(1)&&(guard1))>>s(1)
        >>CALL(MAC::action1)>>s2|
        TILL(clk)>>(a(1)&&b(1)&&(guard2))>>s(1)
        >>CALL(MAC::action1)>>s3;
 } };
```

Figure 3: MAC actor model, (left) a graphical model where each $action_i$ executes DGAs corresponding to s_i, patience property is observed by guarding all firing transitions such that **one** token (bubble) should be available on all input (output) ports before firing, e. g. a(1). (right) textual C++ code generated for the actor where **guard1** and **guard2** are functions $(t > 0)$ and $(t \leq 0)$ respectively.

Definition 5. *An **actor** is a tuple $(\mathcal{P}, \mathcal{L}', \mathcal{F}, \mathcal{R})$ where $\mathcal{P}$ is a set of input and output ports $\mathcal{P} = \mathcal{I} \cup \mathcal{O}$, $\mathcal{L}'$ is the set of local variables which define the state of the actor, $\mathcal{F}$ is a set of functions, and $\mathcal{R}$ is the firing FSM.*

Note that $\mathcal{F} = \mathcal{F}_G \cup \mathcal{F}_A$ where $\mathcal{F}_G$ is a set of boolean functions used to guard state transitions, and $\mathcal{F}_A$ is set of firing actions executed upon firing and able to change values of $\mathcal{L}'$. Note also that an EFSM is a restricted form of an FFSM where the relation between FSM states and $\mathcal{F}_A$ is bijective. Therefore, we can describe the behavior of actors using EFSM. Transition guards of an EFSM are described by $G \subseteq \mathcal{G}_{clk} \times \mathcal{G}_I \times \mathcal{G}_O \times \mathcal{F}_G$, where $\mathcal{G}_I$ ($\mathcal{G}_O$) are used to guard that sufficient number of tokens (bubbles) are available on input (output) ports to be consumed (produced), and $\mathcal{G}_{clk}$ is a clock event condition used if clock synchronization is required. Figure 3 depicts the actor model and the generated code for module MAC. Clocks in SysteMoC are defined per actor. Therefore, transitions can be synchronized by setting an equal clock period (main clock) for all actors in the network and setting all firing transitions to have guard $\mathcal{G}_{clk}$. It is simple to model actors that need multi-cycles for their computation by setting their clock to an integer number of the main clock period. It is even possible to configure delay time for individual firing transitions by using Virtual Processing Component (VPC) which is a supporting framework to SysteMoC.

3. Synthesis details

We discuss in this section the synthesis details of an SM and an SC separately.

3.1. Synthesis of a Synchronous Module

Given an SM defined by $(\mathcal{P}, \mathcal{L}, \mathcal{CF}, \mathcal{DF})$, we need to generate its corresponding SysteMoC actor $(\mathcal{P}, \mathcal{L}', \mathcal{F}, \mathcal{R})$. To this end, we need to generate and synthesize the EFSM ($\mathcal{R}$) based on the given $(\mathcal{CF}, \mathcal{DF})$. Additionally, we synthesize $\mathcal{F} = \mathcal{F}_G \cup \mathcal{F}_A$ where each firing action in $\mathcal{F}_A$ is generated from DGAs of a corresponding state in $\mathcal{R}$, while $\mathcal{F}_G$ are generated from transition guards of $\mathcal{R}$.

Finally, actor variables have to be synthesized such that $\mathcal{L}' = \mathcal{L} \cup \mathcal{L}_E$ where $\mathcal{L}_E$ are extra variables required to handle the semantic mismatch between the two different MoCs e. g. output variables are writable only in actors whereas they are both readable and writable in synchronous guarded actions. Details of SM synthesis have been briefly discussed here. We refer to [Ben13] for complete details including code generation templates.

Our EFSM generation algorithm proceeds by evaluating (to a boolean canonical form) all guards of $\mathcal{CF}$ based on the label variable environment of current state. Then, a subset $\mathcal{CF}'$ is defined such that its guards didn't evaluate to **false** in the previous step. Later, $\mathcal{CF}'$ goes through *case discrimination* to identify the variable environment of each of the next state(s) to be visited. The algorithm starts with the label environment of the initial state, i. e. , only label *start* is assigned value **true**. Attaching DGAs to a state is done by evaluating all guards of $\mathcal{DF}$ based on the state's label environment and attaching the corresponding subset $\mathcal{DF}'$ with guards that were not reduced to **false**. As for actor variable $\mathcal{L}'$, we provide in Table 1 a summary of the generated variables. The rationale behind that will get clearer next in our discussion of firing action generation.

Table 1: Synthesizing variables of a synchronous module.

Variable flow	Variable type	Generated variables
Input	Memorized	Port variable only.
	Event	Port variable only.
Output	Memorized	Port variable and local carry variable .
	Event	Port variable, a local carry variable, and a flag variable.
Local	Memorized	A local direct variable, a local carry variable, and a flag variable.
	Event	A local direct variable, a local carry variable, and a flag variable.

An actor firing action proceeds in two phases, namely, immediate and delayed. In the immediate phase, (1) the reaction to absence for all variables of type *event* are executed and flags of delayed assignments are checked to be executed. To this end, all local variables and output variables of type *event* should have a boolean flag that is set when a delayed value is available from previous reaction to be assigned. (2) immediate guarded actions are ordered and executed, (3) by this time, all variable values in the current reaction are known and output values can be propagated on ports. The delayed phase can now proceed where (1) DGAs writing to local variables are executed first. However they are rewritten such that writing is done to a *carry* variable and not to the actual *direct* variable. That is because the current value of local variables might be used by transition guard functions $\mathcal{F}_G$ to determine the next transition. Finally, (2) DGAs writing to output variables are ordered and executed.

Definition 6. ***Guarded action dependencies****: let $G = \langle \gamma \Rightarrow x = \tau \rangle$ be a guarded action where γ is the boolean guard, x is the variable assigned a value, and τ is an expression. Let $FV(\tau)$ be the set of free variables in expression τ. We define the following:*

$$rdVars(\gamma \Rightarrow x = \tau) = FV(\gamma) \cup FV(\tau) \qquad rdVars(\gamma \Rightarrow next(x) = \tau) = FV(\gamma) \cup FV(\tau)$$
$$wrVars(\gamma \Rightarrow x = \tau) = \{x\} \qquad wrVars(\gamma \Rightarrow next(x) = \tau) = \{x\}$$

Definition 7. ***Guarded actions order****: for guarded actions $G1$ and $G2$, we say that $G1 < G2$ iff $wrVars(G1) \subseteq rdVars(G2)$.*

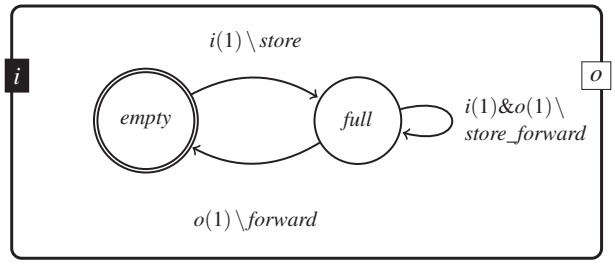


Figure 4: Actor model of a Channel Buffer (CB). Channel buffers are chained with smoc_fifo to build a synchronous channel where each CB contributes to a delay of one clock cycle.

We provide a partial order relation on guarded actions in Definitions 6 and 7. Ordering of guarded action in immediate and delayed phases requires finding a total (sequential) order for DGAs based on the analyzed partial order. In that regard, a topological sorting is done such that for each two DGAs where $G_1 < G_2$: G_1 should appear before G_2 in the immediate phase (RAW dependency) and G_1 should appear after G_2 in the delayed phase (WAR dependency). Consider for example DGAs of state s_1 in Figure 2. DGA (RAW) ordering in the immediate phase should be β_3, β_1, and β_2. However, no delayed assignments exist in s_1. Therefore, the delayed phase shall be empty in its firing action.

3.2. Synthesis of a Synchronous Channel

We now consider the synthesis of an SC defined by tuple $(sP, dP, \mathcal{D}, \mathcal{C})$. Note that it is important for us to make SMs completely independent of their communication over their SCs. In that way, we can reuse generated SM code and simulate them with different SC configuration of $\mathcal{D}$ and $\mathcal{C}$. Unfortunately, SysteMoC supports only one FIFO type for communication between actors which is defined in the class **`smoc_fifo`**. A **`smoc_fifo`** can be considered as an SC that has zero delay i. e. $\mathcal{D} = 0$. Therefore, we had to simulate the clock cycle delay on an SC by introducing Channel Buffers (CB). Basically, a CB is a basic SysteMoC actor that has the sole purpose of delaying its input by one clock cycle. Therefore, to model an SC with delay of $\mathcal{D}$ clock cycles, we need to generate a chain of $\mathcal{D}$ number of CBs connected by **`smoc_fifo`**.

The actor model of a CB is depicted in Figure 4. Required storage capacity $\mathcal{C}$ of the SC can be distributed among **`smoc_fifo`** in the chain. Our arrangement separates SC delay $\mathcal{D}$ from SC capacity $\mathcal{C}$ which are implemented by CB and **`smoc_fifo`** respectively. Note that an CB is essentially a single C++ template class in the generated source code. Hence, the C++ compiler would instantiate as many classes as needed depending on the declared SC token type $\mathcal{T}$. Note that it is also possible to implement different $\mathcal{D}$ and $\mathcal{C}$ of an SC using a single complex CB instead of our chain of simple CBs. However, the C++ compiler will then need to instantiate a different C++ class for each different combination of $\mathcal{D}$, $\mathcal{C}$ and $\mathcal{T}$ which may result in a larger executable. Note also that asynchronous communication can be modeled using a single CB that has its clock period set to arbitrary time period rather than to the main clock period as in synchronous channels.

4. Experimental results

We have developed a synthesis library that is capable of generating code for most features of Quartz including all of its data types. The library was developed in F#.NET and it has about 3900 Lines of Code (LoC). We took advantage of the data types already supported by SystemC, e. g. , bitvectors (**`sc_bv`**) were elegantly mapped to their Quartz counterparts. Synthesis of aggregate data types (tuples) was also straightforward by mapping them to C++ structs. We also supported **`assert`** statement generation to make sure that the original specifications are not violated at run-time, e. g. , out-of-bound array accesses. We consider here a simple partitioning strategy where the synchronous system is partitioned to two modules, namely, one that provides input stimuli (driver) and another that reacts based on that input (main). Therefore, the resulting actor network consists of two actors representing two SMs. Example results of the experimentations conducted on different benchmarks is given in Table 2. We list the example alongside, the time required to generate code for it on a standard PC, the effective number of LoC, the number of canonicalized boolean expressions during EFSM generation, and the total number of states in the EFSM. Number of canonicalized boolean expressions is listed since its the most expensive operation in terms of required computation.

Table 2: Experimental results

Model	Time	LoC	Boolexps	States
Heron Sqr Root	0.1 s	462	11	3
Cruise Control	1 s	1266	335	11
SHA (Basic)	3 s	5076	553	60
SHA (Optimized)	12 s	38012	7301	163

The tested benchmarks were Heron (Newton) square root algorithm, a simple car cruise control model, and an implementation of the SHA2-256 hashing algorithm. Thanks to the robust support for bitvectors in SystemC, we have not had any problem in synthesizing the SHA2 model, although it utilizes some sophisticated bitvector operations. SHA2 model was implemented in two versions, a basic version that maps the standard directly and uses all 64 scheduling values W_t, and an optimized version that starts hashing a block as soon as a hash word has been received. It uses the last 16 scheduling values only which is typical for commercial cores. The optimized version has more states since it requires more control. The generated code is then compiled using g++ and linked against SysteMoC, SystemC and some Boost libraries. Note that generated LoC depends on the number of states in the EFSM and the number of DGAs attached to each state. Comparing basic and optimized SHA2 implementations reveals that LoC figure grows rapidly due to increased number of DGAs per state. That issue should be handled by a smarter scheme for DGA code sharing between different firing actions. We leave that for a future work.

5. Related work

Brandt et al. [BGS10] considered the synthesis of a non-partitioned synchronous system represented by guarded actions to SystemC directly. In contrast, we discussed the synthesis of distributed synchronous specifications to SysteMoC. Halbwachs et al. [HM06] discussed a method

for simulating asynchronous tasks in Lustre. Their implementation was based on SCADE, which is the commercial Lustre tool. This work is concerned instead with the refinement and simulation of synchronous models using open source libraries. Generally, there is not much work in the area of synchronous language synthesis for simulation purposes. We note that it is important to preserve the control states of SMs for better analyzability. Hence, system partitioning schemes that convert a synchronous model directly to a asynchronous one, e. g. Baudisch et al. [BBS10], are not suitable for elastic network representation.

6. Conclusion

We discussed in this work a semantics-preserving synthesis procedure for elastic networks. That enables a large body of theoretical work on synchronous system distribution, e. g. endo/isochrony, to be exercised and improved. It also makes it possible to gradually refine a synchronous model from perfect synchrony to a completely asynchronous actor network model. Additionally, analysis developed for actor networks like [FZK+10] can be utilized in the refined synchronous model. We think that our synthesis procedure is a step towards combining both MoCs in a formal model-based design flow that can potentially have better analyzability and expressiveness. For example, designers can benefit from the composability of synchronous models and combine it with the mapping and scheduling properties of data-flow actor models to generate safety-critical multi-processor software.

Acknowledgment

We would like to thank Joachim Falk of the HW/SW Co-Design group at Erlangen-Nuremberg University for providing us with SysteMoC 1.0 beta version. This version supports clock synchronization and it is still not publicly released to the time of this writing.

References

[BBS10] Baudisch, D., J. Brandt, and K. Schneider: *Multithreaded code from synchronous programs: Extracting independent threads for OpenMP*. In *Design, Automation and Test in Europe (DATE)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.

[BCE+03] Benveniste, A., P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone: *The synchronous languages twelve years later*. Proceedings of the IEEE, 91(1):64–83, 2003.

[BCL00] Benveniste, A., B. Caillaud, and P. Le Guernic: *Compositionality in dataflow synchronous languages: Specification and distributed code generation*. Information and Computation, 163(1):125–171, 2000.

[Ben13] Ben Khadra, M.A.: *A model-based approach to sychronous elastic systems*. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany, March 2013. Master.

[Ber98] Berry, G.: *The foundations of Esterel*. In Plotkin, G., C. Stirling, and M. Tofte (editors): *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 1998.

[BGS10] Brandt, J., M. Gemünde, and K. Schneider: *From synchronous guarded actions to SystemC*. In Dietrich, M. (editor): *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.

[CCKT09] Carmona, J., J. Cortadella, M. Kishinevsky, and A. Taubin: *Elastic circuits*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD), 28(10):1437–1455, October 2009.

[EJL+03] Eker, J., J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong: *Taming heterogeneity – the Ptolemy approach*. Proceedings of the IEEE, 91(1):127–144, January 2003.

[FHT06] Falk, J., C. Haubelt, and J. Teich: *Efficient representation and simulation of model-based designs*. In *Forum on Specification and Design Languages (FDL)*, pages 129–135, Darmstadt, Germany, 2006. Electronic Chips and Systems Design Initiative (ECSI).

[FZK+10] Falk, J., C. Zebelein, J. Keinert, C. Haubelt, J. Teich, and S.S. Bhattacharyya: *Analysis of SystemC actor networks for efficient synthesis*. ACM Transactions on Embedded Computing Systems (TECS), 10(2):18:1–18:34, December 2010.

[GBS13] Gemünde, M., J. Brandt, and K. Schneider: *Clock refinement in imperative synchronous languages*. EURASIP Journal on Embedded Systems, 3:1–21, August 2013. http://jes.eurasipjournals.com/content/2013/1/3.

[Gir05] Girault, A.: *A survey of automatic distribution method for synchronous programs*. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–20, Edinburgh, Scotland, UK, 2005. unpublished workshop proceedings.

[HCRP91] Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud: *The synchronous dataflow programming language LUSTRE*. Proceedings of the IEEE, 79(9):1305–1320, September 1991.

[HM06] Halbwachs, N. and L. Mandel: *Simulation and verification of asynchronous systems by means of a synchronous model*. In *Application of Concurrency to System Design (ACSD)*, pages 3–14, Turku, Finland, 2006. IEEE Computer Society.

[Lee09] Lee, E.A.: *Computing needs time*. Communications of the ACM (CACM), 52(5):70–79, May 2009.

[LM87] Lee, E.A. and D.G. Messerschmitt: *Synchronous data flow*. Proceedings of the IEEE, 75(9):1235–1245, September 1987.

[LS03] Logothetis, G. and K. Schneider: *Exact high level WCET analysis of synchronous programs by symbolic state space exploration*. In *Design, Automation and Test in Europe (DATE)*, pages 10196–10203, Munich, Germany, 2003. IEEE Computer Society.

[LTL03] Le Guernic, P., J. P. Talpin, and J. C. Le Lann: *Polychrony for system design*. Journal of Circuits, Systems, and Computers (JCSC), 12(3):261–304, June 2003.

[Sch09] Schneider, K.: *The synchronous programming language Quartz*. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

Semi-Symbolische Analyse eines ΣΔ-Modulators

Kristin Krüger, Carna Radojicic, Christoph Grimm

TU Kaiserslautern
AG Design of Cyber-Physical Systems

Zusammenfassung: *Der Beitrag befasst sich mit der semi-symbolischen Simulation und Analyse eines Sigma-Delta-Modulators. Ziel der semi-symbolischen Analyse ist, Zusammenhänge zwischen Schaltungsparametern und Performancemaßen aufzuzeigen. Im Unterschied zur rein symbolischen Analyse werden nur lineare Zusammenhänge zu Schaltungsparametern symbolisch betrachtet, was für das Verständnis und die Analyse von Schaltungen in der Regel ausreichend ist.*

Im Beitrag demonstrieren wir am Beispiel eines Sigma-Delta Modulators, wie die Grenze zwischen diskreter und kontinuierlicher Modellierung/Simulation/Analyse überschritten werden kann. Wir zeigen auf, wie (lineare) symbolische Zusammenhänge zwischen Technologieeigenschaften und Performancemaßen z.B. im Frequenzbereich sowie Worst-Case Grenzen der Performancemaße bestimmt werden können.

1. Einleitung

Eingebettete Systeme bestehen heute aus eng miteinander vernetzter digitaler Hardware, analoger Peripherie und komplexen Softwaresystemen. Durch die so entstehende Komplexität und Heterogenität entsteht eine „Verifikationslücke“; derartige Systeme können nicht mehr mit ausreichender Coverage verifiziert werden.

In der Anwendung in der Industrie wird die Coverage durch geschickt gesteuerte Multi-Run Simulationen (z.B. Monte-Carlo [1], Importance-Sampling [2], Design-of-Experiments [3]) und systematisches Vorgehen bei der Verifikation (Verifikationsplanung, UVM) erhöht. Der Vorteil dieser Ansätze ist die breite und allgemeine Anwendbarkeit der zugrundeliegenden Simulationsmethoden. Leider wird durch diese Ansätze die Verifikationslücke nicht vollständig geschlossen.

In der aktuellen Forschung bieten formale Methoden verschiedene Alternativen, die Verifikationslücke zu schließen. Hierzu gehören Model Checking von hybriden Systemen [4], Equivalence Checking von analogen Schaltungen [5] oder symbolische Analyse analoger Schaltungen (z.B. [6]). Diese Ansätze ermöglichen eine hohe Coverage, sie sind aber weit davon entfernt, für komplexe heterogene Systeme anwendbar zu sein.

Wünschenswert wäre eine Kombination der breiten Anwendbarkeit der Simulation mit der hohen Coverage und den guten Analysemethoden von formalen und symbolischen Verfahren. R. Bryant hat in [7] bereits gezeigt, dass mit einem Simulator eine Verifikation mit hoher Coverage möglich ist. Notwendig ist hierzu die Möglichkeit, neben numerischen Werten auch mit Mengen von Werten zu simulieren. Bryant verwendet in der Logiksimulation den Wert „X“ zur Darstellung der Werte „0“ oder „1“. Ein analoges Vorgehen für gemischt analog/digitale, signalverarbeitende Systeme ist in [8, 9] vorgestellt worden: Die semi-symbolische Simulation analog/digitaler Systeme. Diese ist erfolgreich für die Analyse signalverarbeitender, linearer Systeme mit einfachen Nichtlinearitäten angewendet worden. Eine Herausforderung sind aber nach wie vor diskontinuierliche Funktionen wie z.B. Komparatoren (Vergleich) und Begrenzung bzw. Sättigung.

In diesem Beitrag wird ein einfacher Ansatz zur Modellierung und symbolischen Simulation auch von Komparatoren und Begrenzung am Beispiel eines Sigma-Delta Modulators beschrieben und demonstriert. Bereits die Modellierung und Simulation von Sigma-Delta Modulatoren ist eine Herausforderung. Diese werden zunächst knapp skizziert. In Abschnitt 2 wird die semi-symbolische Simulation eingeführt. Abschnitt 3 zeigt, wie diese zur Modellierung und Analyse eines ΣΔ-Modulators verwendet werden kann. Die Herausforderung ist dabei das Überschreiten der Grenze zwischen digitaler und analoger Modellierung (Komparatoren, Begrenzungseffekte, etc.). In Abschnitt 4 wird gezeigt, wie daraus unter Beibehaltung der symbolischen Informationen die Performancemaße im Frequenzbereich bestimmt werden können. Abschnitt 5 schließt mit einer Diskussion und Ausblick auf weitere Arbeiten.

1.1. Herausforderungen bei der Analyse von ΣΔ-Modulation

Die ΣΔ-Modulation ist eine Form der Digital-Analog-Wandlung, bei der nur ein einfacher 1-Bit Wandler verwendet wird. Die Genauigkeit wird im zeitlichen Mittel durch Messen des mittleren Ausgabewerts (durch einen Integrator) und entsprechender Variation der erzeugten Bitfolgen beinahe beliebig erhöht. Die Analyse eines ΣΔ-Modulators ist aus mehreren Gründen sehr komplex und charakteristisch auch für weit komplexere Systeme:

- Die Performance bzw. Genauigkeit ergibt sich wegen der statistischen Funktionsweise nur durch Beobachtung bzw. Simulation über einen sehr langen Zeitraum und nach sehr vielen Iterationen über die Schnittstellen zwischen diskreter und kontinuierlicher bzw. kontinuierlicher und diskreter Modellierung. Dies ist eine große Herausforderung für alle bekannten Verfahren einschließlich Modellierung und Simulation.
- Zur Verbesserung der Genauigkeit werden Verfahren zur Rauschformung („Noise Shaping“) angewendet. Quantisierungsrauschen wird bei kleinen Frequenzen reduziert und bei größeren Frequenzen erhöht; dadurch besitzen niederfrequente Signalanteile einen hohen Rauschabstand. Die höheren Frequenzen werden aufgrund des schlechten Signal-Rauschverhältnisses mit einem Filter entfernt. Dies ist eine Herausforderung vor allem für formale Methoden, wo Eigenschaften im Frequenzbereich bislang nicht üblich sind.
- Zum Noise-Shaping werden digitale Filter (oder SC-Schaltungen) höherer Ordnung verwendet. Allerdings können Schwingungen und Instabilitäten in der Schaltung mit steigender Ordnung auftreten.
- Die Genauigkeit hängt ganz wesentlich auch von technologischen und schaltungstechnischen Eigenschaften ab wie zum Beispiel dem Matching von Kapazitäten oder der Begrenzung von Integratoren. Formale Methoden ermöglichen es aber bislang nicht, derart technologienahe Eigenschaften zu berücksichtigen.

In Kapitel 3 wird ein ΣΔ-Modulator 3. Ordnung mit SC-Filter mit Hilfe semi-symbolischer Simulation analysiert und die Eigenschaften im Frequenzbereich symbolisch analysiert und auf Technologieparameter zurückgeführt. Zunächst wird aber der verwendete mathematische Ansatz eingeführt: Das (semi-symbolische) Rechnen mit Bereichen auf der Basis der affinen Arithmetik.

2. Semi-Symbolische Simulation mit Affiner Arithmetik

Semi-Symbolische Analyse kann auf Schaltungs- (z.B. [10]) oder Blockdiagrammebene (vgl. [11]) erfolgen. Für einen bestimmten Testvektor gibt sie eine symbolische Repräsentation aller möglichen Ausgangssignale zurück. Die symbolische Repräsentation umfasst sowohl ideale Werte wie auch den Einfluss von Unsicherheiten und Parametern auf das Ausgangssignal.

Die Affine Arithmetik [12] nutzt Wertebereiche, um Abweichungen von einem Idealwert x_0 darzustellen. Diese Abweichungen, deren Ursachen beispielsweise Alterung, Rauschen oder Offsets sein können, werden durch das Abweichungssymbol ε_i (noise/ deviation symbol) beschrieben, wobei i für eine der miteinander unkorrelierten Ursachen für die Abweichung steht. ε_i ist ein unbekannter Wert, der im Intervall [– 1, 1] liegt und mit der partiellen Abweichung x_i (partial deviation) skaliert wird. Ein affiner Ausdruck sieht aus wie folgt:

$$\tilde{x} = x_0 + \sum_{i \in N_{\tilde{x}}} x_i \varepsilon_i \qquad \varepsilon_i \in [-1, 1] \,.$$

Der Zahlenbereich $N_{\tilde{x}}$ umfasst alle natürlichen Zahlen, die die Abweichungen $x_i \varepsilon_i$ im affinen Ausdruck $\tilde{x}$ darstellen. Lineare Operationen werden wie folgt definiert:

$$\tilde{x} \pm \tilde{y} = (x_0 \pm y_0) + \sum_{i \in N_{\tilde{x}}} \varepsilon_i (x_i \pm y_i)$$

sowie

$$c\tilde{x} = cx_0 + \sum_{i \in N_{\tilde{x}}} c\varepsilon_i x_i \,.$$

Durch diese symbolische Repräsentation wird für lineare Operationen eine Überapproximation vermieden. Nichtlineare Operationen können ebenfalls definiert werden, allerdings verursachen sie Überapproximation. Dies ist akzeptabel für schwache Nichtlinearitäten, für starke Nichtlinearitäten jedoch meist nicht. Zur semi-symbolischen Simulation wird in einem Simulator der Datentyp von elektrischen oder abstrakten Größen (z.B. „*double x*“) durch einen Datentyp Affine Form (z.B. „*AAF x*“) ersetzt und die arithmetischen Operationen in der Simulation durch die wie oben definierten Operationen ersetzt.

In Abbildung 1 ist die Ausgabe der symbolischen Transientensimulation dargestellt. Die grau hinterlegten Bereiche markieren die Ober- und Untergrenze für den Bereich, in dem sich der affine Ausdruck $\tilde{x}$ aufgrund aller Abweichungen ε_i bewegen kann.

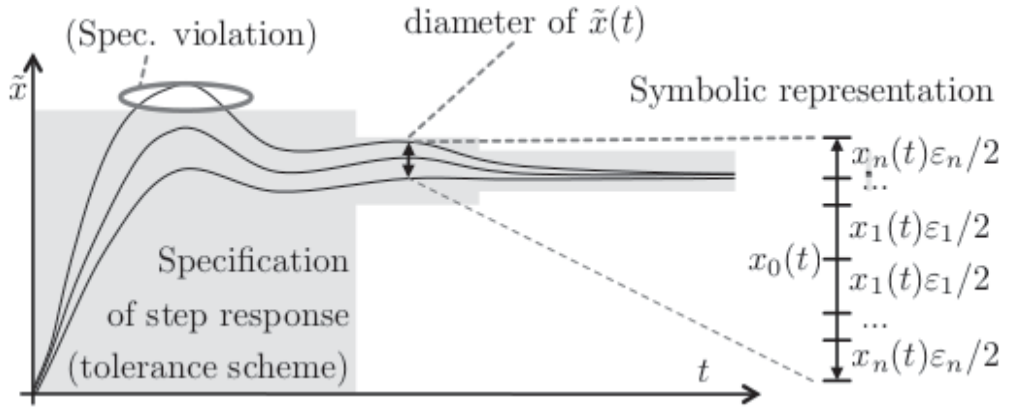


Abbildung 1: Visualisierung der Semi-Symbolischen Analyse

3. Semi-Symbolische Modellierung eines ΣΔ - Modulators

In diesem Abschnitt wird die Modellierung eines Sigma-Delta-Modulators dritter Ordnung mit Switched-Capacitor (SC)-Technik beschrieben. Anhand einer (semi-symbolischen) Simulation soll dann das Verhalten des Sigma-Delta-Modulators bestimmt werden. Dabei soll insbesondere der Einfluss von Störungen wie dem Rauschen mithilfe der symbolischen Darstellung analysierbar und verlässlich in Worst-case Schranken bestimmt werden.

Für die semi-symbolische Simulation wurde SystemC AMS als Grundlage genutzt. Die Module wurden mithilfe von SystemC AMS TDF (Timed Data Flow) implementiert. Dabei wurde der zur Modellierung analoger Größen verwendete Datentyp „double“ durch einen abstrakten Datentyp (ADE) „AAF“ ersetzt, der wie in Abschnitt 2 beschrieben einen numerischen „central value“ sowie eine symbolische Linearkombination von Abweichungen enthält. Alle Operationen wurden durch affine Operationen ersetzt; nichtlineare Operationen durch sichere Einschlüsse.

Um das Prinzip zu zeigen, werden nur einfache Störungen betrachtet, wie beispielsweise Rauschen (für Noise Shaping) und die Prozessvariationen der Kondensatoren, deren Auswirkungen ebenso wie die Möglichkeiten, jene zu mindern, bekannt sind. Komplexere technologische Zusammenhänge können analog modelliert werden; Details sind in [2] beschrieben. Hier wird insbesondere die Modellierung bislang nicht möglicher nicht-kontinuierlicher Funktionen (Komparator) und Effekte (Begrenzung) beschrieben.

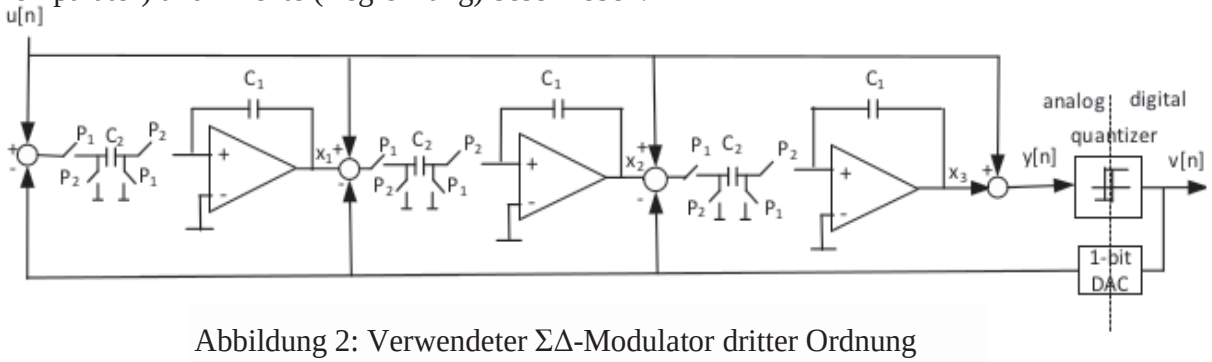


Abbildung 2: Verwendeter ΣΔ-Modulator dritter Ordnung

3.1. Modellierung des Komparators

Der Komparator wird modelliert, indem sein Eingangssignal mit einem bestimmten Schwellenwert verglichen wird. Dabei muss beachtet werden, dass durch AAF das Eingangssignal sozusagen als „Wertebereich“ vorliegt. Liegt der Wertebereich des Signals nun teilweise über und teilweise unter dem Schwellwert, wird das Systemverhalten mithilfe eines Splitter-Modules in zwei Fälle aufgeteilt und die semi-symbolische Simulation kann beide Möglichkeiten abhandeln. Diese Unstetigkeiten können anschließend mit einem Merge-Modul zusammengefasst und zu einem kontinuierlichen Verhalten zurückgeführt werden. Der Komparator beinhaltet ein Splitter-Modul in Form eines TDF-Blocks.

Das Splitter-Modul betrachtet zuerst den Teil des Signals für den Fall, dass das Signal über dem Schwellwert liegt. Dafür verwenden wir die Bezeichnung „1“. Für den Fall, dass das Signal unter dem Schwellwert liegt, wird analog die Bezeichnung „0“ verwendet.

Nun wird ein zusätzlicher Zeitschritt eingefügt, um beide Werte des Komparators (0 und 1) innerhalb eines Simulationslaufs betrachten zu können. Hierbei muss beachtet werden, dass sich nur der Wert des Signals am Ausgang ändern darf; das Eingangssignal aus dem vorhergehenden Zeitschritt muss demnach beibehalten werden. Dafür wurde das Modul one2many implementiert und am Eingang des Modulators eingefügt.

Außerdem muss berücksichtigt werden, dass sich durch das Einfügen eines weiteren Samples die Abtastrate erhöht. Deshalb wurde das Splitter-Modul als SystemC AMS Dynamic Timed Data Flow (DTDF) [13] implementiert, wodurch die Schrittbreite während der Simulation passend geändert werden kann.

3.2. Modellierung der Integratoren

Integratoren wurden, wie in Abbildung 2 dargestellt, in Schalter-Kondensator-(SC)-Technik implementiert. Das verwendete, zeitdiskrete Modell dazu besteht aus einem Verzögerer, einem Modul, das die Verstärkung des Integrators darstellt, und einem weiteren Modul, das die Sättigung modelliert. Als Eingangssignale erhalten die Integratoren die diskontinuierliche Ausgabe des Splitter-Moduls des Komparators, weswegen sie in der Lage sein müssen, ihre simulierte Zeit und ihre Zustände zurückzusetzen. Am Ausgang des Integrators befindet sich zudem ein Merge-Modul, das mithilfe von AAF das Signal in eine kontinuierliche (stetige) Form wie im Folgenden beschrieben zurückwandelt.

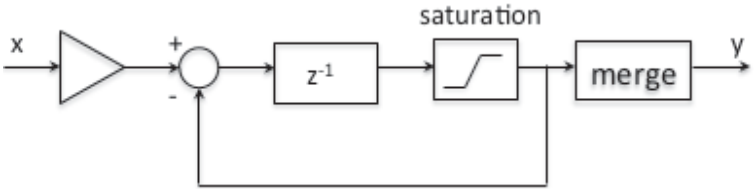


Abbildung 3: Integrator

Im Folgenden stellen $\tilde{y}_H$ und $\tilde{y}_L$ die beiden Fälle für einen affinen Term $\tilde{y}$ (ein Sample eines Signals) dar, bei dem $\tilde{y}$ über bzw. unter dem Schwellwert v_{TH} liegt. Da sie ebenfalls als affine Terme modelliert werden, gilt:

$$\tilde{y}_H = y_{H0} + \sum_{i \in N_{\tilde{y}_H}} \varepsilon_i y_{Hi} \qquad \varepsilon_i \in [-1, 1]$$

sowie

$$\tilde{y}_L = y_{L0} + \sum_{i \in N_{\tilde{y}_L}} \varepsilon_i y_{Li}\, \varepsilon_i \qquad \varepsilon_i \in [-1, 1]\,.$$

Die Grundidee des Merge-Moduls besteht darin, $\tilde{y}_H$ und $\tilde{y}_L$ als Grenzen eines Intervalls zu behandeln. Daraus ergibt sich:

$$[\tilde{y}_L, \tilde{y}_H] = \tilde{y}_c + \varepsilon_{N_{\tilde{y}}+1}\, \tilde{y}_{diam} \qquad \varepsilon_{N_{\tilde{y}}+1}\, \epsilon\, [-1, 1]\,, \qquad (1)$$

wobei $\tilde{y}_c$ den „center value“, $\tilde{y}_{diam}$ den Durchmesser des Intervalls $[\tilde{y}_L, \tilde{y}_H]$ darstellt und $N_{\tilde{y}}$ den vereinigten Zahlenbereich von $N_{\tilde{y}_L}$ und $N_{\tilde{y}_H}$. Der „center value“ wird errechnet aus:

$$\tilde{y}_C = \frac{(\tilde{y}_H + \tilde{y}_L)}{2} = \frac{y_{H0} + y_{L0}}{2} + \sum_{i \in N_{\tilde{y}}} \varepsilon_i \left(\frac{y_{Hi} + y_{Li}}{2}\right).$$

Der Durchmesser errechnet sich aus:

$$\tilde{y}_{diam} = \frac{|\tilde{y}_H - \tilde{y}_L|}{2} = \frac{1}{2}\left| y_{H0} - y_{L0} + \textstyle\sum_{i \in N_{\tilde{y}}} \varepsilon_i (y_{Hi} - y_{Li}) \right|, \text{ wobei } \tilde{y}_{diam} \subseteq [0, l].$$

Durch die Multiplikation von $\varepsilon_{N_{\tilde{y}}+1}$ und $\tilde{y}_{diam}$ in (1) ergeben sich neben affinen Termen auch quadratische. Um ein rein affines Ergebnis zu erhalten, wird $\tilde{y}_{diam}$ überapproximiert. Das Maximum *l* ergibt sich wie folgt:

$$l = \max\left(\frac{|\tilde{y}_H - \tilde{y}_L|}{2}\right) = \frac{1}{2}(|y_{H0} - y_{L0}| + \sum_{i \in N_{\tilde{y}}} |y_{Hi} - y_{Li}|, \text{ da gilt}$$

, da gilt

$$|\tilde{y}_H - \tilde{y}_L| = \left|y_{H0} - y_{L0} + \sum_{i \in N_{\tilde{y}}} \varepsilon_i (y_{Hi} - y_{Li})\right| \leq |y_{H0} - y_{L0}| + \sum_{i \in N_{\tilde{y}}} |y_{Hi} - y_{Li}|.$$

Es macht hierbei keinen Unterschied, ob $\tilde{y}_H$ und $\tilde{y}_L$ als Intervallgrenzen vertauscht werden.

Die Abweichung $\varepsilon_{N_{\tilde{y}}+1}$ in (1) wurde hinzugefügt, um den Einschluss des vollständigen Ergebnisraumes zu garantieren. Der tatsächliche Wert von $\varepsilon_{N_{\tilde{y}}+1}$ hängt stark von den Abweichungen ab, die Unsicherheiten modellieren. Diese Abhängigkeit wird für jeden Zeitschritt betrachtet. Besitzen der aktuelle und vorhergehende Zeitschritt dieselben Abweichungssymbole, so wird dasselbe Abweichungssymbol zur Modellierung der Überapproximation genutzt. Bestehen hingegen keine Abhängigkeiten, wird ein neues Abweichungssymbol eingeführt.

Die Verstärkung des Integrators entspricht dem Verhältnis der Kapazitäten C_2/C_1 zueinander. Die Werte der Kapazitäten sind in Tabelle 1 gegeben, ihre Toleranzen werden über Abweichungen ε_i beschrieben. Es wird angenommen, dass 30% der Toleranz von C_2 mit C_1 korreliert ist, daher wurden dieselben Symbole verwendet. Für den unkorrelierten Anteil wurden entsprechend unterschiedliche ε_i benutzt. Durch die partielle Korrelation der Kondensatoren werden Prozessvariationen zum Großteil behoben.

Tabelle 1: Parameter der Integratoren

Integrator	$\boldsymbol{C_2}$ $[\boldsymbol{pF}]$	$\boldsymbol{C_1}$ $[\boldsymbol{pF}]$
1. Integrator	0.07333 + $\varepsilon_1$30% + $\varepsilon_2$5%	1 + $\varepsilon_1$30% + $\varepsilon_3$5%
2. Integrator	0.2881 + $\varepsilon_1$30% + $\varepsilon_2$5%	1 + $\varepsilon_1$30% + $\varepsilon_3$5%
3. Integrator	0.7997 + $\varepsilon_1$30% + $\varepsilon_2$5%	1 + $\varepsilon_1$30% + $\varepsilon_3$5%

3.3. Modellierung der Begrenzung/Sättigung des Integrators

Da wir mit Affiner Arithmetik arbeiten, kann die Begrenzung nicht durch einen Vergleich mit einfachen Werten dargestellt werden. Stattdessen wird sie mit dem Intervall $[V_{satlow}, V_{sathigh}]$ beschrieben. Wenn nun überprüft werden soll, ob ein Signal $\tilde{x}$, das ebenfalls als affiner Term in der bereits vorgestellten Form vorliegt, den Bereich der Begrenzung überschreitet, müssen vier Fälle unterschieden werden:

1) $\forall \varepsilon_i \in [-1,1], i \in N_{\tilde{x}} : \tilde{x}$ liegt im Intervall $[V_{satlow}, V_{sathigh}]$
2) $\forall \varepsilon_i \in [-1,1], i \in N_{\tilde{x}} : \tilde{x} > V_{sathigh}$
3) $\forall \varepsilon_i \in [-1,1], i \in N_{\tilde{x}} : \tilde{x} < V_{satlow}$

4) ein Teil von $\tilde{x}$ liegt unter V_{satlow} und/ oder $V_{sathigh}$ und umgekehrt

Im ersten Fall muss nichts unternommen werden, da keine der Grenzen überschritten wird. In Fall 2 und 3 wird das Signal $\tilde{x}$ lediglich auf den entsprechenden Wert gesättigt (in Fall 2 auf $V_{sathigh}$, in Fall 3 auf V_{satlow}). Bei Fall 4 hingegen müssen die Grenzen von $\tilde{x}$ entsprechend anpasst werden. Überschreitet das Signal beispielsweise beide Grenzen, muss $\tilde{x}$ so modifiziert werden, dass seine neue Grenzen dem Intervall $[V_{satlow}, V_{sathigh}]$ entsprechen. Dafür wird ein neues Modell eingeführt:

$$\tilde{x}^{sat} = x_0 + \sum_{i \in N_{\tilde{x}}} x_i \varepsilon_i + \left(\frac{V_{satlow} + V_{sathigh}}{2} - x_0 \right) + \varepsilon_{N_{\tilde{x}}+1} x_{N_{\tilde{x}}+1}$$

wobei

$$x_{N_{\tilde{x}}+1} = \frac{\left(V_{sathigh} - \tilde{x}_{high}\right) - \left(V_{satlow} - \tilde{x}_{low}\right)}{2}.$$

Die Werte $\tilde{x}_{high}$ und $\tilde{x}_{low}$ stehen für die obere und untere Grenze des Wertebereichs von $\tilde{x}$:

$$\tilde{x}_{high} = x_0 + \sum_{i \in N_{\tilde{x}}} |x_i|$$

$$\tilde{x}_{low} = x_0 - \sum_{i \in N_{\tilde{x}}} |x_i| .$$

4. Analyse im Frequenzbereich

Zur Analyse im Frequenzbereich ist es erforderlich, Abweichungssymbole für einzelne Rauschquellen (Quantisierungsrauschen in diesem Fall) einzuführen. Hierzu wird die im Folgenden beschriebene Methode verwendet.

Bei der Quantisierung eines analogen Signals entstehen Quantisierungsfehler, sobald das Signal nicht der Quantisierungsstufe entspricht und somit gerundet werden muss. Die Quantisierungsstufen eines m-Bit Quantisierers können errechnet werden über:

$$Q = 1/(2^m - 1) .$$

Das quantisierte Signal y_Q ergibt sich durch Runden des Eingangssignals y[n]. Der Quantisierungsfehler ergibt sich somit zu:

$$e[n] = y_Q[n] - y[n] \in \left(-\frac{Q}{2}, \frac{Q}{2}\right].$$

Wenn das Eingangssignal y[n] gleichverteilt ist, ist die Wahrscheinlichkeitsverteilungs-funktion des Quantisierungsfehlers ebenfalls gleichverteilt und hat den Wert 1/Q im Intervall $(-Q/2, Q/2]$. Der Mittelwert des Fehlers e beträgt 0:

$$m_e = \int_{-\infty}^{\infty} e p_e(e) de = 0$$

und die Varianz ergibt sich zu:

$$\sigma_e^2 = \int_{-\infty}^{\infty} e^2 p_e(e) de = \frac{Q^2}{12}.$$

Der Quantisierungsfehler kann somit als gaussverteilte Zufallsvariable behandelt werden, die einen Mittelwert von 0 und eine Standardabweichung von $\sqrt{\sigma_e^2}$ besitzt. Da die Zufallsvariablen e[n] unkorreliert sind, müssen zeitlich miteinander unkorrelierte Abweichungssymbole verwendet werden. Sie modellieren die dynamische Unsicherheit eines Signals. Um zwischen verschiedenen Ursachen für die Unsicherheiten unterscheiden zu können, wird ε für die Bereichsdarstellung und γ für die Gauss-Verteilung verwendet. Der Quantisierungsfehler ergibt sich nun zu:

$$\tilde{y}_Q = \tilde{y} + e[n] = \tilde{y} + \gamma[n]\sigma_e \, .$$

Sigma-Delta-Modulatoren haben ein Filter, um das Quantisierungsrauschen aus dem Signalband zu schieben. Damit ergibt sich das Ausgangssignal des Quantisierers zu:

$$\tilde{y}_e[n] = \sum_{i=0}^{m} e[n-i]a_i - \sum_{i=1}^{m} \tilde{y}_e[n-i]b_i \, ,$$

wobei m für die Ordnung des Sigma-Delta-Modulators steht. Die Filterkoeffizienten a_i und b_i ergeben sich für den verwendeten Sigma-Delta-Modulator 3. Ordnung zu:

$$a_0 = 1.0, \qquad a_1 = -3.0, \qquad a_2 = 3.0, \qquad a_3 = -1.0$$

$$b_0 = 1.0, \qquad b_1 = C_2^{(3)}/C_1^{(3)} - 3.0$$

$$b_2 = 3.0 + \frac{C_2^{(2)}C_2^{(3)}}{C_1^{(2)}C_1^{(3)}} - 2 * \frac{C_2^{(3)}}{C_1^{(3)}} \, , \qquad b_3 = -1.0 + \frac{C_2^{(3)}}{C_1^{(3)}} + \frac{C_2^{(1)}C_2^{(2)}C_2^{(3)}}{C_1^{(1)}C_1^{(2)}C_1^{(3)}} - \frac{C_2^{(2)}C_2^{(3)}}{C_1^{(2)}C_1^{(3)}}$$

4.1. Ergebnisse

Die Ausgabe des ΣΔ – Modulators wurde mithilfe der Fast Fourier Transformation (FFT) mit einer Abtastrate von $T_S = 0.125\mu$s an $N = 8192$ Punkten berechnet. Als Eingangssignal diente ein Sinussignal mit einer Frequenz $f = 3.9$kHz und einer Amplitude von 600mV. Dazu wurde die erweiterte Version der FFT [14] genutzt, um mit affinen Termen umgehen zu können:

$$\tilde{X}(k) = \sum_{n=0}^{N-1} x_0[n] e^{-j\left(\frac{2\pi k}{N}\right)n} + \sum_{n=0}^{N-1} \sum_{i \in N_{\tilde{x}}} \varepsilon_i x_i[n] e^{-j\left(\frac{2\pi k}{N}\right)n} \, .$$

Da 8192 Samples des Modulators benötigt wurden, wurde das System für 8192∗0.125μs = 1024 μs simuliert. Die Simulation einschließlich der Berechnung der FFT benötigte 110s. Die Ergebnisse sind im Folgenden dargestellt.

Tabelle 2: Ergebniss

	Center value	**Partielle Abweichungen, [Minimum, Maximum]**
$C_2^{(1)}[pF]$	0.07333	0.07333 + $\varepsilon_1$30% + $\varepsilon_2$5% = [0.047645, 0.09899]
$C_1^{(1)}[pF]$	1	1 + $\varepsilon_1$30% + $\varepsilon_3$5% = [0.65, 1.35]
$C_2^{(2)}[pF]$	0.2881	0.2881 + $\varepsilon_1$30% + $\varepsilon_2$5% = [0.187265, 0.3889]
$C_1^{(2)}[pF]$	1	1 + $\varepsilon_1$30% + $\varepsilon_3$5% = [0.65, 1.35]
$C_2^{(3)}[pF]$	0.7997	0.7997 + $\varepsilon_1$30% + $\varepsilon_2$5% = [0.519805, 1.0796]
$C_1^{(3)}[pF]$	1	1 + $\varepsilon_1$30% + $\varepsilon_3$5% = [0.65, 1.35]
Ausgabe FFT ($f_{in} = 3.9$kHz)	0.637146	0.604032 + 0.0ε_1 + 0.013476ε_2 – 0.013476ε_3… = [0.0420469, 1.16659]

Die Ausgabe der FFT ist, wie in Tabelle 2 ersichtlich, ebenfalls in semi-symbolischer Form. Sie berücksichtigt nicht nur die Abweichungssymbole ε_1, ε_2 und ε_3 (vgl. Tabelle 1), sondern auch alle anderen, um einen sicheren Einschluss zu garantieren. Die Ursachen für die Abweichungssymbole liegen nicht nur bei der nicht-linearen Berechnung des Verhältnisses C_2/C_1, sondern auch bei dem Merge-Verfahren sowie bei der Modellierung der Sättigung und des Rauschens. Auffallend ist, dass sich der korrelierte Anteil der Abweichungen der Kondensatoren (ε_1) aufhebt.

Abbildung 4 zeigt die Auswirkungen der Abweichungen für alle Abweichungssymbole. In Schwarz ist der „center value“ dargestellt, die rote bzw. blaue Linie geben das Minimum bzw. Maximum an. Die höchste Spitze im Frequenzspektrum befindet sich bei 3.9 kHz, da das Eingangssignal diese Frequenz besitzt.

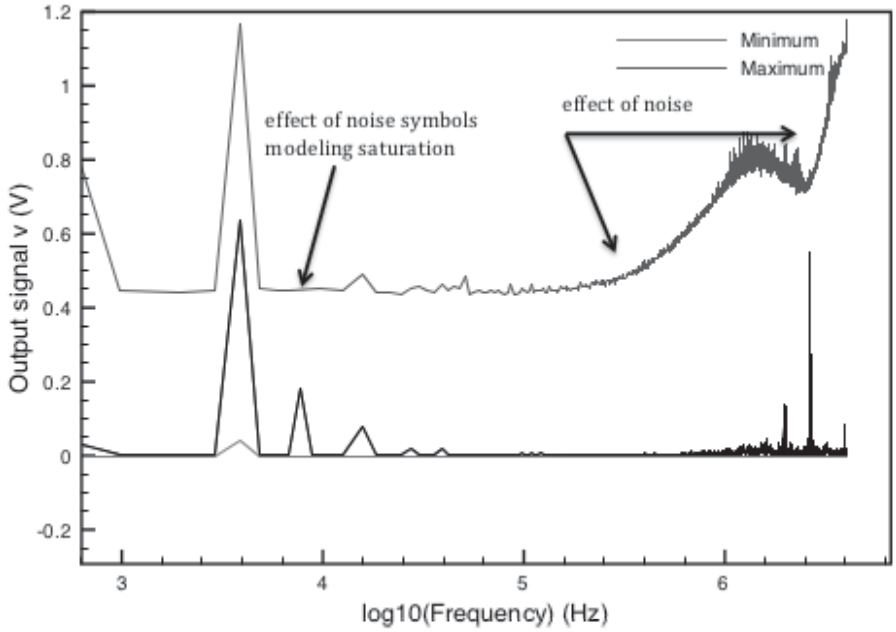


Abbildung 4: Ausgabe der FFT

5. Ausblick

In dem Beitrag konnte gezeigt werden, dass es möglich ist, die Grenze zwischen diskreter und kontinuierlicher Modellierung auch für symbolische und formale Verfahren zu überschreiten. Die Kombination symbolischer und numerischer Ansätze ermöglicht hierbei insbesondere auch die Anwendung auf nicht-triviale Systeme wie den ΣΔ – Modulator. Allerdings erfordert der Ansatz noch sehr viel weitere Forschung und Entwicklung, da er sich in vielen wichtigen Details bis hin zur Verifikationsmethodik grundlegend von bekannten und etablierten Verfahren unterscheidet. Insbesondere der Modellierungsprozess ist noch zu aufwändig und erfordert gründliche Überlegungen:

1. Die Erfahrung, weitere Experimente und theoretische Überlegungen haben gezeigt, dass der Ansatz für stabile und robuste Systeme mit Abweichungen bis zu ca. 20% brauchbar ist.
2. Der Ansatz ist für Oszillatoren und potentiell instabile Systeme unbrauchbar, da dann die Überapproximation exponentiell mit der Zeit wächst; umgekehrt kann aber die Stabilität von z.B. IIR-Filtern nachgewiesen werden.
3. Die Erstellung von Modellen ist, wie am Beispiel gezeigt, sehr aufwändig. Dies kann aber durch systematische Einbettung von vorgefertigten Modellen und – in zukünftigen Arbeiten – durch Formulierung eines allgemeinen Ansatzes erreicht werden.

4. Der Ansatz bietet die (bislang nicht genutzte) Möglichkeit, die Genauigkeit von Modellen zu reduzieren und mit allgemeinen, sicheren Einschlüssen eines im Detail unbekannten Verhaltens zu arbeiten. Diese Möglichkeit soll in weiteren Forschungen noch weiter erschlossen werden.

Referenzen

[1] R. Y. Rubinstein, *Simulation and the Monte Carlo Method*. New York, NY, USA: John Wiley&Sons, Inc., 1981.

[2] K. Antreich, H. Gräb, and C. Wieser, "Practical Methods for Worst-Case and Yield Analysis of Analog Integrated Circuits," *International Journal of High Speed Electronics and Systems*, vol. 4(3), pp. 261–282, 1993.

[3] M. Rafaila, C. Grimm, C. Decker, and G. Pelz, "Sequential design of experiments for effective model-based validation of electronic control units," *e&i Elektrotechnik und Informationstechnik*, vol. 127, pp. 164–170, 2010.

[4] T. Henzinger and H. Wong-Toi, "Linear phase-portrait approximations for nonlinear hybrid systems," *Hybrid Systems III: Verification and Control*, vol. LNCS 1066, pp. 377–388, Springer, 1996.

[5] L. Hedrich and E. Barke, "A Formal Approach to Nonlinear Analog Circuit Verification," in *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, 1995, pp. 123–127.

[6] R. Sommer, E. Hennig, M. Thole, T. Halfmann und T. Wichmann, „Analog Insydes 2 – New Features and Applications in Circuit Design", in *Proceedings of the Sixth International Workshop on Symbolic Methods and Applications in Circuit Design*, 2000

[7] R. E. Bryant: "A methodology for hardware verification based on logic simulation", in *Journal of the ACM*, Volume 38 Issue 2, April 1991, pp. 299-328.

[8] C. Grimm, W. Heupke, and K. Waldschmidt, "Analysis of mixed-signal systems with affine arithmetic", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 118–123, 2005.

[9] D. Grabowski, M. Olbrich, and E. Barke, "Analog circuit simulation using range arithmetics," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '08)*, pp. 762–767, IEEE Computer Society Press, Seoul, Korea, March 2008.

[10] D. Grabowski, M. Olbrich, and E. Barke, "Analog Circuit Simulation Using Range Arithmetics," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference (ASP-DAC '08)*. Seoul, Korea: IEEE Computer Society Press, 2008, pp. 762–767.[11] C. Grimm, W. Heupke, and K. Waldschmidt, "Analysis of Mixed-Signal Systems with Affine Arithmetic" *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 24, no. 1, pp. 118–123, 2005.

[12] M. Andrade, J. Comba, and J. Stolfi, "Affine Arithmetic (extended abstract)," *Interval '94, St.Petersburg, Russia*, 1994.

[13] M. Barnasconi, K. Einwich, C. Grimm, T. Maehne, and A. Vachoux, "Advancing the SystemC Analog/Mixed-Signal (AMS) Extensions: Introducing Dynamic Timed Data Flow," *Open SystemC Initiative (OSCI), White Paper*, 2011.

[14] F. Schupfer, M. Kärgel, C. Grimm, M. Olbrich, and E. Barke, "Towards abstract analysis techniques for range based system simulations," in *Forum on Specification and Design Languages 2010 (FDL '10)*, 2010, pp. 1–6.

Automatic detection of sticky clock gating functions

Maarten Boersma, Ulrike Schmidt, Markus Kaltenbach
IBM Deutschland Research and Development GmbH
Schönaicher Strasse 220, 71032 Böblingen, Germany
{mboersma,schmidtu,kaltenba}@de.ibm.com

Abstract

We present a robust and straightforward method to automatically detect a common mistake in advanced clock gating functions in a design under verification. This type of mistake leads to unnecessary power consumption, but not to functional fails, so it escapes in a purely functional verification methodology.
The proposed method can be used as an enhancement to the existing HDL simulation flow and does not require insight in design details. The method can be easily scaled out to very large designs and can even be applied before all clock gating functions are implemented. This allows clock gating optimization to become an integral part of the design cycle.
We successfully applied the method during the design phase of the Decimal Floating Point Unit and Vector Scalar Unit for IBM's POWER8* microprocessor core. The issues we discovered led to pre-silicon design fixes, reducing the number of clocked register bits by 13% to 76%, depending on the workload scenario.

1. Introduction

Energy efficiency of a chip design is of increasing importance, driven by battery lifetime, cooling, or power supply requirements. During the design phase, significant effort is spent on minimizing both passive and active power consumption in order to meet the energy efficiency target.
Clock gating is a well known logic technique to increase energy efficiency of a design by reducing the active power dissipation [1], [2]. In high-performance microprocessor designs, clock gating functions are very fine-grained, leading to hundreds of clock gating domains within a single execution unit. With today's aggressive power dissipation targets, advanced clock gating functions are implemented that are usually manually engineered [3], [4].
The primary goal of verification work is to get confidence in the functional correctness of a design. However, the raising importance of energy efficiency requires to also verify the implemented power saving mechanisms.
We propose an enhancement to the functional verification flow in order to confirm that the implemented clock gating functions match with the intention of the designer.

*Trademarks of IBM in USA and/or other countries.

2. Clock gating background

In high performance microprocessor designs, local clock buffers (LCBs) are used to buffer the global clock signal to groups of register bits [5]. A conceptual view of a local clock buffer is given in Figure 1 (copied from [4]). The global clock input connects to the clock grid; the local clock output pin is connected to the clock inputs of the individual register bits.

Clock gating functions, computing the enable signal, are implemented per LCB, not for each

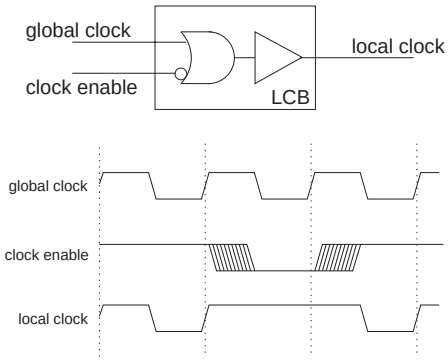


Figure 1: LCB with clock gating support (conceptual)

individual register bit. Register bits sharing the same enable signal are said to be on the same clock domain and can connect to a common LCB.

In order to perform clock gating, an enable input pin on the local clock buffer is provided. If the enable signal is active, the clock is propagated to the registers. If the enable signal is inactive, the clock propagation is blocked inside the LCB.

If the enable signal is inactive, power consumption decreases in three ways. First, the local clock signal is quiet and its capacity is no longer charged or discharged. Second, no energy is spent on changing the state of the connected registers. Third, because the output of the register does not change, switching of downstream logic is prevented.

For a given workload, the quality of the implemented clock gating scheme in the design is expressed as the average number of enabled (clocked) register bits per cycle. A lower number is better, because the energy consumption decreases with the number of clocked register bits.

3. Sticky clock gating functions

Although we found a lot of recent work on the design of aggressive clock gating functions (see for example [3], [4], [6], [7]), the verification of the enable signals seems to have low priority.

However, in our high-performance designs, hundreds of clock gating functions are manually engineered and explicitly written into the RTL design source. Obviously this is an error-prone designer task, giving raise to two kinds of errors:

- If an enable signal is inactive when it has to be active, data that are necessary in the next cycle will not be available. This will usually lead to a functional fail and is found with a state-of-the-art functional verification flow.

- If an enable signal is active when it could have been inactive, the design may still operate correctly. Typically, the only symptom of the problem is a waste of active power dissipation, because data are unnecessarily transferred to register outputs. Modern designs with a tight power budget cannot afford these kinds of mistakes.

In our designs, enable signals are usually computed locally from a $valid$ signal from an earlier cycle. The logic to compute an enable signal can be viewed as a small state machine. In the simplest form there are two states, on and off.
For the registers in these state machines, it has been suggested to use clock domains that are always active [1]. Because of timing constraints the enable signals need to be generated close to the domains that they control, so many additional LCBs would be necessary, distributed across the chip. These additional LCBs add active chip area, so introducing clock gating in this way is a trade-off and will not always reduce power consumption.
Instead, in modern designs, the state machine registers can be clock gated as well. Ideally they connect to the same LCB as the data registers. The state machine registers are active one cycle longer than what is necessary for the data, in order to return the state machine to the off state (although it is even possible to avoid the additional enable cycle by using the technique described in [8]).
In the context of clock gated state machines, we observe that a particular mistake is frequently made during the design phase: the state machine gets clock gated while an enable output is still active. This prohibits further state transitions. If this happens, the enable signal locks to on, or, in other words, the clock gating function is said to be sticky. Figure 2 illustrates this for a single clock domain. At $t = t_b$, the design is idle and the clock domain is inactive. At $t = t_1$ the clock domain is enabled, in order to perform some operation. At $t = t_2$ the operation has completed and the clock domain should become inactive, but stays active in presence of the design mistake. At $t = t_e$ the design returns to a low-power idle state and all clock domains should be inactive - but the clock domain connecting to the sticky enable signal is still active.

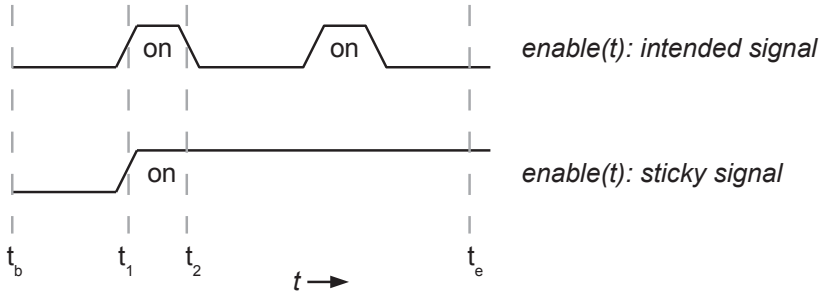
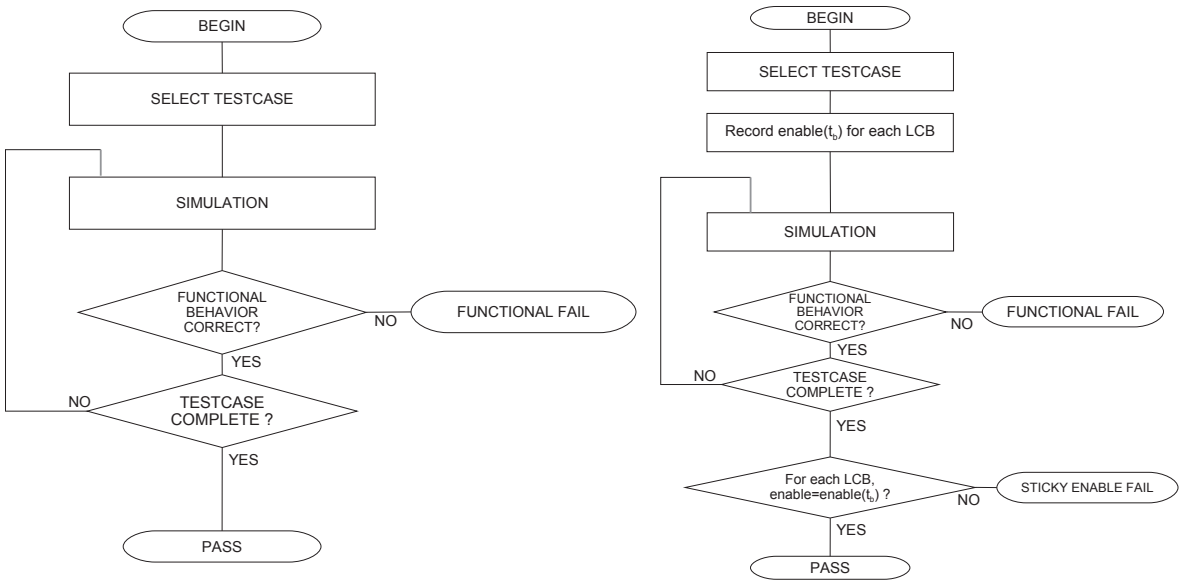


Figure 2: Intended versus sticky clock gating functions

If this mistake escapes into silicon, the state machine will lock into the active state sooner or later, and after this occurs, the enable signal can never turn off again. Hence the connected clock domain, intended to be clock gated, is in practice always active. This makes the active power dissipation higher than expected and leads to a waste of energy.
Our method verifies that these state machines are not clock gated while being in the on state.



(a) Purely functional **(b)** Enhanced for sticky enable detection

Figure 3: Traditional and enhanced verification flow

4. Detection approach

Our starting point is a state-of-the-art verification flow as conceptually shown in Figure 3a. At the beginning $(t = t_b)$ and at the end $(t = t_e)$ of the process, we assume that the design is in an idle state. For example, the idle state can be reached by explicitly initializing all state elements to zero, or by starting from an idle state from a previous simulation run. Uninitialized values (X or U) in the state machines controlling the enable signals are not allowed at $t = t_b$.
If the design is functionally correct, the test passes. If unexpected behavior is observed, the test fails and requires attention.

To detect sticky clock gating functions, we move to the enhanced verification flow of Figure 3b. For any clock domain $i \in 1..N$ in the design, we express the value of the clock gating function at time t as a binary value $enable_i(t)$.
Because the design is idle at $t = t_b$, the initial values of the enable signals are well-defined. After a successful verification run, the design is supposed to return to a (different) idle state, for which we want to verify that the values of the enable signals match with their initial values.
Therefore, if simulation does not fail because of a functional problem, we compare the final value $enable_i(t_e)$ to the initial value $enable_i(t_b)$ for every clock domain i. Depending on the comparison of those values, clock domain i behaves as shown in Table 1.

We define $delta_i(t) = [enable_i(t_b) \neq enable_i(t)]$. The fail condition for clock domain i can now be written as $delta_i(t)$, observed at $t = t_e$.
The test fails if the fail condition is true for any of the N clock domains, hence the overall fail condition can be expressed as the or-reduction of all $delta_i(t)$ signals, observed at $t = t_e$:

$$delta_{any}(t) = \bigcup_{i=1}^{N} delta_i(t) \quad ; \quad fail = delta_{any}(t_e)$$

$enable(t_b)$	$enable(t_e)$	Scenario	Test Result
0	0	Gated clock domain, enable signal not sticky	PASS
1	1	Always-on clock domain	PASS
0	1	Gated clock domain, enable signal sticky	FAIL
1	0	Gated clock domain, enable signal active in initial state	FAIL

Table 1: Simulation scenarios and fail condition for a clock domain

The third line in Table 1 represents successful detection of a sticky clock gating function. In this scenario, energy is wasted *after* the clock domain is used. The last line in the table stands for another scenario where energy is wasted *until* the clock domain is used. Although this is not the design mistake we are primarily looking for, this scenario is also detected by our method and is also worth a design fix.

5. Implementation

The logic implementing the $delta_{any}(t)$ function is shown in Figure 4. Only the AND gates are part of the design under verification, and correspond to the logic gates that block the propagation of the global clock signal through the LCBs. The remaining logic is present in the simulation model, but not in the hardware.
The registers holding the initial values of $enable_i$ are clocked once during the initial idle state, such that they load their enable signal after initializing the design, but before the testcase causes design activity.

The single output of the instrumentation logic is the signal $delta_{any}(t)$ and is observed by the simulation environment at $t = t_e$.
To describe the logic we chose the IBM-internal instrumentation language Bugspray [9]. We suggest SystemVerilog or PSL as alternative in non-IBM environments.
The instrumentation code is automatically generated by parsing a list of all the clock domains in the design HDL. An example of the resulting code is shown in Figure 5. The square brackets describe the registers that hold the initial value of the $enable$ signals. The final line in Figure 5 implements the N-way or-reduction of all $delta(t)$ signals.

A single change was necessary in the simulation environment: upon successful completion of a functional test, the $delta_{any}$ signal from the simulation model is probed. If it is active, an error is issued. If the design was indeed idle in the beginning and end of test case, a simulation waveform needs to be inspected and the clock gating implementation can be reviewed.
The cost of the method is the addition of the instrumentation logic to the simulation model. Since, for N clock domains, the instrumentation logic consists of N registers, N XOR gates and $N - 1$ OR gates, we expect no measurable slowdown in simulation throughput.

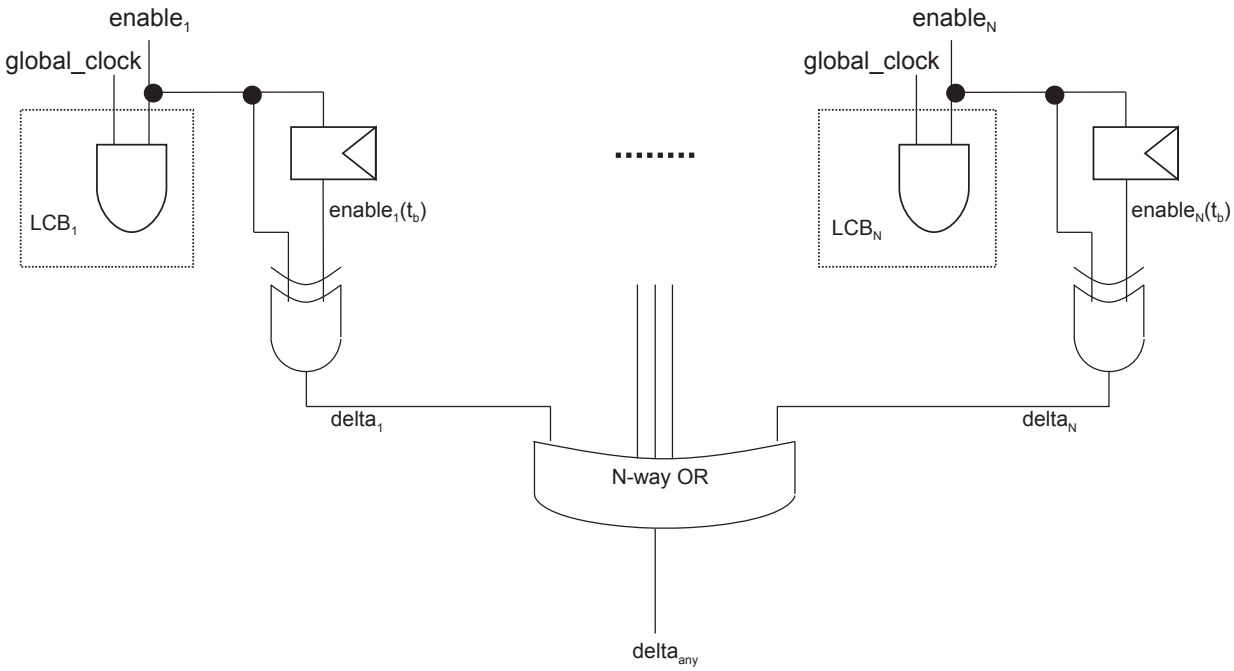


Figure 4: Instrumentation logic for detection of sticky clock gating functions

