Controller Synthesis with Uninterpreted Functions

by

Georg Hofferek

A PhD Thesis
Presented to the Faculty of Computer Science and Biomedical Engineering,
Graz University of Technology (Austria),
in Partial Fulfillment of the Requirements for the PhD Degree

Assessors

Prof. Roderick Bloem (Graz University of Technology, Austria)
Prof. Jie-Hong Roland Jiang (National Taiwan University, Taiwan)

July 2014

TU Graz
Graz University of Technology

Institute for Applied Information Processing and Communications (IAIK)
Faculty of Computer Science and Biomedical Engineering
Graz University of Technology, Austria

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Mathison Turing

Abstract

Concurrency plays a crucial role in today's computing systems. For hardware systems, *pipelining* is an important and widespread mechanism to increase throughput. Such advantages, however, come at a price: concurrent systems require additional control to avoid issues such as inadequate synchronization, data races, or simultaneous access to shared resources. Controllers for concurrent systems are inherently difficult to implement and test. They are, however, rather easy to specify: the controller should make sure that the (externally visible) behavior of the concurrent system is equivalent to the behavior of a corresponding sequential reference system.

Using pipeline controllers as an example, we consider cases where the specification takes the form of a formula $\forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$, where $\overline{x}$ is a vector of first-order variables, representing the state and/or inputs of the system (on which the control signals may depend); $\overline{c}$ is a vector of Boolean variables, representing the signals issued by a controller; $\overline{x}'$ is a vector of auxiliary first-order variables; and Φ is a (quantifier-free) formula in a decidable first-order theory fragment. In particular, we use the theory of *uninterpreted functions* to abstract complex datapath elements of the system. This abstraction is crucial to reduce specifications to tractable sizes. Moreover, for correctness with respect to concurrency, it suffices to ensure that the same operations are executed on the same operands. Detailed semantics of single operations are irrelevant. Thus, uninterpreted functions are a very suitable method of abstraction in this setting.

By finding *certificates* for the variables in $\overline{c}$, we *synthesize a controller* that is correct-by-construction with respect to the specification. We present several ways to compute such certificates, in particular two methods based on *Craig interpolation*. The first one is an iterative method, that computes one certificate at a time. For n certificates, it requires n calls to an SMT solver, producing n refutation proofs. As an alternative, in our second approach we generalize Craig interpolation to what we call *n-interpolation*. It allows us to compute n coordinated interpolants from one single refutation proof. However, n-interpolation imposes two requirements on the proof: it must be *colorable* and *local-first*. We describe how a standard proof, obtained from an SMT solver, can be transformed to satisfy these requirements. Alternatively, we show how *modular SMT solving* can be used to directly obtain a colorable, local-first proof.

We have implemented the interpolation-based approaches in a prototype tool called SURAQ. Using this tool, we are able to synthesize a controller (with two Boolean control signals) for a DLX processor with a five-stage pipeline. The total time required for synthesis is approximately one hour and 15 minutes.

Acknowledgements

Today's research is not an individual but a group effort. Therefore, numerous people deserve to be thanked for their contribution and cooperation without which this thesis would not be what it is now. First of all, I would like to thank Roderick Bloem for being the best supervisor a PhD student can ever wish for. He always took ample time to discuss progress and setbacks with me, and provide me with his expertise and criticism. I learned a lot from and feel greatly indebted to him.

I also want to thank Jie-Hong Roland Jiang; first and foremost for agreeing to be the external reviewer for this thesis. In addition to that he was also great to work with, in particular during the preparation and execution of the QUAINT project, which was essential to this thesis.

I am also thankful to Ashutosh Gupta, who co-developed the concept of n-interpolation. He also provided different perspectives on issues I got stuck with, which greatly helped to get me unstuck.

The people who shared an office with me for some time during the creation of this thesis also deserve to be thanked: I am very thankful to Karin Greimel, who was very helpful in getting me acquainted to the field of formal methods. I also want to express my gratitude to Robert Könighofer, who was always available for fruitful discussions and for bouncing ideas and approaches back and forth.

Bettina Könighofer, Christoph Hillebold, and David Kofler all deserve to be thanked for their contribution to the implementation of SURAQ. Bettina helped a lot in refactoring parts of the code under lots of time pressure before a paper deadline. Christoph made formula objects immutable and unique. He also wrote the parser for VERIT proofs and helped in doing a first analysis of them. David implemented the replacement of equality predicates as part of his Bachelor's thesis.

Furthermore, I would like to mention Johannes Winter, who provided me with his vast expertise on debuggers, whenever I ran into tool-related problems.

Moreover, I thank Georg Weissenbacher, for the fruitful discussions on modular SMT solving, and Georg Schadler, for working on the corresponding implementation.

I am very grateful to Pascal Fontaine and Bruno Woltzenlogel Paleo for helping me with installing, understanding, and using their tools VERIT and SKEPTIK. Moreover, the meetings I had with them in Vienna, and also the (sometimes lengthy) email conversations were really productive and helpful.

I would also like to thank the Institute for Applied Information Processing and Communications (IAIK) for providing the best working and learning envi-

ronment[1] a PhD student can ask for. I always felt extremely happy to work here and to be part of this great team.

Last but not least, I want to thank my wife Maria for enduring all the ups and downs, all the progress and the delays of this thesis with me. I am grateful for all her patience and I apologize for sometimes making her wait so long when I was working long hours.

Funding. The research presented in this thesis was funded in part by the European Commission through projects COCONUT (FP7-2007-IST-1-217069) and DIAMOND (FP7-2009-IST-4-248613), by the Austrian Science Fund (FWF) through project QUAINT (I774-N23) and the national research network RiSE (S11406-N23), as well as by Graz University of Technology.

Georg Hofferek
Graz, July 2014

[1]Except for the building the institute resides in, which — among other issues — is an unbelievably ugly abomination, giving you the impression that the blueprints were recycled from a prison building project.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Glossary

array property fragment A decidable fragment of the theory of arrays that allows limited use of quantifiers [BMS06]. 6, 19

Burch-Dill equivalence An equivalence criterion for pipelined circuits based on the Burch-Dill paradigm. 30–32, 37, 40, 111, *see* Burch-Dill paradigm
Burch-Dill paradigm A way to formally specify correctnes of a pipelined processor, by showing that (1) flushing the pipeline followed by the execution of one instruction in the non-pipelined implementation and (2) performing one step in the pipeline followed by flushing it are commutative. This was proposed by Burch and Dill in [BD94]. 4–8, 30, 32, 35, 38, 111
Burch-Dill style 5, 6, 8, 13, 33, 40, 41, 100, 104, *see* Burch-Dill paradigm

chordal A graph is said to be chordal, if it contains no chord-free cycles of length greater then three. 23, 52
clause A disjunction of literals. 16, 17, 22, 25, 27–29, 64, 72, 73, 78–80, 85, 93
congruence closure A theory solver for $\mathcal{T}_U^{\mathrm{qf}}$. 25, 27, 86, 92
congruence graph A graph in which terms that belong to the same congruence class are connected by paths. 25, 70

fragment A syntactically restricted subset of all formulas in first-order logic, or a first-order theory. 4, 6, 17, 19, 20, 33–35, 41, 79
  conjunctive fragment A fragment that allows only formulas which are conjunctions of literals. 17, 24, 27
  quantifier-free fragment A fragment that does not allow the use of quantifiers. 4, 17, 18, 79

literal A (theory) atom, or its negation. 8, 16, 17, 23, 25–28, 54, 63, 65, 69–72, 74–80, 85–88, 94
local-first A property of resolution proofs that are used for interpolation. In a local-first proof, both premises of resolution steps with a local resolving literal are derived from the same partition. x, 8, 9, 58, 64, 66–68, 73, 74, 80, 81, 93, 103, 104, 106, 112, 113, 130

PSPACE The set of all decision problems that can be solved by a Turing machine using only a polynomial amount of space. 9, 44, 55, 56

QSAT The problem of deciding whether or not a quantified Boolean formula (without free variables) is satisfiable. 55, 56, 83–86, 89

refutation proof A formal proof that some formula is not satisfiable. 8, 27–30, 58, 62–64, 68, 74, 80, 92, 93, 104, 109, 112, 113
resolving literal A literal that occurs in opposite polarity in two clauses and is used to resolve them. 28, 64, 67, 68, 73

SAT The propositional satisfiability problem. 1, 2, 13, 21, 24, 25, 27, 61, 93, 103–105
Skeptik A proof compression tool. See `https://github.com/Paradoxika/Skeptik` and [BFW14]. 14
SMTLIB The Satisfiability Modulo Theories Library. See `http://smtlib.org/`. 90–92, 94
Surak A legendary Vulcan logician, philosopher, and scientist. Many consider him the father of the modern Vulcan civilization. 89, 114
Suraq $\underline{\text{S}}$ynthesizer using $\underline{\text{U}}$ninterpreted Functions, A$\underline{\text{r}}$rays $\underline{\text{a}}$nd E$\underline{\text{q}}$uality. 8, 9, 13, 14, 58, 89–94, 96, 97, 102, 104, 106, 112, 129, 130

theory lemma A formula (within this thesis: usually a clause) that can be proven from the axioms of the theory in question. 13, 25–30, 64, 65, 68–73, 79, 80, 92, 93, 102–106
theory solver An algorithm to decide satisfiability of the conjunctive fragment of a theory. 24, 25
transitivity-congruence chain A path in a congruence graph. 26, 70–72
Tseitin variable A variable introduced during Tseitin's encoding. 17, 94
Tseitin's encoding A procedure to transform a formula with arbitrary structure into an equisatisfiable conjunctive normal form, by introducing a linear number of additional propositional variables [Tse68]. 17, 24, 78, 79, 91, 92

veriT A proof-producing, open-source SMT solver for $\mathcal{T}_U^{\text{qf}}$ See `http://www.verit-solver.org/` and [BdODF09]. 14

Acronyms

ALU Arithmetic Logic Unit. 2

BDD Binary Decision Diagram. 52–54, 58, 83, 84

CNF Conjunctive Normal Form. A conjunction of clauses. 16, 24, 25, 27–29, 63, 91, 92

DAG Directed Acyclic Graph. 28, 91, 94, 95, 98, 104, 105, 113

LTL Linear Temporal Logic. 11

QBF Quantified Boolean Formula. 8, 9, 56, 84–86

SMT Satisfiability Modulo Theories. 1, 2, 7–9, 13, 16, 19, 20, 27, 28, 34, 58, 61, 62, 67, 69, 70, 74–80, 83, 85, 88, 90, 92–94, 102–104, 106, 109, 112, 113, 130

Notation

Notation	Description	Page List
$AC(\phi)$	The set of all constraints obtained by applying the index set construction [BM07] to ϕ.	22, 23, 47, 48, 55
$args(f)$	The set of all terms (or tuples of terms) used as arguments for function f in a formula.	22
$\mathsf{A}[i]$	The value of array A at index i (“array-read” function).	18, 19, 21, 34, 38, 39, 47, 48
$\mathsf{A}\langle i \lhd j\rangle$	The array A after writing j to index i (“array-write” function).	18, 19, 21, 34, 38, 39, 47
$a := b$	Denotes assignment of value b to a. The assignment symbol “:=” will be used instead of “=”, whenever necessary to avoid confusion with the equality predicate.	16, 19, 34, 37, 39, 53, 79
$\mathbb{B}$	The set of Boolean values: $\{\top, \bot\}$.	16, 17, 34
$CC(\phi)$	The set of all congruence constraints obtained by performing Ackermann's reduction on ϕ.	22, 23, 49, 50
$clause(n)$	The clause of proof node n.	28–30, 63
$C\vert_\phi$	The clause C with all literals that contain symbols that do not occur in ϕ removed. That is, $C\vert_\phi = \bigvee_{l \in Lits(C),\, l \preceq \phi} l$.	17, 63
$\mathbb{D}$	*Domain of discourse*; a possibibly infinite set of elements which are the subjects of statements in first-order logic.	16, 17, 34

Notation	**Description**	**Page List**
expand_negate(Φ)	The expansion and negation of a formula Φ as shown in the proof of Theorem 4.	45, 59–63
$\mathcal{I}$	Index set (in the context of array properties).	21, 47, 48, 55
λ	A term that represents “any other index” during the reduction of array properties to $\mathcal{T}_U^{\text{qf}}$.	21, 47, 48, 55
$Lits(\phi)$	The set of literals occurring in ϕ.	xxiii, 16, 17, 26, 74
$[\![\cdot]\!]_{\mathcal{M}}$	The value of “$\cdot$” under model $\mathcal{M}$. The subscript $\mathcal{M}$ is dropped, if the model is clear from context.	17, 46–52, 62
no_array(ϕ)	The formula obtained from $\mathcal{T}_A^P$-formula ϕ by removing array-write expressions, replacing universal quantifiers in array properties with finite conjunctions over the index set and replacing all array-read expressions with fresh uninterpreted function instances.	22, 23, 47, 48
no_equal(ϕ)	The formula obtained from $\mathcal{T}_E^{\text{qf}}$-formula ϕ by replacing all equality atoms with fresh propositional variables. Symmetric atoms $a = b$ and $b = a$ are replaced by the same propositional variable. Reflexive equalities $a = a$ are replaced by $\top$.	23
no_func(ϕ)	The formula obtained from $\mathcal{T}_U^{\text{qf}}$-formula ϕ by replacing all uninterpreted function (predicate, respectively) instances with fresh domain (propositional, respectively) variables.	22, 23, 49, 50
$a \rightsquigarrow b$	A path from a to b in a congruence graph.	xxiv, 26
$\phi \vdash \psi$	Denotes that ϕ entails ψ. That is, ψ is a logical consequence of ϕ. If used with a subscript $\phi \vdash_{\mathcal{T}} \psi$, the entailment holds within theory $\mathcal{T}$. In particular $\vdash_{\mathcal{T}} \psi$ is used to denote that ψ is a lemma of $\mathcal{T}$.	25, 27, 30, 66
rule(n)	The proof rule of proof node n.	28

Notation	**Description**	**Page List**
$\mathcal{S}$	A fragment of $\mathcal{T}_A$, following the grammar given in Definition 20 (page 34).	34, 35, 37, 45, 90
$skel(\phi)$	The propositional skeleton of ϕ.	20, 24, 25, 51, 52, 85, 86
$\mathcal{S}^{\mathrm{Q}}$	The specification language used for controller synthesis. Corresponds to a closed formula $\forall\overline{x}\,.\,\exists\overline{c}\,.\,\forall\overline{x}'\,.\,\Phi$, where Φ is a formula in $\mathcal{S}$.	x, 35, 38, 41, 43–46, 52, 55–58, 84, 87, 90, 97, 114
$\mathcal{S}^{\mathrm{Q}^+}$	A generalized version of $\mathcal{S}^{\mathrm{Q}}$ that allows an arbitrary number of quantifier alternations, as long as all existential quantifiers are over Boolean variables only.	35, 44, 45, 55, 56, 58
$symb(\phi)$	The set of non-logical symbols (variables, function, and predicate symbols) occuring in ϕ.	17, 30, 59, 62
$\phi \preceq \psi$	Denotes that $symb(\phi) \subseteq symb(\psi)$.	17
$\mathcal{T}_A^{\mathrm{p}}$	Array property fragment of the theory of arrays.	19, 21–23, 35, 45, 48, 94
$\mathcal{T}_A$	Theory of arrays.	18, 19, 21, 33, 34, 41
$TC(\phi)$	The set of all transitivity constraints obtained by applying the graph-based reduction [BV00] to ϕ.	23, 51
$\mathcal{T}_E$	Theory of equality.	x, 18, 46, 49, 50
$\mathcal{T}_E^{\mathrm{qf}}$	Quantifier-free fragment of the theory of equality.	18, 21–23, 51
$\mathcal{T}_U$	Theory of uninterpreted functions and equality.	x, 17, 18, 46, 47, 49, 58, 70, 79, 88

Notation	**Description**	**Page List**
$\mathcal{T}_U^{\mathrm{qf}}$	Quantifier-free fragment of the theory of uninterpreted functions and equality.	18, 21–23, 25–27, 49, 58–60, 62, 79, 91, 93
$vars(\phi)$	The set of all variables occuring in ϕ.	35, 46

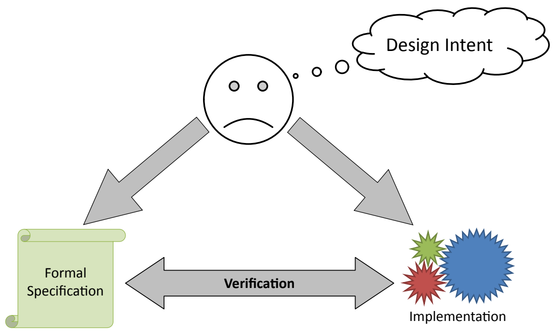
1 Introduction

1.1 Background and Motivation

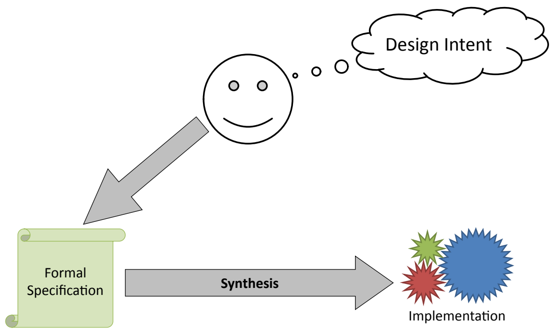
Today's computing systems are getting increasingly complex, while at the same time their role in our society becomes more and more important. In fact, lives may depend on the correct operation of digital systems embedded in things like cars, aircraft, and critical medical care equipment. In critical areas, testing alone may not be sufficient to ensure correctness of a system. Formal verification is one way to alleviate the problem. A verification procedure produces a proof for the fact that a system conforms to a given specification. Obviously, an (informal) specification in natural language is not suited for formal verification. Instead, a precise and unambiguous formalism is needed to encode the specification. Thus, the need for formal verification caused a need for formal specifications of systems. Once such specifications are available, a natural question is why one actually has to bother doing implementations manually. After all, formal specifications should, ideally, already contain all the information to automatically synthesize correct-by-construction implementations. (See Figure 1.1.)