```
delta(0)   <= fpu.multiplier.ex1.enable xor [fpu.multiplier.ex1.enable];
delta(1)   <= fpu.multiplier.ex2.enable xor [fpu.multiplier.ex2.enable];
:             :                         :     :
delta(9)   <= fpu.rounder.ex6.enable    xor [fpu.rounder.ex6.enable];
delta_any <= or(delta(0 to 9));
```

Figure 5: Sample code for instrumentation logic

6. Results

We applied our approach during the design phase of two execution units that are both part of the IBM POWER8 microprocessor core, namely the Vector Scalar Unit (VSU) and the Decimal Floating Point Unit (DFU).
By looking at the commit messages from the version control system of the HDL source, we identified six hardware changes that were triggered by our new verification method. These fixes addressed sticky clock gating functions, together controlling about 2,600 register bits. Since, in total, the units contain about 51,000 registers, about 5% of the total number of register bits were affected.
As argued before, after design initialization, sooner or later the registers controlled by sticky clock gating functions will become always active. Therefore for our result evaluation we consider them as always-active clock domains.
For power consumption prediction, we consider two workload scenarios [3]. The first workload is the idle scenario where the VSU and DFU execution units are not active. The second workload is a power virus, designed to maximize design activity. An approximate per-cycle average of clocked register bits, for either scenario, is given in Table 2.
We identified sticky clock gating functions only for domains that are not used in the power virus workload. This is because the clock gating quality for the power virus workload was already carefully monitored and improved before we started our automated approach.

	total	active per cycle (idle scenario)	active per cycle (max power scenario)
without design fixes	51,000	3,400 (6.7%)	20,600 (40%)
with design fixes	51,000	800 (1.6%)	18,000 (35%)
improvement		76%	13%

Table 2: Approximate cumulative register bit counts in POWER8 VSU and DFU

With our method, we reduced the number of clocked register bits by 5% of the total number of registers, early in the design process. For the maximum power scenario, this corresponds to a 13% reduction, whereas for the idle scenario, the reduction is 76%.
It is difficult to express the results in Watt, because actual power consumption depends on technology, operating conditions, and input data patterns. Therefore, we use the number of clocked register bits as first-order estimate of active power consumption, as in [1] and [4].

7. Discussion

We see our method as straightforward and robust enhancement to any existing functional verification flow. The idea can be applied in both simulation and formal environments, but for detection of sticky clock gating functions, enhancing the simulation flow suffices.
In the results we presented, we only considered the design fixes that were actually committed into the design repository. In our experience, issues were often found by designers before committing logic changes.

Our method recognizes the increasing importance of energy efficiency of chip designs. It gives confidence in the correctness of the clock gating functions without introducing an additional verification process. Hence it makes the design of clock gating functions an integral part of the design cycle, instead of considering it an optional optimization effort after functional verification.
Because of these advantages, we are rolling out the method to other designs and other projects.

References

[1] H. Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy, "Deterministic clock gating for microprocessor power reduction," in *In Proc. of 9 th Int'l Symp. on High Performance Computer Architecture (HPCA)*, pp. 113–122, 2003.

[2] C. M. Abernathy, G. Gervais, and R. Hilgendorf, "Method and apparatus for dynamic power management in an execution unit using pipeline wave flow control." United States Patent 7,137,013, November 2006.

[3] V. Zyuban, J. Friedrich, C. J. Gonzalez, R. Rao, M. D. Brown, M. M. Ziegler, H. Jacobson, S. Islam, S. Chu, P. Kartschoke, G. Fiorenza, M. Boersma, and J. A. Culp, "Power optimization methodology for the IBM POWER7 microprocessor," *IBM J. Res. Dev.*, vol. 55, pp. 267–275, May 2011.

[4] J. Preiss, M. Boersma, and S. M. Mueller, "Advanced clockgating schemes for fused-multiply-add-type floating-point units.," in *IEEE Symposium on Computer Arithmetic*, pp. 48–56, IEEE Computer Society, 2009.

[5] J. D. Warnock, L. J. Sigal, D. F. Wendel, K. P. Muller, J. Friedrich, V. V. Zyuban, E. H. Cannon, and A. J. KleinOsowski, "POWER7 local clocking and clocked storage elements," in *ISSCC*, pp. 178–179, IEEE, 2010.

[6] E. Arbel, C. Eisner, and O. Rokhlenko, "Resurrecting infeasible clock-gating functions.," in *DAC*, pp. 160–165, ACM, 2009.

[7] T.-K. Lam, X. Yang, W.-C. Tang, and Y.-L. Wu, "On applying erroneous clock gating conditions to further cut down power," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, ASPDAC '11, (Piscataway, NJ, USA), pp. 509–514, IEEE Press, 2011.

[8] T. Gemmeke, J. Leenstra, and J. Preiss, "Method to reduce power consumption within a clock gated synchronous circuit and clock gated synchronous circuit." United States Patent 7,639,046, December 2009.

[9] J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, B.-L. Chu, M. L. Behm, J. R. Baumgartner, R. D. Peterson, J. Abdulhafiz, W. E. Bucy, J. H. Klaus, D. J. Klema, T. N. Le, F. D. Lewis, P. E. Milling, L. A. McConville, B. S. Nelson, V. Paruthi, T. W. Pouarz, A. D. Romonosky, J. Stuecheli, K. D. Thompson, D. W. Victor, and B. Wile, "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems," *IBM J. Res. Dev.*, 2002.

Vergleich der Beschreibung und Simulation einer Befehlssatzarchitektur in LISA und CoMet.

Roberto Urban, Kai Lehninger, Maximilian Heyne, Mario Schölzel, H.T. Vierhaus

rurban@informatik.tu-cottbus.de,

{lehnikai | heynemax}@tu-cottbus.de,

{mas | htv}@informatik.tu-cottbus.de

BTU Cottbus - Senftenberg, Lehrstuhl Technische Informatik

Kurzfassung

CoMet (Compilerzentrierter Mikroprozessorentwurf) unterstützt einen alternativen Entwurfsprozess anwendungsspezifischer Mikroprozessoren, indem er einen anderen Ansatz der Entwurfsraumexploration umsetzt. Dieser Ansatz stellt den Compiler ins Zentrum des Entwurfs. Der Entwurf wird durch eine schrittweise Transformation der im Compiler-Backend genutzten Zwischensprachen in niedrigere Abstraktionsniveaus vorangetrieben. Für die Spezifikation aller Zwischensprachen wird die von uns entwickelte Sprache MaMa verwendet. Gleichzeitig liefert die MaMa-Spezifikation einer Sprache die Konfiguration des Simulators für diese spezielle Sprache, indem die Befehlssatzarchitektur einer virtuellen Maschine beschrieben wird. Damit ist eine Simulation des Zwischencodes in jedem Abstraktionsniveau sofort möglich und liefert Profiling-Ergebnisse für die weiteren Schritte. Am Ende des Prozesses steht eine Zwischensprache, die als Assemblercode fungieren kann. Die Spezifikation dieser Sprache entspricht der Beschreibung einer Befehlssatzarchitektur eines Prozessors und ist mir einer Beschreibung in LISA vergleichbar. In dieser untersten Abstraktionsebene wird in diesem Beitrag ein vorhandener Prozessor sowohl mit MaMa als auch mit LISA beschrieben und verglichen. Der Vergleich soll den unterschiedlichen Aufwand beleuchten, die Befehlssatzarchitektur in den beiden Sprachen zu beschreiben, zu simulieren und zu validieren. Wir wollen weiterhin Vorteile, Nachteile und Möglichkeiten beider Ansätze in diesem Abschnitt des Entwurfs aufzeigen.

1. Einleitung

Der Entwurf anwendungsspezifischer Prozessoren ist nach wie vor im Bereich eingebetteter Systeme von Bedeutung. Immer, wenn besondere Anforderungen an das Verhältnis zwischen Geschwindigkeit, Platzbedarf, Leistungsaufnahme und Kosten des Prozessors gestellt werden, kann es sich auch trotz einer Vielzahl guter Standardprozessoren und IP-Cores herausstellen, dass es notwendig oder günstiger ist, einen auf die Anwendung optimierten Prozessor zu entwickeln. Dadurch, dass der optimierte Prozessor nur die Befehle im Befehlssatz verwendet, die auch für die Anwendung benötig werden, wird dieser Prozessor in der Regel weniger Platz benötigen und weniger Strom verbrauchen. Zusätzlich besteht die Möglichkeit eine schnelle und effektive Abarbeitung des Anwendungsalgorithmus mit speziellen Befehlen zu unterstützen. Besonders für echtzeitkritische Anwendungen ist es von großem Vorteil, das Verhalten des Prozessors bis ins Detail bestimmen und optimieren zu können.

Um für einen bekannten Algorithmus eine Zielarchitektur zu bestimmen, die den Algorithmus innerhalb der Vorgaben abarbeitet, ist viel Erfahrung oder viel Zeit bei der Entwurfsraumexploration nötig. Die Entwurfsraumexploration beinhaltet verschiedene Schwierigkeiten. Als Erstes stellt sich die Frage, wie die Zielarchitektur beschrieben werden soll. Hier unterscheidet man zwischen Strukturbeschreibungen und Befehlssatzarchitekturen (ASIPs). Eine Strukturbeschreibung der Architektur kann beispielsweise mit einer Hardwarebeschreibungssprache (HDL, z.B. VHDL, Verilog) oder mit einer abstrakteren Architekturbeschreibungssprache (ADL) erstellt

werden. Für die Beschreibung eines optimierten Befehlssatzes der Zielarchitektur wurden Beschreibungssprachen wie LISA [1] und nML[2] bereits vorgestellt.

Danach stellt sich das Problem der Simulation. Die ursprüngliche Anwendung muss in den Maschinencode der jeweiligen Architektur übersetzt und eine Testumgebung erstellt werden, um die Korrektheit des Entwurfs zu überprüfen und Profiling-Daten zu sammeln. Da es für die entwickelte Architektur typischerweise noch keinen fertigen Compiler gibt muss dieser erstellt oder zumindest konfiguriert werden. Zur Unterstützung des Prozessorentwurfs und der Entwurfsraumexploration wurden bereits eine Vielzahl verschiedener Ansätze präsentiert, die auf einer ADL Beschreibungen [3] [4] oder einer Befehlssatzbeschreibung [5] [6] basieren. Einige Ansätze kombinieren beide Konzepte [7] [8]. Es wurden auch bereit Ansätze [9] [10] vorgestellt, das Problem der Compiler-Generierung zu umgehen, indem man den Compiler ins Zentrum des Entwurfs stellt. Allerdings wird hier der Entwurfsraum durch vorgegebene Architekturen stark eingeschränkt. Allen Ansätzen ist gemeinsam, dass sie ein zyklisches Vorgehen bei der Entwurfsraumexploration verfolgen. Eine breitere Anwendung in diesem Umfeld erfährt der Entwurf von ASIP Architekturen mit LISA. Aus diesem Grund wollen wir unseren Ansatz in CoMet mit dem Entwurf in LISA vergleichen und zeigen, dass der Entwurfsprozess mit LISA bereits von CoMet abgedeckt werden kann.

Mit CoMet wird ein neuer Weg des Entwurfs beschritten. Speziell die Entwurfsraumexploration unterscheidet sich deutlich von allen bisherigen Ansätzen. Es wird eine verfeinernde Entwurfsmethodik verfolgt, in der ohne Zyklen eine passende Zielarchitektur entworfen werden soll. In [11] wurde dieses Konzept bereits vorgestellt. Der Entwurfsprozess nähert sich ohne vollständig bekannte Entwurfsalternativen über eine Verfeinerung der im Compiler genutzten Zwischensprachen schrittweise der optimalen Architektur. Ausgehend von dem abstrakten Zwischencode eines C-Compiler-Frontends entsteht bei jedem Transformationsschritt ein neuer Zwischencode, der sofort simuliert werden kann. Das Vorgehen wird fortgesetzt bis auf unterster Abstraktionsebene ein Zwischencode entsteht, der dem Assemblercode einer Zielarchitektur entspricht. Die MaMa-Spezifikation der entsprechenden Zwischencodesprache enthält die nötigen Detailinformationen, aus denen die Zielarchitektur abgeleitet werden kann. Damit ist diese MaMa-Spezifikation auf unterster Abstraktionsebene vergleichbar mit einer LISA-Spezifikation der Zielarchitektur. Im folgenden Abschnitt 2 wollen wir die Unterschiede und Gemeinsamkeiten beider Beschreibungen herausstellen.

2. Vergleich der zwischen LISA und CoMet

Im Fokus der ersten Arbeiten an dem Projekt CoMet stand die Entwicklung eines konfigurierbaren Simulators für Zwischensprachen. Diesem Simulator kommt im vorgesehen Prozess des compilerzentrierten Entwurfs eine zentrale Bedeutung zu, da es für die Verfeinerung der Zwischencodesprachen detaillierter Profiling-Informationen bedarf, die durch eine sofortige Simulation der Zwischencodes gewonnen werden. Hierfür ist es notwendig die Zwischensprachen zu definieren und den Simulator entsprechend zu konfigurieren. Daher war der erste Schritt die Entwicklung einer möglichst einfachen Beschreibungssprache (MaMa), mit der sich ein großes Spektrum an Zwischensprachen auf verschiedenen Abstraktionsniveaus beschreiben lässt. Sie beschreibt die Syntax und Simulationssemantik für Zwischencodesprachen. Mittels dieser Spezifikation soll ebenfalls der Simulator für die entsprechende Zwischencodesprache konfiguriert werden, so dass er in der Lage ist, Zwischencodeprogramme dieser Sprache zu erkennen und zu simulieren. Im folgenden Abschnitt wollen wir nun die bekannte und bereits verwendete Beschreibungssprache LISA mit MaMa vergleichen.

Eine Spezifikation besteht bei beiden Ansätzen aus zwei Teilbereichen. Zum Einen ist das die Beschreibung der Speicherstruktur, zum Anderen die Beschreibung des Befehlssatzes. Im Fol-

genden werden diese Teilbereiche genauer beleuchtet und Hinweise zum Erstellen der Spezifikation gegeben. Ein weiterer wichtiger Aspekt ist die Simulation der Anwendung auf der beschriebenen Architektur. Wir wollen daher aufzeigen, wie die Simulation in beiden Ansätzen umgesetzt ist.

2.1 Speicherbeschreibung

In MaMa gliedert sich die Beschreibung der Speicherstruktur in die Definition einer Speicherklassenhierarchie mittels des Schlüsselworts MEMORYCLASS und in eine Instanziierung der konkreten Speicher mittels des Schlüsselworts MEMORY. Speicherklassen setzen sich in der Hierarchie aus bereits Definierten Klassen zusammen. In Abbildung 1a ist diese Struktur an einem Beispiel verdeutlicht. Die kleinste Speicherklasse „Bit“ ist vorgegeben und eine Klasse die auf Bit referenziert, definiert immer den unteren Abschluss der Hierarchie. Es können in einer Hierarchie mehrere Klassen existieren, die auf Bit referenzieren. Jeder Speicherklasse können Datentypen zugewiesen werden. In der Abbildung 1a wird der Klasse „Word“ ein 12 Bit breiter Integer-Datentyp zugewiesen. In diesem Beispiel ist es während der Simulation nur möglich, Werte dieses Datentyps in den Instanzen dieser Klasse zu speichern. Einer Klasse können ebenso mehrere Datentypen zugewiesen werden. Die Bezeichner der instanziierten Speicher (im Beispiel R, MEM, PC, SR) werden später in der Semantik-Beschreibung des Befehlssatzes verwendet. Durch diesen Ablauf lassen sich beliebig komplexe Speichermodelle beschreiben und erstellen. Dadurch ist es möglich, sehr leicht komplexere Datenstrukturen zu erzeugen und diese auch in hohen Abstraktionsniveaus zu verwenden.

Speicherhierarchie	`MEMORYCLASS Word = Bit[12] (T12BitInteger);`
	`MEMORYCLASS RegisterBank = Word[31];`
	`MEMORYCLASS Instruction = Bit[24] (TProgramCode);`
	`MEMORYCLASS ProgramMemory = Istruction[4096];`
	`MEMORYCLASS DataMemory = Word[4096];`

Speicherinstanzen	`MEMORY R RegisterBank;`
	`MEMORY DMEM DataMemory;`
	`MEMORY PMEM ProgramMemory;`
	`MEMORY PC Word;`
	`MEMORY SR Word;  //Statusregister`

Abbildung 1a: Speicherstruktur in MaMa am Beispiel

```
RESOURCE {
       PROGRAMM_COUNTER uint16 PC;;
       REGISTER unsignedbit[12] R[31];
       REGISTER unsignedbit[1] SR[12];
       RAM unsignedbit[24] PMEM{
              SIZE(4096);
              BLOCKSIZE(24);
              FLAGS( R | W ); };
       RAM unsignedbit[12] DMEM{
              SIZE(4096);
              BLOCKSIZE(12);
              FLAGS( R | W ); };
       MEMORY_MAP{
              RANGE(0x0000, 0x0fff) -> PMEM[(23..0)]
              RANGE(0x1000, 0x1fff) -> DPMEM[(11..0)]
       }
};
```

Abbildung 1b: Auszug aus der Speicherstruktur in LISA

Die Beschreibung der Speicherstruktur unterscheidet sich in LISA deutlich. Zum Vergleich wurde in Abbildung 1b das gleiche Beispiel umgesetzt. Nach dem Schlüsselwort RESOURCE wird der Speicher mit vorgegeben Konstrukten instanziiert und parametrisiert. Die Möglichkeit eigene Speicherhierarchien anzulegen, besteht hier nicht. Es existiert eine Auswahl an verschiedenen Speichermodellen. Im Beispiel wurden die gebräuchlichsten Klassen REGISTER und RAM verwendet. Da im Simulator die Speicherklasse RAM immer durch 32-bit Integer repräsentiert ist, muss hier noch ein MEMORY_MAP folgen, um die Adressen der Instanziierten Speicher auf diese Adressen des simulierten RAMs abzubilden. Der PC besitzt wie in MaMa eine Sonderrolle, wird hier aber zusätzlich mit einer eigenen Speicherklasse instanziiert.

2.2 Beschreibung des Befehlssatzes

Im Anschluss an die Speicherstruktur wird der Befehlssatz angelegt. Befehle müssen in MaMa durch ihre Syntax und ihre Semantik spezifiziert werden. Die Syntaxbeschreibung beginnt mit dem Schlüsselwort IDEF und endet mit der öffnenden geschweiften Klammer, die gleichzeitig den Beginn der Semantikbeschreibung signalisiert. Für jeden Zwischencodebefehl wird seine Syntax durch eine Folge von Mustern (Syntaxelementen) definiert. Die Syntaxelemente werden durch Anführungszeichen gekennzeichnet. An Stellen, an denen im Zwischencodeprogramm konkrete Zahlen (z.B. Speicherindizes, Adressen oder Konstanten) erwartet werden, befinden sich Platzhalter der Form *%x*, wobei *x* die Position des Platzhalters im Befehl nummeriert. Die Platzhalter werden bei 0 beginnend von links nach rechts durchnummeriert. In der Simulation werden an den Stellen der Platzhalter die erwarteten Werte eingelesen und so verwendet, wie sie in der Semantikbeschreibung der Befehle festgelegt wurden.

```
// Transfer Befehl: R-F | MOV
IDEF "MOV" "R"%0 "," "R"%1 {
       R[%0]<=R[%1], PC<=PC+1
}
// Bitweises Logische UND: R-F | AND
IDEF "AND" "R"%0 "," "R"%1 {
       R[%0]<=R[%0] & R[%1];
       SR[0]<=~(R[%0][0]|R[%0][1]|R[%0][2]|R[%0][3]|R[%0][4]|R[%0][5]|R[%0][6]|
       R[%0][7]|R[%0][8]|R[%0][9]|R[%0][10]|R[%0][11]),
       SR[4]<=R[%0][0]^R[%0][1]^R[%0][2]^R[%0][3]^R[%0][4]^R[%0][5]^R[%0][6]^
       R[%0][7]^R[%0][8]^R[%0][9]^R[%0][10]^R[%0][11],
       PC<=PC+1
}
// Bedingter Sprung (falls Z=0): J-F | JNZ
IDEF "JNZ" "R"%0 {
   if SR[0]==0
       then { PC<=R[%0] }
       else { PC<=PC+1 }
}
```

Abbildung 2a: Auszug aus der Befehlssatzbeschreibung in MaMa

Anschließend wird innerhalb der geschweiften Klammern die Semantik des Befehls wird mittels Registertransfer-Operationen (RT-Operationen) beschrieben. In der Abbildung 2a ist das beispielhaft für drei Assemblerbefehle (MOV, AND, JNZ) gezeigt. RT-Operationen manipulieren die Inhalte der angesprochenen Speicherklassen. Ein Befehl kann aus einer oder mehreren RT-Operationen bestehen, die parallel oder als Sequenz ausgeführt werden können. Zwei RT-Operationen, die parallel in einem Takt ausgeführt werden sollen, sind mit Komma getrennt. Zwei RT-Operationen, die nacheinander ausgeführt werden sollen, sind mit Semikolon getrennt.

Im AND-Befehl dieses Beispiels können die ALU-Statusbits SR[0] und SR[4] erst berechnet werden, wenn das Ergebnis vorliegt. Daher ist hier nach der ersten RT-Operation ein weiterer Takt mit drei parallelen RT-Operationen nötig. Da der Zwischencode selbst und auch der Befehlszähler (PC) in spezielle Speicherklassen abgebildet werden, kann sowohl Daten- als auch Steuerfluss im Simulator über RT-Operationen organisiert werden. In jedem Befehl wird der Wert des PC weitergesetzt oder mit einem neuen Wert geladen, um Verzweigungen, Schleifen oder Funktionssprünge zu implementieren. In Abbildung 3 ist im untersten Befehl (JNZ) ein solcher Sprungbefehl dargestellt. Hier ist auch, mit dem „if-then-else"-Konstrukt, die einzige Sonderform der RT-Operationen dargestellt. Entspricht ein Zwischencodebefehl der in der Spezifikation festgelegten Syntax, so greift der Simulator auf die entsprechende Semantikbeschreibung zu und führt die RT-Operationen aus. Auf diese Weise konfiguriert die Mama-Spezifikation Simulator und beschreibt gleichzeitig Semantik einer Zwischencode-Sprache.

```
OPERATION MOV1{
      DECLARE{     GROUP p0 = {reg};
                   GROUP p1 = {reg};}
      SYNTAX {     "MOV" "R" p0 "," "R" p1}
      CODING {     0b0000110 0b0[7] p1 p0}
      BEHAVIOR {   R[p0] = R[p1];}
}
OPERATION AND1{
      DECLARE{     GROUP p0 = {reg};
                   GROUP p1 = {reg}; }
      SYNTAX {     "AND" "R" p0 "," "R" p1}
      CODING {     0b0001010 0b0[7] p1 p0}
      BEHAVIOR {   R[p0] &= R[p1];
                   SR[0] = !R[p0];
                   SR[0] = (SR[0] || R[p0] == 0xfff); }
}
OPERATION JNZ1{
      DECLARE{     GROUP p0 = {reg};}
      SYNTAX {     "JNZ" "R" p0 "," "R" p1}
      CODING {     0b0111111 0b0[7] p1 0b00001}
      BEHAVIOR {   if(!SR[0])PC = R[p1].to_uint32(0,12);}
}
```

Abbildung 2b: Auszug aus der Befehlssatzbeschreibung in LISA

In der Beschreibung des Befehlssatzes findet man sehr viele Parallelen. In der Abbildung 2b sind zum Vergleich die drei Assemblerbefehle MOV, AND, JNZ aus der Abbildung 2a in einer LISA Beschreibung umgesetzt. Die Beschreibung der Syntax und der Semantik (BEHAVIOR) ähneln sich in beiden Beschreibungssprachen, wobei der Vorteil der LISA Beschreibung in der einfachen Verwendung der C-Syntax für die Verhaltensbeschreibung liegt. Unterschiede existieren bei der Notwendigkeit der Angabe einer binären Kodierung (CODING) des Befehlswortes und dem Umgang mit den Platzhaltern. In LISA ist es notwendig das exakte Format der Platzhalter in DECLARE zu beschreiben und somit jeden erwarteten Wert (Konstante, Speicherindex, Adresse) einen separaten Platzhalter zu definieren. Das Verhalten der Platzhalter und binäre Codierung des erwarteten Wertes wird an anderer Stelle vorgenommen. In der Abbildung wurde das Format „reg" verwendet, was einen 5 Bit breiten Speicherindex der Registerbank beschreibt. Die Verwendung der Platzhalter in MaMa ist weniger restriktiv und bietet somit mehr Freiheiten aber auch Risiken. Der Entwurf wird zum Einen nicht durch festgelegte Schemen eingeschränkt, zum Anderen kann es zu Fehlern in der CoMet-Simulation kommen, wenn das Format der übergeben Wertes nicht mit der Verwendung übereistimmt. In LISA wird eine pha-

senweise Abarbeitung der Befehle angenommen, daher sind die zusätzlichen Befehlsphasen „fetch“, „decode“ und „nop“ zu definieren. Die Behandlung des Befehlszählers wird hier nicht in den Befehlen selbst, sondern in der „fetch“-Phase vorgenommen, die für jeden Befehl ausgeführt wird. Der PC kann jedoch auch hier z.B. in einem Sprungbefehl nochmals verändert und somit der Befehlsfluss gesteuert werden. Es besteht die Möglichkeit Befehlsklassen zu definieren. Hierfür kann in einem übergeordneten Befehl bestimmte Verhaltens- und Codierungsmuster beschrieben werden. Die untergeordneten Befehle dieser Befehlsklasse werden dann nur noch mit den kleineren Unterschieden in den Details definiert. Durch diese hierarchische Befehlsstruktur kann, je nach Zielarchitektur, Beschreibungsaufwand gespart werden.

2.3 Entwurfsumgebung und Simulator

In der MaMa-Spezifikation ist eine virtuelle Maschine beschrieben, mit der sich nur spezielle Zwischencodes simulieren lassen. Diese virtuelle Maschine wird durch den Simulator nachgebildet. Im Simulator ist ein Parser integriert, der zuerst die Spezifikation einliest und sich auf die spezifizierte Syntax und Semantik konfiguriert. So konfiguriert, kann der Parser später Zwischencodeprogramme erkennen und einlesen. Das heißt es werden zu einer MaMa-Spezifikation nur entsprechenden Zwischencodeprogramme simuliert, die der Syntax der beschriebenen Sprache entsprechen. Dieses Zusammenspiel zwischen der Beschreibungssprache und dem konfigurierbaren Zwischencodesimulator wurde in [12] bereits vorgestellt. Daher soll an dieser Stelle der grundlegende Simulationsablauf nur kurz wiederholt werden. Die Simulation läuft in folgenden Schritten ab:

1. Der im Simulator integrierte Parser wird gestartet und liest die MaMa-Spezifikation (Speicherhierarchie, -instanziierung, Befehlssatz) ein. Die Speicherinstanzen werden angelegt. Zum Befehlssatz wird ein Befehlsbaum der Sprache angelegt.

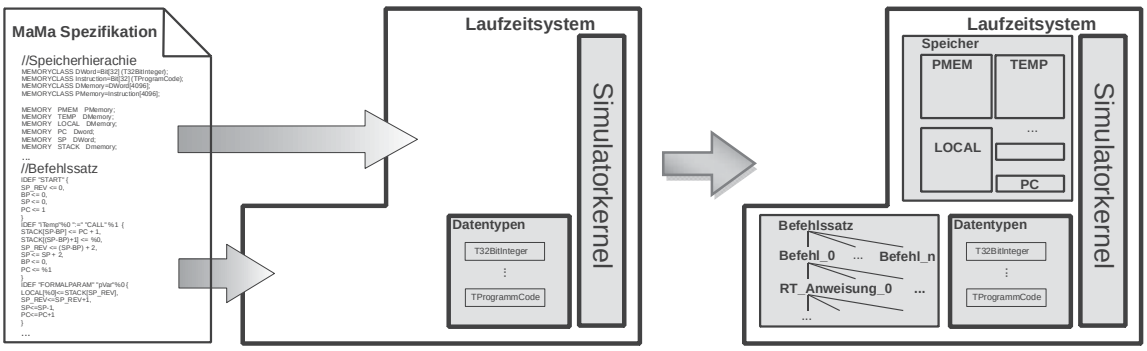


Abbildung 3a: Konfigurierung des Simulators

2. Der Parser liest unter Abgleich jedes Befehls mit dem Syntaxbaum des Befehlssatzes den Zwischencode ein. Die für die Simulation relevanten Informationen werden innerhalb des Laufzeitsystems im Programmspeicher abgelegt.

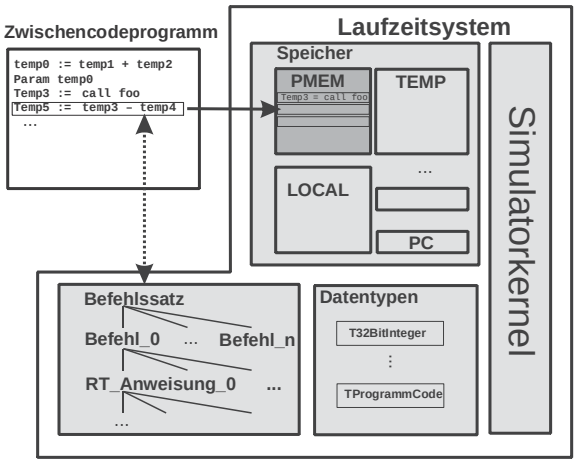


Abbildung 3b: Einlesen des Zwischencodeprogramms

3. Die Simulation wird gestartet indem der im PC gespeicherte Wert als Index über den Programmspeicher interpretiert wird. Die RT-Operationen aus der Semantikbeschreibung der Befehle werden durch den Simulator abgearbeitet und so die Speicherinhalte und der PC manipuliert, wodurch der Programmablauf in der virtuellen Maschine simuliert wird.

In den Abbildungen 3a und 3b ist dargestellt, wie das Laufzeitsystem für die Simulation in den ersten beiden Schritten angelegt und konfiguriert wird. Die Funktionalität der RT-Operationen, die während der Simulation abgearbeitet werden wird zu großen Teilen durch die in den Speichern abgelegten Datentypen bestimmt. Mehrere Basisdatentypen stehen unabhängig von der aktuellen Konfiguration und der simulierten Zwischencodesprache zur Verfügung. Sind neue Datentypen erforderlich können diese als C-Klassen im vorgegebenen Schema beschrieben werden und der Simulator muss neu übersetzt werden. Während der Simulation werden verschieden Profiling-Informationen gesammelt. Das sind Gesamtzahl und Häufigkeit der abgearbeiteten Instruktionen, Anzahl der RT-Operationen, Anzahl der benötigten internen Takte der virtuellen Maschine, sowie Art (Lesen oder Schreiben) und Häufigkeit der Zugriffe auf die Speicherobjekte. Mit diesen Informationen kann die Optimierung für den nächsten Zwischencode bestimmt werden und so der nächste Schritt im Entwurfsprozess vorangetrieben werden.

Zum Erstellen und Simulieren der LISA-Spezifikation wurde der „Synopsys Processor Designer“ genutzt. Die Simulation in beiden Umgebungen unterscheidet sich deutlich. Während in CoMet der Befehlssatzsimulator die RT-Operationen direkt interpretiert, wird im Befehlssatzsimulator eines in LISA-Designs eine Binärcodesimulation durchgeführt. Die Software erstellt aus der LISA Beschreibung automatisch einen passenden Simulator, Assembler und Linker. Der zu simulierende Assemblercode muss mit diesen Werkzeugen in Binärcode der Zielarchitektur übersetzt werden. Für die Simulation steht im der Entwicklungsumgebung der „Processor Debugger“ zur Verfügung. Um die Simulation zu starten, muss dann die übersetzte Anwendung mit diesem Werkzeug geladen werden.

2.4 Entwurfskonzepte

Hinter den bisher verglichenen Aspekten stehen zwei unterschiedliche Entwurfskonzepte. Wenn man die Leistungsfähigkeit der Ansätze abschließend bewerten will, muss der gesamte Entwurfsprozess verglichen werden. Mit LISA wird wie in Abbildung 4a und wie in allen bisherigen Ansätzen ein zyklischer Entwurfsprozess unterstützt. Es wird zu Beginn eine erste Architektur erstellt und diese dann über Simulation der Anwendung validiert. Sind die Vorgaben

erfüllt kann vom Prozessor ein HDL-Design generiert werden. Sind die Vorgaben nicht erfüllt muss das Design angepasst werden. Wenn eine komplexere Anwendung in einer Hochsprache vorliegt, muss zur die Simulation für jedes neue Design ein Compiler für die Hochsprache angepasst werden. Der Entwicklungsprozess kann je nach Erfahrungsgrad des Entwicklers mehr Iterationsschritte enthalten und wird damit sehr zeitintensiv.

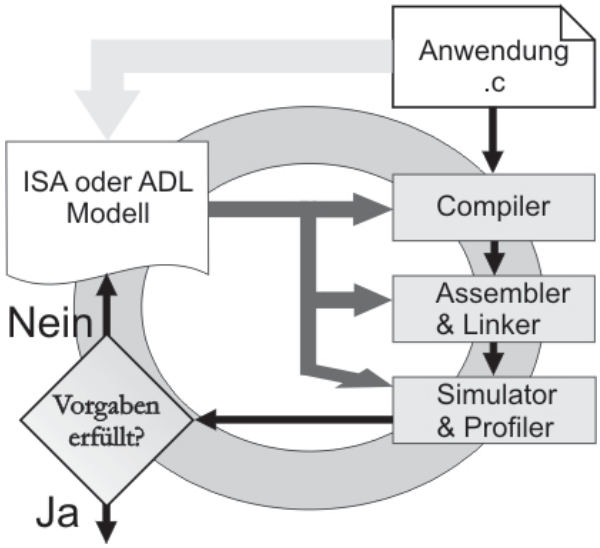


Abbildung 4a: Zyklisches Entwurfsprozess mit LISA (vgl. [13])

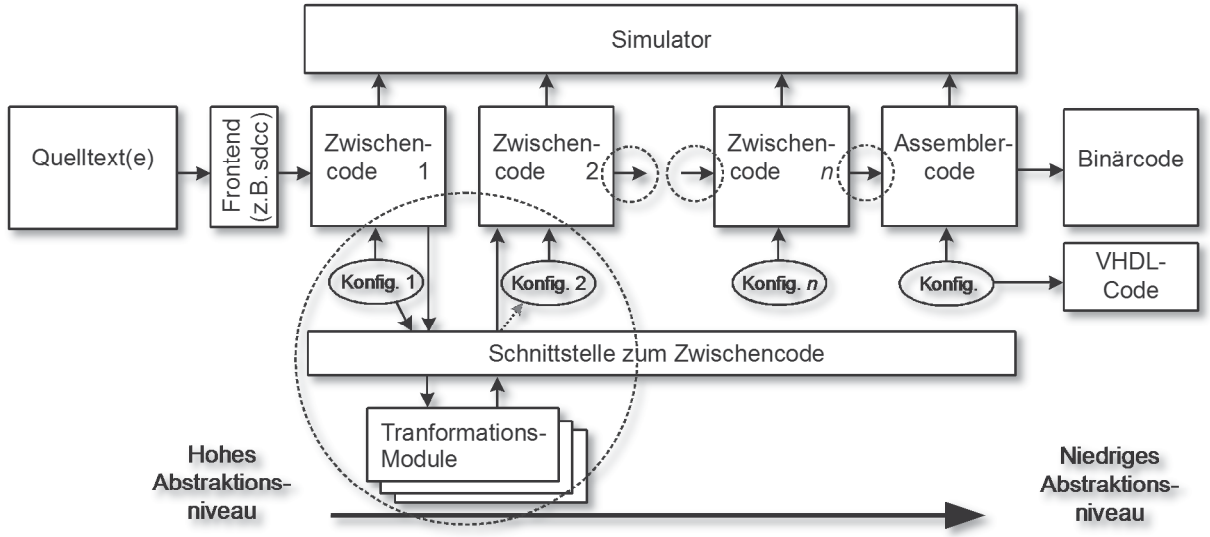


Abbildung 4b: Schrittweiser Entwurfsprozess in CoMet [11]

Mit CoMet wird ein alternativer Entwurfsprozess unterstützt, der von der Hochsprache beginnend Zwischencode generiert und diesen immer weiter bis zum Binärcode des Prozessordesigns verfeinert. Damit wird mittels dieser Zwischencode-Transformationen das Compiler-Backend nachgebildet. Die Notwendigkeit einer einfachen Beschreibung der Zwischencodesprachen war der Grund für die Entwicklung von MaMa. Die einzelnen Transformationen der Zwischensprachen werden über sogenannte Übersetzungsmodule durchgeführt. Das sind C-Algorithmen, die einen Zwischencode in einen neuen Zwischencode übersetzen und dabei Veränderungen und Optimierungen vornehmen. Module werden in der Entwicklungsumgebung als DLLs eigebun-

den. Vorhandene Module können wiederverwendet und angepasst werden oder es müssen neue Module geschrieben, kompiliert und eingebunden werden. Die Entwicklungsumgebung CoMet unterstützt das Erstellen, Kompilieren und Einbinden neuer Module durch vorgefertigte Templates und die Möglichkeit den MinGW C-Compiler einzubinden. In der Abbildung 4b wird der vorgestellte Entwurfsprozess komplett dargestellt. Eingekreist ist die Verfeinerung mittels der Transformationsmodule, die über eine Schnittelle auf die Zwischencode-Informationen zugreifen. Diese wurde vorher für die Simulation vom Parser eingelesen, aufbereitet und stehen als interne Datenstrukturen zur Verfügung. Jedes Modul erzeugt automatisch einen neuen Zwischencode, zu dem eine neue MaMa-Spezifikation erstellt wird. In der neu generierten MaMa-Spezifikation ist die Syntax und alle verwendeten Befehle bereits vorgeben. Speicherstruktur und Semantikbeschreibung des Befehlssatzes muss danach noch ergänzt werden. Je nach Transformationsschritt können aber ein Großteil der Informationen aus der vorhergehenden Spezifikation mit entsprechenden Veränderungen übernommen werden. Das Verfahren wurde exemplarisch mit Modulen zur Befehlsauswahl und Registerallokation umgesetzt und ein Zwischencode des Compiler-Frontends bis zum Assemblercode übersetzt. Am Ende des Prozesses fehlt bisher noch der letzte Syntheseschritt. Dabei soll aus der letzten MaMa-Spezifikation ein VHDL-Modell der Zielarchitektur generiert werden. Dieser Schritt ist Gegenstand weiterer Untersuchungen.

3. Ergebnisse und Ausblick

Für den Vergleich der Spezifikationen wird eine 12-Bit Prozessorarchitektur in LISA und MaMa nachgebildet. Der Prozessor in der Havard-Architektur hat einen getrennten Programmspeicher mit 24-Bit Instruktionsworten und eine Datenspeicher mit 12-Bit Datenworten. Die Befehle werden in einer 3-stufigen Pipeline abgearbeitet, die jedoch in den Beispielen nicht modelliert wurde. Der Prozessor hat acht verschiedene Befehlsformate und 62 Befehle im Befehlssatz.

Da der komplette Befehlssatz und die Prozessorarchitektur bekannt sind, lässt sich der Prozessor sowohl in MaMa als auch in LISA sehr schnell beschreiben. Es hat sich gezeigt das beide Beschreibungssprachen einen ähnlichen Beschreibungsaufwand verlangen. In der Abbildung 5 wird gezeigt, dass die Beschreibung in MaMa verglichen mit LISA und bei vergleichbarem Beschreibungsstil ca. 68% der Codezeilen benötigt. In MaMa werden neben der zusätzlichen Beschreibung aller Speicherklassen in einer Hierarchie zwei Speicherinstanzen mehr benötigt, um die Simulationssemantik zu definieren.

Vergleich	MaMa - Spezifikation	LISA - Spezifikation
Codezeilen (ohne Kommentare)	**542**	786
Speicherinstanzen	8	6
Anzahl der beschrieben Befehle	64	69
binäre Kodierung	**optional**	fest
Pipeline - Architekturen	nein	**ja**
Simulation	**direkt**	benötigt Assembler, Linker, Simulator
Simulationsdauer (normiert)	x 778	**1**

Abbildung 5: Vergleich LISA und CoMet

Vorteil der LISA-Spezifikation ist die einfachere Beschreibung der Semantik der Befehle. Die Möglichkeit die Befehlsklassen hierarchisch zu organisieren, konnten wir nicht gewinnbringend

umsetzen. Durch die phasenweise Abarbeitung der Befehle und durch die Definition der Platzhalter war ein geringer Mehraufwand in der Beschreibung des Befehlssatzes notwendig. Ein weiterer Vorteil von LISA ist es, Pipeline-Architekturen einfach beschreiben zu können. Für eine Vergleichbare Beschreibung und Simulation wurde diese große Stärke in den Beispielen nicht umgesetzt. In MaMa ist eine die Modellierung von Pipeline-Architekturen bisher nicht möglich. Die Umsetzung dieser Erweiterung ist Gegenstand derzeitiger Untersuchungen.

Vorteile der MaMa-Spezifikation liegen in der einfacheren Simulation, die ohne weitere Zwischenschritte möglich ist. Vor allem der Verzicht einer binären Kodierung der Befehlswörter erleichtert eine schnellere Beschreibung der Befehle. Für eine Synthese wird es zu einem späteren Zeitpunkt auch in MaMa möglich und notwendig eine binäre Codierung der Befehle anzugeben.

In der Simulationszeit liegt ein großer Nachteil der CoMet-Architektur. Der auf hohe Flexibilität ausgelegte konfigurierbare Zwischencode-Simulator kann mit dem LISA-Befehlssatzsimulator nicht mithalten und benötigt in einer Vergleichssimulation 778 Mal so viel Zeit. Da das Assemblerprogramm für die MaMa-Spezifikation auf einem Windows- und für die LISA-Spezifikation auf einer Linux-System simuliert wurden, wurde für die Geschwindigkeitsermittlung ein identisches C-Programm auf beiden Maschinen berechnet und dieser Zeitfaktor zu den Simulationszeiten ins Verhältnis gesetzt. Die Laufzeit der CoMet-Simulation ist verglichen mit der LISA-Simulation extrem hoch. LISA spielt hier den Vorteil einer auf eine Ebene optimierten Binärcodesimulation aus. Der Zwischencodesimulator in CoMet ist bisher nicht auf Laufzeit optimiert, da er so offen und flexibel wie möglich konzipiert wurde, um Zwischencodes auf allen Abstraktionsebenen zu simulieren und das Konzept des compilerzentrierten Entwurfs umzusetzen. Diese Flexibilität wird dadurch erreicht, dass die Daten mittels Strings gespeichert und verarbeitet werden. Diese String-Operationen verursachen den Großteil des Laufzeitunterschieds. Hier ist ein Ansatzpunkt den Zwischencodesimulator einem späteren Profiling zu unterziehen und diese String-Operationen zu minimieren.

Im Allgemeinen hat sich gezeigt, dass beide Sprachen viele ähnliche Elemente aufweisen und haben eine vergleichbaren Aufwand auf dieser Ebene des Entwurfs. Der mit LISA vorgesehene Entwurfsprozess, wie in Abbildung 4a dargestellt, kann somit ebenfalls in CoMet umgesetzt werden. Die Leistungsfähigkeit beider Ansätze lässt sich abschließend jedoch nicht anhand dieser Beispielspezifikation beurteilen, da beiden eine vollständig andere Entwurfskonzepte zu Grunde liegen. Das gilt besonders wenn die Anwendung in C-Code oder einer anderen Hochsprache vorliegt. Hier ist dann der Vergleich der Aufwände und Hindernisse entscheidet in LISA einen C-Compiler und in MaMa die einzelnen Transformationsmodule zu entwerfen bzw. anzupassen.

4. Danksagung

Diese Arbeit wurde von der Investitionsbank des Landes Brandenburg (ILB) im Rahmen des Projekts „Compilerzentrierter Mikroprozessorentwurf“ (CoMet) in Kooperation zwischen der Technischen Universität Cottbus-Senftenberg (BTU) und der Gärtner-Electronic-Design GmbH (GED) unterstützt.

LITERATURVERZEICHNIS

[1] S. Pees, A. Hoffmann, V. Zivojnovic und H. Meyr, „LISA - Machine Description Language for Cycle Accurate Models of Programmable DSP Architectures,“ in *Proc. of ACM/IEEE Design Automation Conference*, 1999.

[2] M. Freericks, „The nML Machine Description Formalism,“ TU Berlin, 1993.

[3] P. Leupers und R. Marwedel, „Retargetable Code Generation based on Structural Processor Description.,“ *Design Automation for Embedded Systems,* pp. 1-36, 1 3 1998.

[4] H. Corporaal und H. J. Mulder, „MOVE: A framework for high-performance processor design,“ *in Proc. of the 1991 ACM/IEEE conference on Supercomputing*, New York, 1991.

[5] G. Hadjiyiannis, P. Russo und S. Devadas, „A Methodology for Accurate Performance Evaluation in Architecture Exploration,“ in s *in Proc. of Design Automation Conference (DAC'99)*, 1999.

[6] M. Freericks, „Generation of Hardware Machine Models from Instruction Set Description,“ in *Proc. of the IEEE Workshop on VLSI Signal Processing*, 1993.

[7] S. KOBAYASHI, Y. TAKEUCHI, A. KITAJIMA und M. IMAI, „Compiler Generation in PEAS-III: an ASIP Development System,“ in *Proc. of International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2001.

[8] A. Halambi, P. Grun, G. Vijay und e. al., „EXPRESSION: A language for Architecture Exploration through Compiler/Simulator Retargetability,“ in *Proc. of Design, Automation and Test in Europe (DATE)*, 1999.

[9] M. Schölzel, „Dissertation: Automatisierter Entwurf anwendungsspezifischer VLIW-Prozessoren,“ 12. Dezember 2006. [Online]. Available: http://opus.kobv.de/btu/volltexte/2006/39/.

[10] K. V. Palem, L. N. Chakrapani und S. Yalamanchili, „A framework for compiler driven design space exploration for embedded system customization,“ in *Proc. of the 9th Asian Computing Science Conference*, 2004.

[11] R. Urban, M. Schölzel und H. Vierhaus, „Compilerzentrierter Mikroprozessorentwurf,“ in *Tagungsband DASS 2013. Dresdner Arbeitstagung Schaltungs- und Systementwurf*, 2013.

[12] R. Urban, M. Schölzel und H. Vierhaus, „Ein konfigurierbarer Zwischencodesimulator zum compilerzentrierten Mikroprozessorentwurf,“ in *Proc. of 16th GI/ITG/GMM Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen" (MBMV).*, 2013.

[13] M. Hohenhauer und R. Leupers, C Compilers for ASIPs - Automatic Compiler Generation with LISA, New York: Springer, 2010, pp. 7-13.

Semi-automatische Generierung von Überdeckungsmetriken mittels methodischer Verifikationsplan Verarbeitung

Christoph Kuznik, Bertrand Defo und Wolfgang Müller
Fakultät für Elektrotechnik, Informatik und Mathematik
Universität Paderborn/C-LAB, Fürstenallee 11, D-33102 Paderborn
{*kuznik*, *bertrand*, *wolfgang*}@c-lab.de

Zusammenfassung

Während Verifikationsmethodiken wie OVM und UVM [3] die einheitliche Strukturdefinition und Beschreibung von Testumgebungen erleichtern und zur Interoperabilität beitragen, ist die (Überdeckungs-)Metrik Implementierung der laut Verifikationsplan zu untersuchenden Funktionalität weiterhin ein fehleranfälliger manueller Arbeitsschritt, für den bisweilen zumeist keine unterstützenden automatisierten Prozesse und Werkzeuge genutzt werden. In diesem Beitrag stellen wir eine Methode zur systematischen semi-automatischen Erzeugung von funktionalen Verifikationsmetriken vor. Dabei wird ausgehend vom textuellen Verifikationsplan ein formalisiertes Zwischenformat (ReqIF) [13] verwendet welches in eine Instanz des *Unified Coverage Interoperability Standard* (UCIS) [2] überführt wird. Über Modell-Verknüpfung wird das mit Referenzen auf die konkrete Implementierung versehene UCIS Modell genutzt, um über Code-Generatoren die Verifikationsmetriken direkt für verschiedene Zielsprachen zu generieren und die Resultate der Metriken zurück in das UCIS Modell zu annotieren. Unsere Methode, die einzelnen Prozesschritte und der Werkzeugprototyp werden anhand eines SystemC Modells für einen Abstandsregeltempomat (ACC) evaluiert.

1. Einleitung

Die fortwährende Steigerung der Komplexität von (eingebetteten) Systemen lässt der funktionalen und nicht-funktionalen Verifikation des Systementwurfs eine zentrale Bedeutung zukommen. Um den Fortschritt des Verifikationsprozesses zu erfassen, welcher zuvor im Verifikationsplan festgelegt wurde, finden verschiedene Metriken Anwendung, so z.B. die funktionalen Überdeckungsmetriken (engl. functional coverage). Der Verifikationsplan definiert in zumeist tabellarischer Form die Eigenschaften der Verifikationsumgebung, die zu verifizierenden Szenarien, die damit verbundenen Eingangsgrößen (Stimuli) sowie die dazu passenden Überdeckungsmetriken. Er definiert also, welche Charakteristika des Entwurfes untersucht werden sollen "*Was soll verifiziert werden?*" und mit welche Verifikationstechniken, wie z.B. Verifikationsmetriken, dies erreicht werden soll "*Wie soll verifiziert werden?*". Zur einheitlichen Definition der Testumgebung und zur Unterstützung des Austausches von Daten über Werkzeuggrenzen hinweg wurden Verifikationsmethodiken wie die Universal Verification Methodology (UVM) [3] eingeführt und über Accellera standardisiert. UVM stellt Basisklassen und Strukturen bereit um möglichst effizient und

modular wiederverwendbare Testumgebungen zu definieren. Die beschriebenen Technologien wurden in erster Linie für die Zielsprachen IEEE-1800 SystemVerilog und IEEE-1647 e entwickelt und implementiert, was jedoch proprietäre Simulatoren voraussetzt. Um die Verifikation mittels funktionaler Überdeckungsmetriken und einer korrespondierenden Verifikationsmethodik auch im IEEE-1666 SystemC Umfeld zu ermöglichen, wurde in früheren Arbeiten die System Verification Methodology (SVM) [14, 15] entwickelt. Als modulare Klassenbibliothek für SystemC erlaubt sie auch eine einfache Integration eigener Werkzeuge, so zum Beispiel der in früheren Arbeiten entwickelten Bibliothek für funktionale Überdeckung im SystemC Referenzsimulator [8, 9]. Die eigentliche Implementierung der Verifikationsmetriken für die jeweilige Zielsprache ist jedoch weiterhin ein zumeist manueller Arbeitsschritt und damit tendenziell langwierig sowie bezogen auf die Intention des Verifikationsplanes stark fehleranfällig. Zur effizienteren Konstruktion der Verifikationsmetriken der Testumgebung stellen wir in diesem Beitrag eine Methode zur systematischen Auswertung des Verifikationsplanes zur semi-automatischen Generierung von Überdeckungsmetriken vor. Unser Ansatz basiert dabei auf:

(1) Methodische Erfassung der Überdeckungsmetrik und Ziele in einem modell-basierten Verifikationsplan, wobei diese jedoch weiterhin unabhängig von der späteren Implementierungssprache der Metrik gehalten werden soll.

(2) Nutzen des *Unified Coverage Interoperability Standard* (UCIS) [2] als Anwendungsspezifisches Zwischenformat für die Überdeckungsmetrik bereits *vor* dem Simulationslauf, modellbasierte Verknüpfung von Entwurf und formalisierten Verifikationsplan zur Anwendung von Metrik Code-Generatoren sowie spätere Rückannotierung der Metrikergebnisse in das UCIS Modell.

Über Techniken der modellbasierten Entwicklung (MDE) kombinieren wir das UCIS Überdeckungsmetrik Modell und ein aus dem Entwurf gewonnenes Modell über Referenzen. Darauf aufbauend können die Überdeckungsmetrik Anteile der Testumgebung über Code-Generatoren generiert werden. Kapitel 2 gibt einen Überblick über die Grundlagen von Accellera UCIS. Darauf aufbauend diskutiert Kapitel 3 unsere Methode und Werkzeugkette im Detail, bevor diese in Kapitel 4 am Beispiel eines Abstandsregeltempomat (ACC) SystemC Entwurfs evaluiert wird. In Kapitel 5 geben wir einen kurzen Überblick über themenverwandte Arbeiten bevor wir die gewonnenen Ergebnisse in Kapitel 6 zusammenfassen und einen Forschungsausblick geben.

2. Unified Coverage Intermediate Format (UCIS)

UCIS [2] wurde entwickelt um den effiziente Austausch und die Weitergabe von Ergebnissen einer Vielzahl von Verifikationsmetriken, sogenannter `coverage producers`, zu ermöglichen. Für diesen Zweck wurde ein Informationsmodell von Accellera definiert, welches verschiedene in der Industriepraxis anzutreffende Metriken abdeckt (Funktional, Formal, Assertions, Code, Zustandsmaschinen) jedoch auch um eigene Metriken erweitert werden kann. Abbildung 1 zeigt die in UCIS berücksichtigten Metriken. `UCIS_CVG_SCOPE` bietet Datenstrukturen zur Erfassung Über `UCIS_GENERIC` können auch selbst-definierte Metrikinformationen im Modell abgelegt werden. Des Weiteren wurde eine standardisierte API zur Arbeit mit UCIS definiert, eine freie Implementierung ist jedoch nicht verfügbar. Insgesamt bietet UCIS damit die Mittel zur Erzeugung (Ansammlung), Integration sowie den Export von verschiedenartiger Verifikationsmetriken über Simulatorgrenzen hinweg. Der offizielle Accellera UCIS v1.0 Standard besteht aus:

- UCIS Spezifikation v1.0, beschreibt das Datenmodell (`coverage scopes`), die berücksichtigten Quellen für Metrikinformationen und dazu spezifizierten API Funktionen.
- UCIS API Header File (.h), für Anwendungsspezifische Anpassungen, z.B. eigene Auswertungsalgorithmen.
- UCIS XML Schema, als formale Definition des XML Austauschformates.

Da der Standard keine Implementierung bereitstellt wurde für unsere Ansätze basierend auf dem XML Schema eine einfache Setzen/Abrufen API generiert.[1]

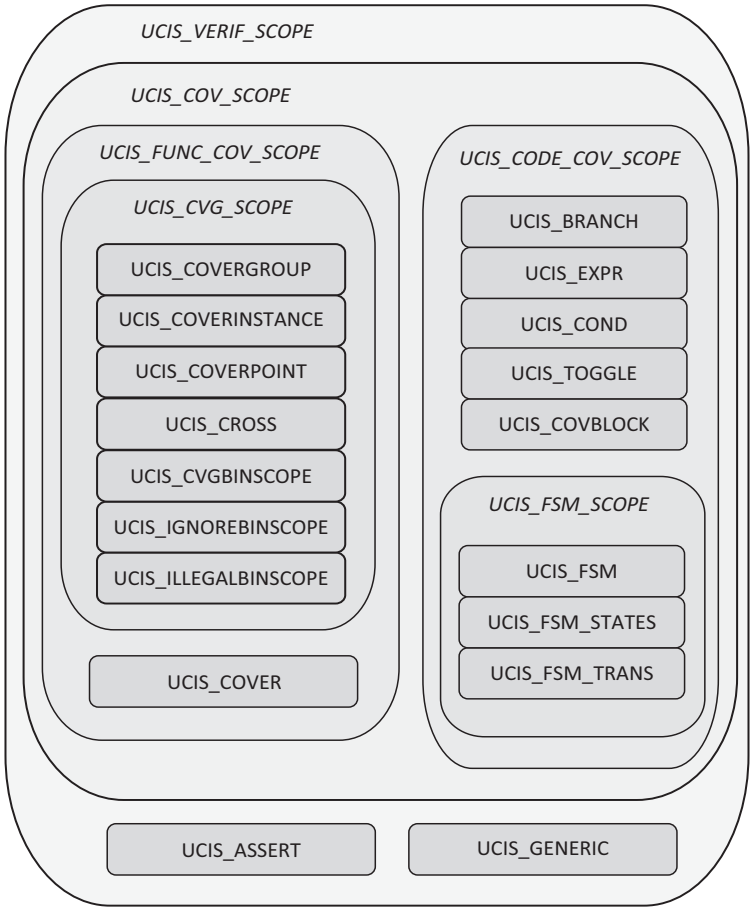


Abbildung 1: Neben funktionalen Überdeckungsmetriken erlauben die UCIS Datenstrukturen auch die Erfassung weiterer `coverage producer` [2].

3. Semi-automatische Generierung von Überdeckungsmetriken mittels methodischer Verifikationsplan Verarbeitung

3.1. Methodik

Die Abbildung 2 zeigt in abstrahierter und vereinfachter Form den typischen Entwurfsfluss beim Systementwurf mit Entwurfsimplementierung und -verifikation als parallele Arbeitsschritte. Basierend auf dem Verifikationsplan werden Testszenarien und Überdeckungsme-

[1] Die genaue Erzeugung des UCIS Metamodelles sowie der UCIS API ist nicht im Fokus dieses Beitrages. Weitere Informationen diesbezüglich und zu Accellera UCIS sind in [10] zu finden.

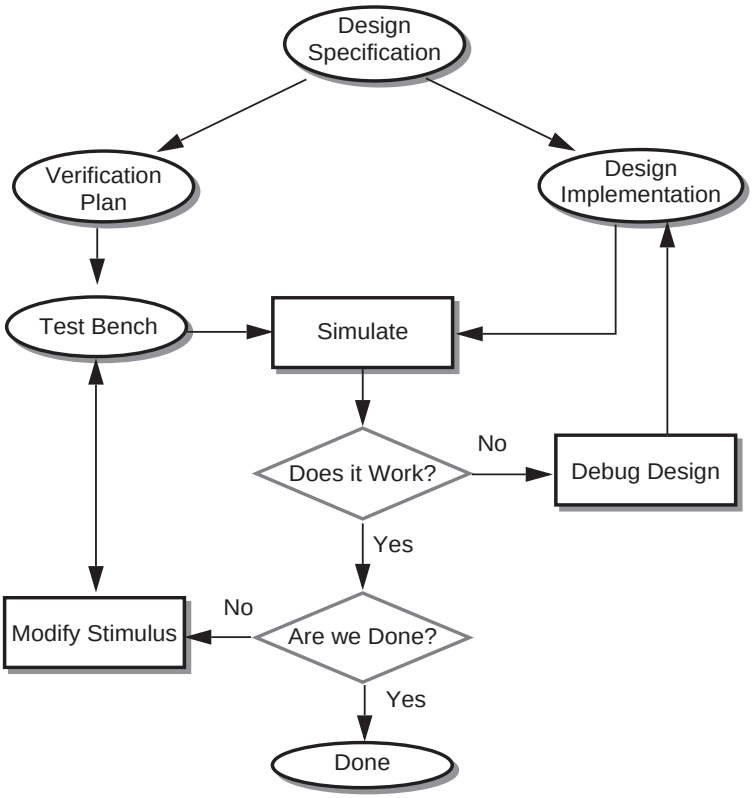


Abbildung 2: Eine vereinfachte Darstellung des Entwurfs- und Verifikationsablaufes [1].

triken definiert und zusammen mit dem Entwurf in Simulationen durch Regression evaluiert. Weist der Entwurf dabei funktionale Mängel auf, muss nachgebessert werden bis die jeweilige Menge an Testszenarien mit befriedigender Abdeckung (`coverage criteria`) fehlerfrei getestet wurde. Die Implementierung der Metrik der laut Verifikationsplan zu untersuchenden Funktionalität in eine Zielsprache ist ein fehleranfälliger, da manueller, Arbeitsschritt, für den bisweilen keine unterstützenden automatisierten Prozesse genutzt werden (siehe Abbildung 3). Um diesen Prozess zu beschleunigen und die Metrikimple-

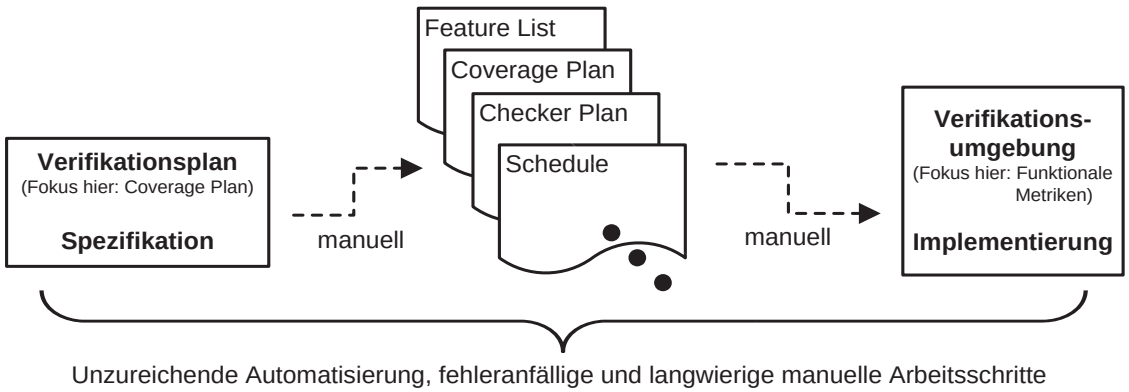


Abbildung 3: Geringe Automatisierung der Metrikimplementierung und hohe Fehleranfälligkeit durch informale manuelle Arbeitsschritte.

mentierungen konsistenter gegenüber den Anforderungen aus dem Verifikationsplan zu gestalten, nutzen wir eine formalere modellbasierte Verarbeitung und Generierung der Überdeckungsmetrik (siehe Abbildung 4).

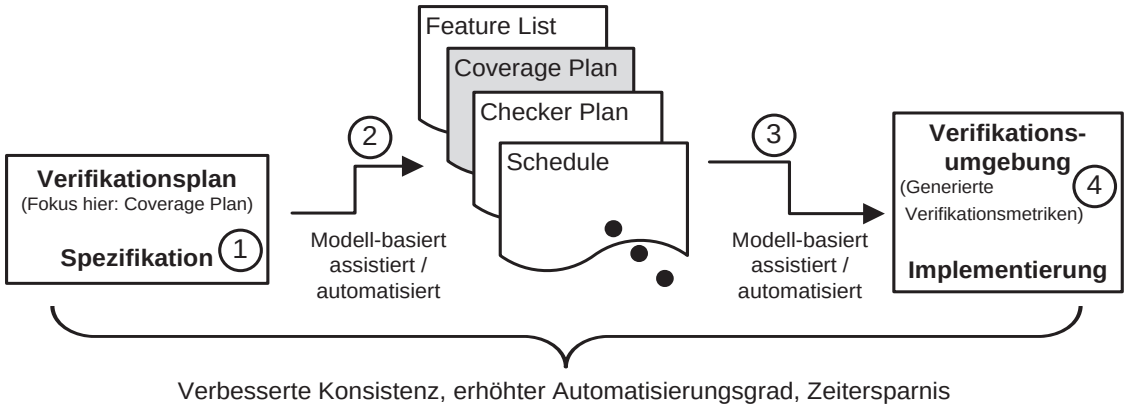


Abbildung 4: Methodische Überdeckungsmetrik Definition (Teil des Verifikationsplanes) verbessert Konsistenz und Automatisierungsgrad der Metrikimplementierung.

Im Detail besteht unsere Methode aus folgenden Prozessschritten:

(1) Erfassung des Metrik Anteiles des Verifikationsplanes in einem spezifischen Metamodell (generiert über OMG ReqIF [13]).

(2) Überführung der Metrikinformation in eine Accellera UCIS Modellinstanz als Metrik-Zwischenformat.

(3) Verknüpfung von Metrik-Zwischenformat und einem Modell des aktuellen Implementierungsstandes des Entwurfes.

(4) Generierung der (Überdeckungs) Metrik Anteile der Testumgebung, basierend auf Informationen der Modellverknüpfung.

Um diese Prozessschritte wirkungsvoll zu integrieren, wurde der zuvor vorgestellte Entwurfsablauf in Abbildung 2 angepasst. Dazu wurden Zwischenschritte eingefügt, die mit den vorgestellten UCIS Modellaktivitäten korrespondieren. Dabei wurden verschiedene Werkzeuge eingesetzt und angepasst um dem Anwender zu assistieren bzw. automatisiert zu verfahren. Abbildung 5 zeigt den integrierten Entwurfsablauf zur semi-automatische Generierung der Überdeckungsmetriken mittels methodischer Verifikationsplan Verarbeitung auf Basis von UCIS. Eine konkrete Anwendung des Entwurfsflusses erfolgt im Abschnitt 4 am Beispiel eines SystemC Modells eines Abstandsregeltempomat (ACC).

4. Durchführung am Beispiel

4.1. Entwurf: Abstandsregeltempomat in SystemC (ACC)

Ein Abstandsregeltempomat (ACC) ist ein Komfortsystem im Bereich der Automobilelektronik welches die Funktionalität des Tempomats um automatische Distanzregelung und damit verbundener Geschwindigkeitsanpassung ergänzt. Abbildung 6 illustriert die Systemfunktionalität. Das SystemC Modell des ACC besteht aus folgenden Modulen (Abb. 7):

(1) *SpeedController* modelliert die geschwindigkeits-adaptive Funktionalität des Tempomaten.

(2) *AccelerationController* modelliert die Standard Tempomat Funktionalität.

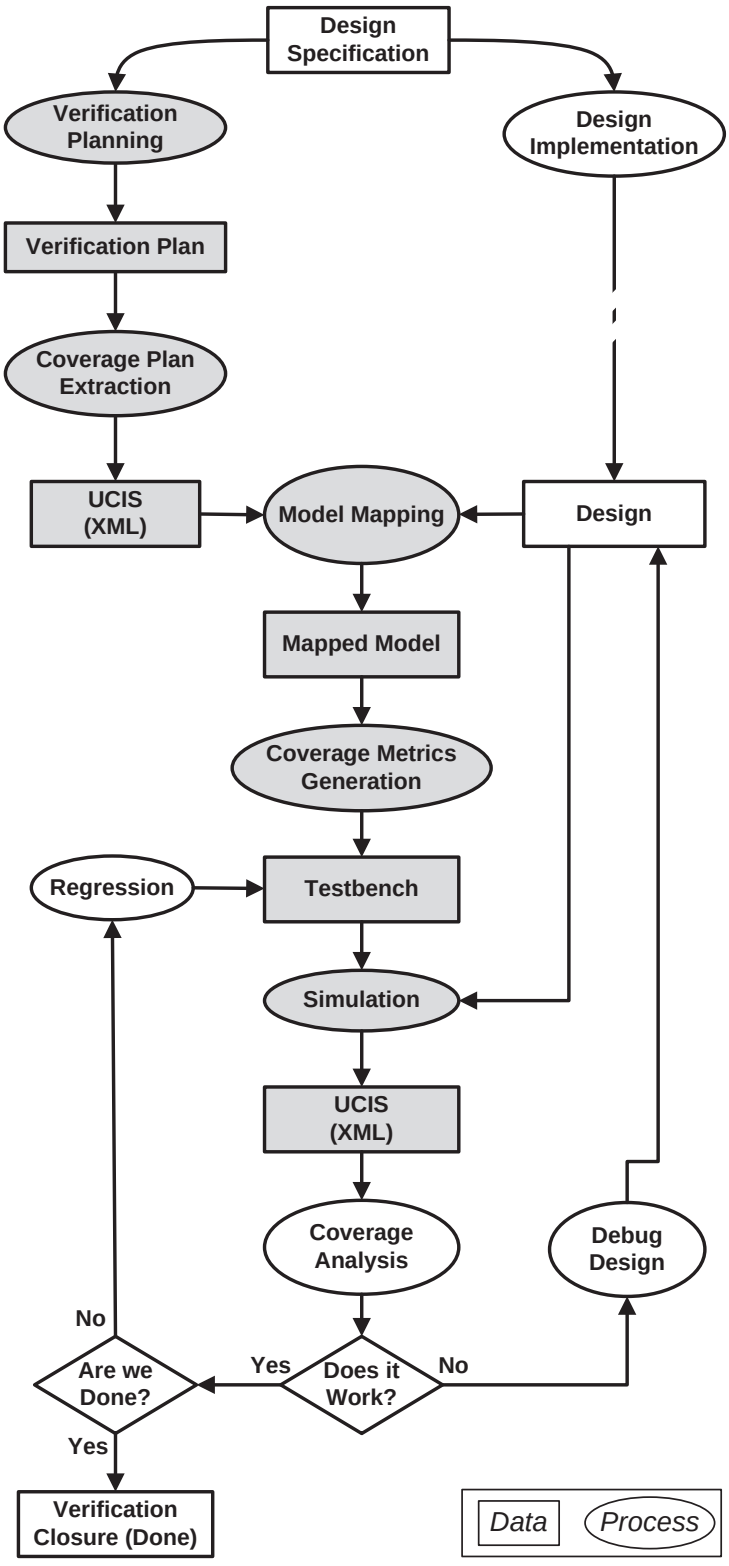


Abbildung 5: Angepasster Entwurfsablauf (von Abbildung 2) zur semi-automatischen Generierung von Überdeckungsmetriken über methodische Verifikationsplan Verarbeitung.

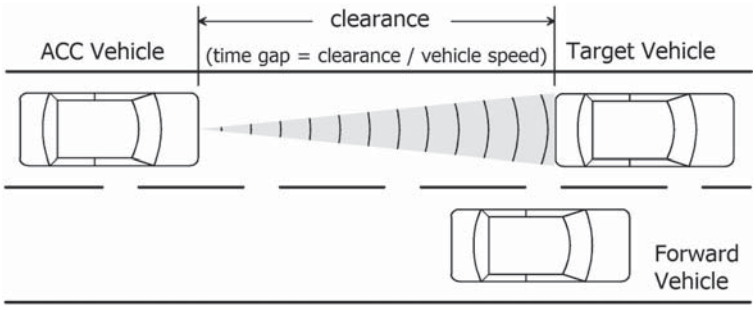


Abbildung 6: Funktionalität eines Abstandsregeltempomat (ACC) Systems.

(3) *EngineController* modelliert ein abstraktes Fahrzeugmodell (Umgebung).

Darüber hinaus besteht das System aus weiteren Modulen, so zum Beispiel einem Radar Modul das vorausfahrende Fahrzeuge in der gleichen Fahrspur detektiert. Da insgesamt mehrere Algorithmen implementiert werden und Datenaustausch zwischen den einzelnen Modulen stattfindet eignet sich dieser Entwurf als Evaluierung für unsere Methode.

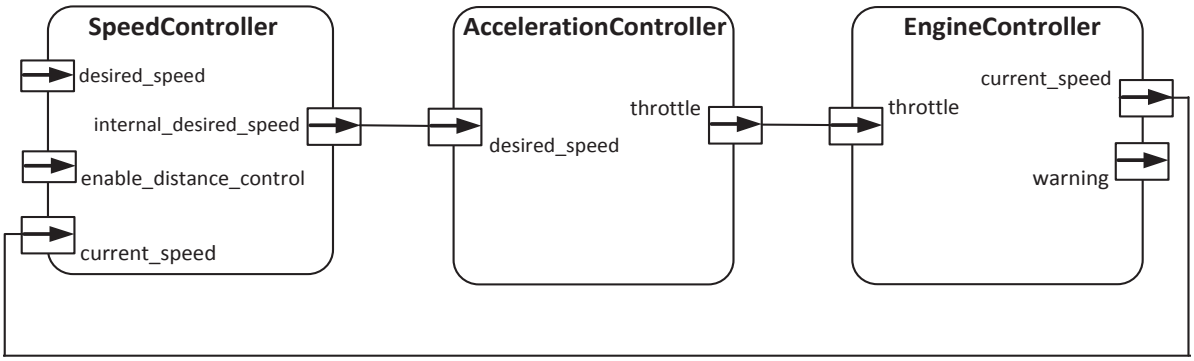


Abbildung 7: Vereinfachtes Strukturmodell des SystemC Abstandsregeltempomaten.

4.2. Annahmen und Vorbedingungen bei Anwendung der Methodik

Für die Anwendung unserer vorgeschlagenen Methode zur semi-automatische Generierung von Überdeckungsmetriken am Beispiel des Abstandsregeltempomat gelten folgende Annahmen:

- Anwendung findet der Accellera UCIS in der Version v1.0.
- Berücksichtigung bei der Metrik-Erfassung und späteren Generierung findet der `UCIS_CVG_SCOPE` Gültigkeitsbereich von UCIS.
- Als Modellierungs- und Simulationsinfrastruktur wird Accellera SystemC Referenz Simulator v2.3 verwendet.
- Ein Modell des aktuellen Entwurfes kann gewonnen werden und für die Verknüpfung mit dem Metrik-Zwischenmodell genutzt werden. Im SystemC Umfeld ist dies zum Beispiel über die Ansätze aus [5, 11] möglich.

Des Weiteren wurde für unsere Untersuchungen eine auf den `UCIS_CVG_SCOPE` Gültigkeitsbereich beschränkte Version der API zum Erstellen und Lesen der UCIS Modelle erstellt.

4.3. Verifikationsplan Erstellung

Die Spezifikation des Beispielsystems berücksichtigt eine Reihe von Nutzungsszenarien, worauf aufbauend der Verifikationplan des Systems entworfen wurde. Zur methodischen Ermittlung der Testfälle fand dabei die Klassifikationsbaummethode (CTM) Anwendung [12]. Für die Implementierungsprachen unabhängige Erfassung der Überdeckungsmetriken nutzen wir das von der Object Management Group (OMG) definierte Requirement Interchange Format (ReqIF) [13], da hier die Information geeignet in der Werkzeugkette erfasst und verarbeitet werden können. Eine tabellarische Ansicht von Überdeckungsmetrik Einträgen im Verifikationsplan des ACC zeigt Tabelle 1. Die Bezeichner korrespondieren zu wichtigen Eingangsgrößen des Systems, so zum Beispiel *desired_speed* und *desired_distance*, welche die gewünschte Geschwindigkeit bzw. die beabsichtigte Distanz spezifizieren. Andere wichtige Kenn- und Eingangsgrößen des ACC Systems sind *current_speed*, *current_distance*, sowie die als Boolean modellierten *enable_acc* und *enable_dist*, die zum Ein- und Ausschalten des Abstandsregeltempomaten bzw. der Distanzregelung verwendet werden. Für jeden Bezeichner wurden die zu verifizierenden Bereiche festgelegt. Zur Veranschaulichung wurde auf eine differenzierte Gewichtung der einzelnen Metrikelemente verzichtet. Das `coverage criteria` liegt jeweils bei 100%. Implementierungsspezifische

Tabelle 1: Auszug der Überdeckungsmetrik für das Module `AccelerationController`.

Name	Range	Type	Weight	Goal
desired_speed	[10:100]	BIN	1	100
current_speed	[0:100]	BIN	1	100
desired_distance	[10:30]	BIN	1	100
current_distance	[0:150]	BIN	1	100
enable_acc	[0:1]	BIN	1	100
enable_dist	[0:1]	BIN	1	100

Details einer konkreten Überdeckungsmetrik wie Konnektierung und Auslösebedingungen werden an diesem Punkt noch nicht betrachtet und erst im späteren Schritt der Verknüpfung von Metrik-Zwischenmodell mit dem Entwurfsmodell referenziert.

4.4. UCIS Metrik-Zwischenmodell und Modell-Verknüpfung

Nach der Erfassung der Überdeckungsmetrik findet nun eine Umsetzung in das UCIS Datenmodell als Zwischenformat statt. Relevante Informationen werden aus dem ReqIF Modell extrahiert und in die UCIS Modellinstanz überführt. Die Abbildung 8 zeigt eine Eclipse Anwendung mit drei Editor Fenstern. Das linke Fenster zeigt das mit Inhalt aus dem Verifikationsplan für den ACC gefüllte UCIS Metrik-Zwischenmodell. Dieses Modell wäre allein nicht für Code-Generierung geeignet da keine Informationen zur konkreten Entwurfsimplementierung existent sind. Das rechte Fenster zeigt das Modell der Implementierung mit konkreter SystemC Module Hierarchie. Im mittleren Fenster können nun die Metrikelemente aus dem UCIS Metrik-Zwischenmodell und dem Entwurfsmodell verknüpft werden. Detailiert betrachtet wird ein drittes Modell instanziert welches jedoch nur Referenzen auf die jeweils anderen beinhaltet. Dieser Schritt erfordert Expertenwissen und kann daher nicht automatisiert werden, jedoch ermöglicht unser Ansatz die Unterstützung des Anwenders.

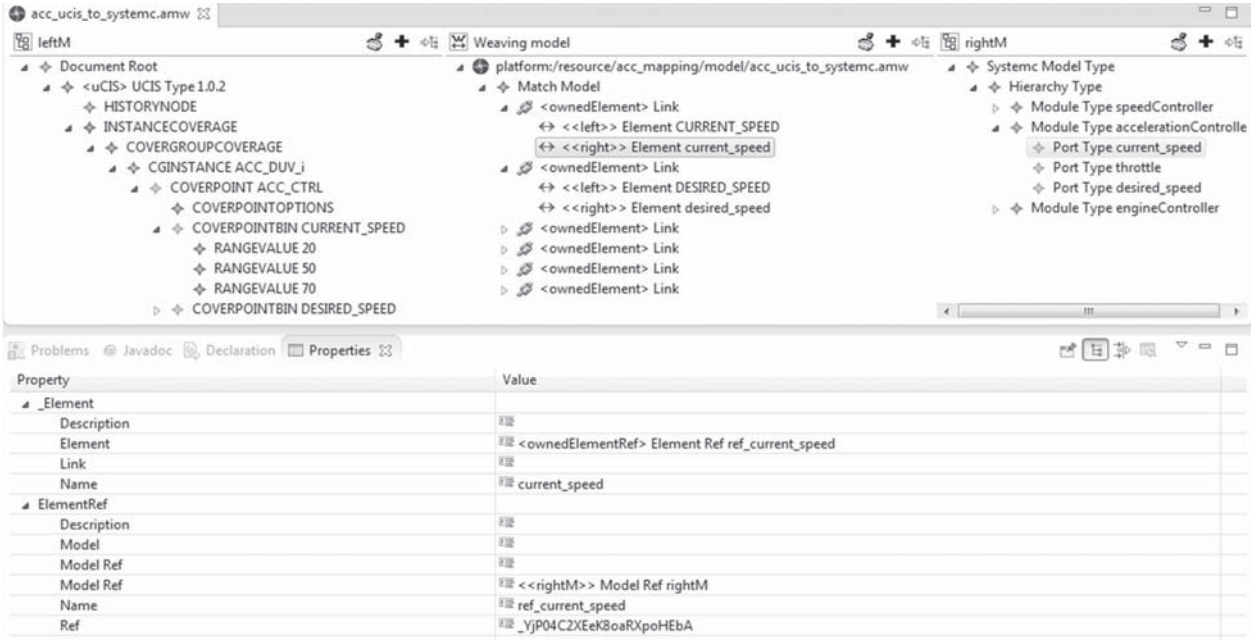


Abbildung 8: Multi-Fenster Editor. Links: UCIS Metrik-Zwischenmodell, Mitte: Modell-Verknüpfung, Rechts: Modell der Implementierung (Entwurf).

4.5. Überdeckungsmetrik Generierung

Die Verknüpfung von Metrik Information aus dem Verifikationsplan und den konkreten Entwurfselementen aus dem Entwurfsmodell erlaubt die automatische Generierung der Metrik in beliebiger Zielsprache (z.B. SystemC, SystemVerilog) über Code-Generatoren. Für unsere Evaluierung findet die Sprache SystemC Anwendung. Hierbei dient die in früheren Arbeiten **(i)** bereitgestellte Bibliothek zur funktionalen Überdeckungsverifikation [9] und **(ii)** die SVM Bibliothek zur Testumgebungs Implementierung [14] als Grundlage. Das Listing 9 zeigt generierten Metrik-Quellcode, spezifisch für unsere funktionale Überdeckung Implementierung. So erzeugen die Zeilen 1-5 zunächst die Instanzen unserer Überdeckungs-Bibliothek bevor dann ab Zeile 7 Metrikelemente angelegt werden. Diese Strukturen werden dann zur Simulationszeit ausgewertet und die Ergebnisse zurück in die vorgesehen Datenstrukturen des UCIS-Zwischenmodells geschrieben.

4.6. Simulation und Regression

Im Simulationsschritt werden die im Verifikationsplan über die Klassifikationsbaum Methode (CTM) erstellten Testszenarien (und damit verbundene Testvektoren) als Stimuli genutzt. Des Weiteren verwenden wir zufallsbasierte Eingangsstimuli die festgelegten Einschränkungen unterliegt. Dabei nutzen wir die Erweiterbarkeit der SVM Bibliothek und verwenden den `MiniSat` basierten `constraint solver` CRAVE für SystemC [7]. Während der Simulation des ACC Systems mit der zuvor beschriebenen Menge an Testvektoren werden die Überdeckungsmetriken sukzessive akkumuliert. Abbildung 10 zeigt einen Auszug der Überdeckungsmetrik nach mehreren Simulationsdurchläufen. Hierbei entsprechen die Treffer den akkumulierten Treffern im jeweiligen Interval des Entwurfselementes. Die Metrikergebnisse können direkt über die Überdeckungsbibliothek ausgewertet werden oder

```
// Init the factory
svm_pFac = svm_Factory::init();

// Acquire existing UCIS model
svm_pFac->setCoverageModel(acc_ucisModel);

// Specify metric, covergroups
cv_pCG = svm_pFac->newCovergroup(this, "ACC_DUV", "ACC_DUV_i");

cv_pCP = svm_pFac->newCoverpoint(cv_pCG, "ACC_CTRL");
cv_pCP->set_weight(1);      // type_options
cv_pCP->set_at_least(100);
cv_pCP->set_goal(90);

(...)

// Specify metric, bin types
cv_pBa = svm_pFac->newBins(cov_pCP, "CURRENT_SPEED", AUTOBINS);
cv_pBa << range(10, 49) << range(50, 69) << range(70, 100);
cv_pBa->connect(current_speed);

cv_pBb = svm_pFac->newBins(cov_pCP, "DESIRED_SPEED", AUTOBINS);
cv_pBb << range(10, 49) << range(50, 69) << range(70, 100);
cv_pBb->connect(desired_speed);

(...)
```

Abbildung 9: Auszug aus der Überdeckungsmetrik für Module `AccelerationController`, spezifisch für Implementierung aus [9].

auch dem UCIS Zwischen-format Modell zurück annotiert werden. Im zweiten Fall kann das UCIS Modell dann als interoperables Metrikinformations Modell genutzt werden.

5. Themenverwandte Arbeiten und Einschränkungen

Datenmodelle zur Speicherung und Integration von Überdeckungsinformation kommen in vielen Entwurfsflüssen der elektronischen Entwurfsautomatisierung vor [6]. Bis zur Definition von Accellera UCIS war jedoch kein interoperables Datenmodell verfügbar. Die Autoren von [4] untersuchen die mögliche Verwertung von UCIS während und nach der Simulation, wie z.B. Steuerung des Testablaufs nach `coverage momentum`. In [16] wird die Verwendung von UCIS für formale Metriken motiviert. Nach unserer Kenntnis und Wissensstand existiert jedoch kein verwandter Ansatz zur Nutzung von UCIS in *frühen* Phasen des Systementwurfes/Verifikationsplan Erstellung, insbesondere nicht als Entwurfsmittel zur späteren Generierung von Metrik Quellcode. Unser Ansatz bedingt die Gewinnung eines Modells des Entwurfes für die weitere modellbasierte Verarbeitung. Limitierungen des Ansatzes ergeben sich damit auch mit den verfügbaren Werkzeugen zur Modellgewinnung und deren Verwertbarkeit bei praxisnahen Entwurfsgrößen und verschiedenen Zielsprachen.

```
BIN:   ACC_SPEED_CTRL:desired_speed::: 20145 Hits
BIN:   ACC_SPEED_CTRL:current_speed::: 21893 Hits
BIN:   ACC_SPEED_CTRL:desired_distance::: 20772 Hits
BIN:   ACC_SPEED_CTRL:current_distance::  23383 Hits
BIN:   ACC_SPEED_CTRL:enable_ac::: 414 Hits
...
```

Abbildung 10: Auszug aus der Überdeckungsmetrik des Modules `ACC_CTRL` nach mehreren Simulationsdurchläufen.

6. Zusammenfassung und Ausblick

In diesem Beitrag haben wir eine Methode eingeführt um ausgehend von Metriken im Verifikationsplan durch formale Erfassung und Verarbeitung eine semi-automatische Generierung von Überdeckungsmetriken für beliebige Zielsprachen zu realisieren. Dabei überführten wie die im Verifikationsplan erfassten Ziele zur Überdeckungsmetrik in ein UCIS-basiertes Zwischenmodell das mit dem Entwurf verknüpft wurde. Der einmalige Aufwand zur Implementierung geeigneter Code-Generatoren innerhalb der modell-basierten Werkzeugkette zahlt sich über höhere Konsistenz gegenüber dem Verifikationsplan, schnellerer Metrik-Implementierung und höherem Automatisierungsgrad aus. Über UCIS als interoperables Austauschformat besteht zudem Anschluss an anschließende Prozesse zur Analyse und Auswertung der Überdeckungsinformationen. In unserem durchgeführten Beispiel wurde allein die Überdeckungsmetrik `UCIS_CVG_SCOPE` berücksichtigt. Mögliche weitere Forschungsaktivitäten sind die Ausweitung der Funktionalität unseres Ansatzes auf zusätzliche Metrikdomänen des UCIS Schema und die verbesserte Integration mit unserer SVM Testumgebungsmethodik Bibliothek.

Danksagungen

Die hier vorgestellten Arbeiten wurden teilweise durch das Bundesministerium für Bildung und Forschung (BMBF) im Rahmen der Förderprojekte EffektiV (01S13022) und ARAMiS (01IS11035) gefördert. Wir bedanken uns bei allen Partnern für die gute Zusammenarbeit.

Literatur

[1] *Questa SIM Users Manual, v10.1a, Chapter 25 Coverage and Verification Management in the UCDB.*

[2] Accellera Organization, Inc. Unified Coverage Interoperability Standard (UCIS), June 2012.

[3] Accellera Organization, Inc. Universal Verification Methodology (UVM), May 2012.

[4] Mentor Graphics Corp. Ahmed Yehia. UCIS Applications: Improving Verification Productivity, Simulation Throughput, and Coverage Closure Process. *Proceeding of Design and Verification Conference (DVCON)*, 2013.

[5] M. D. Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A Generic Model Weaver. *Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles* 2005.

[6] Walter Gude. Questa Verification Management TechTalk.

[7] Finn Haedicke, Hoang M. Le, Daniel Grosse, and Rolf Drechsler. CRAVE: An Advanced Constrained RAndom Verification Environment for SystemC. In *Proceedings of the International Symposium on System-on-Chip 2012. October 11-12, Tampere, Finland*. IEEE, 2012.

[8] C. Kuznik and W. Müller. Verification Closure of SystemC Designs with Functional Coverage. *16th North American SystemC User Group Meeting*, 2011.

[9] Christoph Kuznik and Wolfgang Müller. Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction. *Proceeding of Design and Verification Conference (DVCON)*, 2011.

[10] Christoph Kuznik, Marcio F.S. Oliveira, Bertrand Defo, and Wolfgang Müller. Systematic Application of UCIS to Improve the Automation on Verification Closure. *Proceeding of Design and Verification Conference (DVCON)*, 2013.

[11] Kevin Marquet, Bageshri Karkare, and Matthieu Moy. A Theoretical and Experimental Review of SystemC Front-ends. In Adam Morawiec, Jinnie Hinderscheit, Adam Morawiec, and Jinnie Hinderscheit, editors, *FDL*, pages 124–129. ECSI, Electronic Chips & Systems design Initiative, 2010.

[12] Wolfgang Müller, Alexander Bol, Alexander Krupp, and Ola Lundkvist. Generation of Executable Testbenches from Natural Language Requirement Specifications for Embedded Real-Time Systems. In *DIPES/BICC*, pages 78–89, 2010.

[13] Object Management Group. Requirements Interchange Format (ReqIF) v1.0.1, April 2011.

[14] Marcio F. S. Oliveira, Christoph Kuznik, Wolfgang Mueller, Wolfgang Ecker, and Volkan Esen. A SystemC Library for Advanced TLM Verification. In *Proceeding of Design and Verification Conference (DVCON)*, 2012.

[15] Marcio F.S. Oliveira, Christoph Kuznik, Hoang M. Le, Daniel Grosse, Finn Haedicke, Wolfgang Mueller, Rolf Drechsler, Wolfgang Ecker, and Volkan Esen. The System Verification Methodology for Advanced TLM Verification. CODES+ISSS '12.

[16] Ziyad Hanna Rajeev Ranjan, Ross Weber. On Verification Coverage Metrics in Formal Verification and Speeding Verification Closure with UCIS Coverage Interoperability Standard. *Proceeding of Design and Verification Conference (DVCON)*, 2013.

Increasing Software Reliability by Integrating Formal Verification and Robustness Testing

Stefan Huster, Merdin Macic,
Sebastian Burg, Hanno Eichelberger, Patrick Heckeler,
Jürgen Ruf, Thomas Kropf and Wolfgang Rosenstiel
Computer Engineering Department
University of Tübingen
<lastname>@informatik.uni-tuebingen.de

Abstract

Formal software verification techniques use modular verification strategies to verify the correctness of formally specified programs. Modular verification strategies verify program properties one by one and they postulate all depending properties of each proof. For example, preconditions are postulated when verifying postconditions. This means that if one property cannot be verified, the validity of all depending properties remains unclear. To validate the correctness of unproven properties, latest approaches integrate dynamic testing techniques. But tested code may still contain errors. Current approaches only test unproven properties but not the depending ones. Thereby, modular proofs may depend on defective code. The presented methodology analyses the dependencies between defined properties. Robustness testing is used to test the validity of modular verified properties, in situations in which previously postulated but unproven properties are explicitly violated. This increases the reliability of software, because we test the error handling of depending methods and errors cannot propagate from tested methods to modular verified methods.

1. Introduction

Formal verification techniques prove the compliance of software with its formal specification. Besides complete simulation, formal verification provides the highest reliability. The specification of object-oriented software comprehends mainly preconditions, postconditions and object invariants [Mül02]. In general, a program is regarded as verified if all its postconditions could be verified. Modular verification techniques postulate preconditions and invariants while proving postconditions [Mül02]. This is realised by generating separated proof obligations. A proof obligation[1] comprehends a set of assumptions, a code fragment and a proof goal. The set of assumptions and the goal is expressed as boolean predicate. To verify one proof obligation, one must show, that when postulating the set of assumptions, each possible execution of the defined code fragment fulfils the defined proof goal [Hoa69]. Thereby, dependencies arise between postulated and verified proof obligations which influence the programs correctness.

[1] Also known as "Verification Goal" [MMRV06]

These dependencies become dangerous, if one proof obligation cannot be verified. This invalidates the proof of all depending obligations.
An example of a proof obligation is illustrated in Figure 1. This example contains a method $m1$ with a precondition $p1$ and a postcondition $p2$. This method is called by method $m2$. The contracts $p1$ and $p2$ cause two proof obligations 1 ($PO1$) and 2 ($PO2$). The first proof obligation comprehends no assumptions, code block 1 of method $m2$ as code fragment and the precondition $p1$ as proof goal. This is because the code previous to the call to $m1$ must ensure the precondition $p1$. The second proof obligation postulates the correctness of the precondition and thereby the validity of $PO1$. It contains the code of method $m1$ as code fragment and the postcondition $p2$ as proof goal. If $PO1$ cannot be verified, the validity of $PO2$ is unclear, even if $PO2$ could be verified modularly. This is because method $m1$ might be called with invalid parameters and may violate its postcondition. On this way, the invalid return value may cause an error in code block 2 of method $m2$. Thereby, an error in the code of the unproven $PO1$ propagates to the code of the modular verified $PO2$.

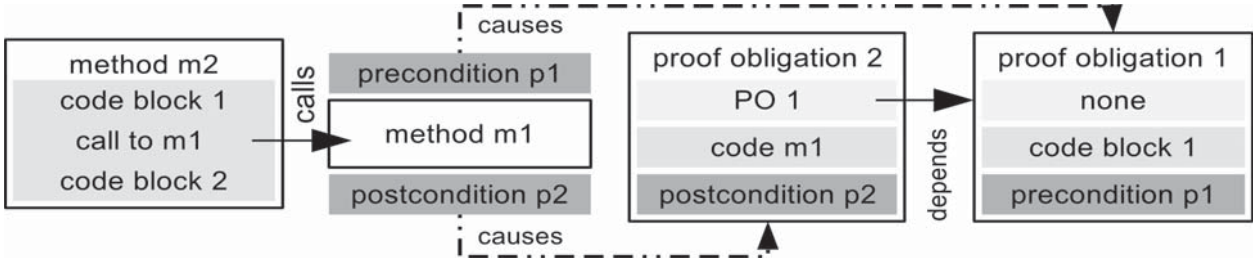


Figure 1: Dependencies between proof obligations

Latest approaches integrate dynamic test techniques (Section 3). They create test suites for code fragments of unverified proof obligations and validate the compliance with defined proof goals. But without complete simulation, tests cannot prove the absence of errors. However, a complete simulation requires to apply all possible input vectors, which is in most cases not applicable. Instead code fragments may depend untested on unproven proof obligations. Therefore, tested code remains as a source of possible error and all depending modular proofs become invalid, because errors may propagate from tested to modular verified code.
The presented methodology uses robustness testing to overcome this risk. In general, robustness testing is used to test the programs behaviour in presence of invalid input parameters. The presented methodology generates proof obligations for modular verification techniques and analyses dependencies between generated proof obligations. If one proof obligation cannot be verified, the obligation itself is tested. Furthermore, depending obligations are tested for their robustness against possible failures in unproven postulated proof obligations. Therefore we prepare test cases which violate explicitly assumed unverified proof goals. These test cases can be applied additional to different test cases. Applied to the example of Figure 1: If the proof obligation $PO1$ cannot be verified, the method $m1$ will be tested with input parameters not fulfilling the precondition $p1$. Compared to the state of the art, our new approach increases the software reliability. This is because modular verification techniques can be applied without the risk of untested failures in modular verified properties, caused by unproven postulated dependencies.

The remainder of this paper is structured as follows: Our methodology is described in Section 2. Related work is listed in Section 3. Results of our experiments are presented in Section 4. We conclude and introduce future work in Section 5.

2. Methodology

The presented methodology combines Formal Verification and Robustness Testing and is abbreviated as FVRT. It is illustrated in Figure 2 and comprehends three steps. In the first step, the Proof Obligation Generator creates a set of Proof Obligations (PO) and Proof Obligation Groups. The Proof Obligation Group of one proof obligation ρ contains all proof obligations depending on ρ. This step is described in detail in Section 2.2. The proof obligations are sent one by one to the formal verification back-end which separates the set of obligations into verified (V POS) and unverified (UV POs) proof obligations. In the second step, each unverified proof obligation is send to the Test Obligation Generator (TOG). A Test Obligation defines for one test case the valid domain of the test vector, the code to test and the test goal. The test goal and test vector are expressed as Boolean predicates. The TOG creates test obligations for the unverified proof obligation and all proof obligations within the corresponding proof obligation group. The generation of test obligations is described in detail in Section 2.3. In the third step, each created test obligation is send to the Test Case Preperator, which generates test case (TC) stubs. This step is described in detail in 2.4.

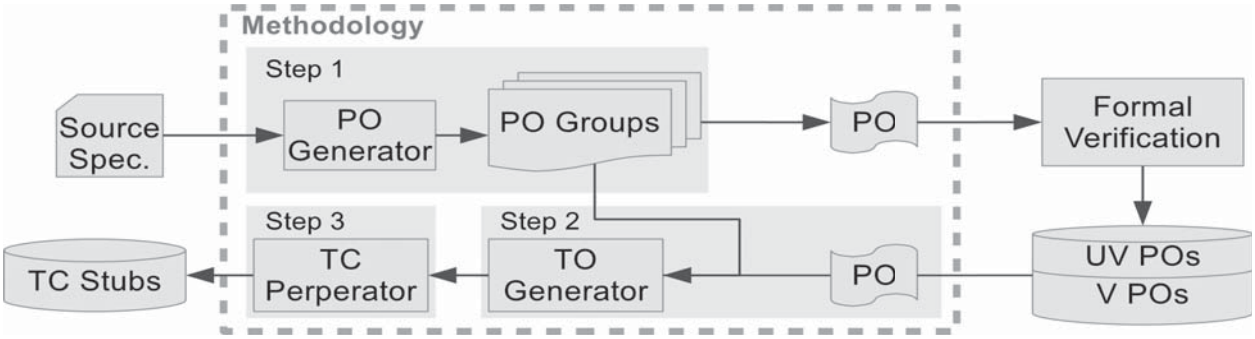


Figure 2: Illustration of our methodology and its environment

2.1. Input

The input to our methodology is the source code and the formal specification of the program under test. The specification of object oriented software bases upon preconditions, postconditions and invariants. In addition, it is common practice to define code assertions to express assumptions about the value of a variable at run time. We use the term "contract" to reference all different specification types listed above.

Definition (Input: Program Under Test, Contracts). The input is a program under test T. The program T comprehends a set of classes K_T. Each class $k \in K_T$ contains a set of methods $m \in M_k$. A method m comprehends a set of statements S_m. For a given statement (set) s, m_s refers to the method m containing s. Each statement can be addressed by S_i $0 \leq i < |S|$. The code fragment between the statement S_i and S_j is referenced by the interval $S_{[i,j]}$. A contract γ is a tuple (t, g) and comprehends a type t and a Boolean predicate g. The type of a contract is either $t = pre$ for preconditions, $t = pos$ for postconditions, $t = inv$ for invariants or $t = ass$ for assertions. Assertions $\gamma \mid t = ass$ comprehend additionally the position p_γ of the assertion within the corresponding method m. The set of contracts of one method m, class k or code fragment $S_{[i,j]}$ is defined by $\Gamma(m|k|S_{[i,j]})$. The set of contracts of one given type is specified by $\Gamma_{\{pre|pos|ass\}}$.

2.2. Step 1: Generating Proof Obligation Groups

Proof obligations are created for each method m_k based on $\Gamma(m)$ and $\Gamma(k)$.

Definition (Proof Obligation (Groups)). A proof obligation $\rho = (t, A, S, g, p)$ is a tuple and it comprehends a type t_ρ, a set of postulated proof obligation A_ρ, a code fragment S_ρ, a proof goal g_ρ, which is expressed as Boolean predicate and a position p_ρ. A proof obligation is verified if g_ρ is always true, after S_ρ has been evaluated assuming the validity of each $a \in A_\rho$. The type of a proof obligations equals the type of the original contract. The position of a proof obligation defines when g_ρ in S_ρ must be valid. The function $\phi(\gamma)$ returns the set of proof obligations, representing the contract γ. The set of all proof obligations is denoted by P. The proof obligation group $\langle\rho\rangle$ of one proof obligation ρ comprehends ρ and all other proof obligations $\dot{\rho} \in P$ which depend on ρ: $\langle\rho\rangle = \rho \cup_{\dot{\rho}}^{P} \{\dot{\rho} | \rho \in A_{\dot{\rho}}\}$.

In order to describe the generation process of proof obligations, we need to introduce some functions: The function $MethodsCalled(S)$ returns the set of called methods in S. The function $MethodCalls(m, S)$ returns the position i of each statement s calling m. Furthermore, we define the context $[\![\gamma]\!]_S$ which translates the variables used within the contract γ to the accessible variable space of the code block S. The translation replaces the variables names in γ by the variable name used at the position of the generated proof obligation. For example, if a precondition of a function $func(int\ a)$ requires $\gamma = a > 0$ and $func$ is called in S with $func(b)$, the context is: $[\![\gamma]\!]_S = b > 0$.

For each precondition $\gamma_{ass} \in \Gamma_{pre}(\hat{m})$ and assertion $\gamma_{ass} \in \Gamma_{ass}(\hat{m})$, we create following proof obligations:

$$\phi(\gamma_{pre}) = \bigcup \forall_k^{K_T} \forall_m^{M_k} \forall_i^{MethodCalls(\hat{m}, S_m)} (pre, A(\gamma_{pre}, S_{\hat{m}[0,i]}), [\![g_{\gamma_{pre}}]\!]_{S_{\hat{m}[0,i]}}, i) \quad (1)$$

$$\phi(\gamma_{ass}) = (ass, A(\gamma_{ass}, S_{\hat{m}[0,p_{\gamma_{ass}}]}), S_{\hat{m}[0,p_{\gamma_{ass}}]}, g_{\gamma_{ass}}, p_{\gamma_{ass}}) \quad (2)$$

In both cases, we create one proof obligation for each code fragment previous to a method call to $\hat{m}$ or respectively previous to the assertion. In both cases, we can use the same search for postulated proof obligations $A(\gamma, S)$. The search is described in Section 2.2.1. Each postcondition $\gamma_{pos} \in \Gamma_{pos}(\hat{m})$ causes following proof obligations:

$$\phi(\gamma_{pos}) = (pos, \Gamma_{pre}(\hat{m}), S_{\hat{m}}, g_{\gamma_{pos}}, |S_{\hat{m}}| - 1) \quad (3)$$

The generation of proof obligations representing invariants is much more complicated. Different methodologies exist to analyse object invariants [MPHL06]. Assertions differ from invariants with respect to their behaviour in inheritance. In addition, invariants may be temporarily invalid without affecting the correctness of the program, e.g. when evaluating re-entrance recursive methods. Our methodology searches for possible execution paths between code positions violating an invariant and other positions depending on that invariant. Then, we create proof obligations for that invariant along the path to verify that the invariant is restored, before a depending code fragment is reached. A formal detailed description of this analysis is far beyond the scope of this paper. However, this paper addresses primary how methods needs to be tested which depends on unproven obligations. The corresponding methodology can be defined independent from the formal definition of the methodology used to create proof obligations for invariants.

2.2.1. Searching Postulated Proof Obligations

For proof obligations representing preconditions or assertions, the set of postulated proof obligations is analysed in the same way. These proof obligations depend on contracts, referencing variables, which influence (transitivity) the value of those variables, referenced by their own proof goal.

Considering the following example: $b \leftarrow func(\ldots); c \leftarrow b/2; d \leftarrow b + 1; assert(d > 0)$. The function $func$ ensures a return value greater than zero. Its return value is stored in b and is used within the value of the assignment to variable c. The variable c is used within the value of the assignment to variable d. Finally, an assertion requires that the value of d is greater than zero. This assertion references the variable d. Its value is directly influenced by the value of variable c and transitively by the value of b. The value of b is limited through the postcondition of function $func$. Therefore, the proof obligation representing the assertion $d > 0$ depends on the postcondition of $func$.
This search algorithm is listed in Algorithm 1 on page 6 and is described below: The function $A(g, S)$ takes two arguments and returns the set of affiliated proof obligations. The first argument g refers to the proof goal g of the proof obligation, whose affiliated obligations should be searched. The second argument S represents the code fragment to analyse. For reasons of simplicity, we assume for the listed algorithm that S is already represented in static single assignment form. We search for variables, influencing those variables, referenced in g. Therefore, we extract the set of referenced variables, by using the function $Vars(g)$. We call these variables "relevant" and they are stored in V_{rel}. In line 4, we start to analyse the given code block bottom up. The lines 5 to 9 search for assertions, limiting the value of any relevant variable. The proof obligation of each found assertion is added to set of affiliated proof obligations P_{rel}. The lines 10-21 search for additional directly and transitivity relevant variables. A variable becomes relevant if its value influences the value of one relevant variable. Therefore, we analyse any kind of assignment to any relevant variable. The function $AssignedVars$ returns the set of variables which are referenced on the left side of any assignment in S_i. The function $AssignedValue$ returns the assigned value. In line 14, all variables referenced on the right side of the assignment (s) are added to the set of relevant variables. On this way the search algorithm considers transitively relevant variables, like the variable b within the example above. Furthermore, we search for method calls in s, because then the value of the assigned relevant variable depends on the postcondition of that method. The proof obligations of affiliated postconditions are added in line 17 to the set of affiliated proof obligations.

2.3. Step 2: Generating Test Obligations

Test obligations define the environment of one test case, including the domain of definition of input parameters and the code and property that should be tested.

Definition (Test Obligation). A test obligation $\tau = (L, m, A)$ is a tuple and comprehends a set of constrains L_τ, the method to test m_τ and a set of test goals A_τ. The constrains L_τ define the domain of definition for the test input parameter. They are expressed as Boolean predicate. Each test goal $a = (g, p) \in A_\tau$ is a tuple and comprehend the actual goal, expressed as Boolean predicate, and its position p in S_{m_τ}. The corresponding test case succeeds if all assertions A_τ evaluate to true when executing m_τ.

The set of test obligations corresponding to one proof obligation ρ is created by the function $\varphi(\rho)$. It is called for each ρ which could not have been formally verified. For reasons of

Algorithm 1 Algorithm to search affiliated proof obligations

function A(g,S)
  $P_{rel} \leftarrow \emptyset$ ▷ The set of affiliated proof obligations
  $V_{ref} \leftarrow Vars(g)$ ▷ Reads variables referenced within the analysed contract
  for $i \leftarrow |S| - 1 \mapsto 0$ **do** ▷ Analysis the given code fragment S bottom up
    for all $\dot{g} \in \Gamma_{ass}(S_{[i]})$ **do** ▷ Analysis related assertions
      if $Vars(g_{\dot{g}}) \cap V_{rel} \neq \emptyset$ **then**
        $P_{rel} \leftarrow P_{rel} \cup \phi(\dot{g})$
      end if
    end for
    $V_{set} \leftarrow AssignedVars(S_{[i]})$ ▷ Searches for assignments to relevant variables
    if $V_{set} \cap V_{rel} \neq \emptyset$ **then**
      for all $\dot{v} \in V_{set}$ **do**
        $s \leftarrow AssignedValue(\dot{v}, S_{[i]})$
        $V_{rel} \leftarrow V_{rel} \cup Vars(s)$ ▷ Adds transitively relevant variables
        $M_{ref} \leftarrow MethodCalls(s)$
        for all $\dot{m} \in M_{ref}$ **do**
          $P_{rel} \leftarrow P_{rel} \cup \phi(\Gamma_{pos}(m))$ ▷ Adds relevant postconditions to P_{rel}
        end for
      end for
    end if
  end for
  return P_{rel}
end function

compactness we refer to m_{S_ρ} by m_ρ and to $[\![\bullet]\!]_{m_{\rho_{[p_\rho]}}}$ by $[\![\bullet]\!]$. The bullet $\bullet$ represents any boolean expression:

$$\varphi(\rho) = (\Gamma_{pre}(m_\rho), m_\rho, (g_\rho, p_\rho)) + TO \tag{4}$$

$$TO = \bigcup_{\dot{\rho}}^{\langle\rho\rangle\setminus\rho} \begin{cases} ((\Gamma_{pre}(m_{\dot{\rho}}) \setminus g_\rho) + \neg g_\rho, m_\rho, A(\rho, \dot{\rho})) & , t_\rho = pre) \\ (t_{\dot{\rho}}, \Gamma_{pre}(m_{\dot{\rho}}), m_{\dot{\rho}}, A(\rho, \dot{\rho})), & ,\text{else} \end{cases} \tag{5}$$

$$A(\rho, \dot{\rho}) = \bigcup_{\ddot{\rho}}^{\ddot{\rho} \in A_{\dot{\rho}} \setminus \rho} (g_{\ddot{\rho}}, p_{\ddot{\rho}}) + \begin{cases} \emptyset & t_\rho = pre \\ ([\![\neg g_\rho]\!], p_\rho) & t_\rho = ass \\ \bigcup_i^{MethodCalls(m_\rho, S_{m_{\dot{\rho}}})} ([\![\neg g_\rho]\!], i) & t_\rho = pos \end{cases} \tag{6}$$

The set of test obligations corresponding to one unverified proof obligation comprehends one test obligation for the unproven proof obligation and robustness tests obligations for all depending proof obligations. The unproven proof obligation is represented as test obligation in Formula 4 by the first summand. This test obligation defines a test case for the method of the unproven obligation and observes the unproven goal. The set of constrains is given by the set of preconditions of the tested method. The robustness test obligations are represented by formula 5. We add one test obligation for each depending proof obligation, testing the method of each proof obligation. If the unproven proof obligation represents a precondition of the tested method, the first case in Formula 6, adds the negated goal to the set of input constrains. In this way, the corresponding test case will test the methods robustness against malicious

input parameters. The second case in formula 6 adds the unproven proof goal negated to the set of test goals. Thereby, the corresponding robustness test case will only succeed if the corresponding postulated contract is violated.

2.4. Step 3: Preparing Test Cases

Preparing test cases means mainly to generate code for Unit Test stubs. Standard Unit Tests call the method under test and compare the methods return value and effects with a previous defined expectation. But standard Unit Tests are insufficient for this approach because we need to add and to modify defined conditions within the methods source code.

Definition (Reimplementation). A reimplementation $m'(\dot{m})$ of a method $\dot{m}$ is a copy of method $\dot{m}$, with the same return type and accepting the same input parameters. A statement s is added at position i to m' by $m'[i] = s$. The position $i = 0$ refers to the first position in m' and $i = |m'| - 1$ to the last position. An empty reimplementation $m'(\dot{m})[*] = \emptyset$ is an empty method with same input parameters and return type as $\dot{m}$.

A test obligation τ is prepared by adding each $l \in L_\tau$ and $a \in A$ as assertion[2] to the reimplementation $m'(m_\tau)$. The prepared test case can be applied by the used by adding corresponding test vectors. The input values can be selected according to predetermined coverage criteria to be met. However, the assertions added by this methodology ensure that the described robustness criteria will be fulfilled by at least on test vector. The test itself is than applied to the reimplementation m'. The input constrains $l \in L_\tau$ are added to the begin of $m'[0] = l$ and the assertions $a \in A_\tau$ to their position $m'[p_a] = g_a$.

2.5. Appliance

The presented methodology combines a static analysis and dynamic robustness tests. The static analysis must be applied, when ever any code fragment has been changed. This is the same for any other formal verification methodology, because every code change may cause additional dependencies to different code fragments or may change the verification results of the corresponding proof obligations. Dynamic robustness tests are created for methods depending on unverified proof obligations. They must be re-evaluated only if the corresponding method has been changed. This is the same for any other dynamic test methodology. Therefore, the presented approach does not cause large overhead, compared to other formal and dynamic approaches, when using during a continues development process.

3. Related Work

This approach bases on basic theories about deductive software verification [Hoa69, Flo67, RC90]. These theories are also used by related proof obligation generators and verification frameworks including Spec# [BLS05] and Java/ESC [FLL+02].

Spec# is a verification framework for .NET languages and was developed by Microsoft. C# programs can be annotated using pre- and postconidtions as well as invariants. Spec# generates proof obligations according to the defined program annotations and transforms the C# programs into the Boogie [BCD+06] verification language. The theorem prover Z3[DMB08] is used as verification back-end.

[2] e.g.as JUnit.Assert when testing Java code

Java / ESC is a static verification framework for Java-programs. The Java / ESC frameworks uses special comments to annotate the program and includes a corresponding proof obligation generator. It supports a set of different theorem provers as verification back-end including Z3.

Our approach integrates static formal verification methodologies, as listed above, with dynamic testing. Most related approaches are those addressing the validation of object oriented software:

Christakis et al. [CMW12] use incrementally different static checkers during the formal verification phase. The dynamic test phase is used to test unverified proof obligations. The Dy'Ta approach [GTXT11] analyse potential defects during its static phase, using the Code Contracts tool [FL11]. The dynamic test phase is used to confirm these potential defects. Both approaches use Pex [TDH08] during their dynamic test phase. Pex is a Microsoft tool for white box test generation for .NET languages. It performs path-bounded model-checking by repeatedly executing the program and solving constraint systems to generate parametrised unit tests, achieving high code coverage. Therefore, Pex integrates the Z3 [DMB08] solver.

Check 'n' Crash [CS05] combines ESC/Java [FLL$^+$02] and JCrasher [CS04]. ESC/Java is used during the static phase to detect possible software defects. Then JCrasher is used as post-processor to analyse if emitted errors really exist. Therefore, JCrasher generates a set of JUnit tests, testing disputable methods under random data. Later, this approach was integrated with Daikon in the DSD-Crasher tool [CSX08]. The DSD-Crasher adds another dynamic test phase at the front of the Check 'n' Crash validation pipeline. This additional phase is used to infer the intended behaviour of the tested program.

Huster et al. [HHR$^+$13] uses a formal verification back-end to separate between verified and unverified proof obligations. Unverified proof obligations are dynamical tested. A control flow analysis is used to separate between relevant and not irrelevant control flows. Relevant code fragments are those fragments which may influence the value of the unverified proof obligation. Test cases to test relevant code fragments are prepared and must be manually completed by adding corresponding test vectors. The test case execution itself is observed and automatically controlled to execute only relevant control flows.

However, no other approach known to the authors of this paper integrates robustness testing to generate test cases to test the software's behaviour in situations in which unproven but postulated proof conditions fail.

4. Case Study

We demonstrate the presented methodology and its benefits using the example in Listing 1. The industrial background of this example is the handling of a CNC-machine of an aluminium bar. The CNC-program of a bar contains a set of works, e.g. drills or millings. To handle the CNC-program, the bar must be clamped on the machine. Therefore, we need to calculate positions for each clamp. The method $CheckClampSit$ (m_c) takes three arguments and checks if the distance between a work and its closest clamp do not exceeds $maxDist$. The set of works and clamps are represented as $Dictionary$ storing the id and the x-position of each clamp and work. The function iterates the work list and search for each work one clamp whose position does not exceed the maximum allowed distance. Therefore, it passes the work and clamp positions to the method $SearchBC$ (m_s). If such a clamp could be found, m_c removes the corresponding work, in line 9 of Listing 1, from the list. If the list is empty at the end of the iteration, the passed clamp situation is valid.