The idea of synthesis was first proposed by Alonzo Church [Chu62] in the early 1960s. Unfortunately, it received only little attention for several decades because of the high computational complexity. However, in recent years, several advances were made by imposing some limitations on the specifications, or by considering only parts of systems, instead of entire systems at once. Moreover, the last decade has brought forward great improvements in the field of SAT and Satisfiability Modulo Theories (SMT) solving. This development was both fostered and witnessed by corresponding competitions: The SAT competition[2]

[2]`http://www.satcompetition.org/`



(a) Normal design setting.



(b) Correct-by-construction synthesis setting.

Figure 1.1: In a normal setting, a user has to create two realizations of the design intent: a formal specification, and an actual implementation. These can then be formally verified against each other. Correct-by-construction synthesis from a formal specification eliminates the need for both manual implementation and verification.

and the SMT competition.[3] Modern solvers are capable to solve many large benchmarks, despite the NP-complete worst-case complexity of the underlying decision problem. As SAT and SMT solving often can act as the underlying decision procedure for a synthesis algorithm, synthesis has greatly benefitted from these improvements as well. Nevertheless, scalability certainly remains an issue for synthesis.

One way to deal with scalability issues is to consider only certain parts of a system for synthesis, while other parts are still implemented manually. We observe that many systems actually consist of two types of components. One type is data-oriented components, also called the datapath of a system. In hardware, that would be components like adders, multipliers, or even complete Arithmetic Logic Units (ALUs), as well as memory and register files. In software, that would

[3]`http://smtcomp.sourceforge.net/`

be (side-effect-free) functions, which compute complex arithmetic operations, cryptographic algorithms, or similar things. Such data-oriented components are usually comparatively easy to implement and test manually, while they are often rather hard to formally specify. For example, consider a 64 bits hardware multiplier. Due to its regular internal structure, it is rather easy to design such a circuit. Once available, it is also rather easy to test. If it works correctly for the few known corner cases (one or both inputs 0, 1, $2^{64}-1$, respectively), plus a few randomly chosen test cases, we have very high confidence that the multiplier circuit is correct. However, giving a formal specification for how all 128 output bits depend on the 128 input bits is a difficult, for practical purposes intractable task.

The second type of components is control-oriented components. In both software and hardware systems, controllers make sure that the data operations are executed in the correct sequence to achieve the overall goal of the computation. Controllers are often more difficult to implement, compared to datapath components. They are also more difficult to test. Certain errors might only be triggered in very special scenarios, when executing a particular sequence of actions, or when using some particular data values. In contrast to datapath elements, there is no internal regularity, so random testing will not help to build trust in correctness either. Yet, the properties that a controller has to ensure are in many cases rather simple to formally specify. Properties like *"Every request will eventually be answered"*, or *"No two simultaneous accesses to this resource may be granted"* map quite nicely to common specification formalisms.

Based on this observation, we would like to have "the best of both worlds". Ideally, we want to have a mixed imperative-declarative paradigm, where we can implement the parts that are easy to implement (and hard to specify), while only specifying the parts that are easy to specify (and hard to implement). The question is how to combine the two parts? One one hand the declarative part will — in general — have to reference datapath elements in order to express the desired properties. But on the other hand, we do not want to spell out all details of those elements. Otherwise, the main advantage of the mixed paradigm would be lost. Instead, the specification will only consider an abstraction of the datapath elements. This abstraction must be detailed enough to facilitate reasoning about correctness, yet abstract enough to be of tractable size. We will show that *uninterpreted functions* can, in many practical cases, be used as such an adequate mean of abstraction. Uninterpreted functions are particularly suited in concurrency settings. Simply put, when doing several operations concurrently, correctness should not depend on the semantics of single operations. Instead, it should suffice to ensure that the same operations are executed on the same data in the same order. Uninterpreted functions enable modeling such properties at minimum cost, as they are not axiomatized in any way, beyond functional consistency.

In hardware, *pipelining* is a widespread technique to achieve concurrency and thus increase throughput. As we will illustrate in Section 1.2, implementing pipeline controllers correctly is extremely difficult. The specification, however, is as simple as stating that *"the (externally visible) behavior of the pipelined system should be the same as the one of a non-pipelined reference system"*. We will present more details on how this property can be formalized in Sections 2.4 and 3.2.

1.2 Problem Description

In its most general form, the main problem we tackle in this thesis assumes the following setting. We have a system whose correctness can be stated in a decidable (fragment of a) first-order theory. We require that there exists a procedure to compute Craig interpolants (see Section 2.3) for this fragment/theory, such that the interpolants are again in the same fragment/theory.[4] Moreover, we assume that some Boolean control signals are not implemented in the system, but should be synthesized instead. We will show how to synthesize these control signals such that the system becomes correct with respect to the correctness criterion that has been stated. To make it more clear, we illustrate this concept with a concrete example, which has been the driving example for this research: Synthesis of a controller for a pipelined microprocessor.

Pipelining a microprocessor, unfortunately, can usually not be achieved by simply inserting pipeline registers at relevant points in the design. The reason is that the non-pipelined reference design that one would start from usually relies on the assumption that whenever an instruction is about to be executed, all previous instructions have completed. This assumption no longer holds for pipelined designs. In a pipeline, one instruction might depend on data that is not yet available, because the previous instruction, which computes this data, is still in the pipeline. An illustrative example can be seen in Figure 1.2. Thus, in addition to the insertion of pipeline registers, a controller is required to make sure that the pipelined processor operates as intended.

Burch and Dill [BD94] showed how to formally verify that a given controller is correct. Their work is based on three important concepts that we will also build upon. First, they used an implicit specification: when starting with the same initial memory content, the pipelined and the non-pipelined implementation produce the same final memory content, when (completely) executing an arbitrary program. Second, they showed how this implicit specification can be turned into a formula, showing that flushing the pipeline followed by the execution of one instruction in the non-pipelined implementation and performing one step in the pipeline followed by flushing it is commutative. We will henceforth refer to this way of specifying pipeline correctness as the *Burch-Dill paradigm*. A more detailed explanation of the Burch-Dill paradigm will be given in Section 2.4. Third, Burch and Dill already realized that a bit-level description of the datapath components in a microprocessor would be intractably large. Thus, they used uninterpreted functions for abstraction.

[4] We will focus on the quantifier-free fragment of the theory of uninterpreted functions and equality, for which this property holds.

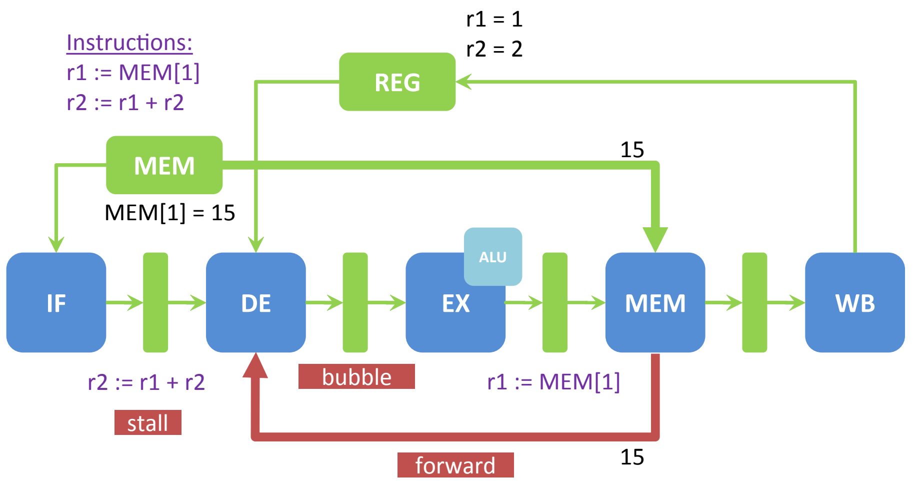


Figure 1.2: Example of Pipeline Concurrency Issues. When a five-stage pipelined processor executes the two statements `r1:=MEM[1]; r2:=r1+r2`, the following concurrency issues occur. At some point, the first instruction `r1:=MEM[1]` has been decoded, but not yet executed, while the second instruction `r2:=r1+r2` has just been fetched and is waiting to be decoded. Since decoding the second instruction requires the value of `r1` that results from executing `r1:=MEM[1]`, the pipeline has to be *stalled*, and a *bubble* (that is, an "empty" instruction) has to be inserted, while the first instruction continues execution. When the first instruction reaches the memory access stage, the new value for `r1` becomes available. This value must now be *forwarded* to the decode stage, to avoid having to wait until write-back to the register file has completed.

We build upon the work of Burch and Dill [BD94] and lift it from verification to a synthesis setting. That is, we assume that the implementation of certain (Boolean) control signals, such as `stall` or `forward` (see Figure 1.2) is not known and should be automatically synthesized so that the pipelined implementation becomes correct with respect to the Burch-Dill paradigm. We create a quantified formula that basically expresses that *for every possible state of the system, there exist values for the control signals, such that for all values of auxiliary variables (required to formulate the correctness criterion), the correctness criterion holds.* Formally, we have

$$\forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi, \tag{1.1}$$

where $\overline{x}$ is a vector of variables describing the state of the pipeline (and memory), $\overline{c}$ is a vector of (Boolean) control signals that should be synthesized, and $\overline{x}'$ are auxiliary variables, necessary to formulate the Burch-Dill style specification Φ. Note that Φ is a formula in a first-order theory. The variables in $\overline{x}$ and $\overline{x}'$ range over the (uninterpreted) first-order domain, whereas the variables in $\overline{c}$ are Boolean. Based on such a quantified formula, synthesis of a controller reduces to finding *certificates* for the existentially quantified control signals. Certificates

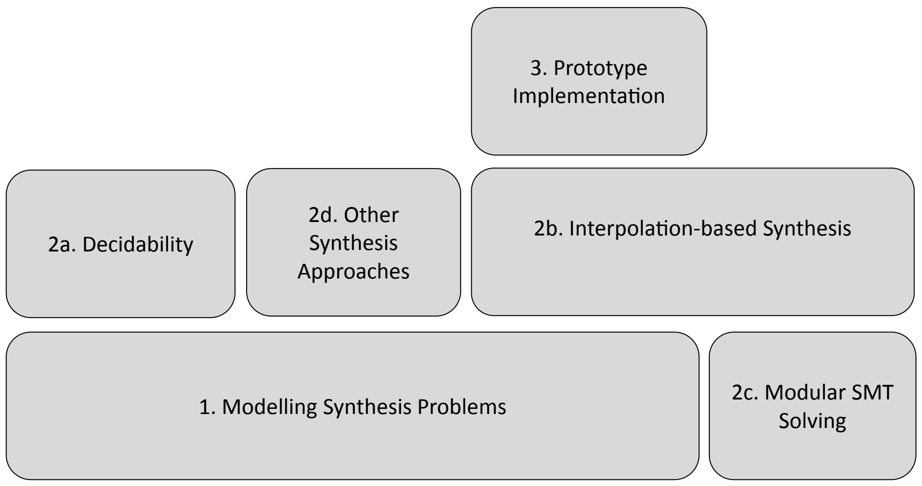

3. Prototype Implementation
2a. Decidability
2d. Other Synthesis Approaches
2b. Interpolation-based Synthesis
1. Modelling Synthesis Problems
2c. Modular SMT Solving

Figure 1.3: Contribution Dependence. The different synthesis approaches (2a, 2b, 2d) build on how synthesis problems are modeled as quantified formulas. Interpolation-based synthesis (2b) is also (partially) based on modular SMT solving (2c). Our prototype implementation (3) is based on interpolation, but not (yet) on modular SMT solving.

are functions that compute the values for variables in $\overline{c}$, based on values for the variables in $\overline{x}$. For pipelined processors, we will use the theory of uninterpreted functions and equality as well as the array property fragment of the theory of arrays to formulate Φ. However, our main approach to compute certificates (see Chapter 5) is not limited to these theories. It can be generalized to any decidable theory (or fragment) for which a suitable interpolation procedure exists.

1.3 Contribution

This section will describe the contribution of this thesis beyond the state of the art. In summary, there are three main contributions: First, a way to properly state a controller synthesis problem as a quantified, Burch-Dill style formula; second, several methods to compute certificates from such formulas; and third, a prototype tool implementing one of the more promising approaches. The rest of this section will describe those contributions in more detail.

1.3.1 Contribution 1 — Stating the Synthesis Problem

The fundament of every synthesis problem is the formal specification that describes what should be synthesized. Thus, the first contribution of this thesis is to show how the Burch-Dill paradigm can be adapted and extended to serve as the specification for our controller synthesis problems. Two main adaptions are necessary to the standard Burch-Dill paradigm. First, we introduce alternating quantifiers to account for the yet unknown control signals. However, by having the existential quantifier over Boolean signals only, we maintain decidability.

Second, the Burch-Dill paradigm is based on describing a *flush*-operation of the pipeline. Such an operation may take several time steps, and thus the formal description refers to several instances of signals and values in the design, each belonging to a particular time step. This is fine in verification, where the entire design (and thus all signals and values) are known. However, in synthesis, the control signals $\bar{c}$ are not known at the time of writing the specification. Thus, referring to multiple instances of them is problematic. Somehow, the specification would also have to ensure that each instance is functionally dependent on all the state variables corresponding to the same time step in the same way. This is problematic, especially with a linear quantifier structure. We show how to work around this problem by using *completion* [HGS03] instead of flushing. More details on this issue will be presented in Section 3.2.

1.3.2 Contribution 2 — Computing Certificates

Based on specifications with the structure shown in Equation 1.1, the main contribution of this thesis is to show ways to compute certificates for the control variables $\bar{c}$. Several different approaches, with different underlying decision procedures have been investigated.

Contribution 2a. The first and rather naive approach for certificate computation is to adapt standard eager-encoding techniques from SMT solving, to reduce our problem to the purely propositional domain. On the propositional level, the specification becomes the characteristic function of a relation, describing possible input/output values. Certificates can then be computed with arbitrary symbolic relation determinization techniques, such as for example a simple cofactor-based approach.

This approach does not scale very well, mainly because the number of transitivity constraints that are needed for a reduction from equality logic to propositional logic is extremely large. The main rationale for investigating this approach was to establish decidability of the problem in an early phase of the research. More details will be presented in Chapter 4.

Contribution 2b. Craig interpolation [Cra57] is one way to symbolically determinize Boolean relations [JLH09]. We improve this method in two ways. First, we show that the reduction to the Boolean level is not necessary and interpolation can be done directly on the theory-level. Second, we show that instead of the standard iterative approach, where one interpolant (corresponding to the implementation of one control signal) is computed at a time and back-substituted into the specification, we can compute multiple coordinated interpolants from a single proof. We will refer to this generalized interpolation method as *n-interpolation*. Details will be presented in Chapter 5.

Contribution 2c. The n-interpolation algorithm mentioned above requires a refutation proof with some additional properties. In particular, it must be *local-first*, meaning that whenever resolution over a literal that is local to one of the partitions over which we interpolate is done, both premises must be derived from the same partition. We show that *modular SMT solving*, an extension of modular SAT solving [BVB+13], can be used to readily obtain a proof with this property. Using tree-shaped dependencies, we can also use modular solving to extend our n-interpolation-based certificate computation to formulas with more than one quantifier alternation, as long as all existential quantifiers are over Boolean variables only:

$$\forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \exists \overline{c}' \,.\, \forall \overline{x}'' \,.\, \ldots \; \Phi. \tag{1.2}$$

This will be presented in Section 5.4.

Contribution 2d. Even though n-interpolation is very promising, we investigate some *alternative approaches* as well, for comparison. One such approach is to reduce our problem to Quantified Boolean Formula (QBF) solving. Naively, this could be done by eagerly encoding theory constraints. An alternative to the eager approach is to add theory constraints lazily, only when needed. A completely different approach is based on the observation that for many practical synthesis problems small and simple solutions do exist. Thus, we "guess" such a solution and try to verify it. If verification fails, we refine the guess based on the counterexample obtained from the verification step. All these alternative approaches will be described in more detail in Chapter 6.

1.3.3 Contribution 3 — Prototype Tool

We have implemented the n-interpolation-based approach in a prototype tool called SURAQ. To the best of our knowledge, this is the first tool that supports correct-by-construction synthesis based on uninterpreted functions as underlying abstraction method. SURAQ was implemented in Java, and consists of roughly 35,000 lines of code. More details on its implementation and experimental results achieved with it can be found in Chapter 7.

1.4 Outline of this Thesis

The structure of this thesis is shown in Figure 1.4. After this introductory chapter, Chapter 2 will revisit the theoretical background on which our research is based. We will also give important definitions and establish notation conventions.

In Chapter 3, we will show how to create a Burch-Dill style specification for a pipelined processor, so that we can use it for controller synthesis. We will highlight how we adapted and extended the Burch-Dill paradigm and why these modifications were necessary.