```
1 class ClampSituation {
2  public bool CheckClampSit(Dictionary<int,int> clamps,Dictionary<int,
   int> works, int maxDist) {
3   Contract.Requires(maxDist>0);
4    foreach (int workId in works.Keys) {
5    int workPos = works[workId];
6    int bcId = SearchBC(clamps, workPos, maxDist);
7    if (bestClampId >= 0) {
8     Contract.Assert(Math.Abs(clamps[bcId] - workPos) <= maxDist);
9     works.Remove(workId);
10   }
11  }
12  return works.Count == 0;
13 }
14 public int SearchBC(Dictionary<int,int> cPos, int wPos, int maxDist)
15 {
16  Contract.Requires(maxDist > 0);
17  Contract.Ensures(Contract.Result<int>() == -1 || (cPos.ContainsKey(
    Contract.Result<int>()) && Math.Abs(cPos[Contract.Result<int>()]-
    wPos) <= maxDist ));
18  foreach (KeyValuePair<int,int> clamp in cPos) {
19   int clampId = clamp.Key; int cPos = clamp.Value;
20   if (cPos - wPos <= maxDist)
21   { return clampId; }
22  }
23  return -1;
24 }
25 }
```

Listing 1: Example for a program under test

This expected behaviour is specified by five contracts. Most interesting is the postcondition of m_s and the assertion of m_c in line 8 of Listing 1. These three contracts ensure that for each work one clamp could be found which is close enough to that work. In the following sections, we describe how these contracts are validated using our methodology and latest approaches.

4.1. Step 1: Generating and Verifying Proof Obligations

The assertion γ_1 (line 8) and the preconditions γ_2 (line 3), γ_3 (line 16) and the postcondition γ_4 (line 17) cause following proof obligations: $\phi(\gamma_1) = \rho_1 = (ass, \{\rho_3\}, S_{m_c[4-7]}, \gamma_1)$, $\phi(\gamma_3) = \rho_2 = (pre, \emptyset, S_{m_c[3-5]}, (maxDist > 0))$, $\phi(\gamma_4) = \rho_3 = (post, \{\rho_2\}, S_{m_s}, \gamma_4)$. The notation $m[i-j]$ refers to the code fragment of method m between the absolute line numbers i and j. The absolute line numbers are printed on the left side of Listing 1. We have used the Microsoft tool CodeContracts[3] to verify the example. This is because other approaches like Key or JML does not support language features like Java templates or container classes. The verification result is listed in Table 1. CodeContracts has been able to verify ρ_1 and ρ_2, but not ρ_3.

[3]Version: 1.5.60502.11 Warning-Level: High

Verification				Testing		
	ρ_1	ρ_2	ρ_3		m_c	m_s
CodeContrats	✓	✓	✗	DyTa	✗	✓
				VeriTatest	✗	✓
				Christakis	✗	✓
				FVRT	✓	✓

Table 1: Verification and Test Results.
FVRT refers to our approach: "Formal Verification and Robustness Testing"

4.2. Step 2: Applying Dynamic Test Techniques

We use formula 4 to generate the proof obligation group of ρ_3 to test the unverified and depending obligations: $\langle\rho_3\rangle = \{\rho_3, \rho_1\}$. The corresponding test obligations are: $\varphi(\rho_3) = (\{\gamma_3\}, m_s, \{(\gamma_3, |m_c| - 1)\})$, $\varphi(\rho_1) = (\{\gamma_2\}, m_c, \{((\neg(bcId == -1 \parallel (clamps.ContainsKey(bcId)\ Math.Abs(clamps[bcId] - workPos) <= maxDist)))), m_{c[7]})\})$. We compare our work with three different approaches integrating formal verification and dynamic test techniques: DyTa, VeriTatest and the approach of Christakis et al. [CMW12]. As illustrated in Table 1, our approach is the only one which also prepares test cases for method m_c. All other approaches test only method m_s.

4.3. Results

As shown in Table 1, current formal verification techniques cannot verify all contracts of the example in Listing 1. Therefore, test techniques are applied to test the unproven postcondition of m_s. Testing m_s with following test manual selected test vectors achieves 100% line and branch coverage: $TC1: \ cPos = \{(1, 110), \}, wPos = 100, maxDist = 20$, $TC2: \ cPos = \{(1, 1000)\}, wPos = 100, maxDist = 20$. But an error occurs, if following test case is applied: $TC1: \ cPos = \{(1, 10), (2, 90)\}, wPos = 100, maxDist = 20$. Then, the function returns 1, even if the distance to that clamp exceeds the $maxDist$-value. This cause a failed assertion in m_c, even if this method has been modular verified. No other approach combining formal verification techniques and dynamic testing analyses this risk and creates corresponding test cases. In the example, the developer could easily fix the error in m_s. But in more complicated and realistic examples, this may not be possible as easy. In this cases, our approach helps to test for robustness against failed assumptions and to add corresponding checks and error handling.

5. Conclusion and Future Work

We have presented an approach to integrate modular formal verification and robustness testing techniques. Unverified contracts may cause errors in program sections, which have been modular verified. Our methodology analyses the dependencies between defined contracts. We test if modular verified contracts still hold, when unverified contracts fail. This causes a higher software reliability, because errors cannot propagate untested from unverified to modular verified methods. The methodology was demonstrated on an example with an industrial background. In this example, one postconditions could not be formally verified. The example has shown that testing, even when achieving 100% line and branch coverage, is no guarantee to detect the methods false behaviour. Thereby, we have demonstrated the insufficiency of testing only

unproven properties and assuming the correctness of modular verified code fragments. We have compared our new methodology with other latest approaches, which integrate formal verification and dynamic test techniques. The presented methodology considers dependencies between contracts and which prepares test cases to prevent the propagation of errors.
Future work may address the automatic generation of test vectors used for robustness testing to minimise the required user interaction. Furthermore, the methodology may be combined with a code change management to analyse which proof obligations and dependencies require new verification after a code version has been saved. This improvement would minimise the overhead when applying the methodology during a continuous development process.

References

[BCD+06] Barnett, Mike, Bor Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino: *Boogie: A modular reusable verifier for object-oriented programs*. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2006.

[BLS05] Barnett, Mike, K Rustan M Leino, and Wolfram Schulte: *The spec# programming system: An overview*. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2005.

[CMW12] Christakis, Maria, Peter Müller, and Valentin Wüstholz: *Collaborative verification and testing with explicit assumptions.* In Giannakopoulou, Dimitra and Dominique Méry (editors): *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 2012.

[CS04] Csallner, Christoph and Yannis Smaragdakis: *Jcrasher: an automatic robustness tester for java*. Software: Practice and Experience, 34(11):1025–1050, 2004.

[CS05] Csallner, Christoph and Yannis Smaragdakis: *Check 'n' crash: combining static checking and testing*. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM, ISBN 1-58113-963-2.

[CSX08] Csallner, Christoph, Yannis Smaragdakis, and Tao Xie: *Dsd-crasher: A hybrid analysis tool for bug finding*. ACM Trans. Softw. Eng. Methodol., 17:8:1–8:37, 2008.

[DMB08] De Moura, Leonardo and Nikolaj Bjørner: *Z3: an efficient smt solver*. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[FL11] Fähndrich, Manuel and Francesco Logozzo: *Static contract checking with abstract interpretation*. In *Proceedings of the 2010 international conference on Formal verification of object-oriented software*, FoVeOOS'10, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[FLL+02] Flanagan, Cormac, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata: *Extended static checking for java*. In *ACM Sigplan Notices*, volume 37, pages 234–245. ACM, 2002.

[Flo67] Floyd, Robert W: *Assigning meanings to programs*. Mathematical aspects of computer science, 19(19-32):1, 1967.

[GTXT11] Ge, Xi, Kunal Taneja, Tao Xie, and Nikolai Tillmann: *Dyta: dynamic symbolic execution guided with static verification results*. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 992–994, New York, NY, USA, 2011. ACM.

[HHR+13] Huster, Stefan, Patrick Heckeler, Jürgen Ruf, Sebastian Burg, Thomas Kropf, and Wolfgang Rosenstiel: *A software testing framework to integrate formal verification results*. In *MBMV*, pages 183–192, 2013.

[Hoa69] Hoare, Charles Antony Richard: *An axiomatic basis for computer programming*. Communications of the ACM, 12(10):576–580, 1969.

[MMRV06] Matthews, J., J S. Moore, S. Ray, and D. Vroon: *Verification Condition Generation Via Theorem Proving*. In Hermann, M. and A. Voronkov (editors): *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2006)*, volume 4246 of *LNCS*, pages 362–376, Phnom Penh, Cambodia, November 2006. Springer.

[MPHL06] Müller, Peter, Arnd Poetzsch-Heffter, and Gary T. Leavens: *Modular invariants for layered object structures*. In *SCIENCE OF COMPUTER PROGRAMMING*, page 2006, 2006.

[Mül02] Müller, Peter: *Modular specification and verification of object-oriented programs*. Springer-Verlag, 2002.

[RC90] Reeves, Steve and Michael Clarke: *Logic for computer science*. Addison-Wesley Wokingham, 1990.

[TDH08] Tillmann, Nikolai and Jonathan De Halleux: *Pex: white box test generation for .net*. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

Verifikation Rekonfigurierbarer Scan-Netze

Rafal Baranowski, Michael A. Kochte, Hans-Joachim Wunderlich
ITI, Stuttgart Universität, Pfaffenwaldring 47, D-70569, Stuttgart
E-Mail: {baranowski, kochte}@iti.uni-stuttgart.de, wu@informatik.uni-stuttgart.de

Zusammenfassung

Rekonfigurierbare Scan-Netze, z. B. entsprechend IEEE Std. P1687 oder 1149.1-2013, ermöglichen den effizienten Zugriff auf On-Chip-Infrastruktur für Bringup, Debug, Post-Silicon-Validierung und Diagnose. Diese Scan-Netze sind oft hierarchisch und können komplexe strukturelle und funktionale Abhängigkeiten aufweisen. Bekannte Verfahren zur Verifikation von Scan-Ketten, basierend auf Simulation und struktureller Analyse, sind nicht geeignet, Korrektheitseigenschaften von komplexen Scan-Netzen zu verifizieren.

Diese Arbeit stellt ein formales Modell für rekonfigurierbare Scan-Netze vor, welches die strukturellen und funktionalen Abhängigkeiten abbildet und anwendbar ist für Architekturen nach IEEE P1687. Das Modell dient als Grundlage für effizientes Bounded Model Checking von Eigenschaften, wie z. B. der Erreichbarkeit von Scan-Registern.

1 Einleitung

Eingebettete Instrumente stellen einen großen Anteil höchstintegrierter Schaltungen dar und werden für Bringup, Post-Silicon-Validierung, Debug und Diagnose benötigt. Diese Instrumente werden auch während des Betriebs im Feld verwendet, z. B. für Power-Up-Initialisierung, Systemüberwachung, Fehlertoleranz oder Reparatur [1, 2, 3]. Rekonfigurierbare Scan-Netze (RSNs) ermöglichen effizienten und skalierbaren Zugriff auf eingebettete Instrumente sowie Test- und Diagnosestrukturen. Der kürzlich vorgestellte IEEE Std. 1149.1-2013 (JTAG) erlaubt schon einfache RSNs, in denen einzelne Scan-Register (Instrumente) aus der Scan-Kette ausgeschlossen werden können [4]. Der Vorschlag IEEE P1687 (IJTAG) strebt die Standardisierung noch flexiblerer Scan-Architekturen an [1].

Abb. 1 stellt ein einfaches RSN dar. Die 1-Bit Scan-Register `S1` und `S2` steuern den Zugriff auf zwei längere Register `S3` und `S4`. Die Scan-In-Daten werden nur durch Register `S3` geschoben, falls zuvor sichergestellt wurde, dass `S1` = `S2` = 1. Die internen Steuersignale kommen aus den Schatten-Latches der Scan-Register `S1` und `S2`, bleiben also während des Schiebens stabil.

Für statische Scan-Architekturen ist die Überprüfung einfacher Entwurfsregeln (DRC) ausreichend, um die Korrektheit des Entwurfs zu gewährleisten [5]. Das Zeitverhalten kann mittels statischer Zeitanalyse (STA) bestimmt werden [6]. Die Funktion komplexerer Strukturen, z. B. entsprechend IEEE Std. 1500, kann durch Simulation validiert werden [7].

Für allgemeine RSNs stellt jedoch die Entwurfsverifikation ein viel schwierigeres Problem dar. Steuersignale für Scan-Register können kombinatorisch von Werten in anderen

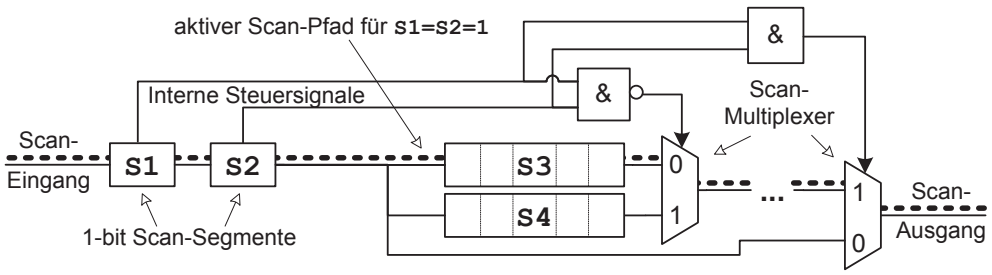


Abbildung 1: Beispiel eines rekonfigurierbaren Scan-Netzes

Scan-Registern erzeugt werden, welche wiederum von Registern in der gleichen oder unterschiedlichen Hierarchiestufe abhängen. Im IP/Core-basierten Entwurf können RSNs von Drittanbietern integriert werden, ohne dass deren Struktur oder Funktion komplett bekannt ist. Folglich können bestimmte Konfigurationen illegal oder widersprüchlich sein und zu Integrationsfehlern führen, z. B. den eingeschränkten Zugriff auf Scan-Register.

Ein Beispiel eines Entwurfsfehlers zeigt Abb. 1. Diese Struktur ist nach IEEE P1687 gültig. Der Zugriff auf das Scan-Register `S4` ist nicht möglich, da es keine Belegung der Scan-Register `S1` und `S2` gibt, so dass Daten durch `S4` geschoben werden können. Es besteht eine *kombinatorische Abhängigkeit*, die nicht erfüllt werden kann. Um diesen Entwurfsfehler zu finden, kann ein Algorithmus zur kombinatorischen Testmustererzeugung (ATPG) verwendet werden. Allerdings können solche Abhängigkeiten auch sequentiell sein: Eine Belegung, die einen Schiebepfad mit dem Zielregister definiert, kann u. U. nicht aus dem Startzustand des RSNs erreichbar sein. Aufgrund solcher *sequentiellen Abhängigkeiten* sind sowohl die Verifikation der Erreichbarkeit und die Berechnung von Zugriffsmustern für RSNs NP-harte Entscheidungsprobleme, ähnlich wie sequentielle Testmustererzeugung für Haftfehler. Sequentielle ATPG Systeme können mehrere Dutzend Taktzyklen berücksichtigen. Ein Zugriff in einem RSN kann aber Tausende von Takten erfordern, was aktuelle Algorithmen überfordert.

Diese Arbeit stellt eine Modellierung für allgemeine RSNs vor, die auch für komplexe Scan-Architekturen anwendbar ist. Das vorgeschlagene Modell ist geeignet für RSNs entsprechend IEEE Std. 1149.1-2013 und IEEE Std. P1687. Im Vergleich zu einem taktgenauen Modell wird in dem vorgeschlagenen Modell das zeitliche Verhalten abstrahiert. Somit wird die sequentielle Tiefe stark reduziert und eine effiziente Verifikation ermöglicht. Das vorgestellte Modell ist eine Verallgemeinerung bisheriger Modellierungsmethoden, die in [8] und [9] vorgestellt wurden, und wurde um die Behandlung von unbekannten oder unspezifizierten Werten erweitert. Das Modell wird beispielhaft zur Erreichbarkeitsanalyse von Scan-Registern mittels Bounded Model Checking verwendet. Die Methode erlaubt die Verifikation sehr großer RSNs und ist allgemeiner als Algorithmen, die nur für bestimmte Unterklassen von RSNs funktionieren, wie z. B. [10] für streng hierarchische Strukturen.

Der folgende Abschnitt stellt die hier betrachteten RSNs vor. Abschnitt 3 beschreibt das vorgeschlagene RSN-Modell mit Zeitabstraktion. Die darauf basierende Verifikation ist in Abschnitt 4 beschrieben. Experimentelle Ergebnisse werden in Abschnitt 5 diskutiert.

2 Terminologie

Rekonfigurierbare Scan-Netze (RSNs) bestehen aus Scan-Registern, Multiplexern und kombinatorischen Elementen. In der Regel wird auf RSNs durch einen JTAG-konformen Test Access Port (TAP) zugegriffen. RSNs können dann als Scan-Register mit *variabler Länge*

betrachtet werden. Der logische Zustand des RSNs bestimmt, welche der Register momentan zugreifbar sind. Der RSN-Zustand wird durch Überschreiben der Registerinhalte geändert.

Der Grundbaustein eines RSN ist ein *Scan-Segment* (siehe Abb. 2). Im einfachsten Fall ist ein Scan-Segment ein Schieberegister mit einem oder mehreren *Scan Flip-Flops* mit gemeinsamen Steuersignalen. Ein Scan-Segment unterstützt bis zu drei Operationen: Während der *Capture*-Operation wird das Register mit Daten von außerhalb des RSNs geladen, z. B. aus einem On-Chip-Instrument. Während der *Shift*-Operation werden Daten vom Scan-Eingang durch die Register-Bits zu dem Scan-Ausgang geschoben. Während der *Update*-Operation übernimmt ein optionales *Schatten*-Latch Daten aus dem Register. Das Steuersignal *select* eines Scan-Segments gibt an, ob das Segment für eine Capture-, Shift- oder Update-Operation aktiviert ist. Wie in JTAG-Testdaten-Registern bleibt das Schatten-Latch während des Schiebens stabil. Ein Scan-Segment mit einem Schatten-Latch kann für die bidirektionale Kommunikation mit einem On-Chip-Instrument genutzt werden. Optionale Elemente eines Scan-Segments sind in Abb. 2a) gestrichelt dargestellt.

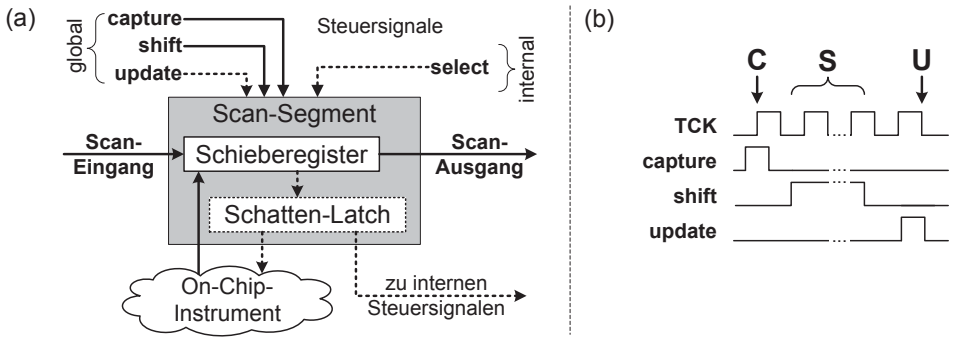


Abbildung 2: (a) Scan-Segment; (b) Capture-, Shift-, Update-Operation (CSU)

Scan-Multiplexer im RSN sind Multiplexer, die den Pfad, durch den Daten geschoben werden, kontrollieren. In Abb. 1 erlauben die zwei Scan-Multiplexer die Scan-Segmente `S3` und `S4` zu umgehen. Das Steuersignal *address* eines Scan-Multiplexers gibt an, welcher Eingang des Multiplexers ausgewählt ist.

Die Steuersignale von Scan-Segmenten und Multiplexern hängen vom Zustand des RSNs ab: Sie können von beliebigen kombinatorischen Logikelementen getrieben werden, welche wiederum von Schatten-Latches von Scan-Segmenten oder von externen Steuereingängen abhängen. In Abb. 1 werden alle Steuersignale von den Schatten-Latches der Scan-Segmente `S1` oder `S2` getrieben.

RSNs haben einen *primären Scan-Eingang* und einen *primären Scan-Ausgang*, sowie globale Steuersignale für die Capture-, Shift- und Update-Operationen. Weitere externe Steuersignale können interne Signale entweder direkt oder durch kombinatorische Logik treiben.

Zwei Scan-Segmente sind *direkt verbunden*, falls ihr Scan-Ausgang und Scan-Eingang durch ein Signal oder einen Multiplexer verbunden sind. Ein *Scan-Pfad* ist eine azyklische Folge von direkt verbundenen Scan-Segmenten, vom primären Scan-Eingang bis zum primären Scan-Ausgang des RSNs. Ein Scan-Pfad ist *aktiv* genau dann, wenn (gdw.) die Select-Signale der Segmente auf dem Pfad aktiv sind, und alle Multiplexer auf dem Pfad die Eingänge auswählen, die auf dem aktiven Pfad liegen. Die Select-Signale aller Segmente, die nicht zum aktiven Pfad gehören, bleiben inaktiv, um Datenverlust zu verhindern. In Abb. 1 läuft der aktive Scan-Pfad über $\texttt{S1}, \texttt{S2}, \texttt{S3}$, falls $\texttt{S1} = \texttt{S2} = 1$.

Eine *Scan-Konfiguration* ist der Zustand aller sequentiellen Elemente und der externen Steuersignale. Eine Scan-Konfiguration ist *gültig* gdw. (i) es einen aktiven Scan-Pfad gibt und (ii) nur Scan-Segmente, die zum aktiven Pfad gehören, ausgewählt sind (select=1). Somit wird sichergestellt, dass Eingabedaten zu den Zielsegmenten und Ausgabedaten korrekt zum primären Scan-Ausgang geschoben werden. Alle Segmente, die nicht zum aktiven Pfad gehören, bleiben stabil.

Ein Zugriff im RSN ist eine atomare Operation mit drei Schritten: *Capture, Shift und Update* (CSU, Abb. 2b). Während Capture registrieren die Segmente auf dem aktiven Pfad neue Daten. Diese Daten werden während der Shift-Phase herausgeschoben und neue Daten werden in die Segmente geschoben. Während Update werden die Daten in Registern auf dem aktiven Pfad im Schatten-Latch übernommen. Lese- und Schreibzugriffe erfordern, dass die Zielregister auf dem aktiven Pfad liegen.

3 RSN-Modellierung auf CSU-Granularität

Die RSN-Modellierung erfasst die Struktur und Funktionalität des RSNs und kann leicht aus einer strukturellen Beschreibung (Gate-, RT- oder höhere Ebene, z. B. Instrument Connectivity Language ICL/P1687) abgeleitet werden. Das Modell erlaubt eine 3-wertige Modellierung der Zustände und Signale mit den Symbolen $\{0, 1, X\}$ nach Kleene's 3-wertiger Logik.

Definition 3.1 *Das CSU-genaue RSN-Modell (Capture-shift-update-Accurate Model, CAM)* $\mathcal{M} = \{S, I, C, c_0, \texttt{Active}\}$ *besteht aus Zustandselementen* S*, externen Steuereingängen* I*, der Menge der Scan-Konfigurationen* $C \subseteq \{0, 1, X\}^{|S \cup I|}$*, der Startkonfiguration* $c_0 \in C$ *und einem Prädikat* `Active`*. Jedes Zustandselement* $s \in S$ *entspricht einem 1-Bit Scan-Register im RSN. Eine Konfiguration* $c \in C$ *bestimmt den Zustand aller Elemente in* S *und der externen Eingänge* I*.* c *kann auch als Funktion interpretiert werden:* $c : S \cup I \to \{0, 1, X\}$*, die jedem* $e \in S \cup I$ *einen Zustand* $c(e)$ *zuordnet. Das Prädikat* $\texttt{Active} : C \times S \to \{0, 1, X\}$ *weist jedem Element* $s \in S$ *in Konfiguration* $c \in C$ *einen Wert* $\texttt{Active}(c, s)$ *zu:*

$$\texttt{Active}(c, s) := \begin{cases} 0 & \textit{falls } \texttt{Select}(c, s) = 0, \\ 1 & \textit{falls } (\texttt{Select}(c, s) = 1) \textit{ und } c \textit{ ist gültig}, \\ X & \textit{sonst.} \end{cases}$$

$\texttt{Select}(c, s)$ *ist der Zustand des* select*-Signals des Scan-Segments* $s \in S$ *in Konfiguration* $c \in C$*. Element* s *ist Teil des aktiven Scan-Pfads gdw.* $\texttt{Active}(c, s) = 1$*.*

3.1 Gültige Scan-Konfigurationen

Um die Prädikate zu erzeugen, wird mittels der Booleschen Funktion $V : C \to \{0, 1\}$ zwischen gültigen und ungültigen Scan-Konfigurationen unterschieden (s. Abschnitt 2). V ist wahr gdw. eine Konfiguration gültig ist, also ein wohldefinierter Scan-Pfad existiert. V wird iterativ als Konjunktion $V(c) = \bigwedge_{s \in S} v(c, s)$ aufgebaut, wobei $v(c, s)$ genau dann wahr ist, wenn die lokale Scan-Konfiguration des Segments s in $c \in C$ gültig ist:

Für ein Segment s mit einem Vorgänger $p \in \texttt{pred}(s)$ und einem Nachfolger $n \in \texttt{succ}(s)$ (Abb. 3a), müssen p und n ausgewählt sein, wenn s ausgewählt ist, damit Scan-Daten nicht verlorengehen: $v(s) := (\mathit{select}(s) = 1) \Rightarrow [(\mathit{select}(p) = 1) \wedge (\mathit{select}(n) = 1)]$.

Für Segment s mit einem Vorgänger p und mehreren Nachfolgern (Abb. 3b) gilt, dass *genau ein* Nachfolger von s ausgewählt sein muss, wenn s ausgewählt ist. Mit

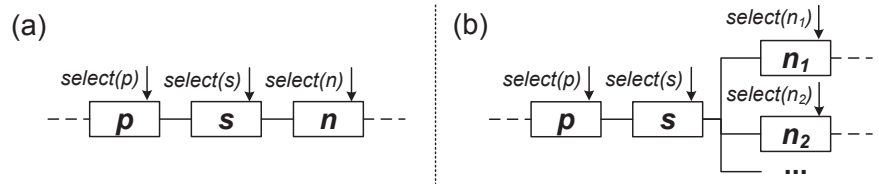


Abbildung 3: (a) Serielle und (b) verzweigende Scan-Strukturen

$A := (select(s) = 1) \Rightarrow \bigvee_{n \in \mathtt{succ}(s)} (select(n) = 1)$ wird sichergestellt, dass *mindestens ein* Nachfolger von s ausgewählt ist. Mit $B := \forall_{n_k, n_l \in \mathtt{succ}(s), n_k \neq n_l} : [(select(n_k) = 1) \Rightarrow (select(n_l) \neq 1)]$ gilt, dass *höchstens* ein Nachfolger von s ausgewählt sein kann. Mit (A) und (B) ergibt sich $v(s) := [(select(s) = 1) \Rightarrow (select(p) = 1)] \wedge (A) \wedge (B)$. Somit wird an Verzweigungen des Pfads sichergestellt, dass nur ein Nachfolger ausgewählt ist.

Entsprechend lassen sich die Funktionen für gültige Konfigurationen für Segmente mit einem Nachfolger und mehreren Vorgängern, und für Segmente mit mehreren Nachfolgern und mehreren Vorgängern ableiten (s. [8]).

Mit der oben eingeführten Funktion V ergibt sich für das Prädikat `Active` :

$$\mathtt{Active}(c, s) := \begin{cases} 0 & \text{falls } \mathtt{Select}(c, s) = 0, \\ 1 & \text{falls } (\mathtt{Select}(c, s) = 1) \wedge V(c), \\ X & \text{sonst.} \end{cases} \quad (1)$$

Die *Select*-Funktionen werden aus den Eingangskegeln der Kontrollsignale im strukturellen RSN-Modell erstellt.

Die Modellierung erfordert, dass der aktive Scan-Pfad nur Scan-Segmente, Multiplexer, Puffer und Inverter enthält, damit Scan-Daten während des Schiebens nicht verändert werden. Dies entspricht auch IEEE P1687. Enthält das RSN komplexere Logik, z. B. Test-Dekompressionsstrukturen, werden diese als Black-Box behandelt und in gültigen Scan-Konfigurationen vom aktiven Pfad ausgeschlossen.

3.2 Übergangsrelation des RSN-Modells

Die Übergangsrelation des CSU-genauen RSN-Modells (CAM) modelliert die Auswirkung einer CSU-Operation, nämlich eine Änderung der Zustände der Scan-Segmente auf dem aktiven Pfad.

Definition 3.2 *Die Übergangsrelation des CAMs* $\mathcal{M} = \{S, I, C, c_0, \mathtt{Active}\}$ *ist die Menge* $T \subseteq C \times C$ *mit allen Paaren von Scan-Konfigurationen* $(c_1 \in C, c_2 \in C)$*, für die gilt:* c_2 *ist durch eine CSU-Operation von* c_1 *erreichbar. Ihre charakteristische Funktion ist:*

$$T(c_1, c_2) := \bigwedge_{s \in S} \big[[(\mathtt{Active}(c_1, s) = 0) \Rightarrow (c_2(s) = c_1(s))] \wedge [(\mathtt{Active}(c_1, s) = X) \Rightarrow (c_2(s) = X)]\big].$$

Die Übergangsrelation definiert die Bedingung für Zustandsänderungen im RSN: Für Elemente s, die nicht zum aktiven Pfad gehören, ändert sich der Zustand nicht zwischen aufeinander folgenden Konfigurationen c_1 und c_2. Außerdem wird angenommen, dass der Zustand von s in c_2 unbekannt ist, wenn die Konfiguration c_1 ungültig ist, oder wenn unbekannt ist, ob s zum aktiven Scan-Pfad gehört ($\mathtt{Active}(c_1, s) = X$). Der Zustand von s kann sich also nur ändern, wenn s in einer gültigen Konfiguration c_1 ausgewählt ist, d. h. $\mathtt{Active}(c_1, s) = 1$.

4 Formale Verifikation

Die folgenden Abschnitte beschreiben die Verwendung des Modells in der formalen Verifikation und seine Einschränkungen. Mittels Bounded Model Checking wird die Erreichbarkeit im RSN überprüft. Schließlich werden *robuste* RSNs definiert und eine Methode vorgestellt, mit der die Robustheit bewiesen werden kann.

4.1 Verifikation basierend auf CSU-genauer Modellierung (CAM)

Das CAM kann als *abstrakte* FSM verstanden werden, die den Zustand des RSNs exakt abbildet, aber das Zeitverhalten abstrahiert. Ein Übergang im CAM entspricht einer kompletten CSU-Operation, also mehreren Takten im RSN. Abb. 4 zeigt ein Beispiel, wo die k Takte einer CSU-Operation zu einem Übergang im CAM zusammengefasst werden.

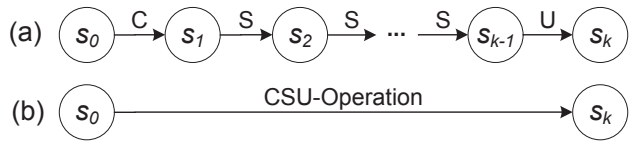


Abbildung 4: Zustandsübergänge während einer CSU-Operation (a) taktgenau; (b) im CSU-genauen Modell (CAM)

Die CSU-genaue Modellierung ist wohldefiniert: Eigenschaften, die im CAM gelten, sind auch für das taktgenaue RSN gültig, falls die internen Steuersignale (z. B. Multiplexer-Adresse) während der Capture- und Shift-Phase stabil sind. Für Signale, die im RSN generiert werden, gilt dies, da sie von Schatten-Latches getrieben werden. Für externe Signale muss dies sowieso im System sichergestellt werden, da sich sonst der aktive Scan-Pfad während des Schiebens ändern und Daten verloren gehen können.

Die CSU-genaue Modellierung ist pessimistisch: Nach Def. 3.2 ist nach einer ungültigen Scan-Konfiguration der Zustand der Scan-Segmente undefiniert, selbst wenn im taktgenauen Modell Segmente einen definierten Zustand besitzen können. So kann mit dem CAM ein Gegenbeispiel für eine Eigenschaft erzeugt werden, die im taktgenauen Modell stets gilt. Sind ungültige Konfigurationen nicht vom Startzustand aus erreichbar (s. Abschnitt 4.3), dann sind im CAM widerlegte Eigenschaften auch im taktgenauen Modell ungültig.

4.2 Bounded Model Checking

In diesem Abschnitt wird die Verwendung des CAM in Bounded Model Checking gezeigt (BMC, [11]): Sei P eine Eigenschaft in temporaler Logik, und T die Übergangsrelation des CAMs eines RSNs. Im CSU-genauen BMC wird nach einer Zustandsfolge (Gegenbeispiel) mit gegebener maximaler Länge gesucht, die P widerlegt. Dabei entspricht die Länge der Anzahl an CSU-Operationen. P ist widerlegt, wenn für eine gewisse Zahl $n \in \mathbb{N}^+$ an Schritten die folgende Formel erfüllt ist: $\varphi^n := I \wedge \left[\bigwedge_{i=0}^{n-1} T_i\right] \wedge (\neg P^n)$, ($I$: charakteristische Funktion der Startzustände von $\mathcal{M}$, T_i: charakteristische Funktion der Übergangsrelation T im i-ten Schritt, P^n: Eigenschaft P über n Schritte). Diese Formel kann in konjunktive Normalform (CNF) überführt und ihre Erfüllbarkeit mit einem SAT-Solver untersucht werden.

Mit BMC kann z. B. die Erreichbarkeit von Scan-Segmenten verifiziert werden. Dazu muss ein aktiver Scan-Pfad vom primären Scan-Eingang über das Segment bis zum Scan-Ausgang existieren. Für ein CAM $\mathcal{M} = \{S, I, C, c_0, \texttt{Active}\}$ ist der Beweis der Erreichbarkeit des Segments $s \in S$ äquivalent mit einem Gegenbeispiel für folgende LTL Formel: $\boldsymbol{G}\,[\texttt{Active}(s) \neq 1]$.

Der Beweis der Erreichbarkeit von s in max. n CSU-Operationen entspricht also einer erfüllenden Belegung für die Formel:

$$\texttt{Check}(s, c_0, n) := \mathbf{1}(c_0) \wedge \left[\bigwedge_{i=1\ldots n} T(c_{i-1}, c_i)\right] \wedge \left[\bigvee_{i=0\ldots n} [\texttt{Active}(c_i, s) = 1]\right],$$

wobei $\mathbf{1}(c_0)$ die charakteristische Funktion der Startkonfiguration $c_0 \in C$ ist. Die Erfüllbarkeit dieser Formel wird für eine steigende Zahl von CSU-Operationen ($n = 1, 2, \ldots$) untersucht, bis eine erfüllende Belegung gefunden wurde oder eine vorgegebene maximale Zahl an Schritten erreicht ist.

4.3 Verifikation der Robustheit

Ein RSN ist *robust*, wenn alle vom Startzustand aus erreichbaren Scan-Konfigurationen gültig sind, d. h. die ausgewählten Scan-Segmente bilden stets einen aktiven Scan-Pfad unabhängig von den Eingangsdaten. Formal ausgedrückt ist ein RSN robust gdw. die LTL Eigenschaft $\boldsymbol{G}\ V$ im CAM gültig ist, also die Boolesche Funktion V (s. Abschnitt 3.1) stets wahr ist.

Für robuste RSNs ist die Verifikation basierend auf dem CSU-genauen Modell *vollständig*: Eine Eigenschaft, die im RSN gilt, ist auch im CAM gültig, d. h. die Abstraktion auf atomare CSU-Operationen verursacht keine falschen Gegenbeispiele, da alle ausgewählten Segmente immer zum aktiven Scan-Pfad gehören (s. Abschnitt 4.1). Entsprechend kann der Zustand eines ausgewählten Segments uneingeschränkt von einer CSU-Operation geändert werden. Die CAM-Übergangsrelation (Def. 3.2) modelliert also exakt die Auswirkung einer CSU-Operation in einem taktgenauen Modell. Da *alle erreichbaren* Konfigurationen gültig sind, kann die Funktion V, welche die Gültigkeit ausdrückt, entfernt und das CAM stark vereinfacht werden: $\texttt{Active}(\texttt{c}, \texttt{s}) := \texttt{Select}(\texttt{c}, \texttt{s})$ für alle $c \in C$ und $s \in S$.

Die Robustheit ($\boldsymbol{G}\ V$) eines RSNs kann mit jedem unbeschränkten LTL Model-Checking-Verfahren auf dem CAM erfolgen. Hier wird eine effiziente SAT-basierte induktive Methode [12] verwendet. Ein RSN mit CAM $\mathcal{M} = \{S, I, C, c_0, \texttt{Active}\}$ und Übergangsrelation T ist robust gdw.:

1. V gilt im Startzustand von $\mathcal{M}$, d. h. $V(c_0)$ ist wahr, und
2. V ist eine Invariante der Übergangsrelation T, d. h. $\forall_{(c_1,c_2)\in T}[V(c_1) \Rightarrow V(c_2)]$.

Gelten beide Bedingungen, ist das RSN robust, da die Startkonfiguration gültig ist und jeder Übergang im CAM die Gültigkeit bewahrt.

Beide Bedingungen können als Erfüllbarkeitsproblem formuliert und mit einem SAT-Solver überprüft werden. $\mathcal{M}$ ist robust, wenn die folgenden Formeln nicht erfüllbar sind:

$$\mathbf{1}(c_0) \wedge \neg V(c_0), \tag{2}$$

$$V(c_i) \wedge T(c_i, c_{i+1}) \wedge \neg V(c_{i+1}), \tag{3}$$

wobei $\mathbf{1}(c_0)$ die charakteristische Funktion der Startkonfiguration ist.

Die erste Anforderung ist notwendig, die zweite hinreichend. Insbesondere gibt es robuste RSNs, in denen die Funktion V keine Invariante der Übergangsrelation ist. Dies gilt, wenn alle gültigen Konfigurationen, die zu ungültigen führen, nicht von der Startkonfiguration aus erreichbar sind. Die hier vorgeschlagene effiziente Methode kann also pessimistisch ein robustes RSN als nicht robust einstufen. Der Vorteil ist allerdings, dass das unbeschränkte LTL-Verifkationsproblem auf eine einfache Boolesche SAT-Instanz abgebildet wird.

5 Evaluierung

Die Skalierbarkeit der CSU-genauen Modellierung wird in BMC-basierten Verifikationsexperimenten evaluiert. Die Experimente werden auf einem Intel Xeon Prozessor mit 3,33 GHz durchgeführt.

5.1 Benchmark-RSNs

Zwei hierarchische Scan-Architekturen werden betrachtet: eine Implementierung mit Segment Insertion Bits (SIB), wie in [13], und eine spezielle Architektur, die zum effizienten Zugriff zwei Betriebsmodi unterstützt (MUX). Die Benchmark-RSNs basieren auf den ITC'02 Schaltungen [14] und werden in [8] ausführlich beschrieben.

Aus Platzgründen werden nur die vier größten Benchmark-RSNs für die zwei Architekturen betrachtet. Die ausgewählten Schaltungen beinhalten bis zu 3 Hierarchiestufen und 5 000 bis 100 000 Scan-Zellen, die in 160 bis 1 200 Scan-Segmente gruppiert sind.

5.2 Verifikation der Erreichbarkeit

Die Erreichbarkeit der Scan-Segmente wird mittels des BMC-Verfahrens aus Abschnitt 4.2 verifiziert. Tabelle 1 stellt den Verifikationsaufwand dar. Spalte "Zugriffslänge" gibt die durchschnittliche und maximale Anzahl der CSU-Operationen (BMC-Schritte) an, die zum Zugriff auf ein Scan-Segment benötigt werden. Unter "Klauseln" wird die maximale Anzahl von Klauseln in der erfüllbaren SAT-Instanz (für den letzten BMC-Schritt) angeführt. t^{max}_{solve} ist die maximale Lösungszeit für ein Scan-Segment, und t_{total} ist die Gesamtverifikationszeit.

Tabelle 1: Verifikation der Erreichbarkeit

Benchmark		Zugriffslänge	Klauseln	Konflikte	t^{max}_{solve}	t_{total}
Design	Architektur	avg / max	max	avg / max	[s]	[s]
g1023	MUX	3,6 / 5	32 917	2,0 / 27	0,03	2,6
	SIB	2,3 / 3	16 826	0 / 0	0,02	1,2
p22810	MUX	3,9 / 7	150 787	5,6 / 88	0,15	36,5
	SIB	2,5 / 4	77 376	0 / 0	0,05	17,2
p34392	MUX	4,4 / 7	67 367	9,1 / 107	0,07	8,1
	SIB	2,7 / 4	33 028	0 / 0	0,03	3,5
p93791	MUX	4,1 / 7	322 891	6,5 / 223	0,57	187,1
	SIB	2,5 / 4	172 082	0 / 0	0,14	94,5

Obgleich die Anzahl von Klauseln in den SAT-Instanzen auf bis zu 323 000 steigt, beträgt die maximale Verifikationszeit für ein Scan-Segment im größten RSN (p93791) höchstens 0,6 Sek. und ca. 0,15 Sek. durchschnittlich. Die Gesamtverifikationszeit vom größten RSN mit über 1 200 Scan-Segmenten beträgt ca. 3 Min.

Bei der Verifikation von SIB-basierten RSNs muss der SAT-Solver kein Backtracking durchführen. Im Gegensatz dazu weisen die MUX-basierten RSNs komplexere sequenzielle Abhängigkeiten auf, die Konflikte bei der SAT-Suche verursachen. Die durchschnittliche und max. Anzahl von Backtracking-Vorgängen ist in Tabelle 1 unter "Konflikte" angegeben.

5.3 Verifikation der Robustheit

Die Robustheit der Benchmark-Schaltungen wird mittels der SAT-basierten induktiven Methode geprüft (s. Abschnitt 4.3). Es wurde erfolgreich bewiesen, dass alle Benchmark-RSNs robust sind, d. h. die Scan-Konfiguration bleibt in diesen Schaltungen für beliebige Eingangssequenzen gültig. Maximal werden 84 Sek. für die Verifikation benötigt.

5.4 Verifikation fehlerhafter RSNs

Im Folgenden werden mittels der CSU-genauen BMC-Methode (s. Abschnitt 4.2) fehlerhafte RSNs verifiziert. Hierzu werden die MUX-basierte Benchmark-Schaltungen aus Abschnitt 5.1 mit folgenden Entwurfsfehlern versehen:

- *Pfad-Bug*: Die Nachfolger von zwei zufällig gewählten Scan-Segmenten oder Multiplexern werden vertauscht.
- *Steuer-Bug*: Zwei *address*-Steuersignale von zufällig gewählten Multiplexern werden vertauscht.
- *Mux-Bug*: Die Scan-Eingänge von einem zufällig gewählten Scan-Multiplexer werden vertauscht.

Ein Scan-Segment wird als unerreichbar klassifiziert, wenn innerhalb von 30 CSU-Operationen kein Zugriff darauf möglich ist (die maximale Anzahl von BMC-Schritten beträgt also 30). Diese Schranke ist signifikant größer als die maximale Zugriffslänge von 7 Schritten in den fehlerfreien Benchmarks (s. Tabelle 1). Um die Unerreichbarkeit im unbeschränkten Sinne zu verifizieren, z. B. für sicherheitsrelevante Eigenschaften, müssen symbolische oder interpolationsbasierte Model-Checking-Verfahren benutzt werden, z. B. [15].

Aus Platzgründen werden nur drei MUX-basierte RSNs (g1023, p22810, und p93791) untersucht. Für jede Art von Entwurfsfehlern und für jedes Benchmark-RSN wurden 100 zufällige Mutationen untersucht. Tabelle 2 zeigt die Verifikationsergebnisse: Spalte “Bugs gefunden” gibt die Rate der fehlerhaften RSNs an, in denen der Fehler detektiert wurde (d. h. innerhalb von 30 CSU-Operationen ist mindestens ein Scan-Segment nicht erreichbar). Spalte “Unerreichbar” nennt den durchschnittlichen Anteil von unerreichbaren Scan-Segmenten in einem fehlerhaften RSN. t^{avg}_{total} und t^{max}_{total} geben die durchschnittliche bzw. maximale Gesamtverifikationszeit für ein RSN.

Tabelle 2: Verifikation der Erreichbarkeit in fehlerhaften RSNs

Benchmark	Entwurfsfehler	Bugs gefunden	Unerreichbar	t^{avg}_{total} [s]	t^{max}_{total} [s]
g1023-MUX	Pfad-Bug	100%	13,3%	2,8	15
	Steuer-Bug	5%	1,9%	1,8	16
	Mux-Bug	100%	20,0%	4,6	18
p22810-MUX	Pfad-Bug	100%	7,6%	29,3	251
	Steuer-Bug	2%	1,0%	21,7	267
	Mux-Bug	100%	8,0%	39,0	363
p93791-MUX	Pfad-Bug	100%	7,9%	139,4	152
	Steuer-Bug	1%	1,0%	109,5	1301
	Mux-Bug	100%	5,9%	172,1	1859

Die durchschnittliche Verifikationszeit der fehlerhaften Benchmarks ist ähnlich wie bei den fehlerfreien Schaltungen (s. Tabelle 1). Im schlimmsten Fall beträgt die Verifikationszeit eines fehlerhaften RSNs 31 Min. Die *Pfad-Bugs* und *Mux-Bugs* werden immer gefunden (jedes fehlerhafte RSN beinhaltet mindestens ein unerreichbares Scan-Segment), wobei bis zu 99% der *Steuer-Bugs* keinen Einfluss auf die Erreichbarkeit von Segmenten haben. Es ist also oft möglich, eine Zugriffssequenz zu allen Scan-Segmenten zu finden, selbst wenn die Steuersignale von zwei zufällig gewählten Scan-Multiplexer vertauscht sind.

Entwurfsfehler führen oft dazu, dass ein ansonsten robustes RSN in eine ungültige Scan-Konfiguration versetzt werden kann. Deshalb können viele Entwurfsfehler effizient durch

die Verifikation der Robustheit detektiert werden. Mittels des Ansatzes aus Abschnitt 4.3 konnte bewiesen werden, dass *alle* untersuchten Bugs die Robustheit der Benchmark-RSNs verletzen. Im schlechtesten Falle werden nur 2 Min. für die Widerlegung der Robustheit eines fehlerhaften RSNs benötigt. Die dabei erzeugten Gegenbeispiele können zur Lokalisierung der Entwurfsfehler benutzt werden.

6 Zusammenfassung

Die Komplexität von rekonfigurierbaren Scan-Netzen und ihr Einsatz im IP-basierten Entwurf erfordern neue Werkzeuge für die Entwurfsverifikation. Diese Arbeit stellt eine Modellierung vor, die durch temporale Abstraktion eine hohe Skalierbarkeit von bestehenden Model-Checking-Verfahren gewährleistet. Die Modellierung unterstützt unbekannte Werte und ist für unterschiedliche konfigurierbare Scan-Architekturen verwendbar. Komplexe Eigenschaften von Scan-Netzen können damit effizient verifiziert werden.

Danksagung: Diese Arbeit wurde durch die Deutsche Forschungsgesellschaft (DFG) mit den Projekten WU 245/13-1 (RMBIST) und WU 245/11-1 (OASIS) gefördert.

Literatur

[1] N. Stollon, *On-Chip Instrumentation: Design and Debug for Systems on Chip.* Springer US, 2011.

[2] E. Larsson and K. Sibin, "Fault management in an IEEE P1687 (IJTAG) environment," in *IEEE Int'l Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2012, p. 7.

[3] J. Rearick and A. Volz, "A Case Study of Using IEEE P1687 (IJTAG) for High-Speed Serial I/O Characterization and Testing," in *Proc. IEEE International Test Conference (ITC)*, 2006, paper 10.2.

[4] "IEEE Standard for Test Access Port and Boundary-Scan Architecture 1149.1-2013," IEEE Computer Society, 2013.

[5] E. Eichelberger and T. Williams, "A logic design structure for LSI testability," in *Proc. Design Automation Conf. (DAC)*, 1977, pp. 462–468.

[6] J. Remmers, M. Villalba, and R. Fisette, "Hierarchical DFT Methodology - A Case Study," in *Proc. IEEE Int'l Test Conf. (ITC)*, 2004, pp. 847–856.

[7] A. Benso, S. Di Carlo *et al.*, "IEEE Standard 1500 Compliance Verification for Embedded Cores," *IEEE Trans. VLSI Syst.*, vol. 16, no. 4, pp. 397–407, 2008.

[8] R. Baranowski, M. A. Kochte, and H.-J. Wunderlich, "Modeling, Verification and Pattern Generation for Reconfigurable Scan Networks," in *Proc. IEEE Int'l Test Conf. (ITC)*, 2012, paper 8.2.

[9] R. Baranowski, M. A. Kochte, and H.-J. Wunderlich, "Scan Pattern Retargeting and Merging with Reduced Access Time," in *Proc. IEEE European Test Symposium (ETS)*, 2013.

[10] F. Ghani Zadegan, U. Ingelsson *et al.*, "Access Time Analysis for IEEE P1687," *IEEE Trans. Computers*, vol. 61, no. 10, pp. 1459–1472, October 2012.

[11] A. Biere, A. Cimatti *et al.*, "Bounded Model Checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[12] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design.* LNCS, Springer, 2000, vol. 1954, pp. 127–144.

[13] F. Zadegan, U. Ingelsson *et al.*, "Design Automation for IEEE P1687," in *Proc. Design, Automation Test in Europe Conference (DATE)*, 2011, pp. 1412–1417.

[14] E. Marinissen, V. Iyengar, and K. Chakrabarty, "A Set of Benchmarks for Modular Testing of SOCs," in *Proc. IEEE Int'l Test Conf. (ITC)*, 2002, pp. 519–528.

[15] K. McMillan, "Interpolation and SAT-Based Model Checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer, 2003, vol. 2725, pp. 1–13.

Efficient SAT/Simulation-based model generation for low-level embedded software

Christian Bartsch, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, Wolfgang Kunz
Electronic Design Automation Group
University of Kaiserslautern
{bartsch,villarraga,bschmidt,stoffel,kunz}@eit.uni-kl.de

Abstract

This paper introduces a method for improving the generation of a model for low-level software called program netlist. The model can be used for formal HW/SW co-verification such as property checking and equivalence checking. The program netlist is based on a finite unrolling of a control flow graph extracted from the machine code of the software.

For generating the model, information on memory addresses accessed by the software, such as jump destinations or multiple memory data accesses, need to be computed. Our approach uses an instruction set simulator to find most of these address values and resorts to an enumerative SAT-based algorithm for computing the remaining, hard-to-determine addresses. In addition to finding these values, the combined algorithm also identifies inactive paths in the software, which is essential for obtaining a compact model. We present results on industrial case studies.

1. Introduction

Hardware-dependent software (HDSW) is an important component of modern embedded systems and Systems-on-Chip (SoCs). HDSW can be identified as low-level software since other software modules are built on top of it. This makes HDSW critical as the functionality of the whole embedded software relies on it. HDSW typically interacts tightly with the surrounding hardware performing complex control intensive tasks which makes it error prone and hard to test.
In [SVF+13] a computational model for the verification of low-level HDSW called program netlist is introduced. The model captures information about the executional paths of the program which greatly reduces the complexity of the verification tasks computed on the model. Model generation is based on path-oriented techniques, such as symbolic execution, which are based on finite unrollings of the control flow graph (CFG) [Kin76, PV09]. This poses various challenges when applied to the generation of computational models for the verification of low-level software.
Compared to the analysis of high-level code, for low-level software the assumption that a complete control flow graph (CFG) is obtained after parsing the source code is not always valid [RLT+10]. A typical case is a jump instruction whose destination is stored in a CPU register (e.g. jump [Rm]). Without making a semantic analysis of the software it cannot be clearly determined during the unrolling which path the execution will take. This situation occurs in code constructs commonly used in low-level software where jump targets are computed. Typical examples are branch tables and call-back functions.

In a similar way, low-level software makes heavy use of indirect memory addressing (e.g. load Rn, [Rm]) which involves arithmetic on memory addresses. A common case is an instruction accessing an element of an array or buffer stored in data memory. This complicates the construction of efficient data memory models [VSB+13]. Again, without a semantic analysis it is not clear what memory location(s) can be reached by the software.
Finally, for generating compact models it is required to perform different optimizations during the unrolling. In particular, path pruning allows to exclude unreachable paths of the software from the model. In order to perform path pruning it is necessary to perform specific checks for every conditional branch found during the unrolling that determine whether the corresponding branch can be reached or not.
All the previous mentioned situations occur thousands of times while unrolling complex low-level software. In consequence, an effective processing strategy is required so that model generation becomes feasible for industrial applications.
In practice, several characteristics of HDSW resulting from typical embedded system applications turn out to be beneficial for an efficient model generation. For instance, the software neither makes calls to unbounded recursions nor does it dynamically allocate memory. Moreover, the possible address ranges of the system are limited and predefined by design. Also, it is a common practice in low-level software to use registers also for storing constant addresses, for example when jumping to a subroutine or when accessing a register of the hardware periphery. Finally, analyzing unrolled versions of the CFG reduces the complexity as more constants can be detected. For instance, accesses to arrays, as mentioned before, normally result in a single constant memory address value.
In this work we efficiently exploit these characteristics to only include the feasible behavior of the software into the computational model. This significantly reduces the state space when performing verification. We employ state-of-the-art instruction set simulation techniques to propagate and discover constant values. Only for difficult cases where simulation does not succeed because the considered values are not constant, e.g., when there is a dependency on primary inputs, we employ SAT solving to determine the range of possible values. SAT is much more costly than simulation and should be avoided whenever possible. As shown in the results, the proposed combination of techniques provides a very effective solution for our intended applications.

1.1. Related Work

The works [SVF+13, VSB+13] present the program netlist, its main characteristics and applications. While in these works it is assumed that the CFG already contains all information needed to perform model generation, this paper presents the details of how to generate program netlists from incomplete CFGs. In particular, it is discussed how to efficiently address the challenges mentioned above.
Similar to our approach the works of [AEO+08, AEF+05] and [CFF+06] also target the verification of low-level software by using symbolic execution. A significant difference of these works when compared to ours is that the software behavior is described by means of symbolic formulas which are constructed while exploring execution paths. These formulas are translated subsequently, according to the particular verification procedure used (uninterpreted functions with equality in [CFF+06] and Boolean logic in [AEO+08, AEF+05]). In our approach, instead of creating symbolic formulas, the software behavior is represented by means of a combinational circuit. In order to perform SAT-based verification, the logic of the circuit is translated into a CNF as done conventionally in hardware verification.

This results in important differences when manipulating incomplete CFGs. For instance, in order to compute targets of indirect jumps, symbolic approaches find potential values by making a syntactic analysis of the symbolic formulas. This however can be only applied if certain restrictions are made to the kind of operations performed by the software on the program counter. Particularly, [CFF+06] allows only load/store operations and [AEO+08] supports only a subset of logical operations. While these restrictions are acceptable for the particular purposes of those works, they would limit the scope for the case of verification of low-level HDSW. Therefore, in our work, no restrictions on program counter operations are imposed. We leave the problem of finding possible address values (semantic analysis) to the intelligence of the simulation and SAT engines.
On the other hand, in order to resolve indirect memory accesses, [CFF+06] models data memory as a linear array. To build the model, the interaction of different instructions through data memory is analyzed. For this purpose, comparisons of symbolic functions are performed. Symbolic functions in this case represent memory addresses. As a main advantage of this analysis, the amount of logic for the model is simplified since logic is added only between instructions that in fact interact through data memory.
In our approach we first make a semantic analysis of the software (using simulation/SAT engine) to compute possible memory address value(s) and then construct memory logic in a similar way to [CFF+06]. However in [CFF+06] calls to the decision procedure are always done, including also cases where memory addresses are constant. In our approach, however, we save these calls by using the results of the simulation leading to a significant speed-up of the model generation.
Finally, checks for implementing path pruning are done by using decision procedures in [AEO+08, AEF+05, CFF+06] for all cases. In our approach however, we call the SAT solver only for situations where the simulation does not succeed.

The rest of this paper is organized as follows. The next section describes the challenges of employing path-oriented techniques to low-level software and how to address those challenges by using instruction set simulation and SAT-solving techniques. Afterwards, we show the effectiveness of the proposed solution with our experimental results. Finally, Sec. 4 concludes the paper.

2. Model generation using incomplete CFGs

The generated program netlist incorporates two basic elements: a formal description of the CPU instructions called instruction cells and a fully unrolled CFG containing all possible execution paths of a program called *execution graph* (EXG) [SVF+13, VSB+13]. As explained in the previous section, efficient generation of program netlists requires efficient processing of indirect jumps, indirect accesses to data memory and liveness checks for path pruning. This section briefly presents the basics concepts for generating program netlists and then later details on how to handle the problems previously mentioned by using a mixed solution including a tailored instruction set simulator and a SAT solver.

2.1. Instruction cells

Instruction cells formally describe the instructions implemented by a given processor. The level of abstraction of the description can vary depending on the particular verification objectives. For example, instruction cells can be modeled at ISA level (in terms of the programmer's visible registers) to realize a representation on a high abstraction level. If a more accurate model is required, for example with cycle accuracy, instruction cells can be modeled at RT level.

We developed a tailored instruction cell language (ICL) to describe instruction cells at the ISA level. This language includes different elements that facilitate the description and that allow automatic translation into other formats. In particular, we are able to automatically generate C++ code describing the functionality of each ISA instruction. As will be explained later, this is required to construct an instruction set simulator. We can also represent instruction cells in an internal data structure from which CNFs can be automatically generated in order to perform SAT solving.

2.2. Execution graph

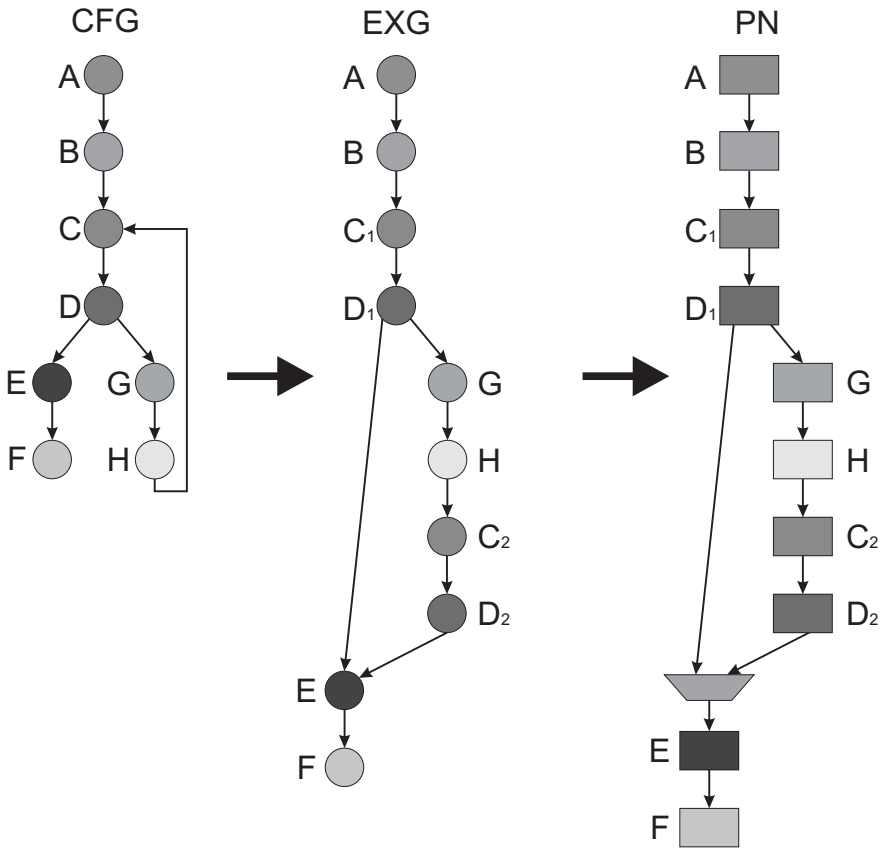


Figure 1: Model generation example: unrolling a CFG into an EXG

An execution graph includes all possible execution paths of a given program. To generate an EXG, first the control flow graph (CFG) of a given machine program has to be extracted. Then, the CFG needs to be fully unrolled into a direct acyclic graph (DAG). Details of this process are explained in [SVF+13]. Figure 1 shows an example.

To generate an EXG with the minimum amount of nodes path pruning and merging can be employed. Both techniques are used on the fly while unrolling the CFG.

Path pruning removes paths that cannot be reached by the program. This reduces significantly the size of the EXG. For that, every successor (branch) of a conditional branch instruction needs to be examined. One check on each of the branches determines whether it is reachable by the program or not. Reachable branches are called “live” and are further unrolled whereas unreachable branches called “dead” are pruned. For example the branch from node D to node G in Fig. 1 is unrolled only once because the branch from node D_2 back to G was found to be dead.

Another technique to compact the EXG is to merge nodes. Two nodes of an EXG can be merged if they represent exactly the same program instruction and if no loops are inserted in the resulting EXG. In Fig. 1, the predecessors of node E in the EXG indicate that two paths have been merged into a single one. As can be seen, no loop is created in the graph.

2.3. Efficient program netlist generation

To generate a program netlist every node of the EXG is replaced by its corresponding instruction cell. Edges of the graph are replaced by the architectural state including all programmer's visible registers. The resulting structure is a combinational circuit without any internal registers. The graph on the right hand side of Fig. 1 illustrates the resulting circuit structure of our example.
In [SVF+13] it is assumed that the CFG already contains all information required to analyze the control flow while performing the unrolling. As mentioned in Sec. 1, when it comes to a CFG that is extracted from a machine program, the resulting CFG will most likely be incomplete. The reason is that not all necessary information is directly available in the machine code, because it depends on the data flow of the program. In those cases a semantic analysis of the software has to be performed in order to deduce the missing information.
Two cases need to be distinguished here: evaluation of the control flow (including indirect jumps and conditional branches) and resolution of the memory behavior (including indirect accesses to memory).
If at a given point of the unrolling an analysis of the semantic of the program is required our algorithm proceeds in two steps. First it calls the instruction set simulator. In turn, the simulator will proceed by forward propagating all constant values contained in the program netlist. If the call succeeds then the information is added to the CFG and the unrolling process can continue. Otherwise, if the simulation fails, the SAT solver is called to compute the required information.
Note that this second call will always succeed. As explained in Sec. 1 this speeds up the process.
For control flow analysis, extracting successors inside an instruction sequence without conditional branches and jumps, i.e., a basic block, is straightforward. However, if a jump instruction is encountered the possible destinations (successors) may not be given directly by the instruction code because that information can be stored in a register. Here, two possible situations may arise. In the simpler one, a constant value loaded from memory and stored in a register is used to calculate the target address. In the second and more complex case, the jump instruction can have different values for the target address. They can depend on the particular execution paths taken by the program or on operations performed by preceding instruction cells to those values. In both cases all possible destinations need to be precisely determined, otherwise performing formal verification on the resulting model can lead to wrong results. In the example of Fig. 1, the destination of the jump instruction at node H is the instruction at node C. This is known in our approach only after examining the semantics of the program by using the combined simulation/SAT solution.
If a conditional branch is found in the unrolling, then it is necessary to check whether each branch can be dead or alive. As explained in Sec. 2.2, unreachable instructions are removed from the model. In these cases we make two calls to the simulation/SAT engine, one for each branch. In Fig. 1, both branches are determined live after the checks performed at branch node D.
Finally, for resolving memory accesses we reduce the complexity of the model by making data memory values path dependent. This means that we create memory logic for a given variable residing in memory only on paths with instructions accessing that variable. Back to our example of Fig. 1, assume that the instruction cell at node C reads a memory value stored at address $Addr_x$. If it is also assumed that the instruction at node G changes the value stored at the same memory

address, then, due to the jump from node H to node C the CFG involves two different accesses to location $Addr_x$. However, after unrolling has been performed (EXG of the bottom part), nodes C_1 and C_2 require only one memory access respectively. Hence, only a single memory value can be reached by each interaction cell. Instruction cells G and C_2 will be directly connected by using memory logic and the memory value read at C_1 can correspond to the initial value of the memory variable. Before taking these decisions, it has to be determined that instruction cells at G, C_1 and C_2 reach the same location $Addr_x$. In each case, this value can depend on the data flow and be stored in a CPU register. Consequently, this information needs to be calculated to define the memory ranges that have to be modeled. In our particular example, three calls to the simulation/SAT engine are made. Note that performing this calculation on the EXG is much easier than on the CFG since more constants can be identified.

2.4. Model enumeration using SAT

There is a large research body on the problem of finding all satisfying values of a propositional formula [JS05]. Here, we present only the basic enumeration algorithm and how it can be employed to solve our particular problem.
Let us assume $addr$ is a bit vector in the program netlist representing the signal we want to inspect. In our case $addr$ can be the memory address signal or the program counter for a given instruction cell. The enumeration algorithm obtains on every iteration a new satisfying value for $addr$. Let us also assume that V_k is the k^{th} satisfying value of $addr$ obtained after the k^{th} iteration of the algorithm. Then, a new element satisfying value V_{k+1} can be extracted from a counterexample refuting the following property.

$$p_{k+1} = \bigvee_{V_i = V_0, V_1, \ldots, V_k} (sig = V_i) \tag{1}$$

The iterative algorithm will execute until the property of Eq. 1 holds, i.e., no new counterexamples are returned by the SAT solver. The constraints in the disjunction are converted to blocking clauses in the SAT instance preventing the solver to visit previous solutions. In our case, we employ incremental SAT solving to speed up the process by reusing information learned by the solver in previous iterations. This scheme produces the exact set of values $addr$ can take.
Obviously, in a general setting, this scheme can run into complexity problems since instructions cells, in principle, could address huge ranges of data memory or jump to many different destinations. However, for our intended application domains, as explained in Sec. 1, the possible solution ranges are limited and restricted leading to a tractable number of SAT calls when computing Eq. 1.

2.5. Constant propagation using instruction set simulation

The SAT-based algorithm of the previous section is very powerful as it is able to find solutions for all cases, independently whether values are constant or not. However, using the SAT solver is very time consuming. The situation is even worse if we take into account that the number of calls required for unrolling complex software can be very large, e.g., on the order of thousands of calls. To reduce run times we propose to implement and to integrate a simulator on an appropriate abstraction level. We avoid the performance problems that can arise when simulating code on low abstraction levels, e.g., at bit level. Therefore, we propose to employ simulation techniques on a higher abstraction level in order to compute memory addresses, jump targets and inactive program paths. Since we are using instruction-based models like assembler code, CFGs and EXGs, the

ISA level is the adequate level of abstraction. However, the performance boost obtained on this abstraction level is bought by loss of information. In particular, information about single bit values and their relationship is lost.
The simulation engine here proposed is capable of simulating arbitrary programs which have been unrolled completely or partially into an EXG. To achieve as much performance boost as possible, we propose an implementation of the instruction cells in a high-level programming language such as C++. In this way, it is possible to take advantage of compiler optimizations for significantly reducing the actual number of processor instructions (of the host machine) needed to simulate a single instruction cell.

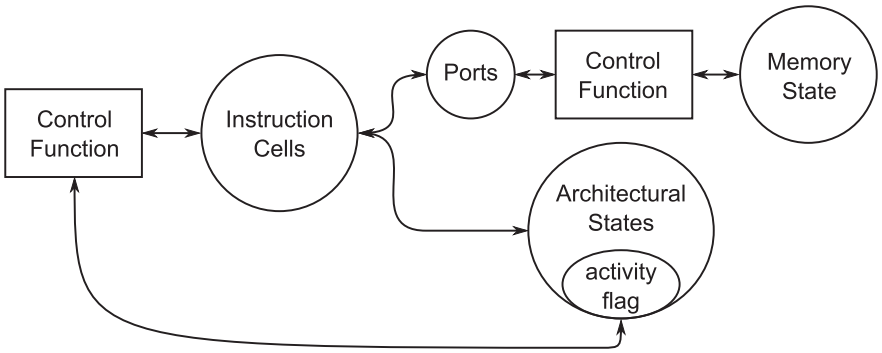


Figure 2: Simulator Components

Fig. 2 depicts the basic components of the proposed simulator. The core of the engine comprises a set of simulation methods, each of them simulating a corresponding instruction cell. Besides this, the simulator contains an internal data structure modeling the architectural state and the memory state as well as functions for controlling the flow of the simulation. One instruction can have several successors and therefore the simulator needs to represent them individually. Besides CPU registers, every architectural state also has a flag which indicates the activity of a corresponding execution path. By using this flag, the simulator can find out what program paths are active at a given point of the simulation. Once it is called, the simulator simulates all active program paths. Importantly, every execution path which is not active is considered inactive (dead) and not reachable by the program. As shown in Fig. 2, information of simulated values for registers of the architectural states can be directly accessed from the instruction cells. Data residing in memory has to be accessed via specific ports representing address and data buses. This facilitates the collection of information of addresses related to memory accesses.
Two functions are required to control the simulation flow. One control function decides which instruction cells has to be simulated next. After this, a call to the simulation method of the corresponding instruction cell is made. Simultaneously, information about active and inactive architectural states is collected. The second function tracks the memory accesses initiated by the memory ports of the instruction cells. It also collects information for every memory access (data and addresses) which is necessary for the overall model generation.

2.5.1. Simulating undetermined data and control flow

A program netlist represents the behavior of a given program along all possible execution paths. Whether a particular execution path is executed or not depends on the concrete assignments to the program inputs. When using formal methods, all valid input assignments need to be considered.

In the program netlist values of the CPU registers or of the memory that depend on the primary inputs are undetermined. This needs to be addressed when simulation methods are applied.
To solve this problem a new flag is introduced in the simulation model. This flag indicates whether the value of a register is valid or not. A valid value corresponds to a constant word stored in the register. Invalid values occur when there exists a dependency with the primary inputs. We call this an *undetermined* value. With this flag, it is possible to identify whether all bits or just a subset of them are undetermined.
Detecting individual undetermined bits is used extensively in case of branch instructions. Imagine that all bits of the status register of the CPU are invalid except the ones required for branch decisions. Without an analysis on register values at bit granularity simulation could lead to false conclusions, for example that both branches of a branch instruction can be reached by the program although in reality this is true only for one of them. Nevertheless, situations where both branches of a conditional branch instruction can be reached by the program can occur. This situation is recorded in the simulation by setting the flags for the corresponding bits in the status register of the CPU. Consequently, both branches are marked active. In these cases, the simulator then simulates both paths until no one can be simulated any further. This happens when the program has ended or an unresolved control flow instruction (a new branch/jump instruction) is found for the first time. This is the topic of the next section.

2.5.2. Computing successors of control flow instructions

As explained in Sec. 2.3 there are two situations where the control flow of a program needs to be evaluated while unrolling incomplete CFGs, namely indirect jumps and conditional branches. To handle them we implement an additional analysis in the simulation engine in order to improve performance.
If a new control flow instruction is found, while unrolling, then a new call to the simulator is made in order to define the possible successors. For the case of a conditional branch instruction this means to check the liveness of each branch. To accomplish this, the simulator traverses every preceding path ending at the control instruction that is being inspected. Taking into account that the simulation is based on concrete values, if there is a merge point preceding the control instruction, then the simulator does not merge the concrete simulated values of the architectural state. Instead of that, each path is simulated individually until the control flow instruction is reached. Note that merging concrete simulation values could introduce nondeterminism into the simulation. By avoiding this, we can save unnecessary calls to the SAT solver. For jump instructions, every path simulation can find at maximum one destination value. In the case that at least one of the simulated paths result in a invalid value then the simulation fails to resolve the control flow and consequently the SAT engine is called.

3. Experimental Results

We have implemented the techniques for efficient generation of program netlists presented in this paper and integrated them into our formal verification platform FCK (Formal Checker Kaiserslautern). This platform is implemented in C++ and can be used for automatic verification of low-level HDSW [SVF+13].
FCK parses machine code generated by GCC from C and assembler programs and produces a CFG. From the CFG, the EXG is generated by unrolling. In the process, whenever the semantic of the program needs to be evaluated in order to resolve control flow or accesses to data memory, the tool

makes calls to the simulation/SAT engines. The SAT engine is called uniquely if the simulation fails. To construct the simulator a set of instruction cells described as a set of C++ methods is used.
The following experiments evaluate the efficiency of model generation with the improvements of the proposed simulation/SAT approach. The low-level software used in the experiments is a LIN driver developed by Infineon Technologies AG, a serial-to-parallel converter (S2PCONV) and a multiplier (MULT).
The serial-to-parallel converter takes a 32-bit serial input sequence and writes it in parallel as one 32-bit word to the output. The detection of the bit sequence is done via repeated polling on a device register. It is guaranteed by the environment that after at most 5 polling accesses a valid input bit is read in.
The multiplier implements a simple 32x32-bit multiplication using addition and shifting instructions.
The LIN driver implements a master node as low-level software [LIN02]. In our version, the driver was adapted to run on the open-source 32-bit 5-stage pipelined processor Aquarius [Ait03]. The driver comprises about 1350 lines of hardware-dependent, low-level C code and inline assembly. It can be configured such that both, transmission and reception modes, are allowed. The driver interacts on one side with a UART containing different registers and on the other side with an application using shared memory.
We use the GCC compiler applying three different optimization levels to the source code, starting from the level zero LIN (l0) with no optimizations being activated, and increasing the aggressiveness of the compiler optimizations up to the maximum level two LIN (l2). LIN (multi) corresponds to a version of the LIN driver for which the ID of the message can be configured by the application. All experiments were performed on an Intel Xeon E5420 CPU at 2.5 GHz with 16 GB RAM.

Table 1: CPU times and solver calls for model generation

	CPU time (s.)		**Calls (Using Simulation)**		
Program	**SAT only**	**SAT and simulation**	**address**	**successors**	**active bit**
LIN (multi)	1791.44	41.27	574 (572)	181 (175)	424 (424)
LIN (l0)	5805.59	98.10	1136 (1136)	161 (161)	401 (401)
LIN (l1)	1062.28	27.02	569 (569)	158 (158)	398 (398)
LIN (l2)	630.74	17.89	496 (496)	75 (75)	342 (342)
S2PCONV	2920.59	2878.97	1448 (263)	0 (0)	2018 (360)
MULT	2.83	0.4	0 (0)	0 (0)	99 (99)

Tab. 1 shows the time required to generate each program netlist. Additionally, the table presents the times needed for finding memory addresses, jump successors and to decide whether branches are alive or not. As can be observed, using the combined approach for finding constant values drastically reduces the CPU time needed for model generation for the multiplier as well as for all variants of the industrial LIN driver. For model generation of the serial-to-parallel converter, however, the runtimes with and without simulation are roughly equal, as can be observed from Tab. 1. The reason for this lies in the repeated polling on the input register which results in a number of program paths that is exponential in the number of access attempts. The simulator needs to enumerate all these paths. To prevent a runtime that is longer than a SAT-only model generation the simulation is stopped if its runtime exceeds a certain limit. In this case the rest of

the model generation is done using only SAT. This also explains the low number of found addresses and active bits using simulation for the S2PCONV.
Tab. 1 also shows the total number of calls made to the combined engine, with the number of calls for which simulation succeeded in parentheses. In most experiments the simulator was able to find all needed information so that the SAT engine was not needed at all. In LIN(multi) not all addresses and successors could be calculated using simulation.
The reason is that some address computations in this benchmark were done using bit manipulation instructions. The simulator, however, cannot analyze values on bit level and was therefore unable to compute the unknown targets.

4. Conclusion

This paper presents a combined SAT- and simulation-based solution for model generation for hardware-dependent low-level software. The technique enhances an existing formal verification framework for low-level software based on program netlists. Whereas the previous approach was only able to handle complete CFGs the new combined SAT/simulation-based approach can identify information missing in the CFG and is able to compute it on-the-fly during construction of the program netlist. By a fine-grained integration of the technique into the model generation process, the computation of missing information can benefit from the already existing partial model. This significantly reduces the number of states that need to be considered.
In the experiments we show that intertwining program netlist generation with computation of missing address values based on our approach exploits synergies that result in a significant speedup of the overall model generation.

References

[AEF+05] Arons, Tamarah, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Lenore D. Zuck: *Formal verification of backward compatibility of microcode*. In *Proceedings of the 17th international conference on Computer Aided Verification*, CAV'05, pages 185–198, Berlin, Heidelberg, 2005. Springer-Verlag.

[AEO+08] Arons, T., E. Elster, S. Ozer, J. Shalev, and E. Singerman: *Efficient symbolic simulation of low level software*. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 825 –830, march 2008.

[Ait03] Aitch, Thorn: *Aquarius: a pipelined RISC CPU*, 2003. `http://opencores.org/project,aquarius`.

[CFF+06] Currie, David, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan: *Embedded software verification using symbolic execution and uninterpreted functions*. Int. J. Parallel Program., 34(1):61–91, February 2006.

[JS05] Jin, HoonSang and Fabio Somenzi: *Prime clauses for fast enumeration of satisfying assignments to boolean circuits*. In *Proc. International Design Automation Conference (DAC)*, pages 750 – 753, 2005.

[Kin76] King, James C.: *Symbolic execution and program testing*. Commun. ACM, 19(7):385–394, July 1976.

[LIN02] LIN Administration: *LIN Specification Package Rev. 1.3*, 2002. `http://www.lin-subbus.org/`.

[PV09] Păsăreanu, Corina S. and Willem Visser: *A survey of new trends in symbolic execution for software testing and analysis*. Int. J. Softw. Tools Technol. Transf., 11(4):339–353, October 2009.

[RLT+10] Reps, Thomas, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal: *There's plenty of room at the bottom: analyzing and verifying machine code*. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 41–56, Berlin, Heidelberg, 2010. Springer-Verlag.

[SVF+13] Schmidt, Bernard, Carlos Villarraga, Thomas Fehmel, Joerg Bormann, Markus Wedler, Minh Nguyen, Dominik Stoffel, and Wolfgang Kunz: *A new formal verification approach for hardware-dependent embedded system software*. IPSJ Transactions on System LSI Design Methodology, 6:135–145, 2013.

[VSB+13] Villarraga, Carlos, Bernard Schmidt, Christian Bartsch, Joerg Bormann, Dominik Stoffel, and Wolfgang Kunz: *An equivalence checker for hardware-dependent software*. In *11. ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 119–128, 2013.

Dynamically Reconfigurable Constant Multiplication on FPGAs

Konrad Möller
University of Kassel
Kassel
konrad.moeller@uni-kassel.de

Martin Kumm
University of Kassel
Kassel
kumm@uni-kassel.de

Björn Barschtipan
University of Kassel
Kassel
bjoern_barschtipan@web.de

Peter Zipf
University of Kassel
Kassel
zipf@uni-kassel.de

Abstract

Multiplication with constants is one of the most important operations in digital signal processing including digital filters and linear transformations like the fast Fourier transform. Runtime reconfiguration of the multiplied coefficients, also called reconfigurable single constant multiplication (RSCM), is an efficient method to reduce resources by time-multiplexing and resource sharing. There are several algorithms to realize RSCM on ASICs by fusing multiple single constant multipliers with multiplexers. This may lead to large multiplexers which do not fit well to the structure of FPGAs, so alternative methods are needed. There is one FPGA specific algorithm (called ReMB method) to generate RSCMs by utilizing an FPGA specific basic element based on 4-input Look-Up Tables (LUT) in Virtex 4 FPGAs. This paper investigates an extension of this heuristic algortihm utilizing the 6-input LUTs in recent FPGAs. It is shown that a reduction of the required basic elements of 18% on average can be achieved by this approach. Moreover, convergence in terms of finding a valid RSCM solution can be guaranteed. This was not the case in the original algorithm.

1. Introduction

Constant coefficient multiplication is the basic operation in many applications of digital signal processing like digital filtering and linear transformations. Thus, much research has been done to find optimal solutions for this operation often referred to as single constant multiplication (SCM). The most efficient way to implement SCM on application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) is multiplierless by a so called adder graph which only consists of additions, subtractions and bit shifts. Though finding an optimal solution for such graphs is NP-complete [BH91], there are optimal solutions for constants of up to 12 bits [DM94], up to 19 bits [GDW02] and up to 32 bits [TN10]. Besides this, there are good heuristics called RAG-n, BHM [BH91] and H_{cub} [VP07] to generate SCMs. Online generators which use these

heuristics are available on the web page of the SPIRAL project [SPI13]. The focus of this paper is on the dynamic reconfiguration of SCMs. One motivation is the reuse of resources by time multiplexing to minimize the design size when the timing is not critical. Another interesting aspect is the use of reconfigurable single constant multiplier blocks (RSCMs) as basic blocks of filters with changing characteristics, e. g., for different frequency regions and adaptive filters. There are several algorithms to create such RSCMs by fusing independently optimized SCMs [THP07,CC09] which where optimized for ASICs. In [THP07] the authors suggest to fuse several optimized SCM graphs by a recursive algorithm called DAG fusion. The first step is a fusion of two SCM graphs with minimal hardware costs. Reconfiguration is realized by multiplexers to switch between the different coefficients. To include more than two coefficients the related SCMs are recursively added to the existing RSCM in the same way. Another ASIC-optimized algorithm to realize RSCMs [CC09] tries to exploit similarities between different coefficients with their canonical signed digit (CSD) representation. Common subexpression elimination (CSE) reduces the number of required adders by searching and fusing identical patterns in the CSD representation of the coefficients. The different shifts and interconnections to realize a specific coefficient are switchable through the introduction of multiplexers. These two approaches lead to large multiplexers and thus are not ideal for FPGAs. There is only one algorithm (called ReMB method) by Demirsoy et al. to generate RSCMs which is optimized for FPGAs [DDK03, DKD05a, DKD05b, DKD04, DKD07]. It is an approach that constructs an RSCM of basic structures that fit into the basic logic elements (BLE) of FPGAs. Therefore, this method was taken as a starting point of our investigation on dynamically reconfigurable constant multiplication on FPGAs. After the analysis of the original algorithm (section 2) some extensions where made to utilize larger LUTs in recent FPGAs and to solve a convergence problem found in the description of the original algorithm (section 3). The experimental results are shown in 4. In section 5 the next steps to be done are reflected and section 6 gives a conclusion.

2. ReMB Method

2.1. Basic Concept of the Algorithm and Metrics

The basic idea of the algorithm is to adapt the basic graph structure to the BLE structure of the underlying FPGA to realize a reconfigurable multiplication of an input x with a constant c_i, where i is the index of the specified coefficients in the coefficient set $C = \{c_1, c_2, ..., c_i\}$. The starting point of Demirsoy et al. is the Virtex 4 BLE structure with a 4-input LUT followed by the arithmetic carry chain. This can be used as a basic element (BE) with a 2:1 multiplexer and an adder as it is shown in Figure 1 a. The arrows denote a specific left shift l_x which equals a power of two multiplication of the corresponding input. The shown BE implements the equations

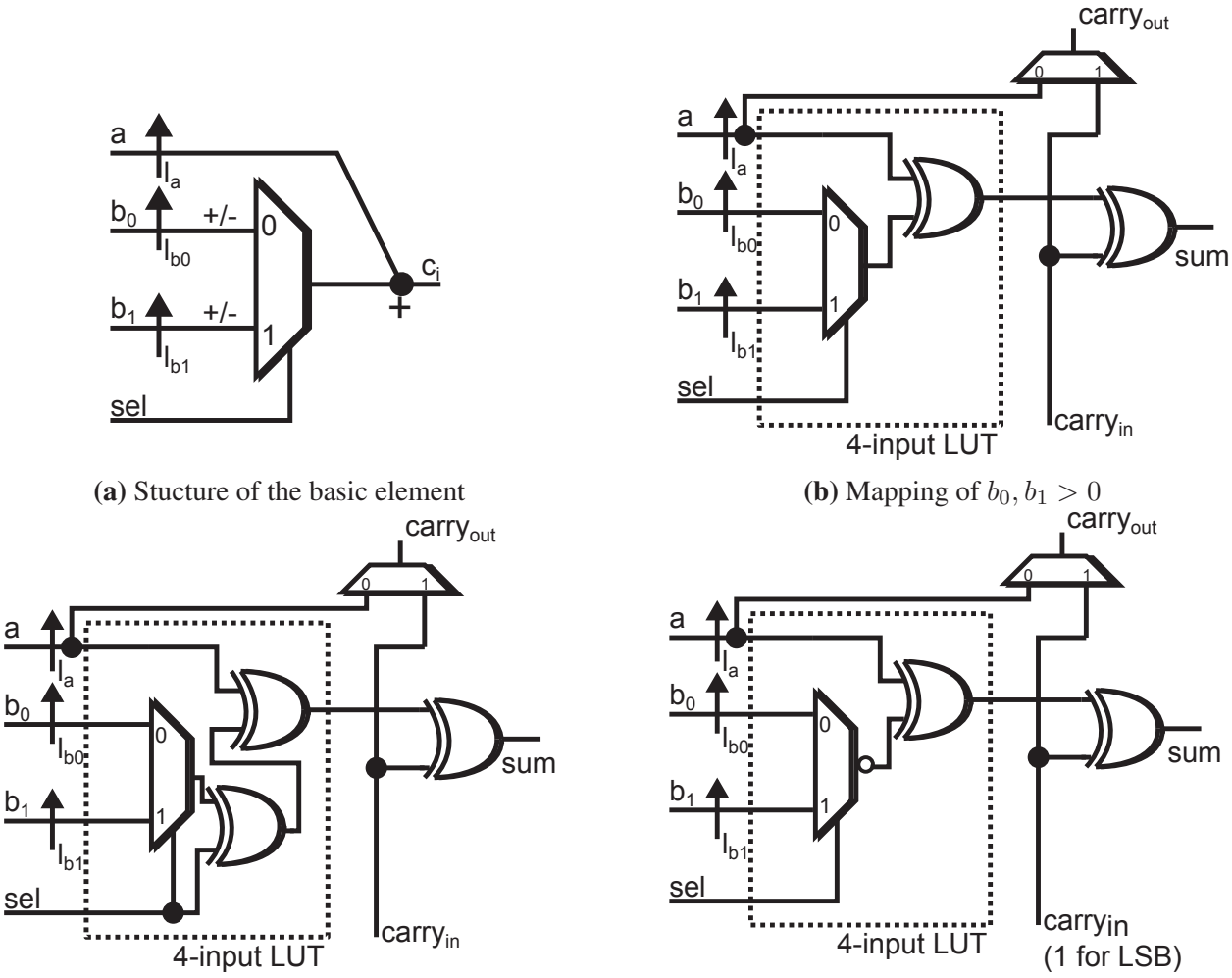
$$c_i = a2^{l_a} \pm b_0 2^{l_{b0}} \text{ or } c_i = a2^{l_a} \pm b_1 2^{l_{b1}} \tag{1}$$

respectively, depending on the selected multiplexer input sel and the sign of b_0 and b_1. In the hardware implementation the shifts can be realized by wires and the addition can be separated into the sum and $\text{carry}_{\text{out}}$ output for each output bit following the relations

$$\text{sum} = a \oplus b \oplus \text{carry}_{\text{in}} \tag{2}$$

$$\text{carry}_{\text{out}} = (a \oplus b)\text{carry}_{\text{in}} + \overline{(a \oplus b)}a \tag{3}$$

for the example in Figure 1 (a). This can be mapped to the LUT and carry chain of a single BLE per bit as it is shown for (1) in Figure 1 (b) for $b_0, b_1 > 0$, (c) for $b_0 > 0, b_1 < 0$ and (d) for $b_0, b_1 < 0$.



(a) Stucture of the basic element

(b) Mapping of $b_0, b_1 > 0$

(c) Mapping of $b_0 > 0, b_1 < 0$

(d) Mapping of $b_0, b_1 < 0$

Figure 1: Mapping of the basic element (a) to the Virtex 4 FPGA BLE (b-d).

Following (1) one BE can produce two possible outputs. One key element of the algorithm is to determine the number of required BE stages to realize a specific coefficient set C. A horizontal cascading of k stages of BEs leads to the definition of the number of different possible constant factors N_{output_k}

$$N_{\text{output}_k} = 2^{k^2-1} \tag{4}$$

beginning with $N_{\text{output}_1} = 2$. An example to show this relation is illustrated in Figure 2. To put it another way each BE can reduce the number of required outputs in an output set that have to be realized, starting with the given coefficient set C as initial output set, until the BE with a single input (x) is reached. The minimum number of required stages in the cascade to realize one specific coefficient is called depth$_{\text{adder}}$. The number of required stages to realize a specific number of different coefficients is called depth$_{\text{mux}}$. These two definitions are necessary to classify

a coefficient set for a RSCM. This circumstance shall be justified by an example. On the one hand it is not possible to realize the coefficient set $C = \{5, 23\}$ with a single BE, because 23 requires at least two basic operations of the kind denoted in (1). On the other hand the set $C = \{5, 7, 9, 15, 31\}$ can only be realized with at least two cascaded BEs (k=2 required (4)) though each single element could be realized with only one BE. The two criteria can be summarized to the basic structure depth (BSD) with the following norm:

$$\text{BSD} = max(\text{depth}_{\text{mux}}, max(\text{depth}_{\text{adder}})) \tag{5}$$

To solve the problem of finding a valid RSCM for the aimed-at coefficient set and thus achieve convergence, the algorithm reduces the BSD with each stage of the cascade.

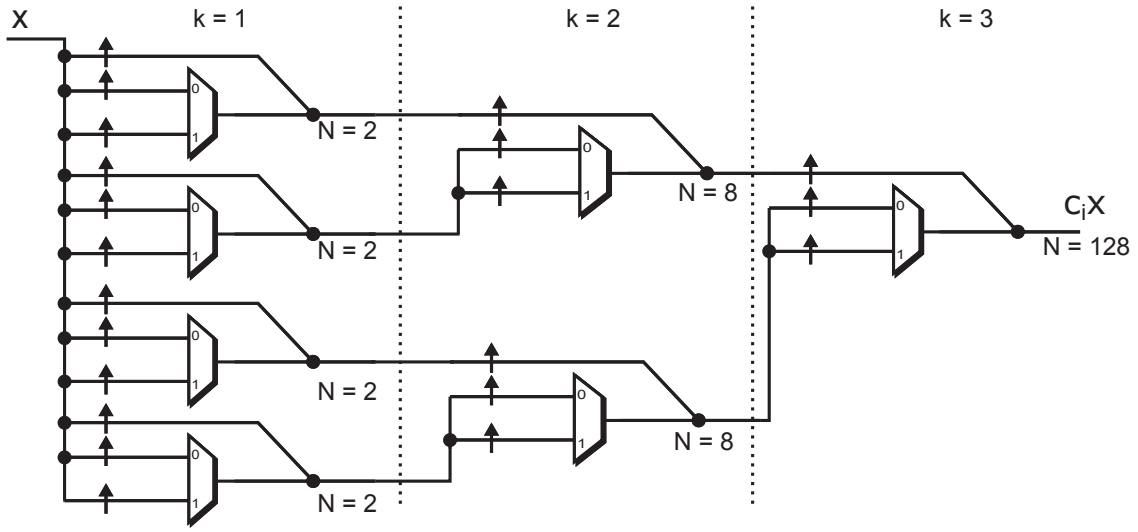


Figure 2: Example of a cascading of several basic elements, N is the number of possible output coefficients.

2.2. The Algorithm

The starting point of the algorithm as shown in Listing 1 is the coefficient set C. That means the generation of the RSCM starts from the output stage (line 1). In each iteration of the algorithm one stage of BEs is determined (lines 4-6) until an input coefficient of 1 is reached. This can be achieved by reducing the BSD in each stage. For that purpose a so called graph table for each coefficient is created (line 4). The graph table of each coefficient contains all combinations of inputs and shifts which fulfill the equation $c_i = a2^{l_a} \pm b2^{l_b}$ (cf. (1)). In the next step the graph tables of all coefficients are combined to fit into basic elements (line 5). This is done by a so called index table that lists the indices of entries in the coefficients' graph tables which can potentially be combined. These can be determined by searching for similarities (e. g. same shift value, same input) of graphs in the graph tables. Beyond that, solutions which are not valid are excluded. Non valid solutions are combinations which would not reduce the BSD or which can not be realized by the BE (an example is given below). Finally, the best combinations in the index table which cover all coefficients are selected (line 6). The choice for the best combinations is made by a score function s defined as

$$s = \sum_i \omega_1(\omega_2 N_1 - \omega_3 \sigma_i^2 + \omega_4(\text{BSD}_o - \text{BSD}_i)) \tag{6}$$

Listing 1: Pseudo code of the algorithm