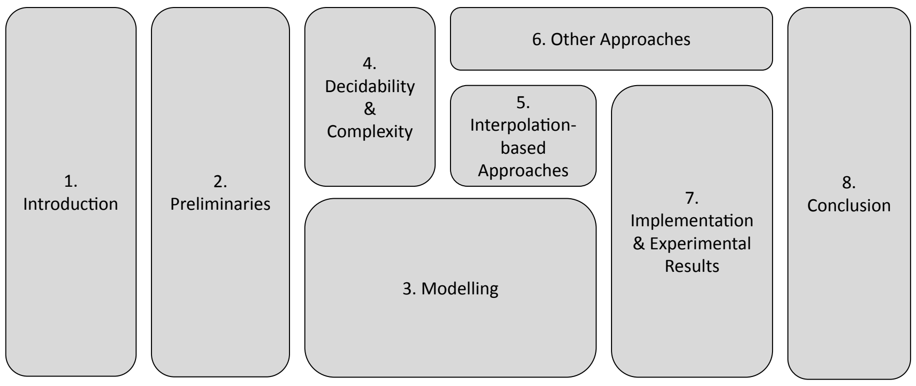

1. Introduction
2. Preliminaries
4. Decidability & Complexity
6. Other Approaches
5. Interpolation-based Approaches
3. Modelling
7. Implementation & Experimental Results
8. Conclusion

Figure 1.4: Chapter Dependence. After introducing the research problem (1), we present necessary preliminaries (2). Building on those, we show how to model synthesis problems with our specification language (3). Moreover, we discuss the decidability and computational complexity of our synthesis paradigm (4). Based on these, we present concrete approaches (5, 6) to solve the synthesis problems, with a focus on interpolation-based approaches (5). Our implementation is also interpolation-based, and the corresponding experimental results are presented (7). Finally, we give a summary and conclusion (8).
Note that the size of the boxes above is *not* intended to correlate with importance or length of any of the chapters.

In Chapter 4, we will show that the problem we consider is decidable in the form in which we stated it. Furthermore we will discuss computational complexity, and show that (the generalized version of) our problem is PSPACE-complete. In order to prove PSPACE-completeness, we will show that we can transform our synthesis problem into a QBF problem, and vice versa.

Chapter 5 is dedicated to interpolation-based approaches for certificate computation. We will first show an iterative approach based on resubstitution. Next, we will present a generalized interpolation scheme called n-interpolation, with which we can compute multiple coordinated interpolants from a single proof. As n-interpolation requires proofs to be local-first, we present two ways to obtain such proofs. The first way is to rewrite a proof obtained from a normal SMT solver. A second approach is to use modular SMT solving.

Alternative approaches for certificate computation, in particular an idea based on lazily encoding theory constraints into a QBF problem, will be discussed in Chapter 6.

Our prototype synthesis tool Suraq, which is based on the n-interpolation approach will be discussed in Chapter 7. This chapter will also give our experimental evaluation of the synthesis flow.

Chapter 8 will summarize our work, point out the most important conclusions, and talk about potential future work.

1.5 Related Work

Declaration of Sources

This section is based on and reuses material from the following sources, previously published by the author:

- [HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42.
- [HGK$^+$13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In Jobstmann and Ray [JR13], pages 77–84.

References to these sources are not always made explicit.

Research on automated synthesis has prospered significantly over the last decade. Before we talk about recent achievements made with respect to synthesis, let us briefly discuss some often neglected correlated aspects of synthesis settings in general. First of all, synthesis removes redundancy from the design process. Instead of creating a formal specification *and* an implementation, the user just creates the specification (see Figure 1.1). While this can save time and effort, it can also be a source of errors. With a manual implementation process, the same mistake would have to happen in the implementation and in the specification in order for the error to remain unnoticed. In a synthesis setting, an error in the specification will lead to a faulty implementation that is not directly detected. As it is not clear whether writing correct specifications is any easier than writing correct implementations, the main underlying problem has just been shifted to a different domain. This issue has so far not really been addressed in literature. Some closely related problems are considered by Könighofer et al. [KHB09, KHB10, KHB13]. They present methods to “debug” formal specifications. On one side, they consider incomplete specifications, that is, specifications that miss some important properties and thus do not constrain the synthesized systems enough. This is addressed by simulating a generalized version of the synthesized system, and letting the user decide whether or not the simulation exhibits undesired behavior. On the other hand, they consider over-constrained specifications, which are unrealizable. This is addressed by presenting and simulating a counterstrategy, which demonstrates the cause of unrealizability. Fixing unrealizable specifications is also the focus of [LDS11] and [AMT13], who each present methods to automatically find assumptions on the environment which fix the problem of unrealizability.

Apart from the completeness and correctness of specifications, there is another issue with synthesis. In most design settings, there are side conditions to the primary goal of implementing a correct system. These are often of a quantitative nature. For example, the final system should be as “small” as possible, with respect to some metric such as lines of code, or number of logic gates. Bounded synthesis [SF07, FS13] is one way to address this issue. It inherently always finds the system with the smallest possible state space. In symbolic synthesis algorithms, small results can be achieved by optimizing the determinization of the winning strategy. A good overview of different methods to do that is given in Section 2 of [EKH12]. There are also other quantitative properties of systems, apart from their size. For example, one would prefer a system that reacts to a request as fast as possible, compared to a system that just waits for some time before reacting — even if the specification only requires that a reaction happens eventually. Quantitative properties of this type have been addressed in [BCHJ09]. Another important property of a system is its robustness against failures of the environment. From a mathematical perspective, the behavior of a system after the environment has made an error is not specified and can thus be arbitrary. However, a system that tries to recover from the error in some way is clearly preferable over a system that just stops operating after the error, even though both are correct with respect to the specification. A comprehensive summary on how to synthesize robust systems is given in [BCG$^+$14].

After this brief overview of side aspects to synthesis in general, let us look at actual synthesis methods. One important branch of synthesis is concerned with synthesis of reactive modules from temporal logic formulas. In the late 1980s, Pnueli and Rosner [PR89] established a doubly exponential lower bound for synthesis from a specification given in Linear Temporal Logic (LTL). Thus, this kind of synthesis was deemed intractable for examples of interesting size, until Piterman et al. [PPS06] showed that by restricting the specification language, the bound could be brought down significantly. Bloem et al. [BGJ$^+$07a] have subsequently shown that this restricted language can effectively be used to synthesize examples of industrial scale. Following that, there has been a proliferation of approaches for synthesis from temporal logic specifications. In fact, there are too many to cite them all. To give just a few examples, consider [JGWB07, SF07, FJR09, SS09, SS13, MS10, FJ12, JB12, KJB13]. Although quite successful, these approaches have only limited applicability to the controller synthesis problems we consider. The reason for that is the lack of abstraction. LTL and similar temporal logic formalisms are based on propositional primitives. Thus, they reason on a bit-precise level. Using such a formalism in our setting means that all system parts that are already implemented would have to be described on a bit-precise level. In many cases, this is not feasible. For example, multipliers — which are common datapath elements — have an exponentially large bit-precise description. Thus, for sufficiently large bit-widths, modeling them on a bit-precise level is intractable.

To cope with this problem, there has been work on how to use abstraction in synthesis settings. Synthesis from temporal logic is often reduced to solving infinite games on finite graphs. The work by de Alfaro et al. [dAR10] shows how to use a three-valued abstraction-refinement to simplify large games. While this might improve the procedures for solving the game, it does not help with respect to the problem of specifying complex implementations on a bit-precise level in a mixed imperative-declarative paradigm. One setting that is based on this paradigm is program repair [JGB05], which aims at replacing faulty parts of programs with synthesized components, such that the overall program becomes correct. Thus, the non-faulty part of the program obviously must be part of the specification for synthesis. In this setting predicate abstraction has been used [GBC06] to avoid having to formalize the program fully on a bit-precise level. Predicate abstraction is also used by Vechev et al. [VYY10], who try to automatically insert synchronization statements into concurrent programs — a problem originally described by Clarke and Emmerson [EC82]. Their work is similar to ours with respect to the fact that they, too, assume that the actual data computation has been implemented, and only the concurrency aspects must be synthesized. Their approach uses a predicate-abstraction-based refinement loop, but instead of just refining the abstraction in each iteration, they also give the complementary option of modifying the program by inserting additional atomic sections. Predicate abstraction, however, is not equality-preserving, unless the abstraction is refined to a level where it is isomorphic to the original concrete instance. This makes predicate abstraction inapplicable in synthesis settings where the specification is based on stating equality between the result computed by a concurrent system and the corresponding result of a reference system. In contrast to this, our abstraction with uninterpreted functions preserves equalities, in many cases. This has been exploited in verification settings; for example, by Burch and Dill [BD94]. However, to the best of our knowledge, we are the first to suggest the use of uninterpreted functions for abstraction in a synthesis setting.

Another approach that employs a mixed imperative-declarative paradigm is *program sketching* [SL13]. Here, the main idea is that the user can leave "holes" in the program, which are filled by the synthesizer. More formally, one can write "`??`" instead of an integer or Boolean constant. The synthesizer will then find constants such that the program satisfies all assertions. To synthesize non-constant expressions, the user must provide a template. For example, to synthesize an expression that linearly depends on a variable `x`, one can write "`?? * x + ??`". Synthesis itself is done by a counterexample-guided refinement loop. In principle, program sketching is quite similar to our work. In both approaches, there are some unknowns — which should be synthesized — in an otherwise completely implemented system. Two key differences are that program sketching can synthesize integer values, whereas we only synthesize Boolean control signals. On the other hand, we synthesize Boolean functions, whereas program sketching only synthesizes constants — unless the user provides a template for a non-constant expression, as illustrated in the example above.

Some work that is rather orthogonal to our own is *functional synthesis* by Kuncak et al. [KMPS10, KMPS13]. Whereas in our setting we assume that data operations are easy to implement, and thus focus on synthesizing control logic, Kuncak et al. focus on synthesizing functions that compute data, based on specifications about the input-output behavior. Their approach is also focused on software and thus supports unbounded data types, for example numbers and data structures. Overall, their work nicely complements our own.

Nurvitadhi et al. [NHKL10, NHKL11] have done work based on the same motivating application. They, too, present a method to synthesize a pipeline controller. Their approach is, however, quite different from ours. They perform data-hazard analysis and resolution, while we start from a logic specification, a Burch-Dill style verification condition, which we use for correct-by-construction synthesis. Thus, we have formal proof that our synthesis result satisfies the original specification. Besides the fundamental internal differences, there are also differences from a user's point of view. For example, we do require a complete datapath for the pipeline, including (potential) forwarding paths. On the other hand, our approach does not need manually implemented `read-enable` and `write-enable` signals, nor do we impose the restrictions of [NHKL11] on the structure of the write interface of the pipeline. Furthermore, even though pipeline controllers are the main motivation for our work, our approach is more general and also extends to other controller synthesis problems, where the specification can be stated as a formula like Equation 1.1 or Equation 1.2.

Concerning the internals of our synthesis approach(es), there is a lot of recent work we rely on and adapt where necessary. Interpolation as a mean of certificate extraction was proposed by Jiang et al. [JLH09]. They consider formulas of the form $\forall \overline{a} \,.\, \exists \overline{b} \,.\, \Phi$, where $\overline{a}$ and $\overline{b}$ are vectors of propositional variables, and Φ is a propositional formula. We extend their work in three ways. First, we allow first-order theory formulas and universally quantified first-order variables. Second, we allow an inner universal quantifier, which binds auxiliary variables on which the certificates for the existentially quantified propositional variables may not depend. Third, we also present a method to compute multiple interpolants from a single proof, whereas [JLH09] proposes only an iterative method, based on resubstitution. Similar iterative resubstitution has also been used in BDD-based temporal logic synthesis approaches [BGJ+07a] and in functional synthesis [KMPS10].

Interpolation in the theory of uninterpreted functions and equality has been described my McMillan [McM05], and optimized by Fuchs et al. [FGG+09]. We generalize their techniques to settings with more than two colors to obtain our colorable proofs. Other recent work on colorable proofs includes [KV09] and [HKV12].

The concept of modular SMT-solving is based on combining ideas from McMillan [McM11] and Bayless et al. [BVB+13]. McMillan [McM11] showed how colorable proofs can be obtained efficiently, by applying interpolation only to non-colorable theory lemmata. Bayless et al. [BVB+13] describe how modular SAT solving works. We propose to combine these ideas, resulting in a modular SMT solver that produces colorable proofs.

Our prototype tool SURAQ relies on a proof-producing SMT solver as underlying decision procedure. For proof production, the current implementation uses the VERIT solver [BdODF09]. For various other tasks (equivalence checks, formula simplification, etc.), SURAQ also uses Z3 [dMB08]. To optimize our synthesis results, we also tried to compress proofs — as smaller proofs yield smaller interpolants. For proof compression, we used the tool SKEPTIK [BFW14], which can read VERIT proofs and compress them with several different algorithms [FMW11, BW13].

2 Preliminaries

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- [HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42.
- [HGK$^+$13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In Jobstmann and Ray [JR13], pages 77–84.

References to these sources are not always made explicit. In particular, definitions and notational conventions are strongly based on the papers listed above. In general, notation largely follows conventions established in literature — [BM07] and [KS08] in particular.

In this section, we will revisit some theoretical background on which our work builds. We assume that the reader is already familiar with first-order logic and theories (in particular, the theories of uninterpreted functions and equality, and the theory of arrays). Thus, we will only briefly recapitulate some important definitions, in order to avoid ambiguities and establish a consistent notation. For a thorough introduction into first-order theories and decision procedures, the reader is referred to [KS08] and [BM07]. In the remainder of this chapter, we will first introduce first-order theories, in particular the *theory of uninterpreted functions and equality*, and the *theory of arrays*, which we will both need to write specifications for pipelined processors in Chapter 3. Next, we will discuss SMT solving, as it is the underlying decision procedure for our synthesis approach presented in Chapter 5. We will also briefly recapitulate *Craig interpolation* and state some relevant definitions, as it is the basis for our n-interpolation-based approach for certificate extraction, which is also presented in Chapter 5.

2.1 Theories in First-Order Logic

2.1.1 Propositional Logic

Propositional logic is a language based on *atomic propositions* which can either be true ($\top$) or false ($\bot$). Let $\mathcal{B}$ be a set of variables ranging over the Boolean domain $\mathbb{B} = \{\top, \bot\}$. Then the syntax of propositional logic is defined as follows:

$$\begin{aligned} \text{atoms} \ni a &:= \top \mid \bot \mid b \\ \text{formulas} \ni \phi &:= a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi, \end{aligned}$$

for each $b \in \mathcal{B}$. We will call anything that conforms to this grammar a *propositional formula*. A model for a propositional formula is a mapping that assigns either $\top$ or $\bot$ to each variable. The semantics of the connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$ are defined as usual.

2.1.2 First-Order Logic

First-order logic is a formal calculus over elements from a (finite or infinite) *domain of discourse* $\mathbb{D}$. Let $\mathcal{X}$ be a set of variables ranging over $\mathbb{D}$, let $\mathcal{B}$ be a set of propositional variables ranging over $\mathbb{B}$, let $\mathcal{F}$ be a set of function symbols, and let $\mathcal{P}$ be a set of predicate symbols. The syntax of first-order logic is defined by the following grammar:

$$\begin{aligned} \text{terms} \ni t &:= x \mid f(t, \ldots, t), \\ \text{atoms} \ni a &:= \top \mid \bot \mid b \mid P(t, \ldots, t), \\ \text{formulas} \ni \phi &:= a \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \exists x \,.\, \phi \mid \forall x \,.\, \phi, \end{aligned}$$

for each $x \in \mathcal{X}$, $b \in \mathcal{B}$, $f \in \mathcal{F}$, and $P \in \mathcal{P}$. Furthermore, we use the following common naming conventions. A *literal* is an atom or the negation of an atom. Literals have a *polarity*, which is *positive* if the literal is an atom, and *negative* if the literal is a negated atom. Let $\mathit{Lits}(\phi)$ denote the set of literals occurring in ϕ. A *clause* is a disjunction of literals. A Conjunctive Normal Form (CNF) is a conjunction of clauses. Every formula can be transformed into an equisatisfiable CNF formula, by introducing a linear number of additional propositional variables, via a procedure called *Tseitin's encoding* [Tse68]. The variables introduced during Tseitin's encoding are called *Tseitin variables.* The symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, and $\exists$ are called *logical symbols.* All other symbols (variables, functions symbols, predicate symbols) are called *non-logical symbols.* We will denote with $symb(\phi)$ the set of non-logical symbols occurring in ϕ. Furthermore, let $\phi \preceq \psi$ iff $symb(\phi) \subseteq symb(\psi)$. For a formula ϕ and a clause C, let $C|_\phi = \bigvee_{l \,\in\, Lits(C),\; l \preceq \phi} l$, that is, the clause C with all literals that contain symbols that do not occur in ϕ removed.

In formulas of the form $\exists x \,.\, \phi$ and $\forall x \,.\, \phi$ the variable x is said to be *bound* by the quantifier. A variable that is not bound by any quantifier is called a *free* variable. A formula without free variables is called a *closed* formula. A formula with free variables is called an *open* formula.

A *model* $\mathcal{M}$ for a formula in first-order logic is a tuple $(\llbracket \mathbb{D} \rrbracket, \llbracket \mathcal{X} \rrbracket, \llbracket \mathcal{B} \rrbracket, \llbracket \mathcal{F} \rrbracket, \llbracket \mathcal{P} \rrbracket)$. $\llbracket \mathbb{D} \rrbracket$ is a set of elements that comprise the concrete domain of discourse used by the model. $\llbracket \mathcal{X} \rrbracket$ is a mapping from $\mathcal{X}$ to $\llbracket \mathbb{D} \rrbracket$, assigning to every $x \in \mathcal{X}$ a concrete object from $\llbracket \mathbb{D} \rrbracket$. $\llbracket \mathcal{B} \rrbracket$ is a mapping from $\mathcal{B}$ to $\mathbb{B}$, assigning to every $b \in \mathcal{B}$ either $\top$ or $\bot$. $\llbracket \mathcal{F} \rrbracket$ assigns a concrete function $\llbracket f \rrbracket$, mapping from $\mathbb{D}^n$ to $\mathbb{D}$, to each n-ary function symbol $f \in \mathcal{F}$. Similarly, $\llbracket \mathcal{P} \rrbracket$ assigns to every $P \in \mathcal{P}$ of arity n a concrete function $\llbracket P \rrbracket$, mapping from $\mathbb{D}^n$ to $\mathbb{B}$. We will use the notation $\llbracket \cdot \rrbracket$ to denote the value of "$\cdot$" in a model. If the model to which we refer is not clear from the context, we will use it as a subscript: $\llbracket \cdot \rrbracket_{\mathcal{M}}$. Based on the usual semantics of Boolean connectives and quantifiers, we say that a model $\mathcal{M}$ satisfies a formula ϕ, written as $\mathcal{M} \vDash \phi$, iff $\mathcal{M}$ makes ϕ evaluate to $\top$.