```
1 active_output_sets = C
2 for i = BSD downto 1
3         for j = size(active_output_sets) downto 1
4                 g = graph_tables(active_output_sets(j))
5                 comb = combine(g)
6                 best_stage_sets(j) = best_combinations(comb)
7         active_output_sets = inputs(best_stage_sets)
```

It is a weighted sum depending on a factor that indicates whether all inputs are 1 (N_1), the variance of adder$_{\text{min}}$ of the inputs (σ_i^2) as well as the difference of BSD of the outputs and inputs of the basic element. In the last step the best combinations are stored as BEs for the specific stage and their inputs are the new output sets for the next iteration (line 7). The selected BEs are optimal referred to the score function for the specific stage, because all possible graph combinations are covered. The impact of this choice on the preceding stage is not considered. Thus, the algorithm does not find a globally optimal solution.

Figure 3 shows an example result of the heuristic algorithm with an exemplary coefficient set $C = \{39, 45, 41, 47\}$. In the first iteration the BE with the inputs $\{33, 31\}$ unshifted, $\{1\}$ shifted by 3 and $\{7\}$ shifted by 1 turn out to be the best choice. In the second iteration the two input sets $\{33, 31\}$ and $\{7\}$ are the new active output sets. Input set $\{1\}$ is the input of the RSCM and thus does not have to be processed in the following iterations. The new active output sets can be realized by one BE with $\{1\}$ shifted by 5, $\{1\}$, $\{1\}$ negated for the output set $\{33, 31\}$ and one BE with $\{1\}$ shifted by 3 and $\{1\}$ negated for the output set $\{7\}$. Finally each input set is $\{1\}$ (overall BSD is 0) and the algorithm stops.

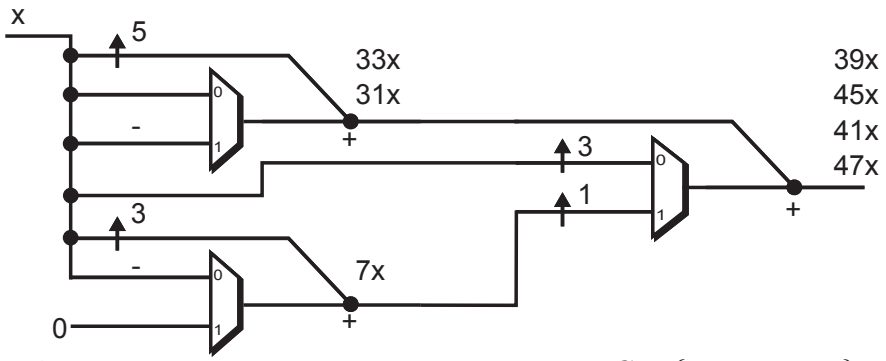


Figure 3: Example RSCM with the coefficient set $C = \{39, 45, 41, 47\}$.

The basic idea of the algorithm is very good as it considers the specific hardware structures the resulting SCM can be mapped to. One disadvantage of the implementation as it is described in [DKD07] is the fact, that not all pairs $\{c_1, c_2\}$ can be realized by a basic element and thus only reducing the BSD does not guarantee that a valid solution can be found (convergence of the algorithm). An example pair is $\{5, 7\}$ which is only realizable as $1 + 4$ and $-1 + 8$ respectively. This case is not covered by (1), because it equals the general pair $\{a2^{l_a} \pm b_0 2^{l_{b0}}, -a2^{l_a} \pm b_1 2^{l_{b1}}\}$. For such cases the authors suggest to search for alternative graphs. In cases where the only valid graph consist at least of one such element the algorithm will not find a solution. So it's a debatable point

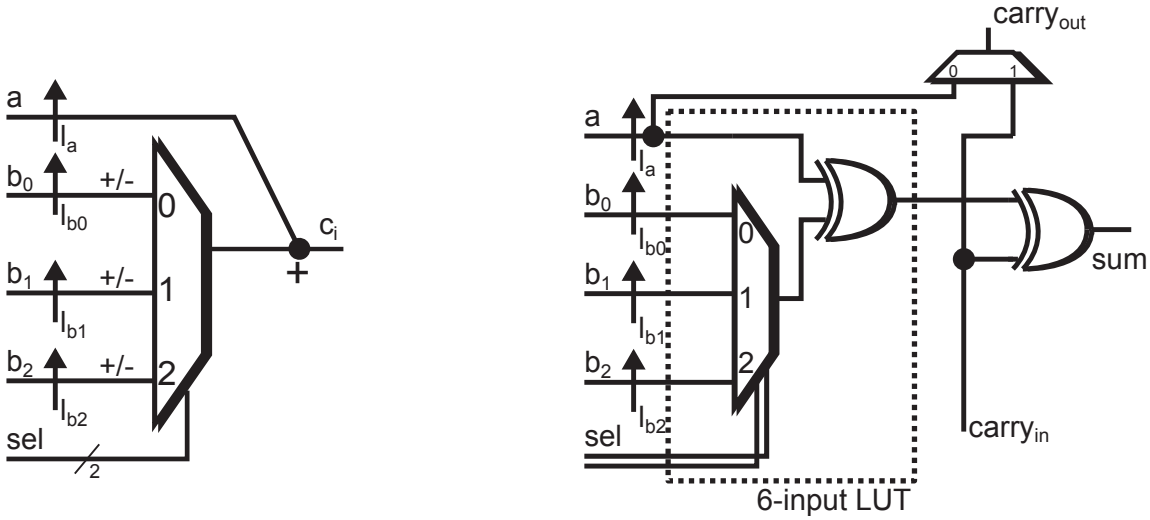
whether the definition of the BSD (5) is valuable. Another disadvantage is, that no negative coefficients are supported by the current implementation of the original algorithm. Negative coefficients during the optimization could in some cases reduce the complexity of the resulting SCM as it was shown by examples but not included in the algorithm in [DKD07]. Further drawbacks are analyzed in section 4.

3. Extended Algorithm

This section shows our extension of the original algorithm for a recent FPGA with 6-input LUTs. A Xilinx Virtex 6 FPGA is taken as an example for the recent Xilinx and Altera FPGAs which provide 6-input LUTs. In the first part new basic elements are introduced to reduce the required resources by BEs with more inputs and other functionality. In the second part the applied changes to the original algorithm are presented.

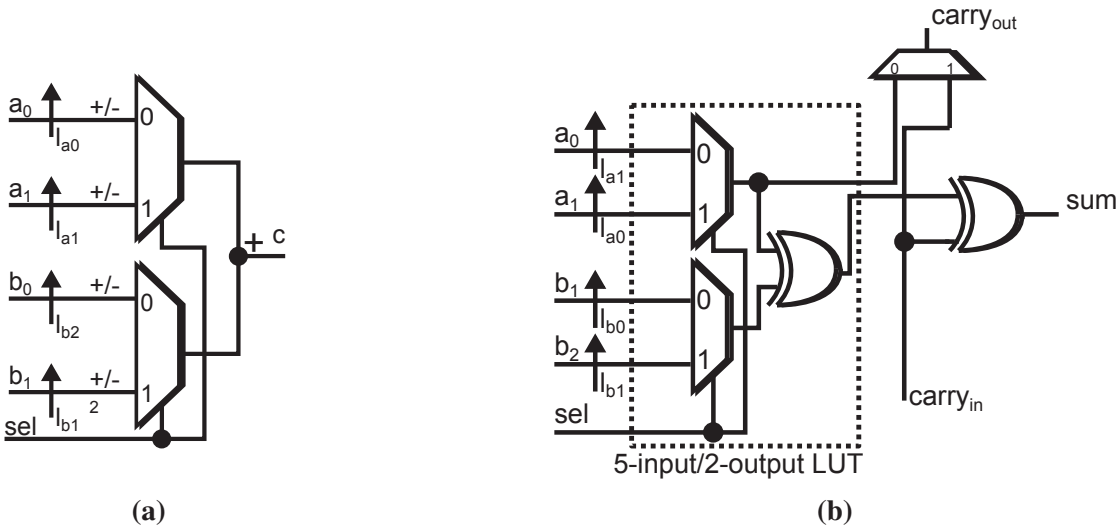
3.1. New Basic Elements

Our straight forward approach is to include a 3:1 multiplexer and an adder into one basic element BE1 which replaces the 2:1 multiplexer-adder-combination of the original algorithm. The realization of that element is shown in Figure 4 (a) and (b). Besides an addition of the inputs b_x each combination with subtracted inputs is possible by analogy with Figure 1 (b-d). The second basic element BE2 (Figure 5 (a) and (b)) uses a feature of the LUTs in Virtex 6 FPGAs. These can be used as a 6-input LUT with one output and as a 5-input LUT with two outputs. The 5-input LUT is used to realize two 2:1 multiplexers whose outputs are added. This was not possible with a Virtex 4 LUT, which offers only one output. BE2 is motivated by the example given in the previous section $C = \{5, 7\}$. This can now be implemented with BE2 by switching between $1 + 4$ and $-1 + 8$ which was not possible in the original algorithm. The basic element in the original algorithm did not support an implementation of $\pm a$ which is now possible through the use of two different a-inputs (a_1 and a_2).



(a) **(b)**

Figure 4: Structure (a) and implementation (b) with a 6-input LUT of basic element BE1.



(a) **(b)**

Figure 5: Structure (a) and implementation (b) with a 2-output/5-input LUT of BE2.

3.2. Changes to the Original Algorithm

The main advantage of our extension of the BE used in the original algorithm to BE1 is that now not two but three coefficients can be realized by one BE1. Thus the number of required basic elements is expected to be reduced by a factor of $\frac{2}{3}$ in the best cases. To include the new BE1 into the algorithm steps 5 and 6 in Listing 1 were changed. More combinations are possible and valid in the extended algorithm, because the definition of the N_{output_k} (4) and thus the BSD calculation were adapted to the larger multiplexer size.

Besides the implementation and integration of the new basic elements BE1 into the algorithm some other changes were made. These are necessary to guarantee that the algorithm finds a valid solution. The problem with the coefficient sets of the kind $C = \{5, 7\}$ that could not be integrated into the SCM graph is solved by splitting C into two sets $C_1 = \{5\}, C_2 = \{7\}$ which are processed separately (called splitting in the following). This leads to the insertion of an additional BE1 for the seperate processing but guarantees convergence of the algorithm.

In order to compensate the drawback of the additional BE1 an operation called fusion, which is possible through the introduction of BE2, was attached at the end of the algorithm. The fusion searches for underutilized BE1s. These are BE1s which result from splitting and have only one used multiplexer input. A pair of those BE1s can be fused to one BE2, like it is shown in Figure 6. This leads to a decrease in the BE overhead caused by splitting. To see the effect of the fusion we generated 92 random graphs with 2-7 coefficients and 6-8 bit word size. In 42 cases splitting was required to get a valid solution with an overall BE overhead of 40%. This means that in the original algorithm no valid solution could be guaranteed for these cases. The BE reduction by fusion was successful in all example cases and lead to a complete compensation of the BEs which were inserted by splitting.

Thus, the improvements of the new algorithm are that it can reduce the number of required basic elements and that a valid solution for all positive input coefficients can be found by splitting and fusion, without additionally required BE resources for our random examples.

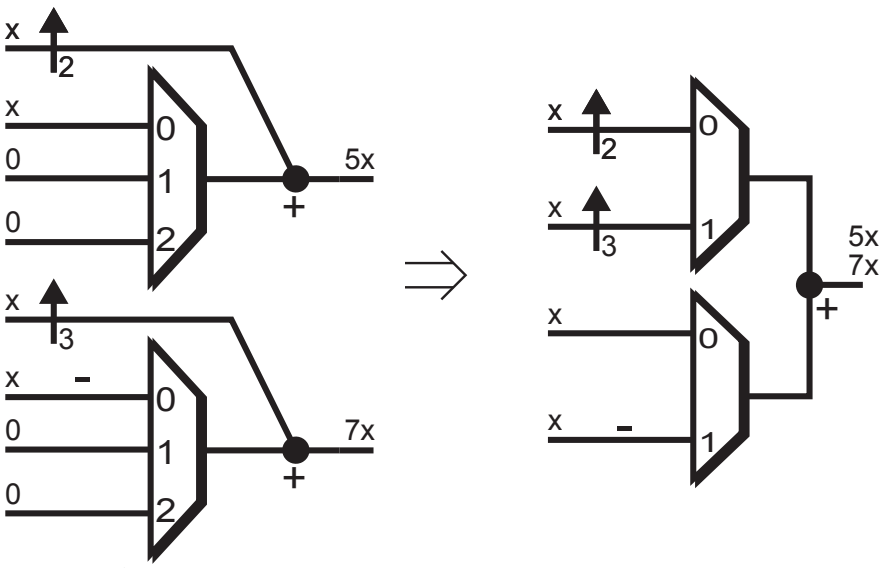


Figure 6: Example for the fusion of two BE1s to one BE2

4. Experimental Results

As there is no analysis of the complexity of the algorithm in previous work we analyse in this section the complexity in terms of resulting graph combinations for different numbers of coefficients per set. Besides this we compare the number of required basic elements for the original algorithm to our extended algorithm to show the impact of the new basic elements to the resulting design size. Finally we compare results of mapping our extended algortihm to mapping a state of the art RSCM method for ASICs (DAG fusion) to a Virtex 6 FPGA to justify specific optimization of RSCM graphs for FPGAs.

4.1. Complexity

Finding an optimal RSCM is a generalization of the SCM problem and therefore also NP-complete. In both versions of the algorithm, the original one and our extended version, all possible graph tables for each coefficient as well as all combinations of these tables are included in the search for the best basic structures of each stage which seems to be very inefficient. The complexity of this search was quantified by 92 graphs already used for the experiment in the section 3.2. The average number of possible graph combinations that have to be evaluated by the score function are shown in Figure 7. An exponential growth of complexity with the number of coefficients in C can be seen. This leads to a memory bottleneck for larger coefficient sets. Putting it on a level with the execution time of the algorithm, the average execution time for seven-constant sets was 155 seconds for an Intel Core i7-3610QM, 2,3 GHz . The memory bottlenecks for set sizes larger than seven in our Matlab implementation confirmes that the exhaustive search in each stage is very inefficient and the realization of larger coefficient sets seems to be impracticable.

4.2. Comparison to the Original Algorithm

The extension of the basic element with a 2:1 multiplexer to a basic element with a 3:1 multiplexer is expected to reduce the number of required basic elements (and thus the required basic logic

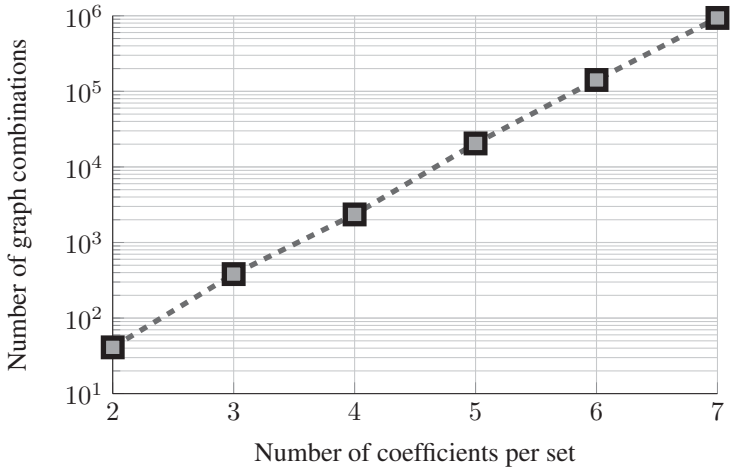


Figure 7: Complexity for different numbers of coefficients per set

elements in the FPGA). This reduction is shown by examples (Table 1) taken from [DKD07] and amounts to 18% reduction on average for our proposed algorithm in these three cases. It can be observed that even for these small examples, considerable reductions can be achieved.

Table 1: Required basic elements of the ReMB method and its extension

coef. set C from [DKD07]	BEs ReMB	BEs ext. ReMB (proposed)
{137,119,113,143}	4	**3**
{1,2,4,8,15,26,50,162,256}	7	**5**
{39,45,41,47}	**3**	**3**

4.3. Comparison to DAG Fusion

As already stated in the introduction, the evaluated algorithm is the only algorithm which has been designed for the RSCM optimization for FPGAs. To see whether it is really necessary to have a specialized FPGA optimization algorithm the results of the proposed algorithm are compared to the results of the DAG fusion algorithm for some examples taken from [THP07] and both synthesized on a Virtex 6 FPGA. The realizations for DAG fusion were created with the SPIRAL online tool [SPI13] that outputs synthesizable Verilog code. It can be seen that less resources (81% of the resources of DAG fusion on average) are required for the result of the specialized FPGA optimization algorithm. These results justify to put some effort into FPGA specific optimization of RSCM graphs.
A general observation of the experiments in 4.1, 4.2 and this section is that the complexity, execution time and resulting design size is heavily dependent on the particular coefficient set.

5. Next Steps to be Done

Some optimization ideas to reduce the number of evaluated graph combinations are already stated in [DKD07]. These inlcude the reduction of the search space by omitting some graphs of the graph table. But there are no strict statements in the text which parts of the search space can be neglected.

Table 2: Comparison of the slices required by DAG fusion and extended ReMB algorithm results

coef. set C from [THP07]	slices DAG fusion	slices ext. ReMB (proposed)
{362,392,473}	25	**20** (80%)
{1,2,4,8,15,26,50,162,256}	28	**16** (57%)
{39,150,196}	21	**18** (86%)
{39,45,41,47}	**11**	**11** (100%)

Another idea is to discard non-valid graph combinations. In the basic algorithm all possible graph combinations are generated and thrown out of the set of possible solutions at the evaluation of the score function. The proposed optimizations for the original algorithm have to be included in our extended algorithm, with some changes. Our idea is to reduce the search space by already rating the specific graphs in the graph tables of the coefficients with weighting functions. Graph tables which bring about non-valid graph combinations or have a low rating will not be included in the graph table of the specific coefficient and thus are not combined with graphs of other coefficients. This will lead to considerable reductions in the number of possible graph combinations which have to be stored and rated by the score function. Moreover the integration of negative coefficients will be included in our algorithm. The current version of our algorithm is written and executed in MATLAB. It generates the RSCM graph and outputs synthesizable VHDL code. The generation of the graph tables as well as the generation and evaluation of the graph combinations results in long runtimes. To obtain better runtime results the algorithm will be implemented in C++.

6. Conclusion

In this paper an algorithm from literature to generate FPGA optimized RSCMs was analysed and extended. The extension of the basic element used in the original algorithm to two basic elements with larger input sizes leads to a reduction of required basic elements of 18% on average. The new basic elements are based on 6-input LUTs which are available in recent FPGAs. Furthermore the convergence in terms of finding a valid solution could be ensured, which was not the case in the original algorithm. Besides this, the analysis of the complexity revealed some general drawbacks of the algorithm: For larger coefficient sets, there are too many graph combinations which have to be stored and scored. This characteristic already existed in the original algorithm and has to be eliminated by future work. The experimental results have shown that the basic element approach is an interesting method for FPGA specific optimization of RSCMs. The FPGA specific approach needed only 81% of the FPGA resources on average compared to the ASIC optimized RSCM algorithm DAG fusion. This reduction has to be confirmed by a larger set of examples. A future requirement is to reduce the drawbacks of the presented approach to make the algorithm also available for larger coefficient sets.

References

[BH91] Bull, D. R. and D. H. Horrocks: *Primitive Operator Digital Filters*. Circuits, Devices and Systems, IEE Proceedings G, 1991.

[CC09] Chen, J. and C. H. Chang: *High-Level Synthesis Algorithm for the Design of Reconfigurable Constant Multiplier*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2009.

[DDK03] Demirsoy, S. S., A. G. Dempster, and I. Kale: *Design Guidelines for Reconfigurable Multiplier Blocks*. Circuits and Systems (ISCAS) Proceedings of the International Symposium on, 2003.

[DKD04] Demirsoy, S. S., I. Kale, and A. G. Dempster: *Efficient Implementation of Digital Filters Using Novel Reconfigurable Multiplier Blocks*. In *Signals, Systems and Computers. Conference Record of the Thirty-Eighth Asilomar Conference on*, 2004.

[DKD05a] Demirsoy, S. S., I. Kale, and A. G. Dempster: *Synthesis of Reconfigurable Multiplier Blocks: Part - II Algorithm*. Circuits and Systems (ISCAS) IEEE International Symposium on, 2005.

[DKD05b] Demirsoy, S. S., I. Kale, and A. G. Dempster: *Synthesis of Reconfigurable Multiplier Blocks: Part I - Fundamentals*. Circuits and Systems (ISCAS) IEEE International Symposium on, 2005.

[DKD07] Demirsoy, S. S., I. Kale, and A. G. Dempster: *Reconfigurable Multiplier Blocks: Structures, Algorithm and Applications*. Circuits, Systems and Signal Processing, 2007.

[DM94] Dempster, A. G. and M. D. Macleod: *Constant Integer Multiplication Using Minimum Adders*. In *Circuits, Devices and Systems, IEE Proceedings*, 1994.

[GDW02] Gustafsson, O., A. G. Dempster, and L. Wanhammar: *Extended Results for Minimum-Adder Constant Integer Multipliers*. Circuits and Systems (ISCAS). IEEE International Symposium on, 2002.

[SPI13] SPIRAL-Project: *http://www.spiral.net*, 2013.

[THP07] Tummeltshammer, P., J. C. Hoe, and M. Puschel: *Time-Multiplexed Multiple-Constant Multiplication*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2007.

[TN10] Thong, J. and N. Nicolici: *A Novel Optimal Single Constant Multiplication Algorithm*. In *Design Automation Conference (DAC), 47th ACM/IEEE*, 2010.

[VP07] Voronenko, Y. and M. Puschel: *Multiplierless Multiple Constant Multiplication*. Transactions on Algorithms (TALG), 2007.

Efficient High Speed Compression Trees on Xilinx FPGAs

Martin Kumm
University of Kassel
Kassel
kumm@uni-kassel.de

Peter Zipf
University of Kassel
Kassel
zipf@uni-kassel.de

Abstract

Compressor trees are efficient circuits to realize multi-operand addition with fast carry-save arithmetic. They can be found in various arithmetic applications like multiplication, squaring and the evaluation of polynomials with application to function approximation. Finding good elementary compressing elements on FPGAs is a non-trivial task as an efficient mapping to look-up tables and carry-chain logic has to be found. It was shown recently that common ternary adders on modern FPGAs outperform previous compressors in terms of area-efficiency, i. e., the number of compressed bits per logic unit. While Altera FPGAs allow a fast and compact implementation of ternary adders we observed that ternary adders on Xilinx FPGAs only achieve about half of the speed of a two-input adder. This work proposes novel compressing elements including different generalized parallel counters and a 4:2 compressor which is based on a modified ternary adder. They provide a better area-efficiency and/or a lower delay than previously proposed compressing elements.

1. Introduction

Various fundamental arithmetic operations involve the addition of multiple numbers. These include multipliers (real/complex), polynomials (incl. squarer) and linear transforms. On FPGAs, efficient implementations for soft-core multipliers [GAKC09, PAI11], large size multipliers using multiple DSP blocks and compressor trees [GAKCL12, BdDI+13], complex multipliers and multivariate polynomials for sine approximation using its Taylor series [BdDI+13] are reported in the literature. The implementation of multi-operand adders on application specific integrated circuits (ASICs) is most of the times best realized by using compression trees. Compared to binary adder trees using common carry propagate adders (CPAs), compression trees typically provide a much higher speed due to the avoidance of long carry chains with nearly the same area requirements. For FPGAs, the situation looks much different. Here, on the one hand, specific carry logic provides the implementation of fast ripple carry adders (RCAs) and, on the other hand, the delay of a simple wire in the routing fabric is much higher. However, only a fraction of a look-up table (LUT) is used when implementing two-input RCAs. Therefore, the main idea for good basic compression elements on FPGAs is to find a way to use the LUTs for additional computations.
A 4:2 compressor for Xilinx' four-input LUT FPGAs with carry logic (Virtex 2/4 and Spartan 2/3) was proposed by Ortiz et al. [OQH+09]. They mapped two cascaded full adders into a single slice

to compress 4 inputs (plus carry-in) to two outputs (plus carry-out). With the advent of modern FPGAs providing up to six-input LUTs, the possibilities for mapping additional compression elements in the LUT are much larger. Fundamental work on this was done by Parandeh-Afshar et al. [PANBI11, PA12] which also offers a good introduction to state-of-the-art compression trees on FPGAs. They used LUTs but also short carry chain paths (2 to 3 full adders in series) to construct generalized parallel counters (GPCs, see Section 2.1 for its definition) as efficient compression elements. They can be used as library elements for the automatic synthesis of compression trees. The problem here is to find a good assignment of the compression elements to the inputs which is always a tradeoff between area and delay. The heuristic algorithm in [PANBI11] focuses on a good utilization of the used compression elements, i. e., the compression element which inputs fit best to the remaining compressor tree inputs is chosen.
An alternative mapping of efficient compressor trees on FPGAs was proposed recently by Hormigo et al. They arranged common carry-save adders (CSA) and ternary adders in a diagonal way such that short paths of the FPGA carry-chain are utilized [HVZ13]. While the previous discussed compressing elements are also very effective for pipelining, the diagonal arrangement may lead to a lot of additional pipeline registers to balance the pipeline.
Another compression tree optimization heuristic was proposed recently by de Dinechin et al. [BdDI+13]. They proposed to represent the problem (i. e., all input bits that have to be added) as a data structure which they call a bit heap. It is a data representation of dot diagrams [EL04] which is a great abstraction of compression trees and is also very beneficial from a software engineering point of view. Their algorithms are available in the open-source arithmetic core generation framework FloPoCo [dDP12, flo13]. They analyzed compression elements for their efficiency and use the most efficient ones in their heuristic instead of forcing a good utilization. Another conclusion from their efficiency analysis was that the ternary adder is the best compression element in terms of efficiency. However, as it was shown by our group, ternary adders are slow compared to common two-input adders on Xilinx FPGAs [KHW+13]. While the performance penalty for Altera FPGAs is 5 to 10% up to 32 bit, it is close to 50% for Xilinx FPGAs which makes the ternary adder unattractive for high speed applications.
The main contribution of this work is the introduction of four new compression elements for Xilinx FPGAs with improved efficiency and less delay. The most efficient and flexible one with highest speed is a 4:2 compressor which is based on a modified ternary adder. It avoids the LUT and routing delays that occur in the ternary adder and provides the same efficiency. Furthermore, it can be pipelined without overhead and can be well integrated into pipelined compressor trees. It was shown by our group that pipelining plays a crucial role in adder-based arithmetic for multiple constant multiplication (MCM) which is similar to the situation in compressor trees [KZ11]. Average speedups of 300% were achieved by spending 18% more slice resources. However, the optimization of pipelined compressor trees was not treated in the literature so far. We show exemplarily that using pipelined compression elements results in fast and compact compressor trees.

2. Previous Work on Compressing Elements

2.1. Classification

The inputs of an n input multi-operand adder where each input i has a word size of b_i bits can be represented as a $n \times b$ bit-array (where b is the maximum of the input word sizes b_i). If a_{ij} denotes

the input bit of row i (input operand) and column j (bit weight) of the bit-array, the corresponding output value v of the multi-operand addition is [EL04]:

$$v = \sum_{i=0}^{n-1} \sum_{j=0}^{b_j-1} a_{ij} 2^j \tag{1}$$

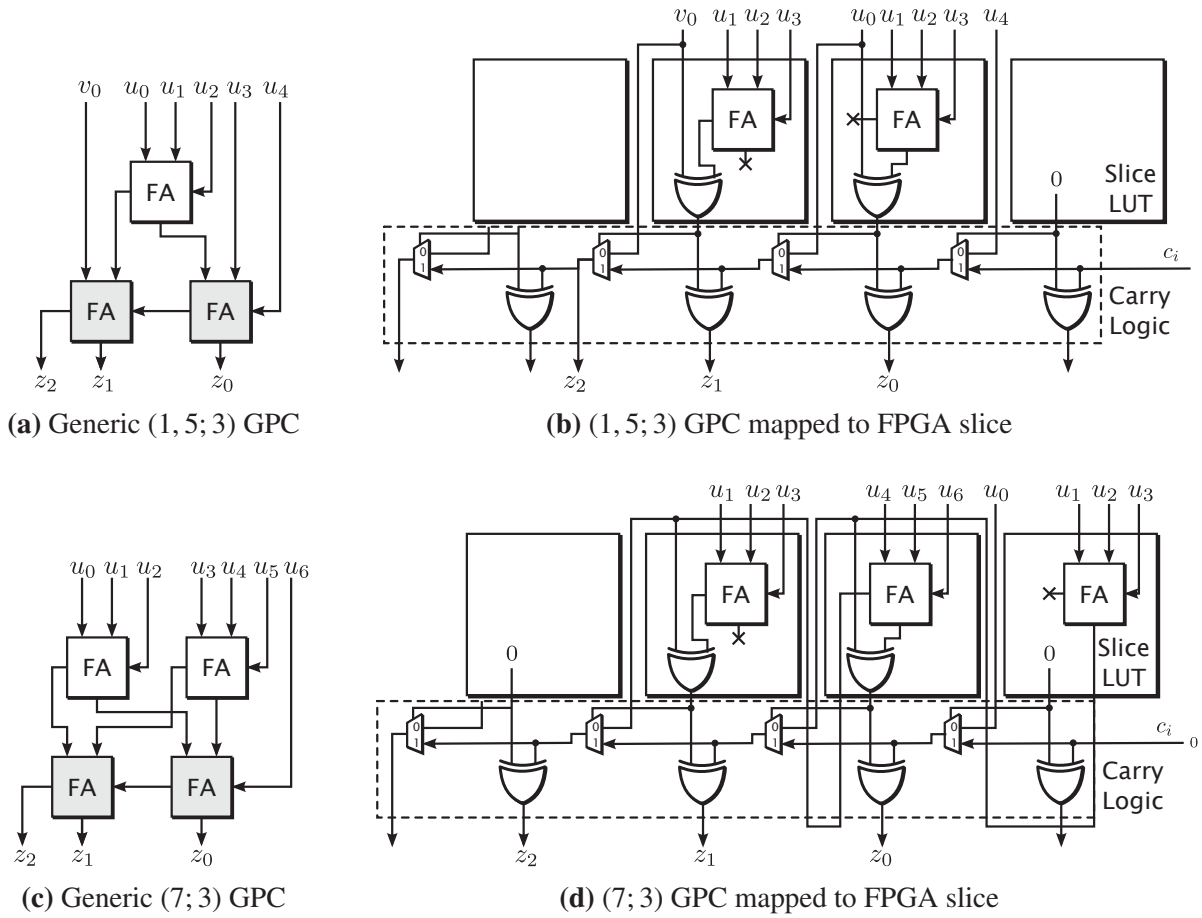
The basic operation in a compressor tree, which we call compressing element, takes a bit-array and produces a smaller bit-array. Compressing elements can be classified by their reduction of rows or columns (or both). A *counter* (or parallel counter) is a circuit that counts the number of input bits of a column which are one. Half-adder and full-adders are examples of 2:2 and 3:2 parallel counters, respectively. Here, $p : q$ denotes number of inputs (p) and outputs (q). *Generalized parallel counters* (GPCs) [PANBI11] which are also called multicolumn counters [EL04] allow that input bits may have different weights (and are thus in different columns). A GPC is commonly denoted as tuple $(p_{k-1}, p_{k-2}, \ldots, p_0; q)$, where p_j represents the number of input bits of weight 2^j and q is the number of output bits. A (3,5;4) GPC, for example, computes the sum of 3 input bits of weight two plus 5 input bits of weight one. The result is a number in the range $0 \ldots 2 \cdot 3 + 5 = 11$ and is represented by a single bit vector with 4 bit. A $p : q$ *compressor* is a circuit that reduces p input bits of the same weight to q output bits with possibly different weights [PA12]. They also provide explicit carry-in and carry-out signals which are *not* counted as input or output bits. To avoid long carry chains, the carry-out typically does not depend on the carry-in. Cascading b $p : q$ compressors reduces p bit vectors of word size b to q bit vectors in a redundant representation. Note that the notation of a compressor is not consistent in the literature as, e. g., full-adders or carry-save adders are sometimes also referred to as 3:2 compressors.

2.2. Generalized Parallel Counters on FPGAs

A set of GPC mappings to FPGAs which utilize LUTs and fast carry chains was proposed by Parandeh-Afshar [PANBI11]. This set is used as basic elements for heuristically synthesizing compressor trees. An example of a $(1, 5; 3)$ GPC is shown in Figure 1a. The main idea is to utilize the FPGA carry chains (shaded boxes in Figure 1a) and to map additional full-adders to the LUTs. The FPGA mapping [PANBI11] for Xilinx FPGAs which provide six-input LUTs that can be configured as two five-input LUTs with shared inputs, namely the Virtex 5-7, Spartan 6, Kintex 7 and Artix 7 families, is shown in Figure 1b. In addition to the carry-chain, one XOR gate has to be realized in the LUT to build a two-input adder. To improve the efficiency, additional full adders are realized in the same LUT. Another example of a $(7; 3)$ GPC (which is also a 7:3 counter) is shown in Figure 1c and its FPGA mapping is shown in Figure 1d. Some other simple LUT based GPCs were also proposed in [BdDI+13]. Here, a $3 : 2$ counter was mapped to a single five-input LUT with two outputs and a $6 : 3$ counter as well as a $(1, 5; 3)$ GPC was mapped to three six-input LUTs.

2.3. Ternary Adders on FPGAs

A ternary adder computes the sum of three input bit vectors $s = x + y + z$. Instead of using two stages of ripple-carry adders, a carry-save adder is used in the first stage which compresses the three input vectors to two vectors which are then merged in a second stage using a common RCA,



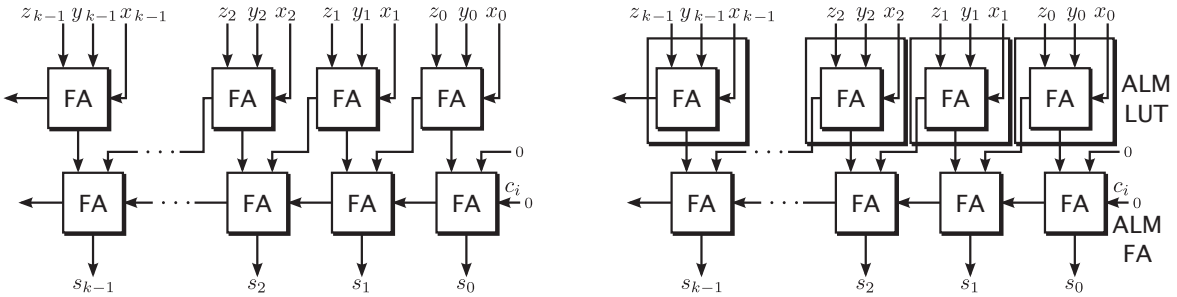
(a) Generic $(1, 5; 3)$ GPC **(b)** $(1, 5; 3)$ GPC mapped to FPGA slice

(c) Generic $(7; 3)$ GPC **(d)** $(7; 3)$ GPC mapped to FPGA slice

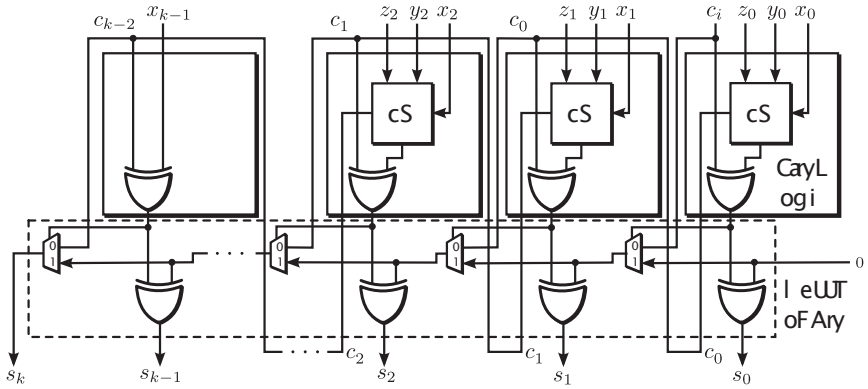
Figure 1: Compression elements from literature [PANBI11]

which is shown in Figure 2a. In principle, on modern FPGAs this uses the same number of full-adders but allows the mapping of the first stage of full-adders to the same LUT used for the RCA of the second stage. Hence, the ternary adder consumes the same resources as the two-input adder. Altera Arria I,II,V and Stratix II-V FPGAs directly support ternary adders. Their adaptive logic modules (ALMs) support a *shared arithmetic mode* in which a LUT output can be directly connected to the full-adder input of the next higher bit [BLSY09]. This allows a direct mapping of the full-adders of the first stage to the LUTs and the second stage of full-adders can be realized by the ALM full-adders as shown in Figure 2b.

Xilinx FPGAs do not provide something comparable to the shared arithmetic mode. Thus, the routing fabric of the FPGA has to be used to connect the carry-out of the first full-adders to the full-adder inputs of the next stage. In addition to that, parts of the full-adders in the second stage have to be realized in the LUT (an XOR gate) which requires an additional input. Thus, four inputs and two outputs of the LUT are needed. This mapping is possible on all modern Xilinx FPGAs and is shown in Figure 2c [SP06]. Note that an additional carry input (c_i) is possible in the first stage while not all inputs can be used in the last stage as the corresponding slice output for an additional full-adder is occupied with the carry-chain output.



(a) Generic architecture **(b)** Altera Stratix ALM mapping [BLSY09]



(c) Xilinx slice mapping [SP06]

Figure 2: Realization of ternary adders

2.4. Evaluation of Compressing Elements

Different performance metrics were proposed to evaluate compressing elements [BdDI+13]. Compressing elements can be characterized by their number of input bits b_i and number of output bits b_o. From these, the most important performance measure is the number of reduced bits $\delta = b_\mathrm{i} - b_\mathrm{o}$ in relation to the hardware resources, which is called the efficiency [BdDI+13]:

$$E = \frac{\delta}{k} \tag{2}$$

Here, k stands for the number of basic logic elements (BLEs) which is defined as the smallest common logic element in the FPGA. For the Xilinx FPGAs considered, it is represented as a quarter of a slice consisting of a six-input LUT which can be configured as two five-input LUTs with shared inputs, the carry-chain logic and two flip-flops. Hence, a common two-input adder consumes one BLE per bit.

Another important measure is the delay of the compressing element. It consists of LUT delays τ_L, routing delays τ_R and carry-chain delays τ_CC. A detailed timing analysis with the tool `trace` revealed that for a Virtex 6 the LUT delay is $\tau_\mathrm{L} \approx 0.3$ ns, a local routing takes about $\tau_\mathrm{R} \approx 0.3 \ldots 0.5$ ns and carry propagation of a carry chain with 4 bits takes 0.06 ns, leading to a single carry delay of $\tau_\mathrm{CC} \approx 0.015$ ns. Thus, the carry-propagation is much less than a LUT delay or a local routing delay which are in the same order of magnitude ($\tau_\mathrm{L} \approx \tau_\mathrm{R} \approx \tau$). This is close to a previous approximation of $\tau_\mathrm{L} + \tau_\mathrm{R} \approx 30\tau_\mathrm{CC}$ [BdDI+13].

An evaluation of compression elements is listed in Table 1. The upper half of the table was published earlier [BdDI+13] and is reproduced here for the sake of comparison. It contains previous compression elements including two-input adders and ternary adders. It can be observed that the best compressing element in terms of efficiency is the ternary adder which approaches $E = 2$ for a large number of BLEs (k) while having a delay between about 3τ for low word sizes ($k < 10$ bit) and about 6τ for large word sizes of $k = 64$ bit. Note that none of the compressing elements except the ternary adder has a better efficiency than $E = 1$.

Table 1: Comparison of different compression elements

GPC / Compressor	b_i	b_o	$\delta = b_i - b_o$	BLEs (k)	Efficiency ($E = \delta/k$)	delay
Naive LUT based GPCs from [BdDI+13][1]:						
(3;2) GPC	3	2	1	1	1	$\tau_L \approx \tau$
(6;3) GPC	6	3	3	3	1	$\tau_L \approx \tau$
(1,5;3) GPC	6	3	3	3	1	$\tau_L \approx \tau$
LUT and carry-chain GPCs from [PANBI11]:						
(6;3) GPC	6	3	3	4	0.75	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
(1,5;3) GPC	6	3	3	3	1	$\tau_L + 3\tau_{CC} \approx \tau$
(2,3;3) GPC	5	3	2	3	0.67	$\tau_L + 3\tau_{CC} \approx \tau$
(7;3) GPC	7	3	4	4	1	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
(1,6;4) GPC	7	4	3	4	0.75	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
(3,5;4) GPC	8	4	4	4	1	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
(4,4;4) GPC	8	4	4	4	1	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
(5,3;4) GPC	8	4	4	4	1	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
(6,2;4) GPC	8	4	4	4	1	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
Adder with k BLE:						
2-input adder	$2k+1$	$k+1$	k	k	1	$\tau_L + k\tau_{CC}$
3-input adder	$3k-1$	$k+1$	$2k-1$	k	$2 - \frac{2}{k}$	$2\tau_L + \tau_R + k\tau_{CC} \approx 3\tau + k\tau_{CC}$
Proposed improved mappings for GPCs, originally introduced in [PANBI11]:						
(6;3) GPC	6	3	3	3	1	$2\tau_L + \tau_R + 3\tau_{CC} \approx 3\tau$
(1,5;3) GPC	6	3	3	2	1.5	$\tau_L + 2\tau_{CC} \approx \tau$
(2,3;3) GPC	5	3	2	2	1	$\tau_L + 2\tau_{CC} \approx \tau$
(7;3) GPC	7	3	4	3	1.33	$2\tau_L + \tau_R + 3\tau_{CC} \approx 3\tau$
(5,3;4) GPC	8	4	4	3	1.33	$2\tau_L + \tau_R + 3\tau_{CC} \approx 3\tau$
(6,2;4) GPC	8	4	4	3	1.33	$2\tau_L + \tau_R + 3\tau_{CC} \approx 3\tau$
Proposed GPCs:						
(5,0,6;5) GPC	11	5	6	4	1.5	$\tau_L + 4\tau_{CC} \approx \tau$
(1,4,1,5;5) GPC	11	5	6	4	1.5	$\tau_L + 4\tau_{CC} \approx \tau$
(1,4,0,6;5) GPC	11	5	6	4	1.5	$\tau_L + 4\tau_{CC} \approx \tau$
(2,0,4,5;5) GPC	11	5	6	4	1.5	$2\tau_L + \tau_R + 4\tau_{CC} \approx 3\tau$
Proposed 4:2 compressor chain with k BLE:						
4:2 compressor	$4k$	$2k+1$	$2k$	k	$2 - \frac{2}{k}$	$\tau_L + k\tau_{CC}$

[1] (6;3) and (1,5;3) are also included in [PANBI11]

3. Proposed Improved Compression Elements

3.1. General Considerations

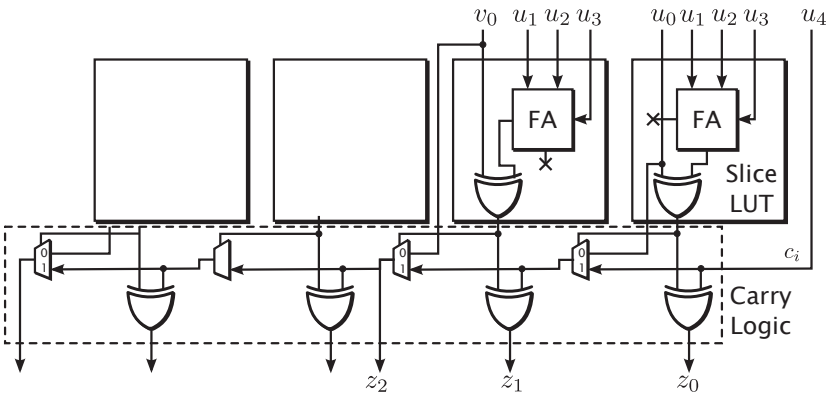
From the efficiency metric of Section 2.4 is becomes clear that 1) the number of input bits per BLE has to be maximized and 2) the carry-chain resources have to be used in an efficient manner. The first issue is not the case for most of the GPCs presented in [PANBI11] as can be seen from the examples in Figure 1b and Figure 1d in which parts of the LUTs are unused. The second issue is obviously not the case for the LUT based GPCs of [BdDI+13] as they don't use any carry-chain resource.
Additional restrictions are given by some limitations of the slice structure. The first limitation concerns the carry-in. There is only one slice input ({A/B/C/D}X [Xil12]) that can be either routed to the carry-in of the first stage or to the 0-input of the carry-chain multiplexer. Alternatively, this carry-chain multiplexer can be fed from the LUT. As one LUT output is used for the second full-adder stage, there are two choices for the least significant stage: either using the second LUT output for additional computations without using the carry-in (this has to be set to zero) or using the second LUT output as routing resource to the carry-chain multiplexer to use the carry-in without doing additional computations. The second limitation concerns the carry-out of the last stage. There are only two outputs per quarter slice which are able to route out the carry-out or the sum result (via {A/B/C/D}MUX output) and the O5 output of the LUT or the sum result (via {A/B/C/D}Q output) [Xil12]. Hence, the O5 LUT output of the last stage can not be routed to a previous stage.

3.2. Improved GPC mappings

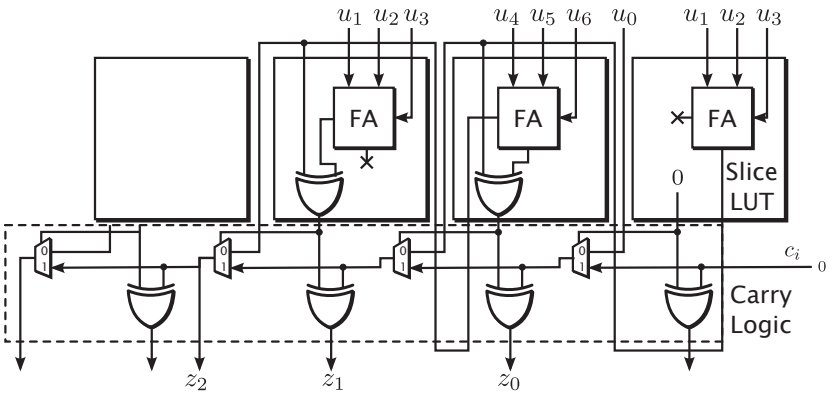
The main limitation of the GPCs proposed in [PANBI11] is that the considerations mentioned above are not fully used, i. e., BLEs are wasted for routing the carry-in or carry-out signals. Examples for improved mappings of the $(1, 5; 3)$ and $(7; 3)$ GPCs of Figure 1 are given in Figure 3. The GPCs for which improved mappings were found are listed in the lower half of Table 1. It can be seen that most of the GPCs can be optimized leading to higher efficiencies.

3.3. Proposed GPCs

The compressing elements considered so far use only a low amount of BLEs. If the number of BLEs is not a multiple of four, the unused parts of the carry chain can not be used for further arithmetic operations – only the LUTs and flip-flops (FFs) can be used. Hence, new GPCs are proposed that try to utilize the full slice as much as possible. The resulting GPC FPGA mappings are shown in Figure 4. Their properties are also listed in the lower part of Table 1.
All GPCs achieve an efficiency of 1.5 while the GPCs (5,0,6;5), (1,4,1,5;5) and (1,4,0,6;5) are designed such that no routing from one LUT to another occurs and, thus, the delay is minimal. If delay or speed is not that critical, the (2,0,4,5;5) GPC may be used in addition to better fit other input patterns.
Their irregular shape, i. e., the large differences of the input numbers for different weights, can be compensated by combining the compressing elements in a shifted manner. Combining two $(5, 0, 6; 5)$ GPCs where one GPC is left shifted by one bit can cover the input pattern $(5, 5, 6, 6)$, which is much more regular. However, such cases should be also found by automated synthesis tools [PANBI11, BdDI+13].



(a) (1,5;3) GPC



(b) (7;3) GPC

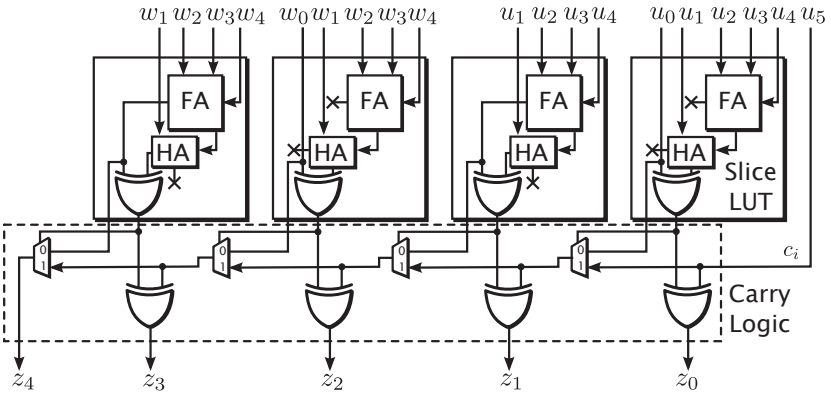
Figure 3: Improved GPC mappings

3.4. Proposed 4:2 Compressor

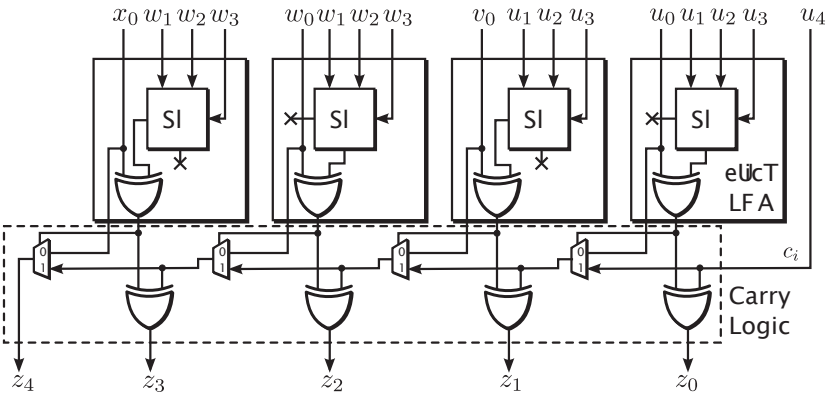
As it was shown before, the ternary adder has the best efficiency for a large number of BLEs (k) or large word sizes but has a poor delay. But its basic configuration can be used as a 4:2 compressor by simply eliminating the carry propagation to the second full-adder stage, leading to one additional input and one additional output per BLE, resulting in the structure shown in Figure 5. Thus, the combinatorial delay is reduced to a single LUT delay while the efficiency remains the same as for the ternary adder. For a full slice configuration with $k = 4$, the efficiency is identical to the low-delay GPCs from the last section (E=1.5) but is further increasing for larger k. Note that the last stage has to be reduced to two inputs due to the output restrictions.

4. Results

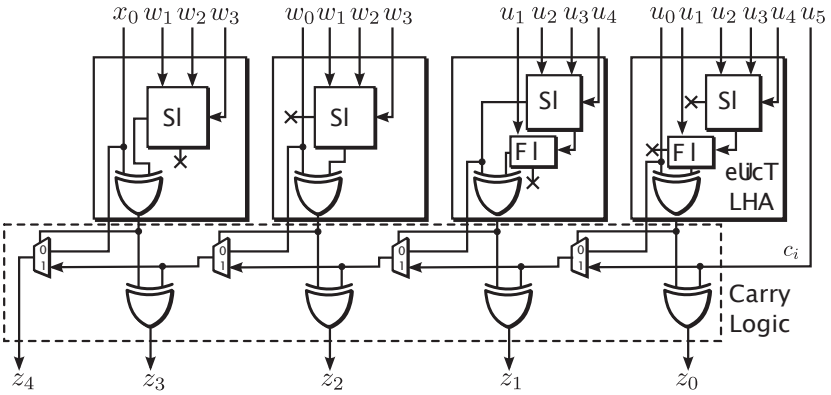
To evaluate the performance of the proposed compressing elements, we designed and synthesized pipelined compression trees with eight 10-bit inputs using different techniques. Even for this relatively low word size, the ternary adder and the 4:2 compressor lead to the best efficiency of $E_k = 1.8$ using $k = 10$ BLEs. The ternary adder tree requires two stages with four ternary adders in total while the compressor tree with 4:2 compressors requires three stages with three 4:2 compressors in total plus one common two-input adder to merge the result.



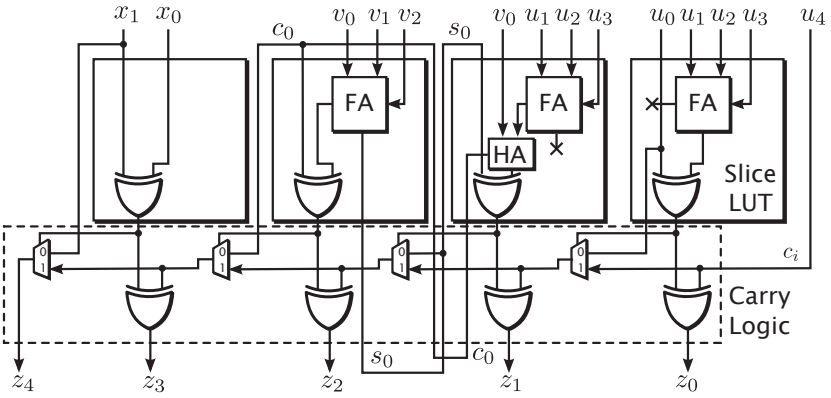
(a) $(5, 0, 6; 5)$ GPC



(b) $(1, 4, 1, 5; 5)$ GPC

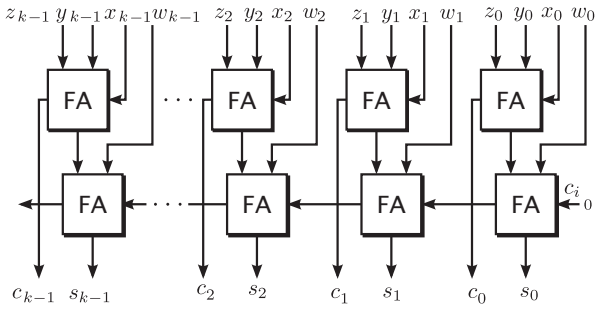


(c) $(1, 4, 0, 6; 5)$ GPC

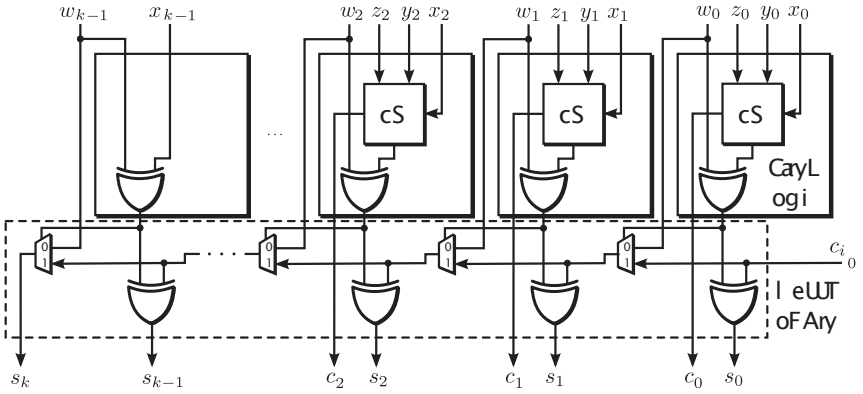


(d) $(2, 0, 4, 5; 5)$ GPC

Figure 4: Proposed GPC slice mappings



(a) General Structure



(b) Slice Mapping

Figure 5: Proposed 4:2 compressor

We compared the results with the bit heap compression tree optimization of the FloPoCo framework [BdDI+13] which uses LUT-based compressing elements for the Xilinx target. It offers an automated pipelining mechanism where pipeline registers are placed according to a timing model of the FPGA and a user-constrained target frequency f_t. The FloPoCo target frequency has be set to f_t = 400 MHz and f_t = 700 MHz. The synthesis was performed for a Xilinx Virtex 6 FPGA (XC6VLX75T-FF784) with speed grade 2. The results are listed in Table 2. The timing results were obtained by registering inputs and outputs to avoid that the critical path is dominated by the routing from the FPGA pin to the input of the circuit. The resource usage results do not include these registers.

As expected, the least slice resources are obtained by using ternary adders or 4:2 compressors. In terms of speed, the 4:2 compressor tree offers a 33% higher speed compared to the ternary adder while using the same amount of slices. The speed can be further increased by pipelining a LUT-based compressor design as shown by the FloPoCo design with f_t =700 MHz, resulting in a circuit with more than 900 MHz. However, it will be hard to keep this speed when the compressor is embedded within a more complex application. Furthermore, this speed has to be paid for by a huge amount of extra resources which is a factor of seven compared to the 4:2 compressor design.

5. Conclusion and Outlook

It was shown in this work that better compressing elements can be found by evaluating the low-level structure of the FPGA. Novel compressing elements for modern Xilinx devices were proposed including different GPCs and a 4:2 compressor based on a ternary adder. All of them provide a

Table 2: Synthesis results for an 8-input 10 bit pipelined adder tree using different methods

Method	Slices	FF	LUT	f_{max} [MHz]	T_{min} [ns]
FloPoCo [BdDI+13] (f_t = 400 MHz)	35	21	70	435.73	2.30
FloPoCo [BdDI+13] (f_t = 700 MHz)	91	145	145	929.37	1.08
ternary adder tree	13	49	49	491.88	2.03
proposed 4:2 compressor	13	65	47	657.46	1.52

better efficiency, a lower delay or both compared to previous compressing elements. They can be pipelined without overhead using the otherwise unused flip-flops in the device. A design example of a pipelined compressor tree showed the effectiveness of the 4:2 compressor.
Further work has to be done in the automatic synthesis of pipelined compressor trees. Previous work only focused on non-pipelined compressor trees although this quite limits the speed of the design. However, the strategy for pipelining is different as each input bit has to be covered by at least a single flip-flop for each compression stage to get a balanced pipeline. Another issue is the selection of the best compressing element. While the used efficiency metric [BdDI+13] assumes that all compressing element inputs can be covered by inputs of the compressor tree, this may not always be the case. Here, one has to find the element leading to the best resulting efficiency. While this was partly solved for the non-pipelined case [PANBI11] it is still open for efficient pipelining.

References

[BdDI+13] Brunie, Nicolas, Florent de Dinechin, Matei Istoan, Guillaume Sergent, Kinga Illyes, and Bogdan Popa: *Arithmetic Core Generation Using Bit Heaps*. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, 2013.

[BLSY09] Baeckler, Gregg, Martin Langhammer, James Schleicher, and Richard Yuan: *Logic Cell Supporting Addition of Three Binary Words*. US Patent No 7565388, Altera Coop., 2009.

[dDP12] Dinechin, Florent de and Bogdan Pasca: *Designing Custom Arithmetic Data Paths with FloPoCo*. IEEE Design & Test of Computers, 28(4):18–27, 2012.

[EL04] Ercegovac, Miloš D and Tomás Lang: *Digital Arithmetic*. Elsevier, 2004.

[flo13] *FloPoCo Project Website*, 2013. `http://flopoco.gforge.inria.fr`.

[GAKC09] Gao, Shuli, Dhamin Al-Khalili, and Noureddine Chabini: *Implementation of Large Size Multipliers Using Ternary Adders and Higher Order Compressors*. In *International Conference on Microelectronics (ICM)*, pages 118–121. IEEE, 2009.

[GAKCL12] Gao, S, D Al-Khalili, N Chabini, and P Langlois: *Asymmetric Large Size Multipliers with Optimised FPGA Resource Utilisation*. Computers & Digital Techniques, IET, 6(6):372–383, 2012.

[HVZ13] Hormigo, J, J Villalba, and E L Zapata: *Multioperand Redundant Adders on FPGAs*. IEEE Transactions of Computers, 62(10):2013–2025, 2013.

[KHW+13] Kumm, Martin, Martin Hardieck, Jens Willkomm, Peter Zipf, and Uwe Meyer-Baese: *Multiple Constant Multiplication with Ternary Adders*. In *IEEE International Conference on Field Programmable Logic and Application (FPL)*, pages 1–8, 2013.

[KZ11] Kumm, Martin and Peter Zipf: *High Speed Low Complexity FPGA-Based FIR Filters Using Pipelined Adder Graphs*. In *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 1–4, 2011.

[OQH+09] Ortiz, M, F Quiles, J Hormigo, F J Jaime, J Villalba, and E L Zapata: *Efficient Implementation of Carry-Save Adders in FPGAs*. In *IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, pages 207–210, 2009.

[PA12] Parandeh-Afshar, Hadi: *Closing the Gap Between FPGA and ASIC: Balancing Flexibility and Efficiency*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2012.

[PAI11] Parandeh-Afshar, Hadi and Paolo Ienne: *Measuring and Reducing the Performance Gap between Embedded and Soft Multipliers on FPGAs*. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 225–231. IEEE, 2011.

[PANBI11] Parandeh-Afshar, H., A. Neogy, P. Brisk, and P. Ienne: *Compressor Tree Synthesis on Commercial High-Performance FPGAs*. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 4(4):1–19, 2011.

[SP06] Simkins, James M and Brian D Philofsky: *Structures and Methods for Implementing Ternary Adders/Subtractors in Programmable Logic Devices*. US Patent No 7274211, Xilinx Inc., March 2006.

[Xil12] Xilinx, Inc.: *Virtex-6 FPGA Configurable Logic Block User Guide (UG364 v1.2)*, February 2012.

Effizienter Design Rule Check von 3D Systemaufbauten mit einer hierarchischen XML-basierten Modellierungssprache

Robert Fischbach, Michael Dittrich, Andy Heinig
Fraunhofer-Institut für Integrierte Schaltungen IIS
Institutsteil Entwurfsautomatisierung EAS
Dresden
vorname.nachname@eas.iis.fraunhofer.de

Zusammenfassung

Die Herstellung einer elektronischen Baugruppe erfordert das Einhalten fertigungsbedingter Regeln. Diese leiten sich aus maschinentechnischen Faktoren ab. Beispielsweise benötigen die Werkzeuge ausreichend Platz zum Anfahren der vorgegebenen Positionen. Typischerweise erfolgt das Beachten solcher Regeln während des Entwurfs manuell auf Basis empirischen Expertenwissens. Steigende Komplexität verlangt dann die Verwendung von zeit- und kostenintensiven Prototypen. Aus diesem Grund kommen beim Entwurf elektronischer Schaltkreise seit vielen Jahren abstrahierte Regeln in maschinenlesbarer Form zum Einsatz. Auch beim Baugruppenentwurf ermöglicht die rechnergestützte Regelprüfung ein systematisches und effizientes Vorgehen bei komplexeren Aufbauten. Mit sogenannten Regelsätzen lassen sich die Entwürfe schon vor der Herstellung auf ihre Fertigbarkeit überprüfen. Für elektronische Baugruppen existieren solche Regelsätze und Vorgehen hauptsächlich im Bereich des Leiterplattenentwurfs. Für komplexere Baugruppen (z.B. Integration mehrerer Dies, 3D-Aufbauten) fehlen etablierte Lösungen. Bedarf besteht vor allem an geeigneten Überprüfungsmechanismen und Regelsätzen für z.B. Bonddrähte und andere Verbindungsstrukturen.

Der in dieser Arbeit vorgestellte neuartige Ansatz ermöglicht die Regelprüfung von 3D-Systemaufbauten. Diese werden in einem ersten Schritt mit Hilfe einer hierarchischen Modellierungssprache beschrieben. Unterschiedlich detaillierte Bonddrahtmodelle und deren Integration in ein komplexeres Gesamtsystem veranschaulichen die Flexibilität dieser Sprache. Liegt eine Systembeschreibung in der vorgeschlagenen Form vor, lassen sich darauf basierend dreidimensionale geometrische Berechnungen ausführen (z.B. Abstandberechnungen, Volumenbedarf). Das stellt die Grundlage für unsere 3D-Regelprüfung dar. Das Einbeziehen, z.B. elektrischer oder materialbezogener Eigenschaften verfeinert das Model und erlaubt umfassende Entwurfs- und Fertigungsregeln.

1. Einleitung

In der klassischen Schaltkreisentwicklung wurden Schaltkreise in einzelne Gehäuse verpackt. Das Gehäuse diente vor allem als mechanischer Schutz. Die deutlichen Grenzen der Schaltkreisintegration, sowie die Herausforderungen der heterogenen Integration unterschiedlicher Technologien auf einem Schaltkreis führten zur Entwicklung fortschrittlicher Systemintegrationsverfah-

ren. Dazu zählen die *system in packages* (SiP, [Bro04]), *embedded wafer level packages* (eWLB, [MOB+08]) oder die 3D-Integration. Besonders anspruchsvoll sind die zunehmende Packungs- und Verbindungsdichte, welche diese Aufbauformen auszeichnen. Die erhöhten Anforderungen an die Aufbau- und Verbindungstechnologien bzw. deren Fertigungsverfahren erfordern technologische Optimierungen, beispielsweise dünnere Bonddrähte, genauere Ausrichtung der Bauteile oder exaktere Verfahrwege der Bondtools. Zusätzlich kann eine bessere Modellierung der beteiligten Strukturen helfen, im Verlauf des Entwurfs der Systeme die anspruchsvollen Anforderungen zu erfüllen. Die geeignete Beschreibung der betroffenen Elemente ermöglicht automatische Regelprüfungen sowie simulative Verfahren. Das gestattet Aussagen über die Fertigbarkeit des geplanten Systemaufbaus während früher Entwurfsphasen. Weiterhin erleichtert dieser Ansatz Optimierungen und das Vergleichen verschiedener Varianten in kurzer Zeit, im Gegensatz zu teuren und langwierigen Untersuchungen mithilfe von Probeaufbauten.
Die Herausforderung bei der Beschreibung solcher Systeme und der Anwendung von Regelprüfungen liegt in dem dreidimensionalen Charakter der zu modellierenden Systemaufbauten. Somit müssen die Systembestandteile effizient als 3D-Komponenten beschrieben werden. In dieser Arbeit erfolgt die Beschreibung mithilfe von CSG-Primitiven (*constructive solid geometry*). Die CSG-Primitive stellen dabei geschlossene Hüllkörper dar, wie zum Beispiel eine Kugel oder ein Quader. Unter Verwendung der mathematischen Operationen Vereinigung, Schnitt und Differenz entstehen aus den CSG-Primitiven neue Körper mit einer ebenfalls geschlossenen Oberfläche.
Abbildung 1 illustriert den in dieser Arbeit vorgestellten Ablauf zur Regelprüfung von 3D-Systemaufbauten. Abschnitt 2 führt in die dazu notwendige Modellierungsmethode ein. Die Anwendung dieser Modellierungsmethode wird im Abschnitt 3 an einem Bonddraht aufgezeigt. Dabei wird der Bonddraht mit verschiedenen Genauigkeiten für verschieden komplexe Regelprüfungen modelliert. Im Abschnitt 4 wird die Beschreibung eines Gesamtsystems gezeigt. Abschnitt 5.1 zeigt die Möglichkeit auf, auch Fertigungswerkzeuge zu berücksichtigen und Abschnitt 5.2 beschreibt schließlich die eigentliche Regelprüfung.

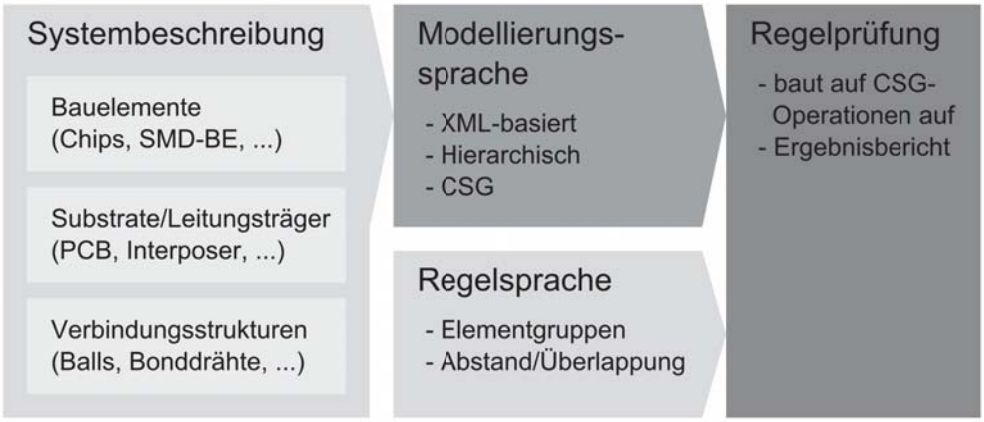


Abbildung 1: Der in der Arbeit vorgestellte Ablauf zur Regelprüfung von 3D-Systemaufbauten basiert auf einer hierarchischen Modellierungssprache. Die einzelnen Systemkomponenten werden als CSG beschrieben. Die Regelprüfung baut auf geometrischen Operationen auf und erlaubt die Erkennung von Elementwechselwirkungen, die durch eine eigene Regelsprache beschrieben sind.

2. Modulare Modellierungsmethodik

Um 3D- und SIP-integrierte Systeme automatisch auf ihre Fertigbarkeit zu überprüfen (simulativ oder analytisch) ist eine dreidimensionale geometrische Beschreibung des Systemaufbaus erforderlich.
Aufgrund der Komplexität des Gesamtsystems sollen Teilkomponenten des Systems nur im jeweils erforderlichen Abstraktionslevel modelliert werden. So reicht beispielsweise für eine simulative Kollisionskontrolle des Bondkopfes mit den bereits gefertigten Bonddrähten eine einfache Modellierung der Schaltkreise als Quader ohne eine detaillierte Darstellung des Schichtenaufbaus. Bonddrähte, Bondkopf und Verfahrwege der Maschine erfordern in diesem Fall eine detailliertere Modellierung. Da deren Form und Verlauf großen Einfluss auf die spätere Fertigbarkeit besitzen.
Während für zweidimensionale Schichtenaufbauten der Schaltkreise mit dem GDSII-Format (u.a. in [Cad13] beschrieben) eine etablierte Beschreibung zur Verfügung steht, ist dies für dreidimensionale Systemaufbauten bisher nicht der Fall. Häufig ist die Beschreibung der 3D-Modelle mit Hilfe von CAD-Formaten wie STEP ([STE]) oder IGES ([IGE]), welche aus dem Maschinenbau stammen. Eine Übersicht über gängige Formate findet sich in [WHK13].
Daher wurde die XML-basierte hierarchische Beschreibungssprache verwendet [WHK13] [LM12]. Der hierarchische und parametrisierbare Aufbau stellt die Austauschbarkeit der Teilkomponenten sicher und ermöglicht deren Beschreibung in verschiedenen Abstraktionslevel. Weiterhin erlaubt die hierarchische Beschreibung eine effiziente Anwendung geometrischer Transformationsoperationen. So kann man beispielsweise bei der Verschiebung eines Teilsystems auf die Annotation an jedem einzelnen Element verzichten.
Das Codebeispiel in Abbildung 2 instanziiert das Drahtbondmodell in Abbildung 3.