2.1.3 First-Order Theories

A *first-order theory* $\mathcal{T}$ is a tuple $(\Sigma_{\mathcal{T}}, \mathcal{A}_{\mathcal{T}})$. The *signature* $\Sigma_{\mathcal{T}}$ is a (possibly infinite) set of variable, predicate, and function symbols. A $\mathcal{T}$-formula $\phi_{\mathcal{T}}$ is a first-order logic formula such that all variables, predicates, and functions symbols occurring in $\phi_{\mathcal{T}}$ also occur in Σ. We will sometimes also use the term "theory" or the symbol $\mathcal{T}$ to denote the set of all $\mathcal{T}$-formulas. $\mathcal{A}$ is a (possibly infinite) set of *axioms.* Axioms are $\mathcal{T}$-formulas where all variables are bound by a quantifier. The intuition behind axioms is that they impose a specific "meaning" onto the symbols in Σ. That is, the axioms are formulas that state properties of symbols in Σ. For a model $\mathcal{M}$, we write $\mathcal{M} \vDash \mathcal{T}$ to indicate that $\mathcal{M}$ satisfies all axioms of $\mathcal{T}$.

A *fragment* of a theory $\mathcal{T}$ is a syntactically restricted subset of all possible $\mathcal{T}$-formulas. There are two fragments which are often of specific interest: The *quantifier-free fragment,* which disallows the use of quantifiers, and the *conjunctive fragment* which allows only conjunctions of literals.

2.1.4 Theory of Uninterpreted Functions and Equality

The first theory we consider is the *theory of uninterpreted functions and equality* $\mathcal{T}_U$. Its signature is $\Sigma_U = \mathcal{X}_U \cup \mathcal{B}_U \cup \mathcal{F}_U \cup \mathcal{P}_U \cup \{=\}$, where $\mathcal{X}_U$ is the set of all variables ranging over $\mathbb{D}$, $\mathcal{B}_U$ is the set of all propositional variables, $\mathcal{F}_U$ is a set of uninterpreted function symbols, $\mathcal{P}_U$ is a set of uninterpreted predicate symbols and "=" is the equality predicate. We will follow the usual convention of writing the equality predicate in infix style. That is, we will write "$a = b$", instead of "$= (a, b)$". Furthermore, we will use $\overline{x}$ as a shorthand notation for $x_1, x_2, \ldots, x_n$; in particular with respect to arguments of functions and predicates, as well as quantifiers. That is, we will — for example — write $f(\overline{x})$ instead o $f(x_1, x_2, \ldots, x_n)$.

The equality predicate is the only interpreted symbol in $\mathcal{T}_U$. Its semantics are defined by the axioms of $\mathcal{T}_U$:

$$\text{Reflexivity:} \quad \forall x \,.\, x = x, \tag{2.1}$$

$$\text{Symmetry:} \quad \forall x \,.\, \forall y \,.\, x = y \rightarrow y = x, \text{ and} \tag{2.2}$$

$$\text{Transitivity:} \quad \forall x \,.\, \forall y \,.\, \forall z \,.(x = y \wedge y = z) \rightarrow x = z. \tag{2.3}$$

For every uninterpreted function symbol $f \in \mathcal{F}_U$ we have:

$$\text{Function congruence:} \quad \forall \overline{x} \,.\, \forall \overline{y} \,.\, \left(\bigwedge_i x_i = y_i \right) \rightarrow f(\overline{x}) = f(\overline{y}); \tag{2.4}$$

and for every uninterpreted predicate symbol $P \in \mathcal{P}_U$ we have:

$$\text{Predicate congruence:} \quad \forall \overline{x} \,.\, \forall \overline{y} \,.\, \left(\bigwedge_i x_i = y_i \right) \rightarrow \big(P(\overline{x}) \leftrightarrow P(\overline{y}) \big) \,. \tag{2.5}$$

The axioms of $\mathcal{T}_U$ specify the usual semantics of the equality predicate: reflexivity (Equation 2.1), symmetry (Equation 2.2), transitivity (Equation 2.3), and congruence with respect to uninterpreted functions (Equation 2.4) and predicates (Equation 2.5).

Henceforth, we will be particularly interested in the quantifier-free fragment of $\mathcal{T}_U$, which we will denote $\mathcal{T}_U^{\mathrm{qf}}$. Moreover, we will sometimes refer to the *theory of equality* $\mathcal{T}_E$, which is a subset of $\mathcal{T}_U$. The signature Σ_E of $\mathcal{T}_E$ does not contain uninterpreted function and predicate symbols. The axioms of $\mathcal{T}_E$ are the first three axioms of $\mathcal{T}_U$ (Equations 2.1, 2.2, and 2.3, respectively). Also for $\mathcal{T}_E$, we will be particularly interested in the quantifier-free fragment $\mathcal{T}_E^{\mathrm{qf}}$.

2.1.5 Theory of Arrays

The *theory of arrays* $\mathcal{T}_A$ was first proposed and axiomatized by McCarthy [McC63]. It is designed for formal reasoning about indexed arrays, by axiomatizing how reading from and writing to an array works. We deviate slightly from standard literature by making $\mathcal{T}_A$ an extension of $\mathcal{T}_U$. That is, we allow the use of uninterpreted functions in $\mathcal{T}_A$-formulas. Thus, the signature of $\mathcal{T}_A$ is $\Sigma_A = \Sigma_U \cup \{\, \cdot[\cdot] \ , \ \cdot\langle \cdot \lhd \cdot \rangle \,\}$, where $\cdot[\cdot]$ is the binary *array-read function* and $\cdot\langle \cdot \lhd \cdot \rangle$ is the ternary *array-write function*. This notation is to be understood as follows: $\mathsf{A}[i]$ denotes the value of array A at index i. By $\mathsf{A}\langle j \lhd x \rangle$, we denote an array identical to A, except that the value at index j equals x. For convenience, we will write $\mathsf{A} = \mathsf{B}$ to express $\forall i \,.\, \mathsf{A}[i] = \mathsf{B}[i]$.

Note that we have introduced first-order logic without distinguishing different data types (also known as "sorts"). This follows the definition given in [HR04]. Thus, there is no formal distinction between array variables and other terms, such as index terms. However, since such a distinction often proves useful in practice, we will make some syntactic restrictions, when we define our specification language in Section 3.1.

The axioms of $\mathcal{T}_A$ are as follows:

$$\text{Array Congruence:} \quad \forall \mathsf{A} \,.\, \forall i \,.\, \forall j \,.\, i = j \rightarrow \mathsf{A}[i] = \mathsf{A}[j] \tag{2.6}$$

$$\text{Read-over-Write 1:} \quad \forall \mathsf{A} \,.\, \forall j \,.\, \forall x \,.\, \mathsf{A}\langle j \lhd x \rangle[j] = x \tag{2.7}$$

$$\text{Read-over-Write 2:} \quad \forall \mathsf{A} \,.\, \forall x \,.\, \forall i \,.\, \forall j \,.\, i \neq j \rightarrow \mathsf{A}\langle j \lhd x \rangle[j] = \mathsf{A}[i] \tag{2.8}$$

Bradley et al. [BMS06] have identified a fragment of $\mathcal{T}_A$ that allows limited use of quantifiers, but is still decidable. They call it the *array property fragment*. In the following we will focus on properties of arrays with *uninterpreted indices*, as presented in [BM07]. Following their definitions, an array property is a formula of the form

$$\forall \bar{i} \,.\, F_{\bar{i}} \rightarrow G_{\bar{i}},$$

where $\bar{i}$ is a tuple of variables, $F_{\bar{i}}$ is the so-called *index guard*, and $G_{\bar{i}}$ is the *value constraint*. The index guard must conform to the following grammar [BM07]:

$$\begin{aligned}
\mathsf{ivar} &:= \ \mathit{evar} \mid \mathit{uvar}, \\
\mathsf{iatom} &:= \ \mathsf{ivar} = \mathsf{ivar} \mid \mathit{evar} \neq \mathsf{ivar} \mid \mathsf{ivar} \neq \mathit{evar} \mid \top, \\
\mathsf{iguard} &:= \ \mathsf{iguard} \wedge \mathsf{iguard} \mid \mathsf{iguard} \vee \mathsf{iguard} \mid \mathsf{iatom},
\end{aligned}$$

where *uvar* is any of the variables from $\bar{i}$, and *evar* is an unquantified variable or constant. In the value constraint $G_{\bar{i}}$, universally quantified variables may only occur inside array reads $\mathsf{A}[i]$. Furthermore, nested reads like $\mathsf{A}[\mathsf{B}[i]]$ are disallowed. The array property fragment $\mathcal{T}_A^{\mathrm{P}}$ consists of formulas that are Boolean combinations of quantifier-free $\mathcal{T}_A$-formulas and array properties [BM07].

2.2 Satisfiability Modulo Theories Solving and Refutation Proofs

Now that we have defined theories, we consider the question of whether or not a $\mathcal{T}$-formula $\phi_{\mathcal{T}}$ is *satisfiable* with respect to the theory. In other words, we ask whether or not there exists a model $\mathcal{M}$ that satisfies $\phi_{\mathcal{T}}$ and also satisfies all axioms of $\mathcal{T}$.

Definition 1 — Satisfiability Modulo Theories (SMT)
A $\mathcal{T}$-formula ϕ is satisfiable modulo $\mathcal{T}$ ($\mathcal{T}$-satisfiable) *iff there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \phi$ and $\mathcal{M} \models A$ for all axioms $A \in \mathcal{A}_{\mathcal{T}}$. A formula that is not ($\mathcal{T}$-) satisfiable is called ($\mathcal{T}$-)* unsatisfiable.

We also consider the dual of satisfiability: validity.

Definition 2 — Validity (modulo Theories)
A $\mathcal{T}$-formula ϕ is valid modulo $\mathcal{T}$ ($\mathcal{T}$-valid)*, iff for all models $\mathcal{M}$ for which $\mathcal{M} \models A$ holds for all axioms $A \in \mathcal{A}_{\mathcal{T}}$, it holds that $\mathcal{M} \models \phi$.*

Lemma 1 — Duality of Satisfiability and Validity
A $\mathcal{T}$-formula ϕ is $\mathcal{T}$-valid if and only if its negation $\neg\phi$ is not $\mathcal{T}$-satisfiable. A $\mathcal{T}$-formula ϕ is $\mathcal{T}$-satisfiable if and only if its negation $\neg\phi$ is not $\mathcal{T}$-valid.

An SMT solver is an algorithm that takes as input a $\mathcal{T}$-formula and outputs whether or not the formula is $\mathcal{T}$-satisfiable. Usually, a solver also produces a satisfying model (if the answer is "satisfiable") or a refutation proof (if the answer is "unsatisfiable") as a byproduct.

We will briefly look at three different approaches for SMT solving: eager encoding, lazy encoding, and DPLL(T). While these concepts are general, we focus on analyzing them for the theories (and fragments) introduced in Section 2.1.

2.2.1 Eager Encoding

The basic idea of eager encoding is to transform the theory formula $\phi_{\mathcal{T}}$ into an equisatisfiable propositional formula ϕ_{prop}. This is achieved in two steps. First, every theory atom of $\phi_{\mathcal{T}}$ is replaced by a fresh propositional variable. We will call this the *propositional skeleton* of $\phi_{\mathcal{T}}$.

Definition 3 — Propositional Skeleton
Let $\phi_{\mathcal{T}}$ be a $\mathcal{T}$-formula. The propositional skeleton *of $\phi_{\mathcal{T}}$, denoted $skel(\phi_{\mathcal{T}})$, is the propositional formula obtained by replacing every theory atom with a fresh propositional variable b.*

Lemma 2
Let $\phi_{\mathcal{T}}$ be a $\mathcal{T}$-formula that is $\mathcal{T}$-satisfiable. Then $skel(\phi_{\mathcal{T}})$ is satisfiable.

Proof
Since $\phi_{\mathcal{T}}$ is $\mathcal{T}$-satisfiable, there exists a model $\mathcal{M}_{\mathcal{T}}$ such that $\mathcal{M}_{\mathcal{T}} \models \phi_{\mathcal{T}}$. We construct a model $\mathcal{M}_{prop}$ that assigns to each variable propositional variable b the same truth value that the theory atom to which b corresponds has in $\mathcal{M}_{\mathcal{T}}$. Clearly, we have that $\mathcal{M}_{prop} \models skel(\phi_{\mathcal{T}})$. **Q. E. D.**

After obtaining the propositional skeleton, the second step is to compute so-called *constraints* that are added in order to block truth assignments to the theory atoms that would contradict the axioms of the theory. As the name "eager encoding" suggests, adding of constraints is done eagerly. That is, a sufficient set of constraints, denoted $Constr(\phi)$, is added such that the resulting propositional formula

$$\phi_{prop} = skel(\phi) \wedge \left(\bigwedge_{\phi_c \,\in\, Constr(\phi)} \phi_c \right) \tag{2.9}$$

is equisatisfiable to the original theory formula $\phi_{\mathcal{T}}$. The formula ϕ_{prop} can then be given to a propositional SAT solver, which will determine its satisfiability.

For $\mathcal{T}_A^{\mathrm{p}}$, there exists a three-stage eager encoding procedure. The first step, which we call *index set construction*, reduces a $\mathcal{T}_A^{\mathrm{p}}$-formula to a formula in $\mathcal{T}_U^{\mathrm{qf}}$ [BM07]. The second step, called Ackermann's reduction [Ack54], removes all uninterpreted function and predicate instances and replaces them with fresh variables of the correct type. This yields a formula in $\mathcal{T}_E^{\mathrm{qf}}$. Finally, a graph-based construction [BV00] reduces the formula to propositional logic. For a $\mathcal{T}_A^{\mathrm{p}}$-formula ϕ, we will compute a set of constraints $Constr_i(\phi)$ in each of the steps. The union of these constraints $Constr(\phi) = \bigcup_i Constr_i(\phi)$ will be sufficient to make the corresponding ϕ_{prop} (see Equation 2.9) equisatisfiable to ϕ. We will now briefly revisit the details of these reduction techniques, as we will later use them as the basis for the validity-preserving reductions described in Section 4.2.

Index Set Construction

Bradley et al. [BM07] show how $\mathcal{T}_A^{\mathrm{p}}$-formulas can be reduced to $\mathcal{T}_U^{\mathrm{qf}}$-formulas. First, all array-write expressions are removed by introducing fresh variables. The corresponding instances of the write-axioms (Equations 2.7 and 2.8) are added to the set of constraints.

Example 1
Let ϕ be a $\mathcal{T}_A^{p}$ formula that contains the array-write term $\mathsf{A}\langle k \lhd x\rangle$. We introduce a fresh variable B and replace every occurrence of $\mathsf{A}\langle k \lhd x\rangle$ with B. We also add the conjunction $\mathsf{B}[k] = x \ \wedge\ \forall i\,.\, i \neq k \rightarrow \mathsf{B}[i] = \mathsf{A}[i]$ to the set of constraints.

Next, we compute the so-called *index set* $\mathcal{I}$. The index set is the union of all terms that are used for array read-access (unless they are universally quantified variables), all terms that occur as an *evar* in the index guards, and the special term λ that represents "any other index". It is important that λ is distinct from any other term in the index set. Thus, we add

$$\bigwedge_{i\, \in\, \mathcal{I}\setminus\{\lambda\}} i \neq \lambda \qquad (2.10)$$

to the set of constraints.

It has been shown [BM07] that considering this finite set $\mathcal{I}$ of indices only is sufficient for proving or disproving satisfiability. Thus, we replace all universal quantifications in the array properties with finite conjunctions over the index set $\mathcal{I}$. That gives us a quantifier-free $\mathcal{T}_A$-formula without array-write expressions. Finally, we replace all array-read expressions with instances of (fresh) uninterpreted functions. For example, $\mathsf{A}[i]$ is replaced by $A(i)$.[5] The resulting formula is in $\mathcal{T}_U^{\mathrm{qf}}$.

[5]Note that for simplicity and readability, we name the fresh uninterpreted functions exactly the same as the corresponding arrays. The distinction is made by using square brackets $[\cdot]$ with arrays, and parenthesis $(\cdot)$ with uninterpreted functions.

Definition 4 — *no_array*()
Let ϕ be a $\mathcal{T}_A^p$ formula. Then no_array(ϕ) is the $\mathcal{T}_U^{qf}$-formula obtained by the transformations outlined above.

Definition 5 — Array Constraints
Let ϕ be a $\mathcal{T}_A^p$-formula. Then $AC(\phi)$ is the set of all constraints obtained by applying the index set construction outlined above.

Ackermann's Reduction

Ackermann's reduction [Ack54] replaces every uninterpreted function instance $f(\overline{x})$ with a fresh domain variable $d_f^{\overline{x}}$, and every uninterpreted predicate instance $P(\overline{x})$ with a fresh propositional variable $b_P^{\overline{x}}$. The constraints which are added make sure that the fresh variables are equal whenever the function (predicate, respectively) instances they represent are equal according to the congruence axioms. For example, let f be a unary function, and let d_f^x and d_f^y be the fresh variables replacing the function instances $f(x)$ and $f(y)$ respectively. Then the following constraints are added during Ackermann's reduction:

$$\bigwedge_{a,b \,\in\, args(f)} \left(a = b \rightarrow d_f^a = d_f^b\right). \tag{2.11}$$

It should be obvious how to generalize this constraint generation to n-ary functions and predicates. Also note that each conjunct in Equation 2.11 can easily be turned into a clause, by using the equivalence $(\varphi \rightarrow \psi) \iff (\neg\varphi \vee \psi)$.

Definition 6 — *no_func*()
Let ϕ be a $\mathcal{T}_U^{qf}$-formula. Then no_func(ϕ) is the $\mathcal{T}_E^{qf}$-formula obtained by replacing every uninterpreted function instance $f(\overline{a})$ with a fresh domain variable $x_f^{\overline{a}}$ and every uninterpreted predicate instance $P(\overline{a})$ with a fresh propositional variable $b_P^{\overline{a}}$.

Definition 7 — Congruence Constraints
Let ϕ be a $\mathcal{T}_U^{qf}$-formula. Then $CC(\phi)$ is the set of all congruence constraints obtained by applying Ackermann's reduction.

Graph-based Reduction

Ackermann's reduction gives us a formula in $\mathcal{T}_E^{\text{qf}}$. The last step required to reduce this to an equisatisfiable propositional formula is to remove equalities. Bryant and Velev [BV00] have introduced such a reduction based on an equality graph. For a $\mathcal{T}_E^{\text{qf}}$-formula ϕ, it proceeds as follows. First, every reflexivity instance $a = a$ in ϕ is replaced by $\top$. Second, every equality atom is rewritten such that the first term precedes the second term with respect to some total order. This takes care of ensuring symmetry. Next, every equality atom $a = b$ is replaced by a fresh propositional variable $b_{a=b}$. In order to take care of transitivity, we construct a so-called *non-polar equality graph*: This graph has a node for every term and an edge for every equality literal (regardless of its polarity) in the formula. This graph is then made *chordal*.

Definition 8 — Chords, Chord-free Cycles, and Chordal Graphs
In a graph $\mathcal{G}$, let n_1 and n_2 be two non-adjacent nodes in a cycle. An edge between n_1 and n_2 is called a chord.

Iff, in a cycle, there exist no non-adjacent nodes that are connected by an edge, the cycle is said to be chord-free.

A graph is called chordal, *iff all of its cycles of length greater than* 3 *are chord-free.*

A graph can be made chordal by adding additional edges. Based on the chordal graph, we can compute the transitivity constraints. For every triangle (x, y, z) in the graph, we add the following constraints:

$$\begin{aligned}
&(b_{x=y} \wedge b_{y=z} \rightarrow b_{x=z}) \wedge \\
&(b_{x=y} \wedge b_{x=z} \rightarrow b_{y=z}) \wedge \\
&(b_{y=z} \wedge b_{x=z} \rightarrow b_{x=y})
\end{aligned} \tag{2.12}$$

Definition 9 — *no_ equal*()
Let ϕ be a $\mathcal{T}_E^{qf}$-formula. Then no_ equal(ϕ) is the propositional formula obtained by the procedure outlined above.

Definition 10 — Transitivity Constraints
Let ϕ be a $\mathcal{T}_E^{qf}$-formula. Then $TC(\phi)$ is the set of all transitivity constraints obtained by applying the graph-based reduction outlined above to ϕ.

Theorem 1
Let ϕ be a $\mathcal{T}_A^{p}$-formula. Then the propositional formula

$$\begin{aligned}
\phi_{prop} \quad = \quad & AC(\phi) \ \wedge \\
& CC(no\_array(\phi)) \ \wedge \\
& TC(no\_func(no\_array(\phi))) \ \wedge \\
& no\_equal(no\_func(no\_array(\phi)))
\end{aligned}$$

is satisfiable if and only if ϕ is satisfiable modulo $\mathcal{T}_A^{p}$.

Proof
See [BM07], [Ack54], and [BV00]. **Q. E. D.**

Concerning computational complexity, this reduction can be done in polynomial time: For a $\mathcal{T}_A^{\mathrm{p}}$-formula ϕ_a of size n_a, the size of the index set is bound by $\mathcal{O}(n_a)$. Also, the number of array properties that occur in ϕ is bound by $\mathcal{O}(n_a)$. Thus, the size of the equisatisfiable $\mathcal{T}_U^{\mathrm{qf}}$-formula ϕ_u obtained by the reductions outlined above is bound by $\mathcal{O}(n_a^2)$. Ackermann's reduction also causes a quadratic blow-up. Let n_u be the size of ϕ_u. The number of function and predicate instances occurring in ϕ_u is bound by $\mathcal{O}(n_u)$. Thus, at most $\mathcal{O}(n_u^2)$ constraints are added, as one constraint for each pair of function/predicate instances is necessary. Let

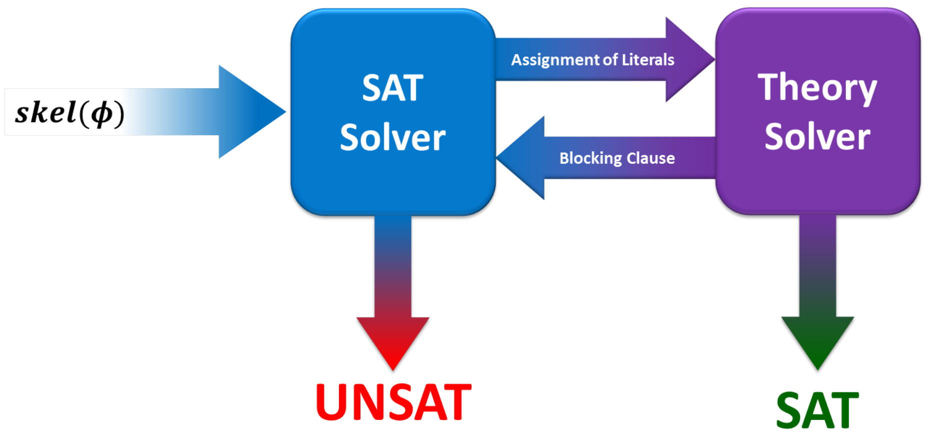


Figure 2.1: Lazy Encoding. The propositional skeleton of ϕ is given to a SAT solver. If a satisfying assignment is found, it is checked by a theory solver. If the assignment is consistent with the theory, ϕ is $\mathcal{T}$-satisfiable. Otherwise, a blocking clause is generated and the SAT solver searches for a new assignment. This is repeated until either a $\mathcal{T}$-consistent assignment is found, or the SAT solver cannot find any more assignments.

the resulting formula be ϕ_e, with size n_e. The number of equalities in ϕ_e is bound by $\mathcal{O}(n_e)$. Thus, the equality graph has at most $\mathcal{O}(n_e)$ nodes and $\mathcal{O}(n_e^3)$ triangles. As the graph-based reduction adds one constraint for each triangle, the size of the resulting propositional formula is bound by $\mathcal{O}(n_e^3)$.

2.2.2 Lazy Encoding

Lazy encoding is based on the interaction between a SAT solver and a so-called *theory solver*. A theory solver is an algorithm that can decide satisfiability of the conjunctive fragment of a theory. In contrast to eager encoding, where a sufficient set of constraints is computed at the beginning, lazy encoding starts with no constraints at all, and lazily adds constraints only when required.

The principle of lazy encoding is shown in Figure 2.1. To decide whether or not a $\mathcal{T}$-formula ϕ is $\mathcal{T}$-satisfiable, the propositional skeleton $\mathit{skel}(\phi)$ is given to a SAT solver.[6] If the SAT solver returns "unsatisfiable", the procedure is done and we know that ϕ is not $\mathcal{T}$-satisfiable. If, however, the SAT solver returns "satisfiable", we obtain a satisfying assignment for the truth values of the theory atoms in ϕ. This assignment is a formula in the conjunctive fragment of $\mathcal{T}$, which we pass to the theory solver. If the theory solver returns "satisfiable", we have found an assignment of truth values to the theory atoms that is *consistent*

[6]We will assume that ϕ is given in CNF. If not we apply Tseitin's encoding to obtain an equisatisfiable CNF.

with $\mathcal{T}$. Thus we know that ϕ is $\mathcal{T}$-satisfiable. If, however, the theory solver returns "unsatisfiable", we have to look for another assignment, as the present one is not consistent with $\mathcal{T}$. To obtain a different assignment, we negate the inconsistent assignment — which conveniently turns it into a clause — and add it as a so-called *blocking clause* to the CNF of $\mathit{skel}(\phi)$. The blocking clause ensures that the next satisfying assignment obtained from the SAT solver (if one exists) is different from the current, $\mathcal{T}$-inconsistent assignment. This loop is repeated until we encounter one of the following two terminal cases. Either the SAT solver suggests an assignment that the theory solver finds to be consistent with $\mathcal{T}$, in which case ϕ is $\mathcal{T}$-satisfiable. Or we have added so many blocking clauses that the SAT solver cannot find any more assignments, in which case ϕ is not $\mathcal{T}$-satisfiable. As every blocking clause excludes (at least) one of only finitely many assignments, the loop is guaranteed to terminate.

Note that all the blocking clauses are *valid* within the theory. That is, every model that satisfies all axioms of the theory also satisfies the blocking clauses. We will thus also refer to these clauses as *theory lemmata*.

Definition 11 — Theory Lemma
A clause C is a theory lemma *of theory $\mathcal{T}$ iff for every model $\mathcal{M}$ it holds that either $\mathcal{M} \models C$, or there exists an axiom $A \in \mathcal{A}_\mathcal{T}$ such that $\mathcal{M} \not\models A$. We will write $\vdash_\mathcal{T} C$ to denote that C is a lemma of theory $\mathcal{T}$.*

Congruence Closure

The well-known *congruence closure* algorithm, which was first introduced in the late 1970ies [NO77, Sho78], is the most common theory solver for $\mathcal{T}_U^{\mathrm{qf}}$. Given a conjunction of $\mathcal{T}_U^{\mathrm{qf}}$-literals, it computes a set of congruence classes, such that the conjunction of literals implies that all terms in the same congruence class are equal. This is done in the following way. First, all terms for which there is a (positive) equality in the conjunction of literals are put into the same congruence class. All remaining terms are put in singleton classes. Next, classes are merged, if they contain common terms. This accounts for the transitivity of the equality predicate. Next, classes are merged based on function congruence. That is, if two classes both contain an instance of the same uninterpreted function, and corresponding parameters are already in the same congruence class (which means that they are equal), the classes of the function instances are merged. These steps are repeated until no more merging can be done. In the last step, all the disequalities from the conjunction of literals are checked against the merged congruence classes. If there is a disequality that contradicts the congruence classes, that is, both its terms are in the same congruence class, the conjunction of literals is unsatisfiable. If no such disequality exists, the conjunction of literals is satisfiable.

The congruence closure algorithm can be used to compute a *congruence graph* [FGG+09, FGG+12] according to the following definition.

Definition 12
A congruence graph *over a set A of atoms is a graph which has terms as its*

nodes. Each edge is labeled either with an equality justification*, which is an equality atom from A that equates the terms connected by the edge, or with a* congruence justification*. A congruence justification can only be used when the terms connected by the edge are both instances $f(a_1, a_2, \dots, a_k)$ and $f(b_1, b_2, \dots, b_k)$ of the same uninterpreted function f. In this case, the congruence justification is a set of k paths in the graph, connecting a_i with b_i, not using the edge labeled by the congruence justification.*

Definition 13 — Transitivity-Congruence Chain
A transitivity-congruence chain *$\pi = (a \rightsquigarrow b)$ is a path in a congruence graph that connects terms a and b. Let $\mathit{Lits}(\pi)$ be the set of literals of the path, which is defined as the union of the literals of all edges on the path. The literal of an edge labeled with an equality justification p is the set $\{p\}$. The set of literals of an edge labeled with a congruence justification with paths π_i is recursively defined as $\bigcup_i \mathit{Lits}(\pi_i)$.*

Transitivity-congruence chains are a useful data structure for splitting theory lemmata in the context of interpolation (see 2.3 for details). The property stated in the following theorem will be very useful.

Theorem 2
The conjunction of the literals in a transitivity-congruence chain $(a \rightsquigarrow b)$ implies $a = b$ within $\mathcal{T}_U^{qf}$. That is, $(\bigvee_{l \in \mathit{Lits}(a \rightsquigarrow b)} \neg l) \vee (a = b)$ is a theory lemma.

Proof
The proof works by induction over the length of the chain. As a base case for induction, we consider a chain $(a_0 \rightsquigarrow a_1)$ of length 1. In this case we have to show that $(a_0 = a_1)$ implies $(a_0 = a_1)$, which is trivial. Next, as induction hypothesis, we assume that for a chain $\pi = (a_0 \rightsquigarrow a_n)$ of length n, it holds that $\bigwedge \mathit{Lits}(\pi)$ implies $a_0 = a_n$. Now we extend the chain by one element: $\pi' = (a_0 \rightsquigarrow a_{n+1})$. We need to show that $\bigwedge \mathit{Lits}(\pi')$ implies $a_0 = a_{n+1}$. We already know that $\bigwedge \mathit{Lits}(\pi')$ implies $a_0 = a_n$, because $\mathit{Lits}(\pi) \subset \mathit{Lits}(\pi')$. Now, we consider two cases:

1. *The edge from a_n to a_{n+1} is labeled with an equality justification $(a_n = a_{n+1})$.* In this case, we need to show that $(a_0 = a_n) \wedge (a_n = a_{n+1})$ implies $a_0 = a_{n+1}$. Since this is an instance of the transitivity axiom (see Equation 2.3), this is clearly the case.

2. *The edge from a_n to a_{n+1} is labeled with a congruence justification.* In this case, the terms a_n and a_{n+1} are instances of an uninterpreted function. Let $a_n = f(\overline{x})$ and $a_{n+1} = f(\overline{y})$. Without loss of generality, we assume that the length of each path in the congruence justification is less than or equal to n. Thus, by our induction hypothesis $\bigwedge \mathit{Lits}(f(\overline{x}) \rightsquigarrow f(\overline{y}))$ implies that for each i we have $x_i = y_i$. Thus, from the function congruence axiom (see Equation 2.4), we conclude that $\bigwedge \mathit{Lits}(a_n \rightsquigarrow a_{n+1})$ implies $f(\overline{x}) = f(\overline{y})$. Now we use the same reasoning as in case 1 to show that $(a_0 = a_n) \wedge (a_n = a_{n+1})$ implies $a_0 = a_n$. **Q. E. D.**

$$\textsc{Hyp}\,\frac{}{C}\,C \in \phi \qquad \textsc{Axi}\,\frac{}{C}\,\vdash_{\mathcal{T}} C \qquad \textsc{Res}\,\frac{a \vee C \quad \neg a \vee D}{C \vee D}$$

Figure 2.2: Sound and complete proof rules for a $\mathcal{T}$-formula ϕ in CNF.

2.2.3 DPLL(T)

The main disadvantage of lazy encoding is that it waits for a full assignment to all theory literals before checking for theory consistency. This leads to very specific blocking clauses and to a large number of iterations between SAT- and theory-solver. A possible solution to this problem is a tighter integration between the propositional SAT solver and the theory solver. The resulting algorithm is called *DPLL(T)*, which is the basis for most modern SMT solvers.

DPLL(T) is based on the DPLL algorithm [DP60, DLL62], and conflict-driven clause learning [MS96, MS99]. It extends these concepts from the propositional to theory level. We assume that the reader is familiar with DPLL and conflict-driven clause learning for the propositional case. If not, Kroening and Strichman [KS08], or the *Handbook of Satisfiability* [BHvMW09] provide a thorough overview of the topic.

The key idea behind DPLL(T) is to embed the check for theory consistency directly into the core of DPLL. Every partial assignment of truth values to theory literals that is tried by DPLL is immediately passed to the theory solver for a consistency check. If the partial assignment is not theory-satisfiable, we immediately learn a blocking clause. This blocking clause is much shorter, and thus more general, than one obtained from a full assignment. Moreover, a good theorem solver provides more than just a yes/no-answer when given a (partial) assignment: it returns one or more strong theory lemmata clauses. That is, short clauses that are valid in the theory and contradict the given (partial) assignment. These can then be added to the CNF of the formula in question. Furthermore, a good solver can support theory propagation. That is, given for example a partial assignment $(a = b) \equiv \top$ and $(b = c) \equiv \top$, the solver can deduce $(a = c) \equiv \top$ and propagate this back to DPLL.

Note that the DPLL(T)-framework is theory-agnostic. That is, it can be used with arbitrary theories for which there is a solver for the corresponding conjunctive fragment. For $\mathcal{T}_U^{\mathrm{qf}}$, the congruence closure-algorithm presented in Section 2.2.2 can be used.

2.2.4 Refutation Proofs

Whenever a $\mathcal{T}$-formula ϕ is not $\mathcal{T}$-satisfiable, we can obtain a *refutation proof.* Such proofs can be obtained from SMT solvers with relatively little overhead. Before we look at the details of proofs, we need to define *proof rules.* Since we only consider formulas in CNF, we only need three different rules, as shown in Figure 2.2.

Definition 14 — Proof Rule
A named proof rule *is a template for a logic entailment between a (possibly empty) set of* premises *and a* conclusion*. Templates for premises are written above a horizontal line, templates for conclusions below. Possible conditions for the application of the proof rule are written on the right-hand side of the line. The* name *of the rule is written on the left-hand side of the line.*

$$\langle\texttt{name}\rangle\ \frac{\langle\texttt{premise(s)}\rangle}{\langle\texttt{conclusion}\rangle}\langle\texttt{condition(s)}\rangle$$

The proof rules given in Figure 2.2 form a sound and complete proof system for proving unsatisfiability of a $\mathcal{T}$-formula ϕ in CNF. The Hyp rule is used to introduce clauses from ϕ into the proof. The Axi rule is used to introduce clauses that are theory lemmata of $\mathcal{T}$. The Res rule is the standard resolution rule to combine clauses that contain one literal in opposite polarity respectively. We will call this literal the *resolving literal*. Based on these proof rules, we define refutation proofs as follows.