```
<element name="assembly">

  <!-- instanciate a bondwire -->
  <include name="bondwire">
    <fieldValue name="diameter" value="1.9"/>
    <fieldValue name="height" value="20"/>
    <fieldValue name="length" value="300"/>
  </include>

</element>
```

Abbildung 2: Instanziierung eines Bonddrahtmodells.

3. Bonddrahtmodellierung und -beschreibung

Der folgende Abschnitt veranschaulicht die oben beschriebene Modellierungsmethodik am Beispiel eines Bonddrahts. Die Geometrie eines solchen Bonddrahtes ist in Abbildung 4 dargestellt. Als Grundlage für Simulationen (z.B. elektrischer Eigenschaften) dienen oft vereinfachte Modelle, welche den Drahtbondverlauf durch Liniensegmente nachbilden. Die Reduktion auf ein 2D-Model ist zulässig, da Bonddrähte typischerweise in einer Ebene verlaufen. In der weiteren Modellierung

```
<element name ="bondwire">
  <interface>
    <field name="diameter" type="double"/>
    <field name="height" type="double"/>
    <field name="length" type="double"/>
  </interface>

  <!-- CSG description of the bond wire -->

</element>
```

Abbildung 3: Rumpf mit Interface eines Bonddrahtmodells.

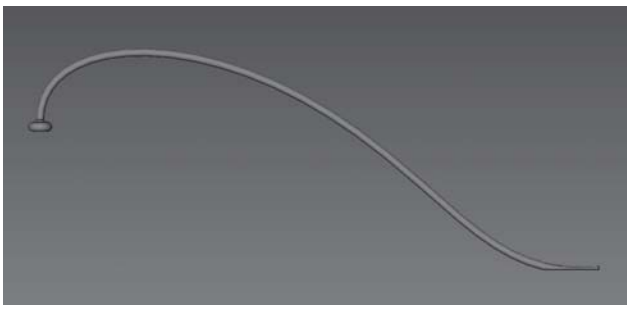


Abbildung 4: CAD-Model eines Au-Thermosonic-Ball-Wedge-Drahtbonds.

entsteht daraus ein 3D-Objekt, welches zusammen mit weiteren Komponenten ein dreidimensionales Gesamtsystem bildet.
Zwei solcher Modelle sind im JEDEC Standard EIA/JESD59 beschrieben [JED97]. Das vereinfachte JEDEC-Model (*simplified bond wire model*, Abbildung 5) reduziert den Verlauf auf drei Liniensegmente. Neben dem Start- und Endpunkt stellt dieses Model nur noch die Höhe des Bonddrahts als Formparameter zur Verfügung.
Eine genauere Beschreibung ermöglicht das bevorzugte JEDEC-Model (*preferred bond wire model*, Abbildung 6), welches ein weiteres Liniensegment zum genaueren Nachbilden der Bonddrahtgeometrie zur Verfügung stellt. Als zusätzliche Formparameter dienen die Aus- und Eintrittswinkel (Alpha und Beta) des Bonddrahts.

Tabelle 1: Gegenüberstellung der untersuchten Bonddrahtmodelle

Model	Mittlere Abweichung	Maximale Abweichung	Mittlere quadr. Abweichung
JEDEC Simplified	65,06 μm	173,77 μm	5937,84 μm^2
JEDEC Preferred	26,38 μm	67,59 μm	1114,53 μm^2
EAS 4 Segmente	22,29 μm	58,15 μm	748,05 μm^2
EAS 10 Segmente	5,21 μm	16,24 μm	43,51 μm^2

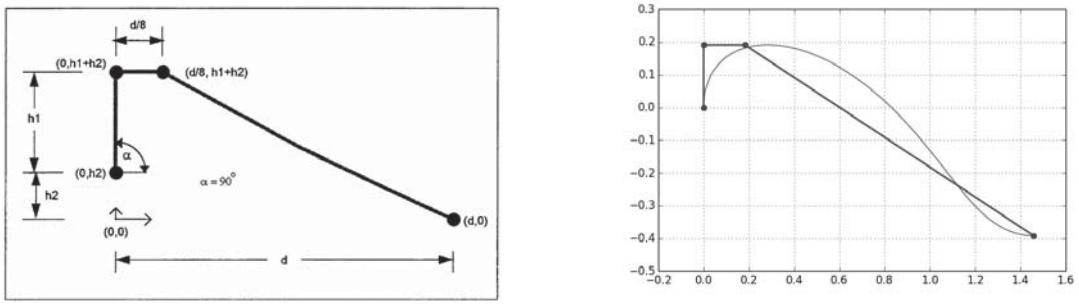


Abbildung 5: Vereinfachtes JEDEC-Bonddrahtmodel (links) und Adaption an einen vorgegebenen Bonddrahtverlauf (rechts). Die grüne Kurve ist der Verlauf der vorgegebenen Bonddrahtgeometrie. Die rote Kurve ist das Model, bei welchem die Verfügbaren Parameter für diesen Verlauf Optimiert wurden (Einheiten in mm).

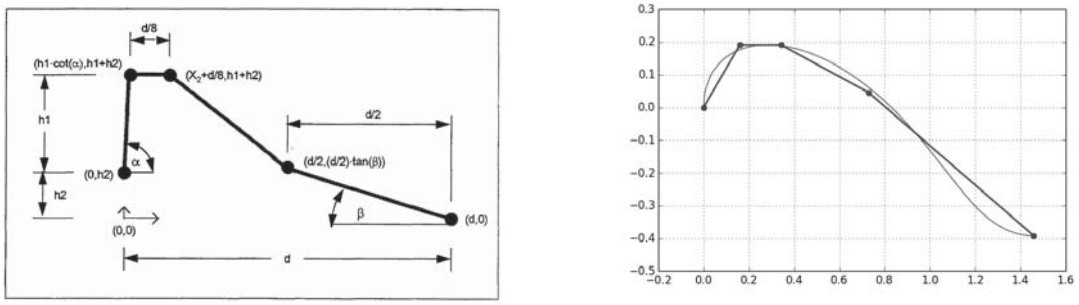


Abbildung 6: Bevorzugtes JEDEC-Bonddrahtmodel (links) und Adaption an einen vorgegeben Bonddrahtverlauf (rechts).

Beide Modelle wurden mit der in Abschnitt 2 beschrieben Methode abgebildet. Durch die Beschränkung auf drei bzw. vier Liniensegmente entstehen jedoch größere Abweichungen (siehe Tabelle 1 sowie Abbildung 5 und 6, jeweils rechts). Das motivierte die Entwicklung eines eigenen Bonddrahtmodels (EAS-Model).
Das EAS-Model bildet den Bonddrahtverlauf mithilfe von zwei kubischen Bézierkurven der Form $C(t) = (1-t)^3 \cdot P_0 + 3 \cdot t \cdot (1-t)^2 \cdot P_1 + 3 \cdot t^2 \cdot (1-t) \cdot P_2 + t^3 \cdot P_3$ ab. Die Fixpunkte P_0 und P_3 der ersten Kurve sind der Start- und der Höchstpunkt der Bondschleife (*bond loop*), welcher durch die Loophöhe und den Abstand vom Startpunkt (x-Richtung) definiert ist. P_2 der ersten Kurve liegt auf der Höhe von P_3 direkt über P_0 (P_2 = $(P_0[x], P_3[y])$). P_0 der zweiten Kurve entspricht P_3 der ersten Kurve. P_3 der zweiten Kurve ist schließlich der Endpunkt des Bonddrahts. Die folgenden drei Formparameter legen die verbleibenden freien Punkte P_1 bzw. P_2 der beiden Kurven fest.
Der Anstiegsfaktor bestimmt, wie steil der Bonddraht nach dem Setzen des Balls nach oben verläuft, und ergibt P_1 der ersten Bézierkurve. Der Looplängenfaktor bildet die Breite der Bondschleife ab und spiegelt sich in P_1 der zweiten Kurve wieder. Der Auslauffaktor gibt an, wie flach der Bonddraht am Ende (Wedge) ausläuft, und bestimmt P_2 der zweiten Kurve.
Die bestmögliche Übereinstimmung des EAS-Models mit einem vorgegebenen Bonddrahtverlauf ergibt sich bei direkter Verwendung der Bézierkurven. Allerdings ist dieser Ansatz für die angestrebte Geometriemodellierung ungeeignet, da letztendlich jede Geometrie in eine diskrete Anzahl Grundprimitive zerlegt wird (z.B. Liniensegmente). Deshalb stellen die Bézierkurven einen Zwischenschritt dar, aus dem heraus beliebige „Stützstellen“ berechnet werden können (5 Punkte in

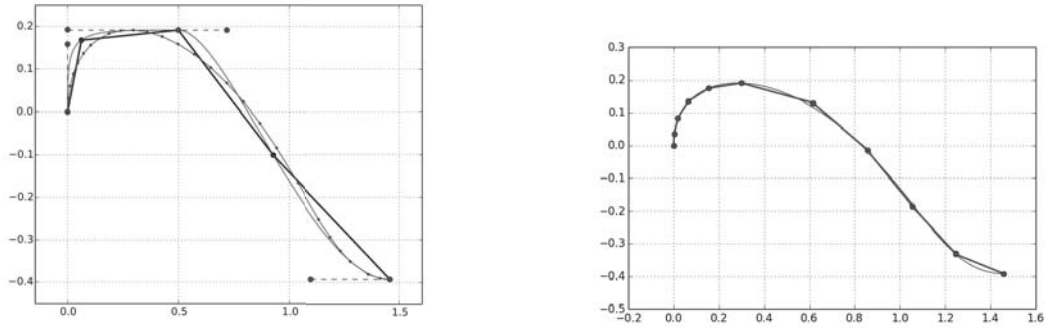


Abbildung 7: EAS-Model (4 Segmente) (links) und Adaption an einen vorgegebenen Bonddrahtverlauf (10 Segmente) (rechts).

Abbildung 7). Mit steigender Anzahl der verwendeten Punkte wächst auch die Genauigkeit des EAS-Modells (siehe Tabelle 1).
Um die Modelle (JEDEC und EAS) an einen gegebenen Bonddrahtverlauf anzupassen findet eine multikriterielle Optimierung der entsprechenden Formparameter statt. Als Zielfunktion dient die mittlere quadratische Abweichung des Ist- zum Sollverlauf des Bonddrahts. Das Ergebnis dieser Optimierungen ist jeweils im rechten Teil der Abbildungen 5 bis 7 zu sehen.
Für die Verwendung in einem komplexen Gesamtsystem können die Bonddrahtmodelle mithilfe der in Abschnitt 2 vorgestellten modularen Methode mit den verschiedenen Genauigkeitsstufen modelliert und instanziiert werden. Für die nachfolgenden Betrachtungen kommt das EAS-Model mit 10 Segmenten zum Einsatz.

4. Beschreibung des Gesamtsystems

Aus den einzelnen Bonddraht-Modellen und den Modellen für die Schaltkreise und die Leiterplatte lässt sich das Gesamtsystem aufbauen. Dabei ist der in Abschnitt 2 beschriebene modulare Ansatz von Vorteil, da die Teilsysteme in eigenen Teilkomponenten beschrieben werden können und die Übersichtlichkeit gewahrt bleibt. Abbildung 8 zeigt ein vereinfachtes Beispiel für die XML-basierte Beschreibung eines Gesamtsystems.
Zusätzlich erlaubt der modulare Ansatz, das Gesamtsystem mit verschiedenen Stufen der Genauigkeit aufzubauen. Das ist beispielsweise im Bereich der Bonddrähte von Vorteil, da diese damit mit den verschiedenen Modellen aus Abschnitt 3 beschrieben werden können.
Abbildung 9 veranschaulicht dieses Konzept: Während einer frühen Entwurfsphase erlaubt beispielsweise ein vereinfachtes Bonddrahtmodel schnelle Berechnungen für erste Entwurfsregelprüfungen (links). Mit voranschreitendem Entwicklungsstand kommen dann genauere Modelle (rechts) zum Einsatz und ermöglichen detaillierte Aussagen über die physikalischen Eigenschaften des Gesamtsystems.

```
<element name="assembly">
  <!-- instanciate substrate -->
  <include name="substrate">
    <fieldValue name="dimX" value="20000"/>
    <fieldValue name="dimX" value="10000"/>
    <fieldValue name="dimZ" value="2000"/>
  </include>
  <!-- instanciate chips -->
  <translate x="5000" y="3000">
    <include name="controller"/>
    <translate x="10000">
      <include name="memory"/>
    </tranlsate>
  </translate>
  <!-- instanciate bondwires -->
  <include name="bondwire">
    <fieldValue name="diameter" value="1.9"/>
    <fieldValue name="height" value="20"/>
    <fieldValue name="length" value="300"/>
  </include>
</element>
```

Abbildung 8: Beschreibung des Gesamtsystems mit der in Abschnitt 2 vorgestellten Beschreibungssprache. Dieses vereinfachte Beispiel verzichtet unter anderem auf die vollständige Ausführung aller Transformationselemente, welche beispielsweise die Bonddrahtinstanzen umschließen würden. Durch den hierarchischen Aufbau lassen sich funktionell zusammengehörige Komponenten einfach gruppieren.

5. Überprüfung des Gesamtsystems

5.1. Simulation des Bondkopfes zur Fertigung des Gesamtsystems

Die Überprüfung der Fertigbarkeit des in der XML-Beschreibungssprache modellierten Systems kann auf verschiedene Arten erfolgen. Für eine akkurate Untersuchung bietet sich eine Simulation des Bondkopfs an. Bei dieser wird der Bondkopf als weiteres Element in der Beschreibungssprache modelliert. Das so erstellte Model erlaubt das Nachbilden des Bondkopfverfahrweges. Zu Kollisionserkennung kommen die CSG-Operationen der Beschreibungssprache zum Einsatz. Eine hohe Genauigkeit erreicht man durch eine ausreichend feine Auflösung des Verfahrweges des Bondkopfes.
Jedoch sorgen die dazu notwendigen vielen Rechenschritte in Kombination mit den CSG-Operationen zu einem hohen Rechenaufwand (im Vergleich zu Kollisionstests eines unveränderlichen Systemaufbaus). Nur wenn das aufzubauende System alle Platzressourcen ausnutzen muss, zahlt sich der hohe Aufwand aus.

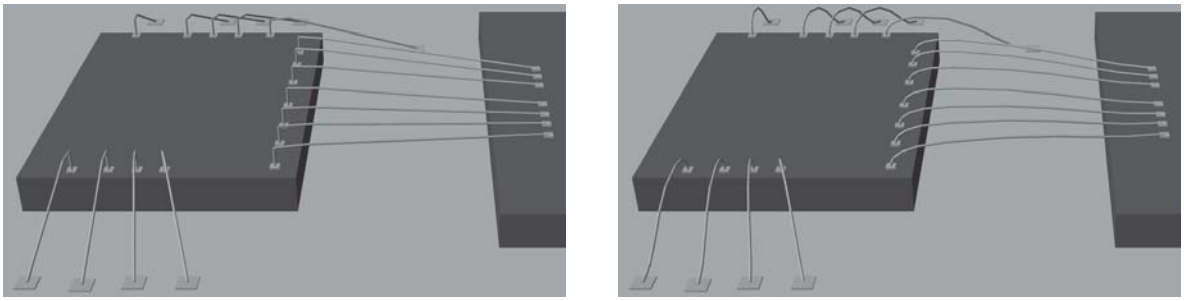


Abbildung 9: Verwendung unterschiedlicher Bonddrahtmodelle innerhalb eines komplexeren Systems. Während das links verwendete JEDEC-Model nur grobe Aussagen zulässt, ermöglicht das rechts dargestellte EAS-Modell (10 Segmente) auch genauere Simulationen.

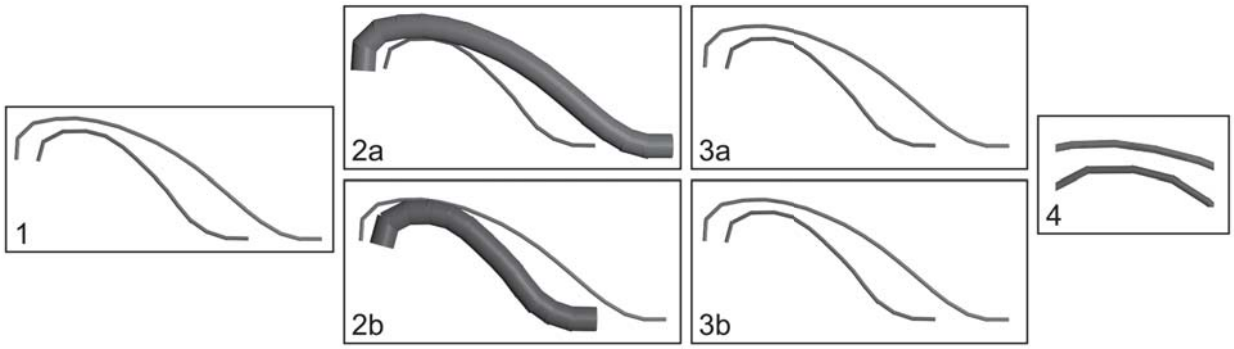


Abbildung 10: Ablauf einer Abstandsprüfung: Die selektierten Objekte (1) werden jeweils um den zu testenden Abstand vergrößert (2). Die Schnittmenge mit den anderen Objekten stellen dann Verletzungen der Regel dar (3). Aus der Gesamtheit dieser Geometrien ergibt sich das Regelergebnis (4), welches im Idealfall wieder als CSG-Objekt angegeben und weiterverwendet werden kann.

5.2. Regelprüfung und Regelsprache

Prädestiniert hingegen sind diese exakten Simulationen vor allem für das Ableiten vereinfachter Entwurfsregeln (*design rules*) für die Regelprüfung. Beispielsweise lassen sich unter Berücksichtigung der Bondprofile Mindestabstände zu benachbarten Komponenten bestimmen. Zu beachten ist eine ausreichend große Sicherheit, damit bei Erfüllung der Regeln immer ein fertigbares System entsteht. Allerdings kann eine solche Regel zu Beispielen führen, in denen die Bonddrähte dichter hätten verlegt werden können, weil eine speziellere Konstellation aufgetreten ist. Solche Anordnungen entstehen es vor allem an den Ecken von Schaltkreisen oder bei gestapelten Schaltkreisen. Der Vorteil von Entwurfsregeln ist deren analytischer Charakter, mit dem man Aussagen über den Grad der Verletzung ableiten kann. Dies ist bei simulativen Verfahren nicht immer gegeben. Für weitere Schritte zur Beseitigung von Fehlern oder zur Optimierung sind solche Aussagen aber sehr nützlich, um die nötige Anpassung des Systems direkt und ohne aufwendige iterative Schleifen zu ermöglichen.
Die klassische Entwurfsregelprüfung im Chipentwurf überprüft Layoutdaten, welche durch zweidimensionale Polygone auf mehreren Lagen (*layer*) beschrieben sind. Geometrische Grundoperationen arbeiten dann auf Kombinationen einzelner Lagen. Dieser Ansatz lässt sich nicht direkt auf dreidimensionale Elemente anwenden, da eine Aufteilung in eine (geringe) Anzahl Lagen ty-

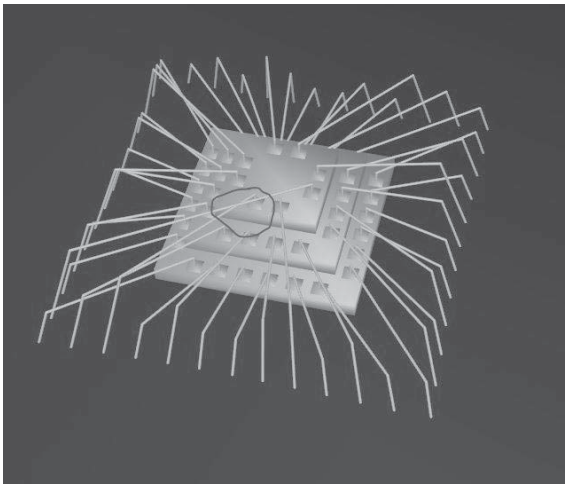


Abbildung 11: Beispiel für die Bonddrahtprüfung.

pischerweise nicht möglich ist. Ein alternativer Ansatz ist die Verwendung von Instanz- und/oder Hierarchieinformationen (z.B. Regeln bezüglich Leiterzugelementen).
Unsere Regelsprache stellt grundlegende Operationen zur Verfügung, aus denen sich komplexere Regeln aufbauen lassen. Ein Beispiel für eine solche Grundoperation ist das Einhalten eines äußeren Abstands, welcher unter Angabe der gewünschten Grenze die verletzten Bereiche für die selektierten 3D-Objekte ermittelt. Die Selektion ist dabei besonders interessant, da diese den oben erwähnten klassischen Ebenenansatz ablöst. Eine Möglichkeit ist das Anbieten von Auswahlfiltern, mit denen sich die zu prüfenden Objekte eingrenzen lassen. Auf diese Weise könnte man z.B. Objekte auswählen, die einem bestimmten Namensschema folgen oder zu einem vorgeschriebenen Instanztyp gehören. Auch Informationen über das Material oder anliegende elektrische Signale können in die Filter integriert werden.
Abbildung 10 zeigt das Ergebnis einer Abstandsoperation. Die Weiterverwendung der Regelergebnisse vereinfacht das Beschreiben komplexerer Regeln. Der Syntax für eine solche Regel sieht wie folgt aus: `regelname { EXTERNERABSTAND wert filter }`. Ein `filter` für Elemente kann z.B. `KOMPONENTENNAME(Wirebond)` oder `ELTERNKOMPONENTE(Substrat1)` sein und lässt sich mit booleschen Operatoren verknüpfen.
Die Grundlage für die 3D-Regelprüfung ist das Vorhandensein effizienter Grundalgorithmen zur Bearbeitung der CSG-Objekte. Allerdings erfordert bereits der Test, ob sich zwei beliebige CSG-Objekte überlappen ein aufwendiges Zerlegen in konvexe Grundelemente. Der Detaillierungsgrad der betrachteten Geometrien hängt außerdem stark von den getroffenen Vereinfachungen ab.
Wie im Abschnitt 5.1 beschrieben, können die Entwurfsregeln für den DRC aus der exakten Simulation abgeleitet werden und sind so ausgelegt, dass bei ihrer Einhaltung das System immer fertigbar ist. Der Bereich für solche Entwurfsregeln für Aufbau- und Verbindungstechniken verläuft von einfachen bis hin zu komplexen Regeln. Eine einfache Regel kann zum Beispiel eine Regel r_1 sein, bei welcher der minimale Abstand zwischen zwei Bonddrähten d_1 und d_2 feststeht (r_1: Abstand(d_1,d_2) $> x$). Abbildung 11 veranschaulicht einen solchen einfachen Abstandstest zwischen mehreren Bonddrähten.
Darauf aufbauend lassen sich komplexere Regeln definieren, in denen beispielsweise an unterschiedlichen Ecken des Schaltkreises unterschiedliche Regeln gelten (wenn nicht Ecke r_1: Abstand(d_1,d_2) $> x_1$ ansonsten r_1: Abstand(d_1,d_2) $> x_2$). Daraus können immer komplexeren

Gebilden entstehen. Da die Regelprüfung für 3D-Geometrien rechenaufwendig ist, kann eine erste Vorabprüfung mit dem vereinfachten Drahtmodel und mit wenigen Geometrieinformationen stattfinden. Wenn diese Vorabprüfung fehlschlägt, können die beteiligten Modelle gegen detailliertere ausgetauscht werden, um nähere Informationen zur Fehlerursache zu erhalten.

6. Zusammenfassung

In der vorliegenden Arbeit wurde eine neue Methode zur Regelprüfung von 3D-Systemaufbauten vorgestellt. Die dazu verwendete hierarchische XML-basierte Modellierungssprache erlaubt auch die Beschreibung komplexerer Geometrien, wie etwa der Verlauf von Bonddrähten. Jedoch liefern die bisher typischen Modellierungsansätze für Bonddrahtgeometrien für die geplanten Simulationen nur eine ungenügende Genauigkeit. Das neu entwickelte EAS-Model gibt solche Verläufe wesentlich genauer wieder.
In Kombination mit weiteren Komponentenbeschreibungen lässt sich so ein Gesamtsystem aufbauen und mit unterschiedlichen Detaillierungsgraden untersuchen. Der Kompromiss zwischen Genauigkeit und Performance erlaubt den flexiblen Einsatz in bestehenden Arbeitsabläufen.

Danksagung

Diese Arbeit ist im Rahmen des Forschungsprojekts ESiMED entstanden, das am 01.09.2012 gestartet ist und vom BMBF unter dem Kennzeichen 16M3201B gefördert wird.

Literatur

[Bro04] Brown, K.M.: *System in package "the rebirth of SIP"*. In: *Custom Integrated Circuits Conference, 2004. Proceedings of the IEEE 2004*, Seiten 681–686, 2004.

[Cad13] Cadence: *Design Data Translator's Reference*, März 2013.

[IGE] *ASME Y14.26M: Digital Representation for Communication of Product Definition Data (IGES-Format)*.

[JED97] *Bond Wire Modeling Standard*, June 1997.

[LM12] Lienig, J. und Dietrich M.: *Entwurf integrierte 3D-Systeme der Elektronik*, Kapitel XML-basierte Sparache für die hierarchische und parametrisierbare Beschreibung von 3D-Systemen. Springer, Heidelberg, 2012.

[MOB+08] Meyer, T., G. Ofner, S. Bradl, M. Brunnbauer und R. Hagen: *Embedded Wafer Level Ball Grid Array (eWLB)*. In: *Electronics Packaging Technology Conference, 2008. EPTC 2008. 10th*, Seiten 994–998, 2008.

[STE] *DIN EN 10303: Produktdatendarstellung und -austausch (STEP-Format)*.

[WHK13] Wolf, S., A. Heinig und U. Knöchel: *XML-Based Hierarchical Description of 3D Systems and SIP*. Design Test, IEEE, 30(3):59–69, 2013, ISSN 2168-2356.

LoCEG: Local Preprocessing in SAT-Solving through Counter-Example Generation

Sebastian Burg, Patrick Heckeler, Stefan Huster, Hanno Eichelberger,
Jörg Behrend, Jürgen Ruf, Thomas Kropf, Oliver Bringmann
University of Tuebingen
{lastname}@informatik.uni-tuebingen.de

Abstract

We introduce a new preprocessing methodology which aims at reducing the time needed for the calculation of a solution for a propositional logic problem in CNF via satisfiability solving(SAT-Solving). We achieve this through the generation of counter-examples which are used to add new clauses to the original instance without affecting the valuation of the given problem. Our new preprocessing technique learns several new clauses per iteration but does not require knowing the whole instance for these calculations. Our experimental results show that our approach reduces the execution time of the solving process, even though we add clauses.

1. Introduction

Instances created from industrial applications represent a set of subproblems within the field of SAT-Solving and propositional logic.
Because of the structured problems given in industrial instances it is possible to verify instances in an adequate time. However, for larger instances a verification run often takes a lot of time and memory in order to find a solution (if a solution is found at all).
In this paper we show a new way to reduce the depth of the search tree and thereby the time needed in the verification process in SAT-Solving. With this we want to enable the formal verification for larger industrial instances.
This paper is structured as follows: First, we introduce preliminaries for SAT-Solving in Section 2. In Section 3, we present our new methodology. The empirical results of our first experiments are shown in Section 4 and discussed in Section 5.

2. Preliminaries

In this section we introduce some basics and notations for SAT-Solving. SAT-Solving describes the process of finding a satisfying set of variables for a propositional logic instance. In most cases this instance is composed as *conjunctive normal form (CNF)* and therefore consists of conjunctions ($\wedge$) of disjunctions ($\vee$). The Instance I is defined as: $I := \bigwedge c \in C$ These disjunctions are called *clauses* c. The set of clauses in an instance I is named C. In clauses literals (l) are disjunctively connected. $c := \bigvee l \in L$ Literals are representations of variables which can also be negated. $l := \{\neg v_i, v_i | v_i \in V\}$ Variables can be $\top$ or $\bot$. We define the set

of variables in an instance V as:$V := \{v,\ v\ \in C\}$ Negations invert these representations.
We use a function $var(l) \rightarrow V$ with $var(l) := v$with $l = v \vee\ l = \neg v$ which returns the corresponding variable to a literal. When used on a clause $var(c_i) := \{var(l)|\ \forall\ l\ \in\ c_i\}$ returns a set of variables in a given clause c_i. These variables can be set to a Boolean value $set(v) \rightarrow \{\top,\ \bot\}$. With the settings for the instance $S := \{set(v)\|\ v \in V\}$, the instance can be valuated $eval(I, S) \rightarrow \{\top,\ \bot\}$.

$$eval(I, S) := \begin{cases} \top, & \text{if } \forall c \in I \exists l \in c, l \rightarrow \top \\ \bot, & \text{otherwise} \end{cases}$$

Furthermore, we use a function which maps a variable to an unique integer value, this is $num(v) \rightarrow \mathbb{N}, v \in V :\ num(v_i) \neq num(v_j),\ \forall\ v_i,\ v_j\ \in V, v_i \neq v_j$. In CNF, the Instance can only be satisfied (SAT) when there is at least one literal in each clause which evaluates to $\top$. When there is no possible setting which satisfies all clauses, then the instance is not solvable ($UNSAT$).

3. Methodology

In our approach we aim at the reduction of execution time. For this our preprocess learns clauses from sub instances of a given instance. This is achieved through the generation of counter-examples which are then used in the same way as learnt clauses from CDCL *(Conflict-Driven-Clause-Learning*, first published by Marques-Silva et.al. [MSS99], approaches.
The process consists of the following steps shown in Algorithm 1. An instance in the described CNF structure is given into this algorithm which iterates a given number of times trying to find multiple clauses in each iteration.

Algorithm 1 Local Preprocessing through Counter-Example Generation

Require: C ▷ Set of Clauses
Require: x ▷ Maximum number of iterations
Require: k ▷ Maximum number of clauses for in sub instance
Require: n ▷ Maximum number of different variables in sub instance
$occurrences[|V|] \leftarrow 0$ ▷ Initialize vector $occurrences$ for all variables to 0
for each $c \in C$ **do** ▷ traversing all clauses in the instance
  for each $v \in var(c)$ **do** ▷ for every variable in the clause
    $occurrences[num(v)] + +$ ▷ increment the occurrences of given variable
  end for
end for
$i \leftarrow 0$
for $i < x$ **do** ▷ For iterations(x) repeat generation loop
  $v_m \leftarrow maxKey(occurrences)$ ▷ Select the variable with the most occurrences
  $S \leftarrow \{\}$ ▷ Initialize S as new empty instance
  for each $c_i \in C$ **do** ▷ For all clauses in the original instance
    if $|S| > k$ **or** $|var(S)| > n$ **then** ▷ If enough clauses and variables are found
      break;
    end if
    if $\exists\ l \in c_i$ with $var(l) = v_m$ **then** ▷ When the clause contains the selected variable
      $S \leftarrow S \cup c_i$ ▷ Add clause to S when clause contains given variable
    end if
  end for
  $\bar{S} \leftarrow cnf(\neg S)$ ▷ Convert negated clauses to CNF
  $S^- \leftarrow findSolutions(\bar{S})$ ▷ Find satisfying Solution (Counter-Examples)
  $S^+ \leftarrow convert(S^-)$ ▷ Convert results
  $C \leftarrow C \cup S^+$ ▷ Add converted Counter-Examples to C
  $i \leftarrow i + 1$
end for

After calculating the occurences for all variables in the instance in lines 1 to 6, we start the iterations limited by x. In the loop in line $11 - 18$ of the algorithm 1 the clauses for the sub instance are selected. In line 15 we decide if the clause fits our heuristic.

A matching clause in our experiments is defined as follows, but the criteria is also subject to the chosen heuristic.
Given the candidate clause $c_c = (\bigvee_{i=0}^{n} l_i)$, with $c_c \in C$ and the corresponding variables $V_c := \{var(l_i) \mid l_i \in c_c\}$, then a matching clause is defined as: $cm := \{\bigvee_{j=0}^{k} l_j \mid \exists\, var(l_x) \in V_c\}$
All selected clauses are then in a sub instance $I_c := c_c \wedge \bigwedge_{i=0}^{k} cm_i$. The number of clauses is limited to a given maximum of clauses and different variables.
After negating the selected clauses in line 19 and converting them to CNF, we start a solving process in line 20 of the Algorithm 1. This process is aimed to find *all* possible solutions which results in the negated sub instance to be SAT.
These results are considered counter-examples for the original instance. They satisfy the negation of parts of the instance. Therefore these solutions would make the original instance UNSAT.
We now have to convert the found solutions into learnt clauses. This is done in line 21 of Algorithm 3. The conversion takes the counter-examples found and transforms them to new clauses as follows.
Given the set of solutions $S := \bigvee(\bigwedge l_s) \ : \ l_s \in I_c$ we build new clauses via negation resulting in CNF conform new clauses. $S' := \bigwedge(\bigvee \neg l_s) \ : \ l_s \in I_c$ These clauses are then added to the original instance. The new clauses do not affect the satisfiability of the original instance.

4. Experimental Results

We implemented the described method in our tool *LoCEG* using python and preprocessed the industrial benchmarks from the SAT Competition 2013. [1]
We use the glucose2.1 solver[2][AS12] in the edition that has won the 2012 SAT Challenge[3] in the Application SAT+UNSAT category. The glucose solver is based on minisat.
First we ran glucose on the original instances for a comparison with a walltime of 3 hours. Afterwards, we applied our tool LoCEG for a maximum of one hour. The created instance file was then solved again by glucose. We let CoProcessor2.0[4] [Man12] preprocess the instances as well and then let glucose do a solving process on these files as well. Furthermore we let CoProcessor2.0 optimize our created instances and again let glucose solve these resulting instances.
For glucose2.1 and CoProcessor2.0 we used the default settings. That means that glucose has its own simplifier enabled, which is based on SatElite. For CoProcessor2.0 a maximum execution time of 10 minutes.
All experiments were done on the High Performance Computing System of the University of Tuebingen[5] which is part of bwGRiD, employing 140 nodes each equipped with Quad-Core Intel Xeon Processor E5440 @2.83 GHz, 8 cores and 16GB of RAM.
In table 1 we show the comparison for the runtime results of the industrial benchmarks from

[1] http://satcompetition.org
[2] http://www.labri.fr/perso/lsimon/glucose/
[3] http://baldur.iti.kit.edu/SAT-Challenge-2012/
[4] http://tools.computational-logic.org/content/coprocessor.php
[5] http://www.zdv.uni-tuebingen.de/dienstleistungen/computing.html

the SAT Competition 2013. For quantitativ the values greater than 1 indicate that the first named method solves the given instances faster in the average. For the quantitativ number and percentage of instances which are improved by the different techniques. For the this we only used benchmarks where every solving tool returned a result within the 3 hours given to the solver, this was the case for 60 instances. Of these 60 instances 17 are SAT and 43 are UNSAT.

		LoCEG compared to original	LoCEG compared to CoProcessor2.0	CoProcessor2.0 compared to original	LoCEG & CoProcessor2.0 compared to original	LoCEG & CoProcessor2.0 compared to CoProcessor2.0
Qualitativ	SAT+UNSAT	1.109	1.443	1.033	0.984	1.074
	SAT	1.216	2.040	0.938	0.614	0.944
	UNSAT	1.067	1.207	1.073	1.130	1.125
Quantitativ	SAT+UNSAT	32(53%)	35(58%)	25(42%)	27(45%)	30(50%)
	SAT	8(47%)	10(59%)	6(35%)	4(24%)	5(29%)
	UNSAT	24(56%)	25(58%)	19(44%)	23(53%)	25(58%)

Table 1: Qualitativ: Runtime comparison. Shown are the averages over the dividends between two methods; Quantitativ: Number of Instances which are solved faster by a method in comparison to other methods.

The results for execution time in absolute numbers shown in Table 1 show that the instances LoCEG creates can in 32 of the 60 cases be solved faster than the original instances and in 35 cases can be solved faster than the optimized instances from CoProcessor2.0. The combination of LoCEG+CoProcessor2.0 improves the execution time only in 27 cases in comparison to the original instances. In 3 cases in combination of LoCEG+CoProcessor2.0 is faster than LoCEG alone. We can observe that we can improve the execution time in 53% of the cases whereas CoProcessor2.0 is only able to improve in 42%. The Combination of LoCEG and CoProcessor2.0 can improve the execution time in 45% of these cases.
The relation to the exact runtime LoCEG can run 10% faster than the original clauses. The result is even better in comparison to CoProcessor2.0, where LoCEG can improve in 44% of the cases. The best results are for SAT Instances where LoCEG can improve the runtime in comparison to CoProcessor2.0 by over 100%.

5. Conclusion

In this paper we presented a new method for learning new clauses in the preprocessing process. This method can reduce the execution time for the solving process.
We showed that our preprocess has a valid foundation and with good heuristics and an improved implementation will be a good addition to the existing preprocessing methods and tools.

References

[AS12] Audemard, Gilles and Laurent Simon: *Glucose 2.1 in the sat challenge 2012*. Proceedings of SAT Challenge 2012; Solver and, page 21, 2012.

[Man12] Manthey, N.: *Coprocessor 2.0 - a flexible cnf simplifier (tool presentation)*. pages 434–440, 2012.

[MSS99] Marques-Silva, Joao P. and Karem A. Sakallah: *Grasp: A search algorithm for propositional satisfiability*. IEEE Transactions on Computers, 48(5):506–521, 1999.

Equivalence Checking on System Level using Stepwise Induction*

Niels Thole*†
*Institute of Computer Science,
Universität Bremen, Germany
nthole@informatik.uni-bremen.de

Görschwin Fey*†
†Institute of Space Systems,
German Aerospace Center, Germany
Goerschwin.Fey@dlr.de

Abstract

We present an algorithm for equivalence checking between two C++ objects that uses stepwise induction. To prevent the effort of checking each state for reachability, we utilize a hypothesis that approximately describes the reachable states.

1. Motivation

Equivalence checking is used in the iterative development of a system to check if the old and the modified models behave equivalently. The special case of equivalence checking in our paper analyses two C++ classes. In this check the two models are equivalent iff the call of equivalent methods on both models always leads to equivalent outputs, when only equivalent methods were called before. We present an algorithm to prove or disprove the equivalence by using stepwise induction. For this proof we use a hypothesis that approximates equivalent states of the two models. Koelbl et al. [KJJP09] also use stepwise induction for equivalence checking. However, they do not further analyze a counterexample if the proof fails and therefore cannot prove that two models are not equivalent. This is possible with our algorithm. Other works on equivalence checking on C programs [GMYF09, MSF06, SBCJ05] handle specific aspects like the equivalence of similar code, array operations and pipelining. Finder et al. [FWF13] use bounded model checking to show the equivalence between two functions. The paper shows the equivalence between two functions and shortens the runtime by using checkpoints.
For the equivalence check C++ classes are interpreted as *Finite State Machines* (FSMs) where states are assignments of variables and the edges correspond to the execution of public methods. For two of those state machines exists a relation between the methods of both models specifying which methods should be equivalent to each other. The state machines are combined by using the synchronous product between them but only keeping the edges where the two inputs are in relation to each other. FSMs are less powerful than C++ and some features from C++ like pointers cannot be part of the checked classes. We also assume that all functions always terminate.

*This work has been supported by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative.

Algorithm 1 Equivalence Check

Input:
cModel1, cModel2: the C++ models;
equiv_methods: relation of equivalent methods;
hyp: hypothesis for an invariant as logical formula;
Output:
decision of equivalence and an invariant (if the models are equivalent)
Description:

```
1: // Look for forbidden states and remove them from the hypothesis
2: pre_hyp = hyp; post_hyp = true; forbidden_states = false;
3: forbidden_states = getStatesWithPredecessors(cModel1, cModel2, equiv_methods, pre_hyp, post_hyp);
4: if (initial_state(cModel1, cModel2) → forbidden_states)
5:    return models are not equivalent;
6: hyp = hyp ∧ ¬forbidden_states;
7: // Look for counterexamples that lead from a state that fulfills the hypothesis to a state that does not fulfill it
8: pre_hyp = post_hyp = hyp;
9: while (generateCounterexample(cModel1, cModel2, equiv_methods, pre_hyp, post_hyp) ≠ null) {
10:    (start, follow, method) = generateCounterExample(cModel1, cModel2, equiv_methods, pre_hyp, post_hyp);
11:    // Get start and its predecessors
12:    post_hyp = start;
13:    predecessors = getStatesWithPredecessors(cModel1, cModel2, equiv_methods, pre_hyp, post_hyp);
14:    // Check if the initial state is one of the predecessors
15:    if (initial_state(cModel1, cModel2) → predecessors){
16:       pre_hyp = follow;
17:       post_hyp = true;
18:       if (generateCounterexample(cModel1, cModel2, equiv_methods, pre_hyp, post_hyp) ≠ null) {
19:          return models are not equivalent;
20:       } else
21:          hyp = hyp ∨ follow;
22:    } else
23:       hyp = hyp ∧ ¬predecessors;
24:    pre_hyp = post_hyp = hyp;
25: }
26: return the models are equivalent and hyp is an invariant;
```

Our algorithm additionally requires a hypothesis for a likely invariant. At this step, the hypothesis is not proven to be an invariant and does not need to be. If the hypothesis is not a correct invariant, it is adjusted by the algorithm.
With the hypothesis the proof of equivalence can be done by stepwise induction. This proof utilizes the generated hypothesis and proves that, if a state fulfills the hypothesis and corresponding methods are called in both models, the methods return the same results and the resulting state fulfills the hypothesis as well. For this check, an underlying model checker is used as a black-box. If the proof is successful, the equivalence of the two models is proven. However, if a counterexample is returned, it is not proven that the models are not equivalent.
We present an algorithm to handle this problem and finally prove or disprove if the two models are equivalent. Additionally, if the models are equivalent, an invariant is returned that is an overapproximation of all reachable states.

2. Our Approach

The goal of this paper is an equivalence check between two software models in C++. This check exploits a hypothesis for an invariant of the two models. The hypothesis should be true in all reachable states of the combined state machine. The hypothesis is meant to be a likely invariant. Reachable states, that do not fulfill the hypothesis, or unreachable and fulfilling states can complicate the equivalence check. Both problems are solved later on.

Algorithm 2 getStatesWithPredecessors

Input:
cModel1, cModel2: the C++ models;
equiv_methods: relation of equivalent methods;
pre_hyp, post_hyp: pre- and post-hypothesis as logical formula;
Output:
a logical formula that describes all starting states of counterexamples and their predecessors that fulfill the pre-hypothesis

Description:

1: $result = false$
2: **while** ($generateCounterexample(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp) \neq$ **null**) {
3: $(start, follow, method) = generateCounterExample(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp)$;
4: $result = result \vee start$;
5: $pre\_hyp = pre\_hyp \wedge \neg result$;
6: $post\_hyp = \neg result$;
7: }
8: **return** $result$

The proof is done with stepwise induction and either shows that when the models are in a state that fulfills the hypothesis and methods that are defined as equivalent are executed, the methods return equivalent results and the following state fulfills the hypothesis, or returns a counterexample. Counterexamples consist of an originating state $start$, a pair of methods that are defined as equivalent $method$ and a following state $follow$. A counterexample is denoted as a triple $(start, follow, method)$ which contains the according information. In the counterexample, the methods either return different results or the following state does not fulfill the hypothesis. Counterexamples are used to adjust the hypothesis until either the equivalence is proven or disproven. Algorithm 1 shows the pseudo code.
The function $generateCounterexample$ returns such a counterexample and receives the two models $cModel1$ and $cModel2$, the relation of methods $equiv\_methods$, a pre-hypothesis pre_hyp, and a post-hypothesis $post\_hyp$ who should hold in the starting and following state respectively.
Another function that is used by our algorithm is $getStatesWithPredecessors$ which returns all counterexamples and their predecessors. The inputs are identical to $generateCounterexample$. Different from $generateCounterExample$, this function returns a logical formula that describes all starting states of existing counterexamples as well as the predecessors of those counterexamples that fulfill the pre-hypothesis. The pseudo code can be seen in Algorithm 2.
In lines 1 – 3 of the algorithm, *forbidden states* are excluded from the hypothesis. We define forbidden states as states in which a call of methods defined as equivalent returns different results and the predecessors of those states.
By setting the post-hypothesis to $true$ any counterexample is generated due to different results from the methods. For each forbidden state found, pre- and post-hypothesis are adjusted to exclude that state. This enables us to find the predecessors of forbidden states and will prevent the generation of a counterexample that has already been found. In line 4, it is checked if the initial state is a forbidden state. If this is true, the models are not equivalent and the algorithm terminates. Afterwards, the hypothesis is advanced to exclude all discovered forbidden states. In the next step in lines 8 – 25 we generate counterexamples where the following state does not fulfill the hypothesis. If there is such a counterexample $(start, follow, method)$ then $follow$ does not fulfill the hypothesis since $start$ cannot be a forbidden state.
We handle the counterexample similar to the previous step where we discovered the forbidden states and exclude $start$ and all its predecessors from the hypothesis, which can be seen in lines 12 – 13. If the initial state is not excluded that way, the counterexample was not reachable and $start$

and its predecessors can safely be excluded from the hypothesis in line 23. However, if the initial state is excluded the state $start$ and therefore the original counterexample is reachable. The counterexample was generated due to the state $follow$ not fulfilling the hypothesis but not due to different results from the methods. If $follow$ is forbidden, which is checked in lines 16 – 18, the models are not equivalent which is returned in line 19. Otherwise in line 21 $follow$ is included into the hypothesis. All successors of $follow$ are checked in the same way. This is repeated until no counterexamples remain.
If there was no counterexample that disproved the hypothesis, the models are equivalent. Additionally, the final hypothesis is an invariant which can be used to speed up further equivalence checks after the modification of one model or other checks.

3. Experimental Results

Experiments were done with CBMC[CKL04] as backend. A simple example was tested with multiple hypotheses. The check took only a few seconds with good hypotheses while $true$ lead to a timeout.

References

[CKL04] Clarke, E., D. Kroening, and F. Lerda: *A tool for checking ANSI-C programs*. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[FWF13] Finder, A., J. Witte, and G. Fey: *Debugging HDL designs based on functional equivalences with high-level specifications*. IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, pages 60–65, 2013.

[GMYF09] Gao, S., T. Matsumoto, H. Yoshida, and M. Fujita: *Equivalence checking of loops before and after pipelining by applying symbolic simulation and induction*. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies*, pages 380–385, 2009.

[KJJP09] Koelbl, A., R. Jacoby, H. Jain, and C. Pixley: *Solver technology for system-level to RTL equivalence checking*. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 196–201, 2009.

[MSF06] Matsumoto, T., H. Saito, and M. Fujita: *Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs*. In *Proceedings of the International Symposium on Quality Electronic Design*, pages 370–375, 2006.

[SBCJ05] Shashidhar, K., M. Bruynooghe, F. Catthoor, and G. Janssens: *Functional equivalence checking for verification of algebraic transformations on array-intensive source code*. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1310–1315, 2005.

Funktionale Abdeckungsanalyse für C-Programme *

Aljoscha Windhorst[1] Hoang M. Le[1] Daniel Große[2] Rolf Drechsler[1,3]
[1]Universität Bremen [2]solvertec GmbH [3]DFKI GmbH
{alomat,hle,drechsle}@informatik.uni-bremen.de grosse@solvertec.de

Zusammenfassung

Die formale Verifikation spielt bei der Entwicklung von sicherheitskritischen Systemen eine wichtige Rolle. Unvollständige Spezifikationen können jedoch zur erfolgreichen Verifikation fehlerhafter Systeme führen. Im Hardware-Umfeld werden deshalb zunehmend Techniken zur funktionalen Abdeckungsanalyse von Spezifikationen diskutiert. Darauf aufbauend stellt diese Arbeit ein Verfahren zur funktionalen Abdeckungsanalyse von C-Programmen vor. Dabei wird untersucht ob eine Spezifikation in Form einer Menge von Zusicherungen das Eingabe-Ausgabe-Verhalten ihrer Implementierung eindeutig beschreibt.

1. Einleitung

Die Programmiersprache C spielt bei der Entwicklung von hardwarenahem Code, etwa auf Betriebssystemebene oder für eingebettete Systeme, eine wichtige Rolle [BM09]. Diese Anwendungsbereiche stellen hohe Anforderungen an die Qualität der Software. In sicherheitskritischen Domänen, in denen sich etablierte Methoden wie Tests oder statische Analysen als unzureichend erweisen, kann mit Hilfe formaler Verifikationsmethoden ein zusätzlicher Sicherheitsgewinn erzielt werden. In der Praxis haben sich zu diesem Zweck die vollautomatischen Modellprüfungsverfahren [CGP99] durchgesetzt, die bereits in Form einer Vielzahl von *Beweisern* implementiert wurden [DKW08]. Die meisten dieser Beweiser ermöglichen den Nachweis der Gültigkeit einer Spezifikation des Eingabeprogramms in Form einer Menge von benutzerdefinierten Eigenschaften. Misslingt der Beweis, wird ein Gegenbeispiel ausgegeben, das dem leichteren Auffinden des ursächlichen Fehlers dient. Entsprechende C-Beweiser werden z.B. in [CKL04, MFS12] vorgestellt. Ihre Leistungsfähigkeit zu steigern ist ein sehr aktuelles und aktives Forschungsfeld [Bey12].
Die Korrektheit einer Implementierung ist jedoch auch mit dem Nachweis der Einhaltung ihrer Spezifikation nicht sichergestellt. Letztere kann mitunter unvollständig sein, wodurch Teile des Systems ungeprüft bleiben können. Dies führt im schlechtesten Fall zu „false positives“: der erfolgreichen Verifikation eines fehlerhaften Programms. Eine entscheidende Frage bei der Formulierung einer Spezifikation lautet deshalb: „Have I written enough properties?“ [KGG99]. Im Bereich der Hardware-Verifikation wurden zu ihrer Beantwortung verschiedene Ansätze einer *funktionalen Abdeckungsanalyse* entwickelt. Viele davon verfahren nach dem gleichen Grundprinzip: Die Äquivalenz des Verhaltens der Ausgänge zweier Modelle M_0 und M_1 des untersuchten Systems wird bewiesen. Sei M_0 ein Modell der Spezifikation, so wählen [KGG99, GKD08] für M_1

*Diese Arbeit wurde zum Teil vom Bundesministerium für Bildung und Forschung (BMBF) im Rahmen des EffektiV-Projekts unter der Vertragsnummer 01IS13022E und von der Deutschen Forschungsgemeinschaft (DFG) im Rahmen des Reinhart Koselleck-Projekts DR 287/23-1 gefördert.

ein Modell der Implementierung. [Cla07, Bor09] hingegen wählen eine zweite Instanz des Spezifikationsmodells und ermöglichen somit eine implementierungsunabhängige Analyse. Für beide Varianten gilt: Zeigen M_0 und M_1 das gleiche Eingabe-Ausgabe-Verhalten, gilt die Spezifikation als funktional vollständig.
Für die formale Verifikation von Software sind uns bisher keine Abdeckungsbegriffe bekannt. In dieser Arbeit wird deshalb ein Verfahren zur funktionalen Abdeckungsanalyse von C-Programmen vorgestellt, das, den Ansätzen aus [KGG99, GKD08] folgend, auf dem Äquivalenzbeweis zweier *Speichermodelle* des Eingabeprogramms basiert: M_{spec} repräsentiert die Zustände des Arbeitsspeichers, wie diese durch die Spezifikation definiert werden. M_{impl} repräsentiert die aus der Implementierung resultierenden Zustände. Erfüllt die Implementierung ihre Spezifikation, wird die Äquivalenz einer Menge von Programmausgaben in M_{spec} und M_{impl} mit Hilfe eines SMT-Beweisers [GD07] nachgewiesen. Als Ergebnis wird entweder die funktionale Vollständigkeit der Spezifikation bezüglich der Implementierung zugesichert oder ein Gegenbeispiel ausgegeben.

2. Einleitendes Beispiel