Definition 15 — Refutation Proof
A refutation proof *is a Directed Acyclic Graph (DAG) (N, E), where $N = \{r\} \cup N_I \cup N_L$ is the set of nodes (partitioned into the root node r, the set of internal nodes N_I and the set of leaf nodes N_L), and $E \subseteq N \times N$ is the set of (directed) edges. Every $n \in N$ is labeled with the name of a proof rule $rule(n)$ and a clause $clause(n)$. The graph has to fulfill the following properties:*

1. *$clause(r) = \bot$.*
2. *For all $n \in N_L$, $clause(n)$ is either a clause from ϕ (if $rule(n) =$ Hyp) or a theory lemma (if $rule(n) =$ Axi).*
3. *The root node r has no incoming edges, the leaves in N_L have no outgoing edges, and all nodes $n \in N \backslash N_L$ have exactly two outgoing edges, pointing to nodes n_1, n_2, with $n_1 \neq n_2$. Using $clause(n_1)$ and $clause(n_2)$ as premises and $clause(n)$ as conclusion must yield a correct instance of proof rule $rule(n)$.*

It is reasonable to assume that DPLL(T)-style SMT solvers produce proofs that conform to this format, or can easily be converted to this format.

2.2.5 Certificates

A significant part of this thesis is concerned with computing *certificates* for quantified formulas. In particular, we focus on certificates for existentially quantified propositional variables after an outer universal quantifier. Informally speaking, a certificate — sometimes also called a *witness* — is a (verifiable) "proof" that a quantified formula is valid.

Definition 16 — Certificate
Let $\overline{x}$ be a vector of variables, let $\overline{c}$ be a vector of propositional variables, and let

$\Phi(\overline{x},\overline{c})$ *be a formula in first-order logic,*[7] *such that* $\forall\overline{x}\,.\,\exists\overline{c}\,.\,\Phi$ *is a valid, closed formula. Then a* certificate *is a function* $\sigma : \mathbb{D}^{|\overline{x}|} \mapsto \mathbb{B}^{|\overline{c}|}$ *such that* $\forall\overline{x}\,.\,\Phi\big(\overline{x},\sigma(\overline{x})\big)$ *is valid.*

2.3 Craig Interpolation

In 1957, William Craig proved an interesting property about formulas in first-order logic [Cra57]. Suppose we have two CNF formulas ϕ and ψ, such that their conjunction $\phi \wedge \psi$ is unsatisfiable. Then there exists a formula χ, called an *interpolant between* ϕ *and* ψ that satisfies the following properties.

1. ϕ implies χ.
2. ψ implies $\neg\chi$.
3. The non-logical symbols that appear in χ also appear in both ϕ and ψ.

This is usually referred to as *Craig's interpolation theorem.*

The formulas ϕ and ψ will be called the *partitions* of formula $\phi \wedge \psi$. We will associate with each partition a unique *"color"*, and we will say that all symbols occurring in a partition (but not in another partition) have this color. Symbols that occur in more than one partition will be called *colorless* or *global*. A term or formula is *colorable* iff it contains only symbols of one color and colorless symbols. Terms or formulas that contain symbols of more than one color are called *non-colorable*.

An interpolant can easily be computed from a refutation proof for $\phi \wedge \psi$ [McM05]. Every proof node is annotated with a so-called *partial interpolant.*

Definition 17 — Partial Interpolant
Let ϕ *and* ψ *be CNF formulas such that* $\phi \wedge \psi$ *is unsatisfiable. Let* n *be a node in the refutation proof of* $\phi \wedge \psi$*. Let* $C = clause(n)$*. A formula* χ_p *is a partial interpolant for* C *between* ϕ *and* ψ *if* ϕ *implies* $C|_\phi \vee \chi_p$*,* ψ *implies* $C|_\psi \vee \neg\chi_p$*,* $\chi_p \preceq \phi$*, and* $\chi_p \preceq \psi$*.*

Lemma 3
If χ *is a partial interpolant for* $C \equiv \bot$ *between* ϕ *and* ψ*, then* χ *is an interpolant between* ϕ *and* ψ*.*

In Figure 2.3, we present interpolating proof rules, following Pudlák's interpolation system [Pud97].[8] In a refutation proof for $\phi \wedge \psi$, these rules annotate (in square brackets) each conclusion with a partial interpolant for the conclusion. Rules iHyp-ϕ and iHyp-ψ are used at leaf nodes that have clauses from ϕ and ψ respectively. Rules iAxi-ϕ and iAxi-ψ are used for leaves with theory lemmata, whose symbols are a subset of the symbols in ϕ and ψ respectively. Note that

[7] Note that Φ does not necessarily have to be quantifier-free.

[8] Pudlák's system is not the only interpolation system. A different system was proposed by McMillan [McM05]. In fact D'Silva et al. [DKPW10] show that many different interpolation systems can be constructed. We focus on Pudláks system because it is symmetric with respect to the partitions.

$$\textsc{iHyp-}\phi\ \frac{}{C[\bot]}\ C \in \phi \qquad \textsc{iHyp-}\psi\ \frac{}{C[\top]}\ C \in \psi$$

$$\textsc{iAxi-}\phi\ \frac{}{C[\bot]}\ C \preceq \phi, \vdash_{\mathcal{T}} C \qquad \textsc{iAxi-}\psi\ \frac{}{C[\top]}\ C \preceq \psi, \vdash_{\mathcal{T}} C$$

$$\textsc{iRes}\ \frac{a \vee C[I_C] \qquad \neg a \vee D[I_D]}{C \vee D[(a \vee I_C) \wedge (\neg a \vee I_D)]}\ a \preceq \phi, a \preceq \psi$$

$$\textsc{iRes-}\phi\ \frac{a \vee C[I_C] \qquad \neg a \vee D[ID]}{C \vee D[I_C \vee I_D]}\ a \preceq \phi, a \not\preceq \psi$$

$$\textsc{iRes-}\psi\ \frac{a \vee C[I_C] \qquad \neg a \vee D[I_D]}{C \vee D[I_C \wedge I_D]}\ a \not\preceq \phi, a \preceq \psi$$

Figure 2.3: Interpolating proof rules.

these rules cannot deal with theory lemmata that are non-colorable. That is, these rules assume that the refutation proof is colorable.

Definition 18 — Colorable Proof
A refutation proof for $\phi \wedge \psi$ is colorable *if every leaf $n \in N_L$ of the proof is colorable. That is, $symb(clause(n)) \subseteq symb(\phi)$ or $symb(clause(n)) \subseteq Symb(\psi)$.*

2.4 Burch-Dill Paradigm

Burch and Dill [BD94] have suggested an interesting paradigm to verify controllers of pipelined processors. The main idea is to compare the externally visible behavior of a non-pipelined reference design with that of the pipelined design. This is illustrated in Figure 2.4. Both the pipelined and the reference design start from the same initial state S_0, representing the contents of the memory, the register file, etc. We assume that the pipeline is initially empty. Now we run an arbitrary program on the reference design. Every step updates the state of the design, leading to a sequence of states $S_0, R_1, R_2, \ldots, R_n$, illustrated in Figure 2.4 by red arrows. Now we run the same program on the pipelined design, leading to a sequence of states $S_0, P_1, P_2, \ldots, P_n$ (blue arrows in Figure 2.4). After reaching state P_n, we flush the pipeline and compare the resulting state to R_n.

Definition 19 — Burch-Dill Equivalence
Let R_n be the state of a non-pipelined reference design, obtained by running a program, starting from an initial state S_0. Let P_n be the state of a pipelined design, obtained by running the same program from the same initial state S_0 and an empty pipeline. Then the pipelined design is Burch-Dill equivalent *to the non-pipelined reference design, if the externally visible components of the*

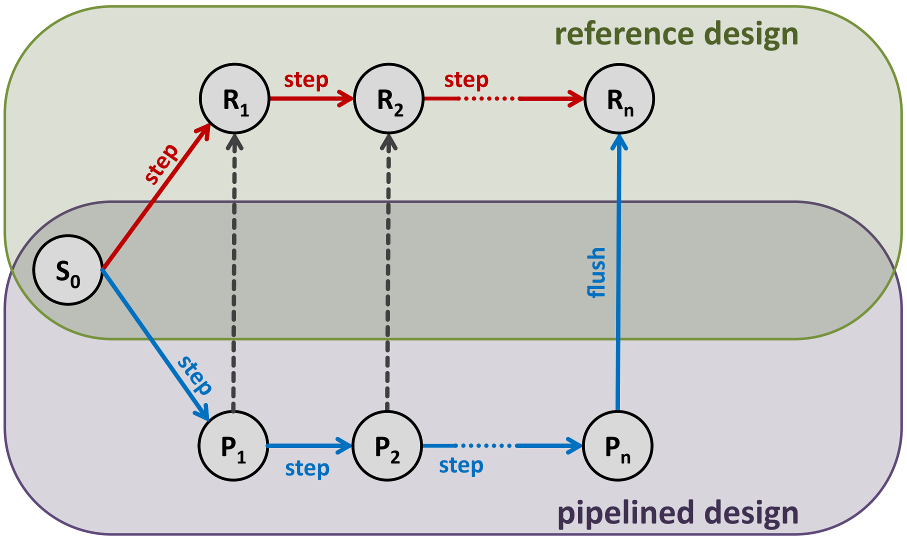


Figure 2.4: Burch-Dill equivalence between a pipelined design and a non-pipelined reference design.

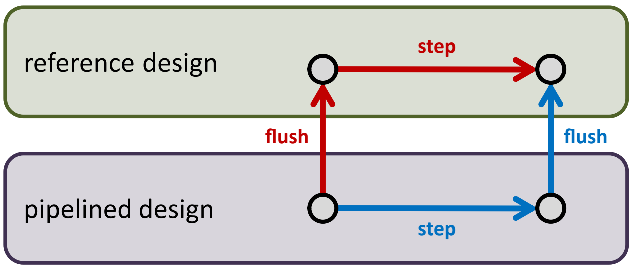


Figure 2.5: To show Burch-Dill equivalence, it suffices to show that the red and the blue path in this diagram are commutative.

state obtained by flushing the pipeline in state P_n are equal to their respective counterparts in state R_n (see Figure 2.4).

Proving Burch-Dill equivalence based on its definition would be rather difficult. One would have to consider all possible programs, of all possible lengths. Fortunately, Burch and Dill have shown an easier way [BD94]. Instead of starting in an initial state with an empty pipeline, let us consider an arbitrary state, where the pipeline is fully (or partially) filled. This state is represented by the lower left circle in Figure 2.5. Now we consider two possible paths. For the red path in Figure 2.5, we flush the pipeline and then perform one step in the non-pipelined reference design. For the blue path, we first perform one step in the pipelined design and then flush the pipeline.

Theorem 3
If, for an arbitrary initial (pipeline) state, the two paths shown in Figure 2.5 are commutative, that is, if we obtain the same resulting state from both paths, then the pipelined design and the reference design are Burch-Dill equivalent.

Proof
Flushing an empty pipeline has no effect on the state. Thus, we can "split" the state S_0 in Figure 2.4 into two copies; one for each design. When we do this, Figure 2.4 basically consists of multiple instances of Figure 2.5, strung together. (This is illustrated by the dashed arrows in Figure 2.4.) Since we assumed the paths in every instance to be commutative, we inductively conclude that also their concatenation is commutative. **Q. E. D.**

There are some more notable facts about the Burch-Dill paradigm. First of all, commutativity as shown in Figure 2.5 is a sufficient, but not a necessary condition for pipeline correctness. One example where this becomes evident is a pipeline that has *invariants*. An invariant is a (logical) statement about the state of the pipeline that always has to hold. States that violate an invariant should not be reachable from a valid initial state. However, the arbitrary start state considered in Figure 2.5 could violate an invariant. Thus, it might be possible that for such a start state, commutativity does not hold. If, however, an implementation ensures that such a state is never reached, it can still be correct. Burch and Dill [BD94] suggested to take invariants into account by only considering start states that satisfy the invariants.

Another important issue is *progress*. There are situations, where executing one step in the pipeline actually does not load a new instruction, for example, because stalling was necessary. In such a case, no instruction should be executed in the reference design either, to keep the outcomes equivalent. This will pose an additional challenge for the synthesis of `stall` signals, which was not present in verification settings. Details will be explained in Section 3.2.

3 Modeling

Declaration of Sources

This chapter is based on and reuses material from the following source, previously published by the author:

- [HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42.

References to this source are not always made explicit. In particular, definitions in Section 3.1 and the example in Section 3.2 are strongly based on the paper cited above.

In this section we will first define the language from which we build the specifications for our controller synthesis problems (Section 3.1). In short, our specifications will be formulas in a special fragment of the theory of arrays $\mathcal{T}_A$ as defined in Section 2.1.5. We will then show how to create a Burch-Dill style specification for a pipelined design, based on a simple toy example (Section 3.2).

3.1 Specification Language

The specifications for the controller synthesis problems we consider should be able to express the notion that *"for all inputs/states, there exist values for the* *control signals, such that (for all values of some auxiliary variables) a correctness criterion ϕ is fulfilled."* Formally, our specifications will be a fragment of the theory of arrays $\mathcal{T}_A$.[9] We will first define the formulas that we will use as the suffix to the $\forall$-$\exists$-$\forall$-quantifier structure[10] mentioned above.

Definition 20 — Fragment $\mathcal{S}$
Let $\mathcal{R}$ be a set of array variables, let $\mathcal{X}$ be a set of variables ranging over $\mathbb{D}$, let $\mathcal{B}$ be a set of propositional variables ranging over $\mathbb{B}$, let $\mathcal{F}$ be a set of function symbols, and let $\mathcal{P}$ be a set of predicate symbols. Let $\bar{i} = \{i_1, i_2, \ldots, i_k\}$ be a finite subset of $\mathcal{X}$, and let $\forall\bar{i}$ be a shorthand notation for $\forall i_1 . \forall i_2 \ldots . \forall i_k$. Then $\mathcal{S}$ is the fragment of the theory of arrays $\mathcal{T}_A$, whose formulas conform to the following grammar:

$$
\begin{array}{rcl}
\mathsf{formula} & := & \mathsf{array\_property} \mid \mathsf{term} = \mathsf{term} \mid \mathsf{array\_term} = \mathsf{array\_term} \mid \\
 & & \mathsf{predicate\_symbol}(\mathsf{term}*) \mid \mathsf{propositional\_var} \mid \top \mid \bot \mid \\
 & & \neg\mathsf{formula} \mid \mathsf{formula} \wedge \mathsf{formula} \mid \mathsf{formula} \vee \mathsf{formula} \mid \\
 & & \mathsf{formula} \rightarrow \mathsf{formula} \\
\mathsf{array\_term} & := & \mathsf{array\_var} \mid \mathsf{array\_var}\langle \mathsf{term} \lhd \mathsf{term} \rangle \\
\mathsf{term} & := & \mathsf{domain\_var} \mid \mathsf{function\_symbol}(\mathsf{term}*) \mid \mathsf{array\_var}[\mathsf{term}] \\
\mathsf{array\_property} & := & \forall\bar{i} \,.\, \mathsf{iguard}_{\bar{i}} \;\rightarrow\; \mathsf{valconstr}_{\bar{i}} \\
\mathsf{iguard}_{\bar{i}} & := & \mathsf{iguard}_{\bar{i}} \wedge \mathsf{iguard}_{\bar{i}} \mid \mathsf{iguard}_{\bar{i}} \vee \mathsf{iguard}_{\bar{i}} \mid \mathsf{atom}_{\bar{i}} \\
\mathsf{atom}_{\bar{i}} & := & \top \mid \mathsf{ivar}_{\bar{i}} = \mathsf{ivar}_{\bar{i}} \mid \mathsf{evar}_{\bar{i}} \neq \mathsf{ivar}_{\bar{i}} \mid \mathsf{ivar}_{\bar{i}} \neq \mathsf{evar}_{\bar{i}} \\
\mathsf{ivar}_{\bar{i}} & := & \mathsf{evar}_{\bar{i}} \mid \mathsf{uvar}_{\bar{i}} \\
\mathsf{uvar}_{\bar{i}} & := & \text{any i} \in \bar{i} \\
\mathsf{evar}_{\bar{i}} & := & \mathsf{function\_symbol}(\mathsf{evar}_{\bar{i}}*) \mid \\
 & & \text{any i} \in \mathcal{X} \setminus \bar{i} \\
\mathsf{valconstr}_{\bar{i}} & := & \mathsf{array\_var}[\mathsf{ivar}_{\bar{i}}] = \mathsf{array\_var}[\mathsf{ivar}_{\bar{i}}] \mid \\
 & & \mathsf{array\_var}[\mathsf{ivar}_{\bar{i}}] = \mathsf{evar}_{\bar{i}} \mid \\
 & & \neg\mathsf{valconstr}_{\bar{i}} \mid \mathsf{valconstr}_{\bar{i}} \wedge \mathsf{valconstr}_{\bar{i}} \mid \\
 & & \mathsf{valconstr}_{\bar{i}} \vee \mathsf{valconstr}_{\bar{i}} \mid \mathsf{valconstr}_{\bar{i}} \rightarrow \mathsf{valconstr}_{\bar{i}} \\
\mathsf{function\_symbol} & := & \text{any } f \in \mathcal{F} \\
\mathsf{predicate\_symbol} & := & \text{any } P \in \mathcal{P} \\
\mathsf{array\_var} & := & \text{any } \mathsf{R} \in \mathcal{R} \\
\mathsf{domain\_var} & := & \text{any } x \in \mathcal{X} \\
\mathsf{propositional\_var} & := & \text{any } b \in \mathcal{B}
\end{array}
$$

[9]Note that, unlike some literature sources, we have defined $\mathcal{T}_A$ to include uninterpreted functions and predicates.

[10]We will later generalize this to quantifier prefixes with an arbitrary number of alternations of $\forall$ and $\exists$ quantifiers. This will be particularly interesting with respect to the modular SMT solving approach that we present in Section 5.4.

Note that the fragment $\mathcal{S}$ is a subset of the array property fragment $\mathcal{T}_A^{\mathrm{p}}$ that adds some additional syntactic restrictions, such as ensuring that array-reads and array-writes are only done on actual array terms.

Definition 21 — Specification Language $\mathcal{S}^{\mathbf{Q}}$
Let Φ be a formula in $\mathcal{S}$. Let vars$(\Phi) = \overline{x} \cup \overline{c} \cup \overline{x}'$ be the set of variables occurring in Φ, partitioned into $\overline{x}$, $\overline{c}$, and $\overline{x}'$, such that $\overline{c}$ contains only propositional variables, and $\overline{x}$ and $\overline{x}'$ contain variables of arbitrary type. Then the specification language $\mathcal{S}^{Q}$ *is the set of all formulas of the form*

$$\forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi. \tag{3.1}$$

Intuitively, $\overline{x}$ is the set of variables that describe the state and/or inputs of the system for which we want to find a controller. The variables in $\overline{c}$ represent the control signals for which we want to find an implementation. And $\overline{x}'$ is a set of auxiliary variables. The quantifier structure shown in Equation 3.1 makes sure that the control signals can depend on the values of variables in $\overline{x}$. However, they must be independent of the values of any variable in $\overline{x}'$.

We also consider a generalized version of $\mathcal{S}^{\mathrm{Q}}$, where we allow an arbitrary number of quantifier alternations. However, we still insist that all existential quantifiers are over propositional variables only.

Definition 22 — Generalized Specification Language $\mathcal{S}^{\mathbf{Q}^+}$
Let Φ be a formula in $\mathcal{S}$. Let vars$(\Phi) = \overline{x} \cup \overline{x}' \cup \overline{x}''' \cup \dots \cup \overline{c} \cup \overline{c}' \cup \overline{c}'' \cup \dots$ be the set of variables occurring in Φ, partitioned into $\overline{x}, \overline{x}', \overline{x}'', \dots,$ and $\overline{c}, \overline{c}', \overline{c}'', \dots,$ such that $\overline{c}, \overline{c}', \overline{c}'', \dots$ contain only propositional variables, and $\overline{x}, \overline{x}', \overline{x}'', \dots$ contain variables of arbitrary type. Then the generalized specification language $\mathcal{S}^{Q^+}$ *is the set of all formulas of the form*

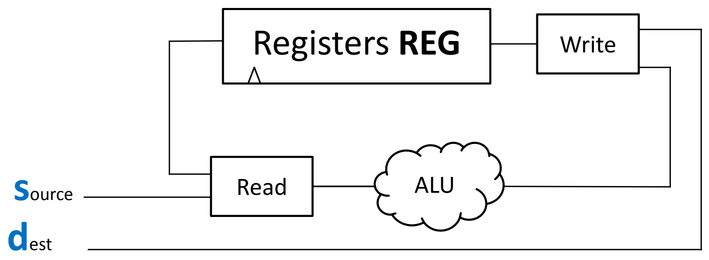
$$\Phi^{Q^+} = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \exists \overline{c}' \,.\, \forall \overline{x}'' \,.\, \dots\ \Phi. \tag{3.2}$$

With this generalization it is, for example, possible to express that certain control signals can depend on more variables than others: In Equation 3.2, signals in $\overline{c}$ can only depend on $\overline{x}$, whereas signals in $\overline{c}'$ can depend on $\overline{x}$ and $\overline{x}'$. It should be noted, however, that all the examples we present in this thesis are based on specifications in $\mathcal{S}^{\mathrm{Q}}$, despite the fact that our methods for solving them can be generalized to $\mathcal{S}^{\mathrm{Q}^+}$ easily.

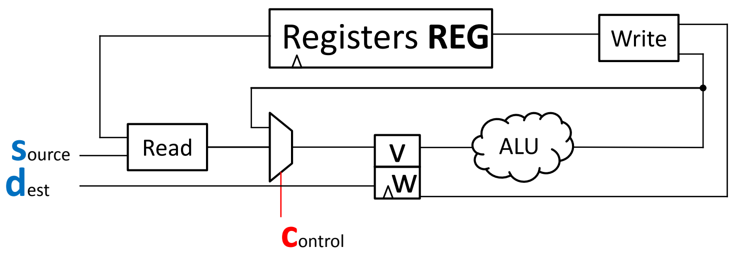
3.2 Creating a Specification

In this section we will show how to write a specification Φ^{Q}, which is a formula in $\mathcal{S}^{\mathrm{Q}}$, for a pipelined processor. The procedure is based on the Burch-Dill paradigm [BD94] (see Section 2.4). We will illustrate the necessary steps with a simple running example (see Figure 3.1). The generalizations to more complex pipelined designs are straightforward.

We assume that we have a (high-level) graphical representation of the pipelined design and the corresponding non-pipelined reference design available. In



(a) Non-pipelined reference design.



(b) Pipelined Design.

Figure 3.1: A simple example of a pipelined microprocessor-like design, and the corresponding non-pipelined reference design.

the pipelined design, some (Boolean) control signals — controlling for example forwarding of data values or stalling the pipeline — are not implemented, but marked for synthesis.

Example 2
In Figure 3.1, we present a sketch of an illustrating toy example, to which all the examples in this section refer. Despite its simplicity, this example features all the important building blocks of a microprocessor-like design. Let us first examine the non-pipelined reference design in Figure 3.1(a). The design has two inputs — s and d — representing a source and a destination address, respectively. The design contains a register file REG, *which is an address-based memory. That is, data words can be read from and written to the register file, using an address to select one specific register in the register file. This is symbolized by the* Read *and* Write *blocks in Figure 3.1(a). The output of the* Read *block is the current value of the register with address s. The* Write *block updates the register file so that in the next time step the register with address d will contain the value present at the* Write *block's input. The ALU block represents an arbitrary combinational function on data words.*

All in all, the design sketched in Figure 3.1(a) does the following. In every time step, the value of the register with address s is read. The function ALU is applied to this value and the result is stored to the register with address d.

This circuit is similar to a microprocessor (although heavily simplified), which also reads operands from memory, processes them, and writes the result(s) back to memory. Despite its simplicity, this circuit will suffice to demonstrate the concepts of our modeling approach.

Figure 3.1(b) shows a pipelined version of the design in Figure 3.1(a), with one stage of pipeline registers v and w. In this design, the value read from the register file is stored to pipeline register v in the first step. The destination address belonging to this value is stored alongside in pipeline register w. In a second time step, the function ALU is applied to the value of v and written back to the register with address w.

This design suffers from the following concurrency-issue. Suppose that the first part of the pipeline wants to read a value from the address to which the second part has to write to in the same time step. In this case, the register file REG *still contains an old value. To address this problem, we add a multiplexer that provides the choice of either reading from the register file, or reading a forwarded value from the second part of the circuit. The choice is made by a (Boolean) control signal c.*

Assume that when $c = \top$ the forwarded data is read, and when $c = \bot$ data from the register file is read. It is easy to see that in this simple example $c := (s = w)$ is a valid implementation of the controller. Setting $c = \top$ whenever $s = w$ will ensure that (after a final flush of the pipeline) the pipelined version of the design will leave the register file in exactly the same state as the non-pipelined version would have, when given the same sequence of inputs.

Let Φ be the $\mathcal{S}$-formula that comes after the initial $\forall$-$\exists$-$\forall$-quantifier prefix of Φ^{Q}. The formula Φ is supposed to express Burch-Dill equivalence between the pipelined and the reference design. Thus, Φ consists of three parts. The first part is a formula, describing the behavior of the design from the (arbitrary) initial state along the *"flush-step"* path (red arrows in Figure 2.5). Analogously, the second part describes the behavior along the *"step-flush"* path (blue arrows in Figure 2.5). The third part asserts that the two final states resulting from the two other parts are in fact equivalent.

Before we show how to obtain these three parts, we first need to discuss how to model the different parts of a design in an $\mathcal{S}$-formula. To model address-based memory (such as register files), we use arrays with uninterpreted indices. One advantage of this approach is that the actual size of the memory (number of addresses) and its width (bits per word) do not matter. Our considerations and computations will be valid for any concrete values for size and width. Inputs and storage elements for single data words (such as pipeline registers) are modeled by domain variables ranging over an infinite or sufficiently large[11] domain $\mathbb{D}$. As an example for such a domain, consider the set $\mathbb{B}^n$, corresponding to n-bit data words of a processor. Again, our approach is independent of the concrete choice for $\mathbb{D}$. Combinational datapath elements, such as the function ALU in our example, are modeled by uninterpreted functions with appropriate arity. This abstraction completely disregards the semantics of the operations implemented

[11] We will discuss the precise meaning of *sufficiently large* in Section 4.2.1.

by the datapath elements and focuses solely on functional consistency. That is, the only important fact we care about is that — given a particular input — the datapath element will always return the same output. We use primes to denote time steps. That is, v' is the value of v after one time step, v'' is the value after two time steps, etc.

To obtain the first part of Φ, we have to model the flushing of the pipeline from an arbitrary current state. This poses **a problem that did not exist in verification approaches** based on the Burch-Dill paradigm. Flushing a multi-stage pipeline requires several time steps. To model each of these time steps, (symbolic) values for the control signals are required. In verification, this is not a problem, as all control signals have been implemented and can thus be replaced with symbolic expressions representing their implementations. In our synthesis setting, however, the implementation of the control signals is not known. Since their concrete value in a real execution can be different in every time step of flushing, we would have to introduce a fresh propositional variable per time step. Suppose we have two such variables c and c'. Eventually, we want to synthesize a function $f_c(\overline{x})$ that computes the value of c depending on the inputs/state modeled by $\overline{x}$. However, now we would have to ensure that the values of c' correspond to the values of a function $f_{c'}(\overline{x}')$ and that whenever $\overline{x} = \overline{x}'$ we have that $f_c(\overline{x}) = f_{c'}(\overline{x}')$. This consistency cannot be expressed in $\mathcal{S}^{\mathbb{Q}}$. To solve this problem, we use an approach resembling the *completion functions* presented by Hosabettu et al. [HGS03]. Instead of actually flushing the pipeline, we model the effect that completing an unfinished pipeline stage would have on the observable parts of the design. After completing one pipeline stage, we consider this stage removed from the circuit and continue completing remaining stages, if any. Unfortunately, for more complex designs, for example, pipelines that do out-of-order and/or speculative execution, modeling the effect of completing the pipeline might be more difficult and not necessarily possible in a stage-by-stage way. It is up to the pipeline's designer to adequately model the completion of the pipeline.

Example 3
For the design in Figure 3.1(b), completion is achieved by updating $\mathsf{REG}[w]$ *to the value* $ALU(v)$*. In our example, there are no more stages to complete, thus flushing this design is modeled by the following formula, where the "ci" subscript symbolizes that we are in the path were we first* <u>c</u>omplete *the pipeline and then perform one* <u>i</u>nstruction *in the reference design.*

$$\mathsf{REG}'_{\mathsf{ci}} = \mathsf{REG}\langle w \lhd ALU(v)\rangle \tag{3.3}$$

After completing the pipeline, we model one step in the non-pipelined reference design. We obtain the first part of our equivalence criterion, which we call Φ_{ci}, by forming the conjunction of the equations obtained from modeling completion and one step of the reference design.

Example 4
One step of the non-pipelined circuit is modeled by the following equation:

$$\mathsf{REG}''_{\mathsf{ci}} = \mathsf{REG}'_{\mathsf{ci}}\langle d \lhd ALU(\mathsf{REG}'_{\mathsf{ci}}[s])\rangle \tag{3.4}$$

The conjunction of Equations 3.3 and 3.4 forms Φ_{ci}.

The second part of Φ, which we denote Φ_{sc}, is obtained by modeling one ***step*** in the pipelined design, followed by ***completion***. This is done similarly to the "*ci*" part. Since Φ_{ci} and Φ_{sc} describe how the elements of the design are *updated* during execution, we define

$$\Phi_{upd} = \Phi_{ci} \wedge \Phi_{sc}. \tag{3.5}$$

Example 5
During one step, the value of register v is fed to the function ALU, the result of which is then written to address w of the register file. Also, the pipeline registers v and w are updated during the step operation. The new value of w is copied from input d. The new value of v depends on the value of control signal c. Overall, we obtain the following formula to model one step of the pipelined design:

$$\begin{gathered}\mathsf{REG}'_{\mathsf{sc}} = \mathsf{REG}\langle w \lhd ALU(v)\rangle \;\wedge\; (w' = d) \wedge \\ \big((c \wedge v' = ALU(v)) \vee (\neg c \wedge v' = REG[s])\big)\end{gathered} \tag{3.6}$$

Completion of the pipeline after this step works analogously to the complete-instruction path. We obtain the following equation:

$$\mathsf{REG}''_{\mathsf{sc}} = \mathsf{REG}'_{\mathsf{sc}}\langle w' \lhd ALU(v')\rangle \tag{3.7}$$

We form the conjunction of Equations 3.6 and 3.7 to obtain Φ_{sc}.

The third part of Φ — which we call Φ_{equiv} — is supposed to ensure that the states obtained by the update rules of Φ_{ci} and Φ_{sc} are identical.

Example 6
In our example, the observable state of the design are the values of the register file. Thus, we obtain

$$\Phi_{equiv} := (\mathsf{REG}''_{\mathsf{ci}} = \mathsf{REG}''_{\mathsf{sc}}) \tag{3.8}$$

To complete our specification Φ^{Q}, we now need to partition all the variables in Φ into sets $\overline{x}$, $\overline{x}'$, and $\overline{c}$. This is done in the following way. All variables that represent (Boolean) control signals for which we want to synthesize an implementation are put into the set $\overline{c}$. Due to the order of the quantifiers (see Equation 3.1), the set $\overline{x}$ should contain all variables on which the value of the control signals can depend. All auxiliary variables, which should not influence the value of the control signals, are put in $\overline{x}'$.

Example 7
For the design in Figure 3.1, we only have one control signal. Thus, $\overline{c} = \{c\}$.

For deciding the value of the control signal, an implementation could look at the current values of the inputs s and d, the pipeline registers v and w, and the register file REG*. Thus, we get $\overline{x} = \{s, d, v, w, \mathsf{REG}\}$. Finally, all other variables were just introduced as auxiliary variables to formulate Burch-Dill equivalence. This gives us $\overline{x}' = \{\mathsf{REG}'_{\mathsf{ci}}, \mathsf{REG}''_{\mathsf{ci}}, \mathsf{REG}'_{\mathsf{sc}}, \mathsf{REG}''_{\mathsf{sc}}, v', w'\}$.*

Now, we set $\Phi = \Phi_{upd} \rightarrow \Phi_{equiv}$. This gives us our final specification:

$$\Phi^{\mathrm{Q}} = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi. \tag{3.9}$$

Intuitively, Equation 3.9 reads: *"For all inputs and states, it holds that there exist (Boolean) values for the control signals, such that for all values of the auxiliary variables that follow the update rules of the design, Burch-Dill equivalence holds."*

Note that all variables in Φ^{Q} are bound by one of the quantifiers. This means that Φ^{Q} is either equivalent to $\top$ (in which case we will call it *valid*), or equivalent to $\bot$. The latter means that it is *not* possible to find control values that ensure correctness for all possible states/inputs; the controller synthesis problem is *unrealizable*. A possible reason for unrealizability is that the datapath does not feature enough options to ensure correct behavior.

Example 8
If we remove the multiplexer for data forwarding from the pipelined circuit in Figure 3.1(b), or if we connect its inputs to wrong wires (e.g. to the output of the w register instead of the output of the ALU function), the controller synthesis problem becomes unrealizable.

In situations like the one in Example 8, it is easy to obtain a *counterexample*; that is, a situation (state and inputs) for which it is impossible to find control values so that the specification is fulfilled. This data is helpful to find out what needs to be added to the datapath in order to make the controller synthesis problem realizable.

Another possible reason for unrealizability is that the specification is missing invariants. As mentioned in Section 2.4, it might be necessary to restrict the scope of checking Burch-Dill equivalence to initial states that satisfy all invariants of the pipeline design. In our setting, we can do this as follows. Let Φ_{inv} be a formula that expresses the invariants of the design, and let $\Phi = \Phi_{upd} \rightarrow \Phi_{equiv}$ be the formula that expresses Burch-Dill equivalence (as outlined above), but does not consider invariants. To take the invariants into account, we use $\tilde{\Phi} = \Phi_{inv} \rightarrow \Phi$ instead of just Φ in Φ^{Q}.

There is yet another important issue where synthesis differs from a verification setting: **ensuring progress**. As we mentioned in Section 2.4, there are situations where a pipeline does not load a new instruction. The Burch-Dill style specification must be written in such a way that in such situations also the reference design does not execute an instruction. This causes a problem when stalling is controlled by a signal that should be synthesized: Always stalling the pipeline is a way to "trivially" satisfy the specification. This is because in case of a stall, the specification just compares whether or not the result of completion ("ci" path) is equal to the result of completion ("sc" path). We propose to solve this problem in the following way. The designer has to provide a formula Φ_{empty} which characterizes an empty pipeline; that is, a pipeline state in which completion would not have any effect on the memory. Obviously, in such a state issuing a stall signal is not meaningful. Thus, we amend our update rules Φ_{upd} by a conjunct $\Phi_{empty} \rightarrow \neg c_{stall}$. This ensures that eventually a new instruction will be loaded into the pipeline.