```
1  int npo2(int x) {
2    x--;
3    for(int i = 1; i < sizeof(int) * 8; i *= 2)
4      x = x | (x >> i);
5    return x + 1;
6  }
7
8  void proof() {
9    /* Vorbedingungen */
10   int x;
11   assume( x > 0 );
12   assume( x < 32768 );
13   /* Verifikation der Spezifikation */
14   int result_impl = npo2( x );
15   assert( result_impl <= ( x << 1 ) );
16   assert( ( result_impl & ( result_impl - 1 ) ) == 0 );
17   /* Definition des Spezifikationsmodells */
18   int result_spec;
19   assume( result_spec <= ( x << 1 ) );
20   assume( ( result_spec & ( result_spec - 1 ) ) == 0 );
21   /* Verifikation der Abdeckungseigenschaft */
22   assert( result_spec == result_impl );
23 }
```

Das obige Beispiel zeigt die funktionale Verifikation der Funktion *npo2* und den Abdeckungsbeweis der dazugehörigen Spezifikation in Form einer Menge von Zusicherungen. Dabei werden die beweiserspezifischen Anweisungen für Annahmen `assume(boolean)` und Zusicherungen `assert(boolean)` eingesetzt. Erstere setzt den übergebenen Booleschen Ausdruck für alle folgenden Beweise als im entsprechenden Zustand gültig voraus, während letztere einen Beweisversuch des übergebenen Ausdrucks herbeiführt. Dies entspricht der Semantik in anderen Beweisern wie z.B. CBMC [CKL04]. Die im Quelltext dargestellte Funktion *npo2* sei das zu verifizierende Programm. Es gibt die nächstgrößere Zweierpotenz (*next power of 2*) zur einzigen Eingabe x zurück. Die Zeilen 10-16 der Funktion *proof* führen die Verifikation der Spezifikation durch, die in Form von zwei Annahmen (Zeile 11-12) und zwei Zusicherungen (Zeile 15-16) erfolgt. Der Aufruf der untersuchten Funktion konstruiert das Speichermodell der Implementierung. Die Rückgabe von *npo2* wird dabei in einer Instanz der Variable *result_impl* gespeichert (Zeile 14). Die Annahmen schränken den Wertebereich der Eingabe x ein. Die zwei Zusicherungen bilden die eigentliche Spezifikation des Programms: `result_impl <= (x << 1)` schließt zu große Werte aus,

`(result_impl & (result_impl - 1)) == 0` verlangt eine Zweierpotenz (nur ein Bit ist gesetzt). Beide Zusicherungen lassen sich erfolgreich verifizieren. Das Programm könnte nun als vollständig verifiziert gelten. Allerdings stellt sich die Frage, ob die verifizierte Spezifikation tatsächlich funktional vollständig war.
Die Antwort liefern die Zeilen 18-22 der Funktion *proof*. Diese Zeilen stellen eine händische Umsetzung der funktionalen Abdeckungsanalyse dar. Zunächst wird in Zeile 18 die Variable *result_spec* instantiiert. Durch die Annahme der erfolgreich verifizierten Zusicherungen (Zeile 19-20) bildet diese genau das spezifizierte Verhalten in Abhängigkeit von der Eingabe x ab. Die folgende Zusicherung *result_impl* $==$ *result_spec* garantiert, dass die Spezifikation den gesamten Funktionsumfang der Implementierung erfasst und umgekehrt. Diese Zusicherung (wird im Folgenden auch Abdeckungseigenschaft genannt) lässt sich für die vorliegende Spezifikation nicht beweisen. Der Beweiser liefert ein Gegenbeispiel der Form $x = 5$, *result_impl* $= 8$, *result_spec* $= 4$. Das Gegenbeispiel zeigt eine Lücke in der Spezifikation von *npo2* auf. Es wurde zwar spezifiziert, dass keine zu großen Zweierpotenzen ausgegeben werden dürfen, zu kleine Zweierpotenzen hingegen sind erlaubt. Aus diesem Grund wurde im Gegenbeispiel *result_spec* mit 4 belegt, obwohl $x > 4$ gilt. Um die Spezifikation zu vervollständigen, wird die Zusicherung `assert(result_impl>x)` hinzugefügt. Diese lässt sich jedoch nicht verifizieren. Das Gegenbeispiel hat die Form $x = 2$, *result_impl* $= 2$. Die Implementierung ermittelt für eine Zweierpotenz als Eingabe die nächste Zweierpotenz einschließlich der Eingabe selbst. Die Spezifikation hingegen fordert für alle Eingaben die *nächstgrößere* Zweierpotenz als Ausgabe. Es wurde ein anderes Programm implementiert als spezifiziert wurde. Trotzdem wäre dieses Programm ohne die Abdeckungsanalyse erfolgreich verifiziert worden. Um den Fehler zu beheben, wird die Dekrementierung in Zeile 2 entfernt. Alle drei Zusicherungen sowie die Abdeckungseigenschaft lassen sich danach erfolgreich verifizieren.

3. Formalisierung

Ein Programm wird genau dann von seiner Spezifikation funktional abgedeckt, wenn diese für jede mögliche Eingabe die Ausgaben des Programms eindeutig definiert und die Ausgaben der Implementierung mit diesen eindeutig spezifizierten Ausgaben übereinstimmen. Programmausgaben erschöpfen sich nicht in Funktionsrückgaben, sondern können durch eine Vielzahl von Mechanismen, wie z.B. Pointern oder Shared Memory, in beliebigen Zuständen erfolgen. Die folgende Definition abstrahiert von der Art des Speicherzugriffs. Ein Speichermodell in einem Zustand $s \in S$ sei definiert als ein Feld von Wörtern $M^s : W \to W$ mit $W \subset \mathbb{N}$: die Menge der Wörter entsprechend der Wortbreite. Eine Programmausgabe $o^s \in O^s$ wird dementsprechend definiert als die Konkatenation einer Menge von Wörtern $M^s(a)M^s(a+1)\dots M^s(a+k)$ mit $a, \dots, a+k \in W$: die Adressen der dazugehörigen Wörter und $k \geq 0$ entsprechend der Breite des Datentyps. Zur Vereinfachung werden ausschließlich Ausgaben im Endzustand eines terminierten Programms betrachtet. Für jede Ausgabe existiert eine Abbildung o_{spec} im Spezifikationsmodell sowie o_{impl} im Implementierungsmodell. Sei I die Menge der aus den Anweisungen der Implementierung resultierenden Eigenschaften, P die Menge der Zusicherungen und A die Menge der Annahmen, so gilt nach dem Beweis der Zusicherungen $\phi : (\bigwedge A \wedge \bigwedge I) \wedge (\bigwedge A \wedge \bigwedge I \Rightarrow \bigwedge P)$ für M_{impl} sowie $\psi : \bigwedge A \wedge \bigwedge P$ für M_{spec}. Die zu beweisende Abdeckungseigenschaft lautet nun $\phi \wedge \psi \wedge (\forall o \in O : o_{spec} = o_{impl})$. Da die Spezifikation in Form von P in beiden Modellen gilt, ist leicht zu erkennen, dass eine Ausgabe o nur dann differieren kann, wenn P sie nicht eindeutig beschreibt. Die Formulierung der Abdeckungseigenschaft lässt sich automatisieren. Dazu werden

Programm	Anzahl Ausgaben	Laufzeit (s)	Verifiziert	Vollständig
npo2	$1 * 32$ Bit	1	ja	ja
BubbleSort	$1...5 * 32$ Bit (Feld)	2869	ja	ja
SimpleWhile100	$1 * 32$ Bit	12	ja	nein
Bitcount16	$2 * 32$ Bit	2	ja	nein
IntSqRoot	$1 * 32$ Bit	4	ja	nein
Queue10	$10 * 32$ Bit (Feld)	12	ja	ja
SelectSort	$1...5 * 32$ Bit (Feld)	108	ja	nein

Tabelle 1: Testergebnisse

die Zusicherungen der Spezifikation in Annahmen umgewandelt und auf M_{spec} angewendet. Anschließend wird die oben genannte Abdeckungseigenschaft bewiesen.

4. Ergebnisse

Die vorgestellte automatische Abdeckungsanalyse wurde in Form eines prototypischen Beweisers implementiert, dessen Laufzeiten mit denen anderer C-Beweiser wie CBMC vergleichbar sind. Mit seiner Hilfe wurde eine Reihe kompakter Algorithmen wie z.B. *npo2*, *BubbleSort* sowie eine Auswahl von Benchmarks aus [MFS12] untersucht (siehe Tabelle 1). Mehrere dieser Algorithmen arbeiten mit Pointern und Feldern dynamischer Größe. Für BubbleSort wurde der Anteil des Abdeckungsbeweises an der Gesamtlaufzeit der Verifikation untersucht. Dieser lag weit unter 50%. Die Mehrzahl der vorliegenden Spezifikationen war unvollständig.
Die Komplexität des Beweisprozesses und die daraus resultierenden Laufzeiten heutiger C-Beweiser erlauben bisher nur die Untersuchung relativ kleiner Programmausschnitte. Deshalb ist ihre Anwendung nur für besonders elementare aber funktional komplexe Teile eines Programms sinnvoll. Genau in diesen Fällen kann auch eine funktional vollständige Spezifikation und der Nachweis ihrer Vollständigkeit von großem Nutzen sein und ist, wie mit dem vorliegenden Verfahren gezeigt wurde, mit moderatem Aufwand auch möglich. Die nächsten Entwicklungsschritte liegen in der Durchführung einer implementierungsunabhängigen Abdeckungsanalyse und der Unterstützung zustandsübergreifender Spezifikationen.

Literatur

[Bey12] Beyer, Dirk: *Competition on software verification*. In: *TACAS*, Seiten 504–524. 2012.

[BM09] Barr, Michael und Anthony Massa: *Programming embedded systems: with C and GNU development tools*. O'Reilly, 2009.

[Bor09] Bormann, Jörg: *Vollständige funktionale Verifikation*. Doktorarbeit, Universität Kaiserslautern, 2009.

[CGP99] Clarke, Edmund M., Orna Grumberg und Doron A. Peled: *Model checking*. 1999.

[CKL04] Clarke, Edmund, Daniel Kroening und Flavio Lerda: *A tool for checking ANSI-C programs*. In: *TACAS*, Seiten 168–176. 2004.

[Cla07] Claessen, Koen: *A coverage analysis for safety property lists*. In: *FMCAD*, Seiten 139–145, 2007.

[DKW08] D'Silva, Vijay, Daniel Kroening und Georg Weissenbacher: *A survey of automated techniques for formal software verification*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 27(7):1165–1178, 2008.

[GD07] Ganesh, Vijay und David L Dill: *A decision procedure for bit-vectors and arrays*. In: *CAV*, Seiten 519–531, 2007.

[GKD08] Große, Daniel, Ulrich Kühne und Rolf Drechsler: *Analyzing functional coverage in bounded model checking*. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 27(7):1305–1314, 2008.

[KGG99] Katz, Sagi, Orna Grumberg und Danny Geist: *"Have I Written Enough Properties?" - A Method of Comparison Between Specification and Implementation*. In: *CHARME*, Seiten 280–297. 1999.

[MFS12] Merz, Florian, Stephan Falke und Carsten Sinz: *LLBMC: Bounded model checking of C and C++ programs using a compiler IR*. In: *Verified Software: Theories, Tools, Experiments*, Seiten 146–161. Springer, 2012.

System Level Modeling of Piezoresistive Effect of Carbon Nanotubes for Sensor Application

Vladimir Kolchuzhin, Jan Mehner
Chair of Microsystems and Precision Engineering, TU Chemnitz

Milind Shende, Erik Markert, Ulrich Heinkel
Chair for Circuit and System Design, TU Chemnitz

Christian Wagner
Center for Microtechnologies (ZfM), TU Chemnitz

Thomas Gessner
Fraunhofer Institute for Electronic Nano System (ENAS) Chemnitz

Summary

The article deals with integration carbon nanotube (CNT) models as components at system level design. The framework VHDL-AMS that uses compact models to describe components performing heterogeneous functions is used for implementation and simulation. The CNT mechanical and electrical compact submodels are based on the analytical models, lumped element models and the simulations results from density functional theory (DFT). The important aspects in developing the system models, the parameters necessary in creating models are presented and discussed in the article.

1 Introduction

Microelectromechanical systems, actuated by electrostatic force, are widely used in various applications such as the pressure sensors, microphones, switches, resonators, accelerometers, gyroscopes, RF-filters, tunable capacitors, and micromirrors. But the accuracy of geometry and therefore the characteristics (sensitivity, signal-to-noise ratio, and frequency range) of MEMS based sensors are limited by the technology tolerances. Today, the nanoscale dimension is explored to look out for new sensing principles. One of them is the piezoresistive effect of carbon nanotubes, useful for a nanoscaled acceleration sensor. Recently, several groups have reported on techniques for making networks of CNTs that can be placed onto metallic pads [CLC10].

The NEMS/MEMS design is a very challenging multilevel and interdisciplinary task. The levels in design include synthesis of lumped elements, process sequence development and mask layout drawing, component and system simulations. Different algorithms and software tools are used for this purpose. Numerical simulations are used both as a design tool and for understanding complex device behavior. Commonly used numerical simulation methods (FDM, FEM and BEM) provide results for a given set of geometrical and material parameters. For CNTs, classical simulation approaches based on finite elements do not hold, because their electrical properties directly depend on the atomic positions of the carbon atoms. Therefore, quantum-mechanical simulations of CNTs are done using density functional theory, which is a standard technique for simulations of systems consisting up to hundreds of atoms. This article presents a system level model of piezoresistive effect of single-walled CNT for sensor application.

2 CNT Compact Models

2.1 Analytical Model

The change in electrical resistance of ideal CNT due to strain can be calculated by an empirical transport formula based on band-gap changes [CLC10]:

$$R = R_C + \frac{1}{|t|^2}\frac{h}{8e^2}\left[1+\exp\left(\frac{E_{gap}(\varepsilon)}{kT}\right)\right] \quad (1)$$

Here, R_C denotes the contact resistivity; t^2 is the transmission probability of the nanotube, usually taken as 1; h, e and k are physical constants; T is the temperature and E_{gap} is the bandgap of the nanotube. The bandgap of a strained CNT is linearly changing with the relative deformation. An analytical formula for CNTs with a large radius with a slight modification is given:

$$E_{gap}(\varepsilon) = E_{gap}^0 + \mathrm{sgn}(2p+1)3t_0\sqrt{2}(1+\nu)\varepsilon\cos 3\theta, \quad (2)$$

where p is a value determined by the chiral indices (n,m) of the nanotube in a way that $p = [n\text{-}m]_3$: $p = 2 \mapsto p = -1$; $t_0 = 2.66$ eV is the hopping parameter; $\nu \approx 0.2$ stands for the Poisson's ratio; θ represents the chiral angle of the CNT and ε denotes the relative strain. The bandgap of the unstrained state can be written for semiconducting CNTs as $E_{gap}^0 = 3a_0t_0/2r_0$, where $a_0 = 1.42$ Å is the lattice constant of graphene and r_0 is the radius of the CNT.

2.2 DFT modelling

DFT is a quantum mechanical theory, which solves the electronic structure problem and provides more reliable results and helps to confirm analytical models [WSG12]. Nevertheless, its computational cost is very high and only up to hundreds of atoms can be calculated. For simulations, we use Atomistix ToolKit from QuantumWise. As the amount of simulated atoms is very small, periodic boundary conditions have been applied to obtain the results for infinite CNTs. Thus, there will be no quantization effects in this direction in the experiments and infinite CNTs may be assumed for simulation.

The results obtained by such kind of simulations are the mechanical and electrical properties of strained CNTs. For the mechanical part, the procedure is as following: the atoms of the CNTs are placed in a unit cell, which is periodic into all special directions. Then, a geometry optimization is performed to relax all internal forces. At each stretch step, the unit cell and the according atomic coordinates are scaled linearly, followed by another geometry relaxation. The total quantum-mechanical energy is saved and traced over the deformation range.

The conductivity of ideal CNTs is obtained by an empirical transport formula (1). The bandgap is evaluated by DFT calculations as well as the analytical formula (2). The (n,0)-CNTs are the most sensitive ones. The bandgap and the total energy dependences on the relative deformation are shown in Fig. 1. It is highlighted that different kinds of CNTs, which are geometrically very similar, show a strongly different behavior. For the (13,0) and (14,0) CNT, a band gap exists at zero strain. They differ in the gradient of the bandgap at positive strain: for the (14,0) CNT, it decreases and for the (13,0) it increases. The (15,0) CNT deviates, as its band gap is approximately zero in the original state and its resistivity increases in both tensile and compressive directions. After reaching the maximum resistivity, it decreases again. The obtained results show only minor deviations of the DFT and the analytical model.

These models are extremely helpful when it comes to compact models for lumped elements simulations for the sensor itself. DFT calculations take hours of time, whereas an analytical model is computed within a fraction of a second. In contrast to DFT, analytical models are not that transferable, e.g. they break down at CNTs whose diameter is smaller than the Van-der-

Waals distance. Within fabrication, such small-diameter CNTs are rarely found and therefore, the analytical model is suitable.

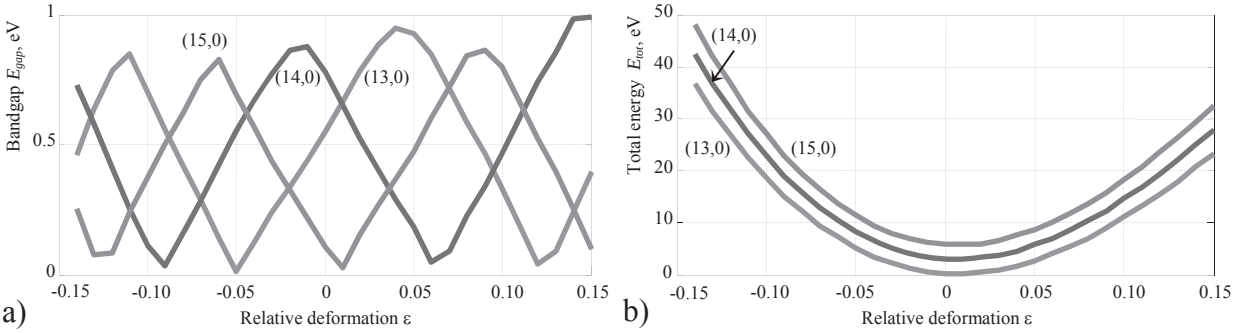


Figure 1: The bandgap and the total energy of the CNTs traced over the relative deformation

3 Implementation in VHDL-AMS

The VHDL-AMS model is similar to a black-box model with terminals relating electrical and mechanical quantities based on the generalized Kirchhoff's laws. The single-walled CNT at the system level is presented as a four terminals device (two mechanical and two electrical terminals), which consist of the nonlinear mechanical spring unidirectional coupled with the strain dependent resistor. The behavioral model of spring is defined by:

$$f(t) = k_{\mathrm{CNT}} u(t) \text{ in linear case, and}$$

$$f(t) = \frac{\partial E_{tot}}{\partial u} \text{ in nonlinear case,}$$

where an axial stiffness of CNT k_{CNT} can be given as: $k_{\mathrm{CNT}} = A \cdot E / L_{\mathrm{CNT}}(0)$. The Van-der-Waals distance (6.3 Å) is the distance of two neighboring carbon sheets in graphite and is used to define the effective cross section area *A*. The next step is to define the behavioral model of the resistor:

$$i(t) = v(t) / R_{\mathrm{CNT}}(\varepsilon),$$

The CNT-model is tested using the simulator ANSYS Simplorer. It is necessary to specify the external circuitry (voltage source E1=1V, and controller units AM1, VM1), loads (mechanical force source F1 and controller units FM1, SM1) and solver parameters (time step size, total simulation time). The schematic of this measurement setup is depicted in Fig. 2.

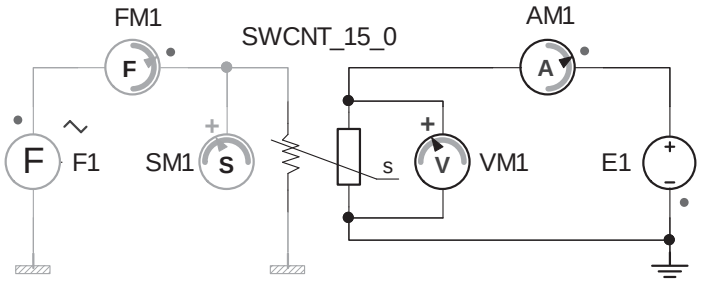


Figure 2: Testbench circuit for VHDL-AMS model of single-walled CNT in Simplorer

The simulation results of an analysis for the semi-metallic CNT (15,0) are shown in Fig. 3. It can be seen that for linearly changing input force, CNT shows nonlinear change in resistivity. The resistance of CNT is also shown in Fig. 3 on a logarithmic scale, as it exponentially depends on the bandgap. The current flowing through the CNT can be calibrated to get percentage change in applied strain. Beyond this simple test case, one can simulate the CNT based mechanical sensor together with any other system environment or complex electronic circuit.

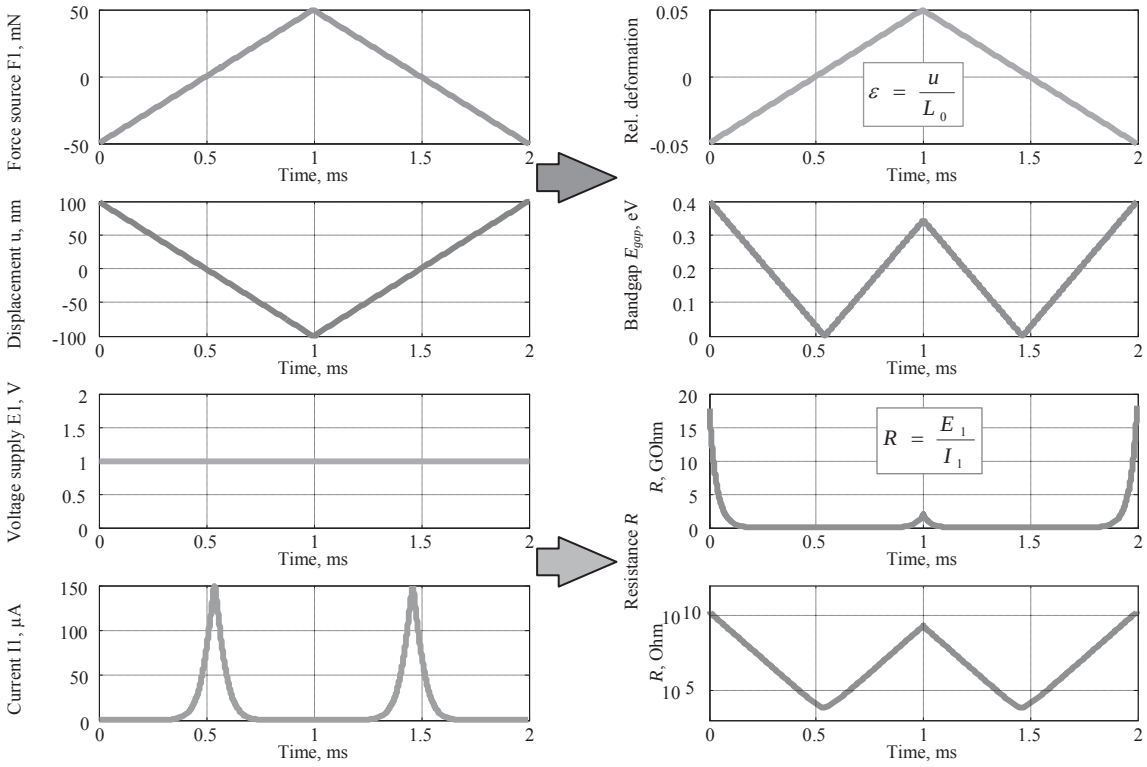


Figure 3: Results of an analysis for the semi-metallic CNT (15,0)

4 Conclusions

The important aspects in developing the system models of CNTs, the parameters necessary in creating models are presented and discussed. Density function theory model is described which is used for quantum mechanical simulation of CNTs. DFT modelling is important because it deals with the electronic properties of CNT that directly depend on the atomic positions of the carbon atoms in the tube. High level behavioral and structural abstract model of CNT strain gauge sensor is discussed using hardware description language VHDL-AMS.
The simulation results are shown. The presented sensor project is in concept development stage. Future work consists of finalizing the detailed specifications of the whole mechanical sensor based on CNT, implementing precise and possibly every minimal electro-mechanical effects of the sensor in HDL models, considering other environmental effects on sensor such as temperature, humidity, pressure etc. Automation of model extraction and co-simulation using different modelling tools are also the part of future work.

Acknowledgments

This work has been done within the Research Unit 1713 which is funded by the DFG.

Literature

[CLC10] M.A. Cullinan and M.L. Culpepper: *Carbon nanotubes as piezoresistive microelectro-mechanical sensors: Theory and experiment*, Ph. Review B82, 115428, 2010.

[WSG12] C. Wagner, J. Schuster, and T. Gessner: *DFT investigations of the piezoresistive effect of carbon nanotubes for sensor application*. Phys. Stat. Sol. B249, 2450-2453, 2012.

Hybride Prototypisierung eines Sensorsubsystems

S. Stieber, Johann-P. Wolff, Ch. Haubelt
Universität Rostock/IMD
sebastian.stieber2@uni-rostock.de

Rainer Dorsch
BOSCH Sensortec GmbH, Reutlingen

Zusammenfassung

In dem vorgestellten Ansatz werden vorhandene Sensoren benutzt, um eine virtuelle Repräsentation eines digitalen Sensorsubsystems zu erschaffen, welches zur plattformunabhängigen Entwicklung von Treibersoftware genutzt werden kann. Die verwendete Methodik zur Synchronisation der verschiedenen zeitlichen Domänen beschränkt sich dabei auf eine Anpassung des High-Level-Modells und zieht keine Änderungen an der Treibersoftware nach sich.

1. Einleitung und Stand der Technik

Die Verwendung von physikalischer Hardware in Simulationssystemen wird als Hardware-in-the-Loop (HIL) bezeichnet. In Verbindung mit der hohen Simulationsgeschwindigkeit von High-Level-Modellen in SystemC lassen sich hybride Prototypen erschaffen, die in Echtzeit agieren und somit frühzeitig einen Eindruck von dem späteren Nutzungsgefühl des Produktes vermitteln können. In der Entwicklung von Sensorsubsystemen können vorhandene Sensoren als Datenlieferant für neu zu entwickelnde digitale Sensorsubsysteme genutzt werden. Dabei stellt die zeitliche Synchronisation zwischen realen Elementen (Sensoren) und der Simulation jedoch ein Problem dar. Die vorgestellte Lösung basiert auf der Idee, den Verlauf der Simulationszeit an die Echtzeit anhand von Synchronisationspunkten anzunähern.
HIL findet bereits seit längerer Zeit Anwendung in der Verifikation von virtuellen Prototypen. Fathy et al. beschreiben die grundlegenden Anwendungsbereiche und deren Einsatzgebiete im Fahrzeugbau [FFHS06]. HIL-Systeme benötigen an der Schnittstelle zwischen Echtzeit- und Simulationszeitdömane Mechanismen zur Synchronisation, um Zeitverzögerungseffekte auszugleichen [CL12]. Defo et al. stellen in [DMR11] eine Methode zur Restbus-Simulation vor, bei der ein bestehendes Flexray-Netzwerk um ein SystemC-Modell, das das Verhalten weiterer Busteilnehmer emuliert, erweitert wird. Untersucht wird dann die Synchronisation zwischen dem schnellen realen Bussystem und dem langsameren simulierten Modell. Wir stellen einen Ansatz vor, in dem das simulierte Modell schneller ist als die physikalische Hardware.

2. Problembeschreibung

Der Prototyp eines Sensors wurde in der Modellierungs- und Simulationssprache SystemC modelliert. Es beinhaltet spezifizierte Datenraten und Registerbeschreibungen, sowie Interrupt-Pins, die beispielsweise erkannte Gesten oder das Vorhandensein neuer Daten signalisieren können. Die

Simulationsgeschwindigkeit dieses bitakkuraten Modells ist sehr hoch, da auf die Simulation zyklengenauer Abläufe verzichtet wird.

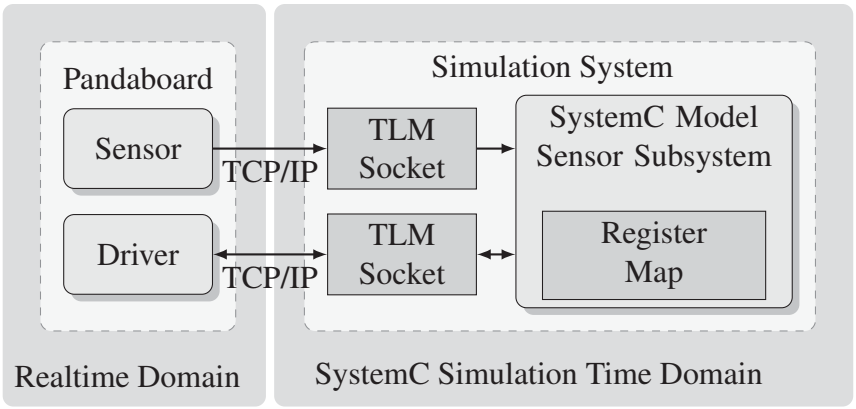


Abbildung 1: Anbindung der realen Sensoren an simuliertes Sensorsubsystem

Der hybride Prototyp (Abbildung 1) besteht aus einem Pandaboard und einem PC mit dem Betriebssystem Ubuntu 12.04, welche über Ethernet miteinander in Verbindung stehen. Die Bewegungsdaten von mehreren vorhandenen Inertialsensoren, welche über den I2C-Bus mit dem Pandaboard verbunden sind, werden periodisch von einem C-Programm ausgelesen und in größerer Anzahl über eine TCP-Verbindung an das High-Level-Modell auf dem PC gesendet. Eine hochfrequente Übertragung einzelner Samples ist aufgrund geringer Busbandbreite und langer Zugriffszeiten nicht möglich. Die Einspeisung der Samples in die Simulation erfolgt über eine SystemC-TLM-Schnittstelle [61312], welche gleichzeitig ein kontrolliertes Voranschreiten der Simulationszeit nach Eingabe eines Sensorsamples veranlasst. Anschließend erfolgt die Verarbeitung der aktuellen Sensorsamples innerhalb des Modells, gefolgt von der Aktualisierung der Registerwerte. Zusätzlich steht, als Abstraktion eines Buszugriffs, eine TCP-Schnittstelle als Zugriffspunkt des Treibers zur Verfügung und dient zur Annahme und Beantwortung von Anfragen an die Register Map. Die einzelnen Bestandteile des Versuchsaufbaus können in unterschiedliche zeitliche Domänen eingeordnet werden. Während das Modell auf Basis der SystemC-Simulationszeit arbeitet, agieren die Sensoren sowie die zu entwickelnde Software auf dem Pandaboard in Echtzeit. Die Verarbeitung der Samples innerhalb der Simulation muss mit der Echtzeit synchronisiert werden, um realitätsgetreue Ausgangssampleraten des Sensorsubsystems zu erzeugen. Abbildung 2 verdeutlicht dieses Problem anhand eines Beispiels. Der zu entwickelnde Treiber liest Sensordaten mit einer festen Datenrate vom Sensorsubsystem. Dabei wird zu jedem Lesezeitpunkt ein neuer Registerzustand des Sensors erwartet. Läuft die Simulation des Sensorsubsystems, wie in a) dargestellt, zu schnell ab, entsteht eine falsche zeitliche Auflösung der durchgeführten Bewegung. Eine folgende Gestenerkennung würde schließlich fehlerbehaftet durchgeführt.
Um diese zeitliche Verzerrung der Bewegungsdaten zu verhindern und eine realitätsnahe Repräsentation des simulierten Sensors zu erzeugen muss die Simulation gezielt verlangsamt werden.

3. Zeitsynchronisation

Es können Synchronisationspunkte im High-Level-Modell definiert werden, die eine Angleichung der Simulationszeit an die Echtzeit bewirken. Dabei muss der Zeitabstand zwischen diesen Synchronisationspunkten an die höchste für das Interface wichtige Änderungsrate angepasst werden.

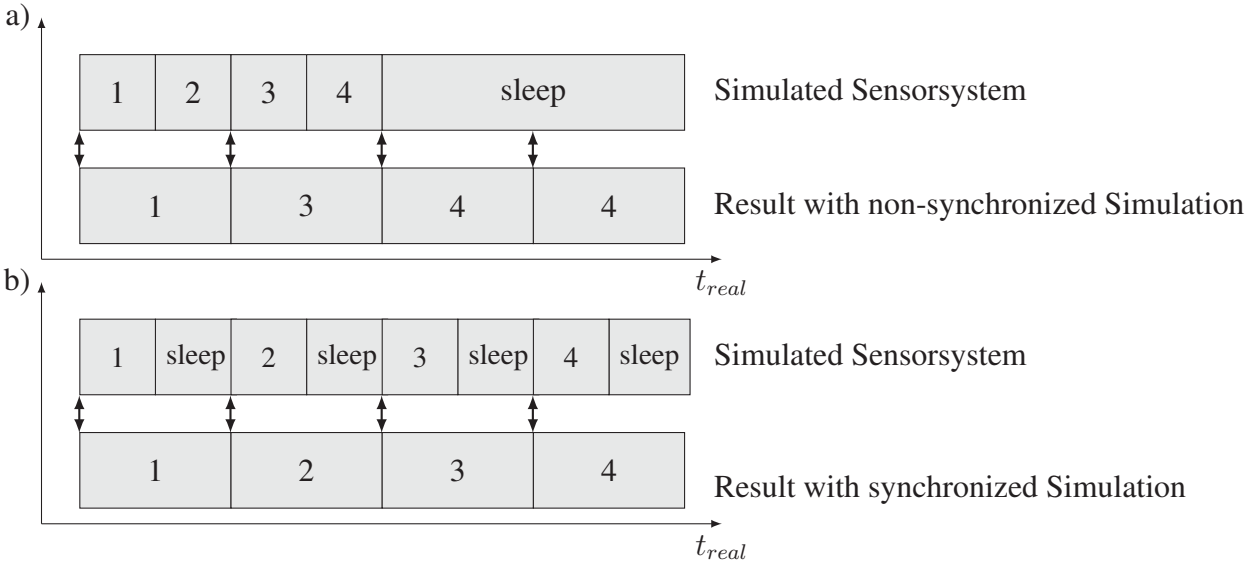


Abbildung 2: Datenabfrage bei nicht synchronisierter Simulation

In der Regel liegen die benötigten Sensordatenraten im Bereich von einigen Millisekunden. Die Werte der Register hängen von diesen Datenraten ab und ergeben somit die angestrebte Granularität der Zeitsynchronisation. An den Synchronisationspunkten kann schließlich die Simulation pausiert werden, bis Simulations- und Echtzeit wieder synchron sind. Aus dem beschriebenen Ansatz resultiert der in Abbildung 2 b) dargestellte Ablauf. Die Verzögerung der Simulation zu jeder Änderung der Registerwerte führt zu einer korrekten Datenabfrage des Treibers und somit zu einem geringeren Fehler bei der nachfolgenden Verarbeitung. Dennoch beeinflussen verschiedene Latenzen die Kommunikation der einzelnen Bestandteile des Systems. Die Gesamtverzögerung ergibt sich, wie folgt:

$$t_{latency} = t_{sampling} + t_{communication} + t_{calculation}$$

Durch die Sammlung der Sensordaten vor der Übertragung an die Simulation entsteht eine Verzögerung $t_{sampling}$. Diese Verzögerung ist abhängig von der Sensordatenrate, sowie der Anzahl der zu übertragenden Sensorsamples. Anschließend entsteht durch die Kommunikation über die Netzwerkschnittstelle eine weitere Latenz $t_{communication}$. Die Verarbeitung des ersten Sensorsamples bis zur ersten Aktualisierung der Register Map stellt eine weitere, allerdings vernachlässigbar kleine, Komponente $t_{calculation}$ der Gesamtverzögerung dar. Nach dieser initialen Verzögerung liegt stets ein konsistenter Zustand der Register vor, welcher sich abhängig von den gewählten Datenraten regelmäßig aktualisiert. Die Gesamtverzögerung $t_{latency}$ bleibt zwar über die gesamte Simulationsdauer vorhanden, jedoch stellt die Synchronisation anschließend die erwartete Reihenfolge der Sensorsamples sicher. Dadurch werden durchgeführte Bewegungen in den nachfolgenden Algorithmen der Treibersoftware realitätsgetreu verarbeitet.

4. Ergebnisse

Das beschriebene Modell umfasst insgesamt eine Größe von etwa 10.000 Zeilen Quellcode. Die Sensorperioden der verwendeten realen Sensoren liegen bei 500 μs. Das Sensor-Modell benötigt in der SystemC-Simulationsumgebung auf dem Referenzcomputer (Intel®Core™i5-3320M @ 2,6

GHz) durchschnittlich 380 μs zur Verarbeitung einzelner Samples und erreicht somit eine Simulationsgeschwindigkeit, die 24% schneller als die Echtzeit verläuft. Die hybride Prototypisierung des vorgestellten Sensorsubsystems besitzt allerdings auch einige Einschränkungen im Vergleich zu einem realen Hardwareprototypen. Die räumliche Trennung der Sensoren und der Treibersoftware von der eigentlichen Simulation führt zu einer, im Vergleich zur Sensordatenrate, hohen Latenz und schränkt somit die sinnvoll simulierbare Sensordatenrate ein. Auch bei einer Verlagerung der Treibersoftware auf die Simulationsplattform kommt es durch ungleichmäßiges Scheduling des zugrunde liegenden Betriebssystems bei sehr geringen Sensorperioden zum häufiger werdenden Verpassen der Synchronisationspunkte und somit zu einer ungleichmäßigen Aktualisierungsrate der Register Map. Durch die direkte Anbindung der Sensoren an die Simulationsplattform könnten die Sensordaten mit deutlich geringerer Latenz erfasst und verarbeitet werden. Somit könnten die beschriebenen Verzögerungen minimiert und die Qualität des hybriden Prototypen weiter erhöht werden.

5. Zusammenfassung

Das SystemC High-Level-Modell des Sensorsubsystem konnte mit Hilfe realer Sensoren virtualisiert und zur Treiberentwicklung genutzt werden. Mit Hilfe der beschriebenen Methodik der Zeitsynchronisation anhand von definierten Synchronisationspunkten wurde der Verlauf der Simulation an den realen Zeitverlauf der Sensoren, sowie der Treibersoftware angeglichen. Da diese Synchronität allein durch Anpassungen am SystemC-Modell des Sensorsystems erreicht werden kann, sind keine diesbezüglichen Änderungen an der Treibersoftware beim Wechsel vom Modell zum realen Sensor nötig. Die zeitliche Auflösung der Synchronisation besitzt dabei Grenzen, sofern herkömmliche Betriebssysteme ohne konkrete Echtzeitanforderungen verwendet werden. Diese Grenzen ergeben sich zum Einen durch die Kommunikationsverzögerung der Treiberzugriffe und zum Anderen durch die Schwankungen im Scheduling des Betriebssystems. Weiterhin ist eine hohe Simulationsgeschwindigkeit des High-Level-Modells Grundvoraussetzung für den Einsatz der Synchronisation. Der Abstraktionsgrad des Modells sollte hoch genug sein, um eine Simulation in Echtzeit oder schneller durchführen zu können. Einzelne Überschreitungen der zeitlichen Anforderungen können durch die Synchronisationstechnik kompensiert werden.

Literatur

[61312] *IEEE Standard for Standard SystemC Language Reference Manual*. IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), Seiten 1–638, 2012.

[CL12] Choi, C. und W. Lee: *Analysis and compensation of time delay effects in hardware-in-the-loop simulation for automotive PMSM drive system*. Industrial Electronics, IEEE Transactions on, 59(9):3403–3410, 2012.

[DMR11] Defo, G., W. Müller und H. Rommel: *Synchronisation eines SystemC Restbus-Simulators mit einem Hardware-In-the-Loop FlexRay Netzwerk.* In: *MBMV*, Seiten 219–228, 2011.

[FFHS06] Fathy, H., Z. Filipi, J. Hagena und J. Stein: *Review of hardware-in-the-loop simulation and its prospects in the automotive area*. In: *Defense and Security Symposium*, Seiten 62280E–62280E. International Society for Optics and Photonics, 2006.

Formale Methoden für Alle

(Erweiterte Zusammenfassung)

Mathias Soeken[1,2] Max Nitze[1] Rolf Drechsler[1,2]
[1]Arbeitsgruppe Rechnerarchitektur, Universität Bremen
[2]Cyber-Physical Systems, DFKI GmbH, Bremen
{msoeken,maxnitze,drechsle}@informatik.uni-bremen.de

Zusammenfassung

Formale Methoden können verwendet werden, um schwierige Probleme exakt zu lösen. Ihre Anwendung ist jedoch nicht trivial und erfordert in der Regel Expertenwissen. Wir stellen das Werkzeug DIAB vor, das formale Methoden näher an Programmierer bringt, mit dem Ziel die Komplexität der Verwendung formaler Methoden an die der Implementierung von Programmen anzunähern. Konkret werden in dieser Arbeit Erfüllbarkeitsbeweiser verwendet, um algorithmische Probleme automatisch aus einer Programmiersprache heraus zu lösen. Die Transformation in das Erfüllbarkeitsproblem übernimmt dabei das Werkzeug.

1. Einleitung und Motivation

Formale Methoden befinden sich heute in etwa dem Zustand in dem sich Programmiersprachen vor 50 Jahren befanden. Mithilfe von formalen Methoden lassen sich zwar viele schwierige Probleme lösen, vorhandene Werkzeuge sind jedoch nicht trivial einzusetzen und ihre Anwendung benötigt insbesondere auch ein gutes Verständnis der zugrundeliegenden Algorithmen. Die Eigenschaft, dass formale Methoden in externen Werkzeugen mit eigenen neuen Programmiersprachen ausgelagert sind, erschwert ihre Verwendung zusätzlich.

In dieser erweiterten Zusammenfassung wird ein Ansatz illustriert, der Programmierern formale Methoden zugänglich macht, ohne dass sie direkt sichtbar sind. Darüberhinaus ist der Algorithmus in eine existierende Programmiersprache eingebettet, in diesem Fall JAVA, so dass die gewohnte Arbeitsumgebung nicht verlassen werden muss.

Das Problem und die Lösungsidee sollen im Folgenden am Beispiel von Graphenfärben verdeutlicht werden, der Ansatz lässt sich aber analog auch auf andere Probleme anwenden. Ein Graph ist genau dann k-*färbbar*, wenn alle Knoten mit einer von k Farben gefärbt sind und benachbarten Knoten unterschiedliche Farben zugeordnet sind. Es soll ein Programm entwickelt werden, das einen Graphen ohne Färbung einliest und anschließend den Graphen mit k Farben färbt, falls eine solche Färbung existiert. Viele praktisch relevante Probleme lassen sich auf das Färben von Graphen abbilden (z.B. Registerallokation [CAC+81]).

Eine naive Umsetzung für dieses NP-vollständige Problem [Kar72] könnte alle Färbungen aufzählen bis eine valide gefunden worden ist. Aufwendigere Implementierungen würden eine *branch and bound* Variante mit *backtracking* verwenden.

Mithilfe des von uns entwickelten Ansatzes DIAB lässt sich das Problem wie in Abbildung 1 gezeigt umsetzen. Der Graph verwendet zwei Datenstrukturen für Knoten und Kanten, wobei der

```
import DIAB;
...

public class Vertex {
  private int id;

  @Variable
  private int color;
  public int getColor() { return color; }
  ...
}

public class Edge {
  private Vertex first, second;

  public Vertex getFirst() { return first; }
  public Vertex getSecond() { return second; }
  ...
}

public class Graph {
  private Collection<Vertex> vertices;
  private Collection<Edge> edges;
  private int k;
  ...

  public void parse(final String filename) { ... }

  @Constraint
  public boolean maximumColor(final Vertex vertex) {
    return vertex.color >= 0 && vertex.color < k;
  }

  @Constraint
  public boolean differentColor(final Edge edge) {
    return edge.getFirst().getColor() != edge.getSecond().getColor();
  }

  public static void main(String[] args) {
    Graph g = new Graph();
    g.parse("some_graph.col");
    DIAB.satisfy(g);
  }
}
```

Abbildung 1: Illustration für die direkte Einbettung formaler Methoden zum Finden einer Färbung

Knoten eine Feldvariable `color` enthält, die durch die Annotation `@Variable` gekennzeichnet ist. Die `Graph` Klasse hat eine Methode zum Parsen von Dateien. Die anderen Methoden sind als `@Constraint` annotiert und beschreiben die Bedingungen für eine valide Graphfärbung. Dies ist eine direkte Übersetzung der Definition für eine Graphfärbung in JAVA Quellcode. Der Code ist ausreichend für die Implementierung; alle durch '`...`' ausgelassenen Zeilen beschreiben Methoden zum Aufbauen des Graphen. In der `main` Methode wird eine Instanz des Graphen erzeugt und eine Datei eingelesen. Danach ist der Graph zunächst ungefärbt. Der Aufruf von `DIAB.satisfy(g)` belegt alle mit `@Variable` annotierten Feldvariablen, die über die Graphinstanz erreicht werden können, so dass alle Bedingungen der Methoden, die als `@Constraint` annotiert sind, erfüllt werden.

Ein ähnlicher Ansatz ist im Tool Pex [TS05] implementiert, der formale Methoden für *white box testing* in .NET Sprachen nutzt. Die Programmiersprache EIFFEL [Mey88] bindet durch das *Design-by-contract* Paradigma formale Methoden auf Syntaxebene an die Sprache.

2. Implementierung

In diesem Abschnitt soll die Implementierung des Algorithmus illustriert werden. Zunächst wird beschrieben wie durch `@Constraint` annotierte Methoden, im Folgenden *Constraint Methoden* genannt, übersetzt werden, so dass formale Methoden angewandt werden können. Im Anschluss werden die aktuell implementierten und geplanten Features diskutiert.

2.1. Übersetzung der Constraint Methoden

Die Implementierung der `satisfy` Methode, wie sie in Abbildung 1 angewendet wird, ist abstrakt wie folgt realisiert: Zunächst werden alle annotierten Constraint Methoden mithilfe der JAVA *Reflection API* ermittelt. Der Bezug der Bedingungen ergibt sich aus den Parametern der Methoden. In der Constraint Methode `maximumColor` muss die Bedingung für alle Knoten gelten, die über den Graphen erreicht werden können. Diese können direkt über die Feldvariable `vertices` und indirekt über die Feldvariable `edges` erreicht werden. Da dies per Reflection API erfolgt, muss `satisfy` keine Eigenschaften der Datenstruktur kennen. Die Constraint Methoden werden zur Laufzeit disassembliert und automatisch über den JAVA Bytecode in prädikatenlogische Formeln erster Ordnung übersetzt. Die freien Variablen werden über die `@Variable` Annotation ermittelt. Alle Formeln werden in einer Konjunktion zusammengeführt und das resultierende Erfüllbarkeitsproblem wird mit einem externen SMT Beweiser, im konkreten Fall Z3 [dMB08], gelöst. Aus einer möglichen erfüllenden Belegung können dann die konkreten Werte für die mit `@Variable` annotierten Feldvariablen ausgelesen und zugewiesen werden.

2.2. Aktueller Stand und nächste Schritte

Unsere Implementierung unterstützt aktuell Programme im Stil des Programms aus Abbildung 1. Das zu lösende Problem muss ein Erfüllbarkeitsproblem sein. Es ist auch möglich externe Bibliotheken in der Implementierung zu verwenden. Beispielsweise haben wir das Graphenfärbenbeispiel auf Basis der JUNG[1] Bibliothek implementiert, anstatt eine eigene Graphklasse zu definieren. Als Datentypen für freie Variablen werden derzeit **`int`** und **`boolean`** unterstützt. Es ist möglich, lokale Variablen in den Constraint Methoden zu verwenden, solange sie **`final`** deklariert sind, d.h. ihr Wert lässt sich nach initialer Zuweisung nicht mehr verändern. Operationsaufrufe in Constraint Methoden sind möglich, solange die aufrufenden Methoden keine Seiteneffekte haben.

In den nächsten Schritten sollen weitere Basisdatentypen wie beispielsweise **`float`** unterstützt werden. Außerdem soll untersucht werden, welche Vorteile funktionale JVM Sprachen wie CLOJURE oder SCALA bei der Übersetzung in prädikatenlogische Formeln haben. Formale Methoden sind nicht nur zum Lösen von Erfüllbarkeitsproblemen hilfreich, sondern auch für Optimierungsprobleme. Dies soll auch untersucht werden. Denkbar sind Programmkonstrukte wie in Abbildung 2 illustriert. Hier wird die Methode `numColors` als Optimierungskriterium durch die Annotation `@Objective` gekennzeichnet. In der `main` Methode wird anstatt `satisfy` die Funktion `minimize` aufgerufen, um die Anzahl der Farben im gefärbten Graph zu minimieren.

[1] jung.sourceforge.net

```
public class Graph {
  ...

  @Objective
  public int numColors() {
    final Set<int> colors = new HashSet<int>();
    for (Vertex v : vertices) {
      colors.add(v.getColor());
    }
    return colors.size();
  }

  public static void main(String[] args) {
    Graph g = new Graph();
    g.parse("some_graph.col");
    DIAB.minimize(g);
  }
}
```

Abbildung 2: Unterstützung von Optimierungsproblemen

3. Zusammenfassung

Wir haben den Ansatz DIAB illustriert, der formale Methoden näher an Programmierer bringt. Anstatt wie bisher auf externe Tools mit eigenen dedizierten Sprachen angewiesen zu sein, ist der vorgestellte Ansatz direkt in die Programmiersprache JAVA eingebettet. Dabei sind die Algorithmen als Klassenbibliothek implementiert worden ohne die Sprache ändern zu müssen.

Literatur

[CAC+81] Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins und Peter W. Markstein: *Register Allocation Via Coloring*. Computer Languages, 6(1):47–57, 1981.

[dMB08] Moura, Leonardo Mendonça de und Nikolaj Bjørner: *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Seiten 337–340, 2008.

[Kar72] Karp, Richard M.: *Reducibility Among Combinatorial Problems*. In: *Complexity of Computer Computations*, Seiten 85–103, 1972.

[Mey88] Meyer, Bertrand: *Eiffel: A language and environment for software engineering*. Journal of Systems and Software, 8(3):199–246, 1988.

[TS05] Tillmann, Nikolai und Wolfram Schulte: *Parameterized unit tests*. In: *European Software Engineering Conference (ESEC)*, Seiten 253–262, 2005.

A Logic for Cardinality Constraints (Extended Abstract)

Heinz Riener[⋆] Oliver Keszocze[⋆] Rolf Drechsler[⋆†] Görschwin Fey[⋆‡]
[⋆]Institute of Computer Science [†]Cyber-Physical Systems [‡]Institute of Space Systems
University Bremen, Germany DFKI GmbH, Germany DLR, Germany
{hriener,keszocze,drechsler,fey}@informatik.uni-bremen.de

Many decision problems from computer science and artificial intelligence are solved by reducing them to the NP-complete *Satisfiability* (SAT) [Coo71, Lev73] problem. Decision heuristics that efficiently decide large instances of SAT have been proposed. For expressing complicated problem constraints, however, richer logic fragments of first-order logic are often more convenient. *Satisfiability Modulo Theories* (SMT) [BSST09] offer a set of standard theories and logics fixing the interpretation of predicate and function symbols. A *theory* refers to a restriction of the non-logical symbols and typically provides domains, predicate symbols, and function symbols. A *logic* defines a language over a theory. For instance, when reasoning about bit-vectors, the theory FixedSizeBitVectors provides a bit-vector sort and a set of bit-vector arithmetic functions and relational predicates, e.g., `bvadd`. The predicate and function symbols refer to their standard meaning, e.g., `bvadd` refers to addition where bit-vectors are interpreted as binary numbers. Non-standard interpretations of these symbols are of no interest in practice. The logic QF_BV defines the language of closed quantifier-free formulæ built over an arbitrary expansion of the FixedSizeBitVectors theory.

A decision procedure for SMT is called an *SMT solver*. A variety of SMT solvers for different theories and logics[1] have been proposed and a standardized textual format, the SMT-LIB format (currently in its second version), for expressing SMT instances has been established [BST12]. This allows for an easy exchange of problem instances and enables the comparison of different SMT solvers.

When modeling problems as SMT instances, many applications demand for constraints expressing that at most k (or at least k) out of n Boolean variables must be true, where k and n are integers with $0 < k \leq n$. These constraints are called *(Boolean) cardinality constraints*. Cardinality constraints arise in many applications including formal verification, automated fault diagnosis, scheduling, and circuit synthesis.

In SMT-LIB, there is no logic providing predicate symbols which express the semantics of cardinality constraints. Thus researchers proposed several different logic encodings to lower cardinality constraints to *Conjunctive Normal Form* (CNF). For SAT, the most popular logic encodings are based on *Binary Decision Diagrams* (BDD) [BBR06, ES06], adder networks [ES06], sorter networks [CZI10, ANORC11], or modulo constraints [Aav11] which are then translated to CNF

[1] A detailed description of the available theories and logics can be found on http://www.smt-lib.org/.

```
(set-logic QF_CBV)
(set-option :cardinality-implementation adder)
(declare-fun x () Bool)
(declare-fun y () Bool)
(declare-fun z () Bool)
(declare-fun v () (_ BitVec 8))
(assert ((_ card_ge 2) x y z))
(assert ((_ card_lt 3) (bvule (_bv4 8) v) y z))
(check-sat)
(exit)
```

Figure 1: An SMT-LIB2 instance using logic QF_CBV

using Tseytin translation [Tse68]. Those logic encodings are especially designed in favor of unit propagation which is one of the main solving steps implemented in *Davis-Putnam-Logemann-Loveland* (DPLL)-based SAT solvers. Intuitively, a logic encoding of a cardinality constraint allows for fast solving if unit propagation is able to infer false for all variables of the constraint when k out of n variables are assigned true. This property is also known as *arc-consistency* in the *Constraint Programming* (CP) community.

For SAT solvers, a logic encoding which uses redundant clauses to enable arc-consistency is preferred over succinct logic encodings. For SMT solvers, results are not so clear: on the one hand, empirical evaluations [SWFD09, ANORC11] showed that SMT solvers have the potential to significantly outperform SAT solvers for particular problem instances. On the other hand, for several problem instances SMT solvers perform worse than SAT solvers. Moreover, different SMT solvers seem to prefer different logic encodings [SWFD09].

We introduce a novel SMT logic, *cardinality constraints over Booleans* (Cardinality)[2], which provides predicate symbols for expressing cardinality constraints natively in SMT-LIB2 using *the theory of Booleans* (Core). This logic can be combined with existing SMT-LIB2 logics. For instance, QF_CBV refers to the logic of quantifier-free bit-vectors with cardinality constraints.

In Figure 1, an SMT instance with two cardinality constraints is shown in the textual SMT-LIB2 format. The syntax of the cardinality constraints is design similar to parameterized bit-vector operations like `zero_extend` or `extract`. For instance, `((_ card_ge 2) x y z))` expresses a greater equal-cardinality constraint that evaluates to true iff at least two out of the three Boolean variables x, y, and z are assigned true. Notice that the arguments of the cardinality constraint can be any expression of Boolean sort. Furthermore, we use the SMT option `cardinality-implementation` to select a solver-specific logic encoding.

The SMT logic provides a standard to describe cardinality constraints in SMT-LIB2 allowing SMT solver developers to select the best implementation with respect to their solving techniques. For instance, recent work [LM12] showed that in case of SAT solving a native representation of cardinality constraints [Mag12] can be utilized to speed-up the solving process. Moreover, expressing problems using native cardinality constraints instead of replacing them with logic constraints that encode their semantics compacts the size of the SMT instances and provides additional syntactic sugar for users. This simplifies the process of creating an SMT instance as well as making the created SMT instances human-readable.

Additionally, we observed that analysis techniques such as finding sets of contradicting clauses, e.g., as proposed in [LS08], benefit from our SMT logic allowing for a coarse-grained pinpointing

[2]The detailed logic description can be found on http://www.informatik.uni-bremen.de/agra/doc/misc/cardinality.smt2

of errors on the clause level. Compared to the coarse-grained analysis, finding contradictions in an SMT instance with several flattened BDDs can be much more time consuming and error prone.

We propose an open-source implementation[3] integrated into metaSMT [HFF+11] to allow for further experimentation. metaSMT[4] is a framework that provides an easy to use *Domain Specific Embedded Language* (DSEL) for C++ similar to SMT-LIB and supports multiple SAT and SMT solvers. We extend metaSMT with an SMT-LIB2 parser which additionally supports the novel logic of cardinality constraints. Currently, our implementation offers different logic encodings to lower cardinality constraints to Boolean primitives similar to miniSAT+ [ES06].

We claim that a standardized SMT logic for cardinality constraints enables the design of a benchmark set allowing for a better exploration of advantages and drawbacks of different logic encodings. Understanding the influence of SMT specific solving techniques like theory propagation have not been investigated so far. We consider this an interesting open research question and envision an SMT solver that automatically chooses the best logic encoding with respect to a given application for the future.

References

[Aav11] Aavani, Amir: *Translating pseudo-Boolean constraints into CNF*. In *Theory and Applications of Satisfiability Testing*, pages 357–359, 2011.

[ANORC11] Asin, Roberto, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodriguez-Carbonell: *Cardinality networks: A theoretical and empirical study*. Constraints, 16(2):195–221, 2011.

[BBR06] Bailleux, Olivier, Yacine Boufkhad, and Olivier Roussel: *A translation of pseudo-Boolean constraints to SAT*. Journal on Satisfiability, Boolean Modeling and Computation, 2(1–4):191–200, 2006.

[BSST09] Barrett, Clark, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli: *Handbook of Satisfiability*, chapter Satisfiability Modulo Theories, pages 825–885. IOS Press, 2009.

[BST12] Barrett, Clark, Aaron Stump, and Cesare Tinelli: *The SMT-LIB standard: Version 2.0*, 2012.

[Coo71] Cook, Stephen A.: *The complexity of theorem-proving procedures*. In *ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CZI10] Codish, Michael and Moshe Zazon-Ivry: *Pairwise cardinality networks*. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 154–172, 2010.

[ES06] Eén, Niklas and Niklas Sörensson: *Translating pseudo-Boolean constraints into SAT*. Journal on Satisfiability, Boolean Modeling and Computation, 2(1-4):1–26, 2006.

[3] https://github.com/agra-uni-bremen/metaSMT

[4] http://www.informatik.uni-bremen.de/agra/ger/metasmt.php

[HFF+11] Haedicke, Finn, Stefan Frehse, Görschwin Fey, Daniel Große, and Rolf Drechsler: *metaSMT: Focus on your application not on solver integration*. In *International Workshop on Design and Implementation of Formal Tools and Systems*, pages 22–29, 2011.

[Lev73] Levin, Leonid A.: *Universal search problems*. In *Problemy Peredaci Informacii 9*, pages 115–116, 1973. Translated in problems of Information Transmission 9, 265–266.

[LM12] Liffiton, Mark H. and Jordyn C. Maglalang: *A cardinality solver: More expressive constraints for free — (poster presentation)*. In *Theory and Applications of Satisfiability Testing*, pages 485–486, 2012.

[LS08] Liffiton, Mark H and Karem A Sakallah: *Algorithms for computing minimal unsatisfiable subsets of constraints*. Journal of Automated Reasoning, 40(1):1–33, 2008.

[Mag12] Maglalang, Jordyn C.: *Native cardinality constraints: More expressive, more efficient constraints*. Honors Projects, Paper 19, 2012.

[SWFD09] Sülflow, André, Robert Wille, Görschwin Fey, and Rolf Drechsler: *Evaluation of cardinality constraints on SMT-based debugging*. In *International Symposium on Multiple-Valued Logic*, pages 209–303, 2009.

[Tse68] Tseytin, Gregory S.: *On the complexity of derivation in propositional calculus*. Studies in Constructive Mathematics and Mathematical Logic, pages 115–125, 1968.