To summarize, we have shown how to create a Burch-Dill style specification for a pipeline controller synthesis problem, based on a fragment of $\mathcal{T}_A$ that we introduced. In the next chapter(s), we will focus on deciding whether or not formulas in $\mathcal{S}^{\mathrm{Q}}$ are valid. Furthermore, for valid formulas we will show different ways to extract certificates for the variables in $\bar{c}$. It should be noted that the methods we are about to present are not limited to the synthesis of pipeline controllers. Instead, they can be applied to any synthesis problem that can be stated in $\mathcal{S}^{\mathrm{Q}}$.

4 Decidability and Complexity

Declaration of Sources

This chapter is based on and reuses material from the following sources, previously published by the author:

- [HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42.
- [EKH12] Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 91–100. IEEE, 2012

References to this sources are not always made explicit. In particular, the proofs in Sections 4.1, 4.2, and 4.3, as well as the reductions in Section 4.2 are strongly based on [HB11].

In Chapter 3, we introduced the class $\mathcal{S}^{\mathrm{Q}}$ of formulas which serve as specifications for the controller synthesis problems we consider. In this chapter, we will first show that the question of validity of formulas in $\mathcal{S}^{\mathrm{Q}}$ is decidable. The proof is constructive, that is, we will show an actual method to solve the decision problem. We will then show how the standard eager-encoding techniques (see Section 2.2.1) can be employed to reduce the problem to the propositional domain. Based on this reduction, we will then show that the problem is $coNP^{NP}$-complete, and that the generalized version of the problem, deciding validity of formulas in $\mathcal{S}^{Q^+}$, is PSPACE-complete.

4.1 Decidability

Theorem 4 — Decidability of $\mathcal{S}^{\mathbf{Q}}$
Let $\Phi^Q = \forall\overline{x}\,.\,\exists\overline{c}\,.\forall\overline{x}'\,.\,\Phi$ be a formula in $\mathcal{S}^Q$. Then the question of whether or not Φ^Q is valid *is decidable.*

Proof
Let $n = |\overline{c}|$ be number of variables in $\overline{c}$. As the variables in $\overline{c}$ are, by definition, all propositional, we can replace the existential quantifier by 2^n disjuncts.

$$\forall\overline{x}\,.\,\exists\overline{c}\,.\,\forall\overline{x}'\,.\,\Phi$$
$$\Updownarrow$$
$$\forall\overline{x}\,.\,\Big(\forall\overline{x}'\,.\,\Phi_{\overline{c}=00\ldots00}\ \vee\ \forall\overline{x}'\,.\,\Phi_{\overline{c}=00\ldots01}\ \vee\ldots\vee\ \forall\overline{x}'\,.\,\Phi_{\overline{c}=11\ldots11}\Big), \tag{4.1}$$

where $\Phi_{\overline{c}=\ldots}$ stands for a copy of Φ in which all instances of the variables in $\overline{c}$ have been replaced by either $\top$ or $\bot$, as indicated by the given bit string. For each of the disjuncts, we now rename the variables in $\overline{x}'$ to obtain

$$\forall\overline{x}\,.\,\Big(\forall\overline{x}'_{00\ldots00}\,.\,\Phi_{\overline{c}=00\ldots00}\ \vee\ \forall\overline{x}'_{00\ldots01}\,.\,\Phi_{\overline{c}=00\ldots01}\ \vee\ldots$$
$$\ldots\vee\ \forall\overline{x}'_{11\ldots11}\,.\,\Phi_{\overline{c}=11\ldots11}\Big). \tag{4.2}$$

This ensures that none of the variables in any of the $\overline{x}'_i$ occur anywhere outside the scope of their quantifier. Thus, we can switch the order of quantification and disjunction. This gives us

$$\forall\overline{x}\,.\,\forall\overline{x}'_{00\ldots00}\,.\,\forall\overline{x}'_{00\ldots01}\,.\ldots.\,\forall\overline{x}'_{11\ldots11}\,.\,\Big(\Phi_{\overline{c}=00\ldots00}\ \vee\ \Phi_{\overline{c}=00\ldots01}\ \vee\ldots$$
$$\ldots\vee\ \Phi_{\overline{c}=11\ldots11}\Big). \tag{4.3}$$

Now, we put a double negation in front, and push one of the negations over the quantifiers to obtain

$$\neg\exists\overline{x}\,.\,\exists\overline{x}'_{00\ldots00}\,.\,\exists\overline{x}'_{00\ldots01}\,.\ldots.\,\exists\overline{x}'_{11\ldots11}\,.\,\neg\Big(\Phi_{\overline{c}=00\ldots00}\ \vee$$
$$\Phi_{\overline{c}=00\ldots01}\ \vee\ldots\vee\ \Phi_{\overline{c}=11\ldots11}\Big). \tag{4.4}$$

By pushing the negation further inside, we get

$$\neg\exists\overline{x}\,.\,\exists\overline{x}'_{00\ldots00}\,.\,\exists\overline{x}'_{00\ldots01}\,.\ldots.\,\exists\overline{x}'_{11\ldots11}\,.\,\Big(\neg\Phi_{\overline{c}=00\ldots00}\ \wedge$$
$$\neg\Phi_{\overline{c}=00\ldots01}\ \wedge\ldots\wedge\ \neg\Phi_{\overline{c}=11\ldots11}\Big). \tag{4.5}$$

Let $\varphi = \varphi_{00\ldots00} \wedge \ldots \wedge \varphi_{11\ldots11}$, where each $\varphi_i = \neg\Phi_{\overline{c}=i}$. Note that φ is a formula in $\mathcal{S}$, which is a subset of $\mathcal{T}_A^{\mathrm{p}}$. Clearly, the formula in Equation 4.5 — which is equivalent to the formula in Equation 4.2 — is valid if and only if φ is not satisfiable (modulo its theory). Satisfiability of formulas in $\mathcal{S}$ is decidable, because satisfiability of formulas in the superset $\mathcal{T}_A^{\mathrm{p}}$ is decidable [KS08]. Thus, we conclude that validity for formulas in $\mathcal{S}^{\mathrm{Q}}$ is decidable. **Q. E. D.**

Definition 23 — *expand_negate*()
Let $\Phi^Q = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$ be a formula in $\mathcal{S}^Q$. Then expand_negate(Φ) denotes the formula $\varphi = \varphi_{00\ldots00} \wedge \ldots \wedge \varphi_{11\ldots11}$ obtained by the expansion, renaming, and negation as outlined in the proof of Theorem 4.

Lemma 4
A formula $\Phi^Q = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$ in $\mathcal{S}^Q$ is valid if and only if $\varphi = expand\_negate(\Phi)$ is unsatisfiable.

Proof
See proof of Theorem 4. **Q. E. D.**

Note that the definition of *expand_negate*(), as well as Lemma 4 can be generalized to $\mathcal{S}^{\mathrm{Q}^+}$, that is, an arbitrary number of quantifier alternations, in a straightforward way.

4.2 Reduction to Propositional Logic

Now that we have established that validity for formulas in $\mathcal{S}^{\mathrm{Q}}$ is decidable, we focus on how to compute certificates for the existentially quantified Boolean control signals. In this section, we will describe a first, somewhat naive approach which is based on a reduction to propositional logic. We will perform three validity-preserving reductions, based on the standard eager-encoding procedure for $\mathcal{T}_A^{\mathrm{p}}$ (see Section 2.2.1). We will also provide a proof of correctness for each of the reduction steps. All proofs share the same basic structure, which we will explain first. Finally, we will show how to extract the certificates from the propositional formula. We will continue to make use of the running example introduced in Section 3.2 to illustrate the reduction steps.

4.2.1 Structure of Proofs

The proofs for each of the reduction steps all share the same basic structure. In each case, we want to prove equivalence between two quantified formulas, with the same quantifier structure: Universal quantification, followed by Boolean existential quantification, followed again by universal quantification. Let the two formulas be $\forall \overline{a} \,.\, \exists \overline{b} \,.\, \forall \overline{c} \,.\, \varphi$ and $\forall \overline{x} \,.\, \exists \overline{y} \,.\, \forall \overline{z} \,.\, \psi$. The proof that validity of the first formula implies validity of the second proceeds as sketched by the following equation.

$$\begin{tikzcd}
\forall\overline{a}\,. & \exists\overline{b}\,. \arrow[d, "\beta"'] & \forall\overline{c}\,. & \varphi \\
\forall\overline{x}\,. \arrow[u, "\alpha"] & \exists\overline{y}\,. & \forall\overline{z}\,. \arrow[u, "\gamma"] & \psi
\end{tikzcd} \tag{4.6}$$

We start with an arbitrary interpretation $\llbracket\overline{x}\rrbracket$ for $\overline{x}$ in ψ. Next, we map these values to corresponding values $\llbracket\overline{a}\rrbracket$ for $\overline{a}$ in φ, according to a mapping $\alpha : \llbracket\overline{x}\rrbracket \mapsto \llbracket\overline{a}\rrbracket$. We then use the assumption of the validity of the first formula to find values $\left\llbracket\overline{b}\right\rrbracket$ for $\overline{b}$ such that for any arbitrary interpretation $\llbracket\overline{c}\rrbracket$ for $\overline{c}$ we have $\left\{\llbracket\overline{a}\rrbracket, \left\llbracket\overline{b}\right\rrbracket, \llbracket\overline{c}\rrbracket\right\} \vDash \varphi$. We use another mapping $\beta : \left\llbracket\overline{b}\right\rrbracket \mapsto \llbracket\overline{y}\rrbracket$ to find values $\llbracket\overline{y}\rrbracket$. Now we still need to prove that for all possible interpretations $\llbracket\overline{z}\rrbracket$ for $\overline{z}$ in ψ, it holds that $\{\llbracket\overline{x}\rrbracket, \llbracket\overline{y}\rrbracket, \llbracket\overline{z}\rrbracket\} \vDash \psi$. To do so, we arbitrarily choose an interpretation $\llbracket\overline{z}\rrbracket$ for $\overline{z}$ and use a mapping $\gamma : \llbracket\overline{z}\rrbracket \mapsto \llbracket\overline{c}\rrbracket$ to find corresponding values $\llbracket\overline{c}\rrbracket$. From the assumption of validity of the first formula we know that $\left\{\llbracket\overline{a}\rrbracket, \left\llbracket\overline{b}\right\rrbracket, \llbracket\overline{c}\rrbracket\right\} \vDash \varphi$. What remains to be shown is that this implies $\{\llbracket\overline{x}\rrbracket, \llbracket\overline{y}\rrbracket, \llbracket\overline{z}\rrbracket\} \vDash \psi$. This last step, as well as the mappings α and γ will be different for each of the proofs. The mapping β will always be the identity mapping, because in our case $\overline{b}$ and $\overline{y}$ are the same set of Boolean control variables. This also means that the control functions which we will eventually compute from the propositional formula will be valid implementations for the original specification. Within each proof, we will only present the mappings α and γ, and we will show that $\left\{\llbracket\overline{a}\rrbracket, \left\llbracket\overline{b}\right\rrbracket, \llbracket\overline{c}\rrbracket\right\} \vDash \varphi$ implies $\{\llbracket\overline{x}\rrbracket, \llbracket\overline{y}\rrbracket, \llbracket\overline{z}\rrbracket\} \vDash \psi$.

Some of the proofs will require that the domain $\llbracket\mathbb{D}\rrbracket$, from which the interpretations for first-order variables are chosen, is large enough so that all variables in a formula can be assigned pairwise different values. Henceforth, when we speak of a ***sufficiently large domain*** (with respect to a formula φ), we will mean that $\big|\llbracket\mathbb{D}\rrbracket\big| \geq \big|\mathit{vars}(\varphi)\big|$. Note that in particular any infinite domain is obviously sufficiently large with respect to any formula.

In the following sections, we show a validity preserving reduction from $\mathcal{S}^{\mathrm{Q}}$ to (quantified) propositional logic in three steps. The first step removes arrays, and yield a formula in $\mathcal{T}_U$ (see Section 4.2.2). The second step removes uninterpreted functions and predicates, resulting in a $\mathcal{T}_E$ formula (see Section 4.2.3). Finally, the third step eliminates equalities and gives us a (quantified) formula in propositional logic (see Section 4.2.4).

4.2.2 Reduction from $\mathcal{S}^{\mathrm{Q}}$ to $\mathcal{T}_U$

We start with a formula $\Phi^{\mathrm{Q}} = \forall\overline{x}\,.\,\exists\overline{c}\,.\,\forall\overline{x}'\,.\,\Phi$ in $\mathcal{S}^{\mathrm{Q}}$. For the reduction to $\mathcal{T}_U$ we take the part Φ (that is, the part of Φ^{Q} without the quantifier prefix), and proceed as in Section 2.2.1. First, all array writes are removed by introducing new variables and applying the write axioms.

Example 9
Consider the term $\mathsf{REG}\langle w \lhd ALU(v)\rangle$ *from Example 3. A new variable* REG' *is introduced, the term is replaced by* REG'*, and the conjunct*

$$\mathsf{REG}'[w] = ALU(v) \land \forall i \,.\, i \neq w \rightarrow \mathsf{REG}'[i] = \mathsf{REG}[i]$$

is added to the list of constraints.

The new variable will be added to $\overline{x}'$, because array-write expressions represent “future” values of an array on which the control signals should not depend. Next, we find the index set $\mathcal{I}$, as outlined in Section 2.2.1. Then, all universal quantifications over array indices are replaced by finite conjunctions over the index set $\mathcal{I}$. Finally, we replace all array reads by uninterpreted function instances. We will denote the set of all function symbols that replace array variables from $\overline{x}$ with $\overline{f}$, and the set of all function symbols replacing array variables from $\overline{x}'$ with $\overline{f}'$.

Example 10
Consider the specification from Example 7. The index set for this example is $\mathcal{I} = \{s, d, w, \lambda\}$*. We now replace all universal quantifications with conjunctions over the index set. For example, the subformula* $\forall i \,.\, i \neq w \rightarrow \mathsf{REG}'[i] = \mathsf{REG}[i]$ *from Example 9 is replaced by*

$$\bigwedge_{i \,\in\, \mathcal{I}} i \neq w \rightarrow \mathsf{REG}'[i] = \mathsf{REG}[i]. \tag{4.7}$$

Next, array reads are replaced by uninterpreted function instances. For example, $\mathsf{REG}'_{\mathsf{ci}}[w]$ *becomes* $REG'_{ci}(w)$*.*

Now we update $\overline{x}$ and $\overline{x}'$. We remove all array variables, as they are no longer present in the formula. The new domain variable λ is added to $\overline{x}'$. We will denote these updated sets $\overline{x}_U$ and $\overline{x}'_U$, respectively.

Theorem 5 — Reduction to $\mathcal{T}_U$
For a sufficiently large domain $\mathbb{D}$*, the formula* $\Phi^Q = \forall\overline{x} \,.\, \exists\overline{c} \,.\, \forall\overline{x}' \,.\, \Phi$ *is valid if and only if the formula*

$$\Phi_U^Q = \forall\overline{x}_U \,.\, \forall\overline{f} \,.\, \exists\overline{c} \,.\, \forall\overline{x}'_U \,.\, \forall\overline{f}' \,.\, \big(AC(\Phi) \rightarrow no\_array(\Phi)\big) \tag{4.8}$$

is valid.[12]

Proof
“$\Rightarrow$”: We assume validity of Φ^{Q}, and proof validity Φ_U^{Q}. Let $[\![\overline{x}_U]\!]$, $\left[\!\!\left[\overline{f}\right]\!\!\right]$, $[\![\overline{x}'_U]\!]$, $\left[\!\!\left[\overline{f}'\right]\!\!\right]$ be arbitrary interpretations for $\overline{x}_U$, $\overline{f}$ $\overline{x}'_U$, and $\overline{f}'$ respectively. Let α be a

[12]Strictly speaking, Φ_U^{Q} is not a $\mathcal{T}_U$-formula. It is not even a first-order formula, because of the quantifications over $\overline{f}$ and $\overline{f}'$. For our purposes, however, this detail is not relevant. We will thus treat Φ_U^{Q} as if it were a $\mathcal{T}_U$-formula.

mapping from an interpretation of function symbols in $\left[\!\left[\overline{f}\right]\!\right]$ to an interpretation of array variables $\overline{\mathsf{R}}$ as follows. For all $[\![f]\!] \in \left[\!\left[\overline{f}\right]\!\right]$ let $\alpha([\![f]\!])$ be an interpretation $[\![\mathsf{R}]\!]$ for an array variable R such that for all i it holds that $[\![f]\!]\,(i) = [\![\mathsf{R}]\!]\,[i]$. For all other (non-array) variables in $\overline{x}_U$, α is an identity mapping. Let γ be a mapping from $\left[\!\left[\overline{f}'\right]\!\right]$ to $\left[\!\left[\overline{\mathsf{R}}'\right]\!\right]$, defined analogously to α. For each array variable R in $\overline{x}$, let its interpretation $[\![\mathsf{R}]\!]$ in $[\![\overline{x}]\!]$ be $[\![\mathsf{R}]\!] = \alpha([\![f_R]\!])$, where f_R is the function symbol that replaces R in Φ^{Q}_U. For each array variable R' in $\overline{x}'$, let its interpretation $[\![\mathsf{R}']\!]$ in $[\![\overline{x}']\!]$ be $[\![\mathsf{R}']\!] = \gamma([\![f_{R'}]\!])$, where $f_{R'}$ is the function symbol that replaces R' in Φ^{Q}_U. Let $[\![\overline{c}]\!]$ be an interpretation for $\overline{c}$ such that for all possible interpretations $[\![\overline{x}']\!]$ we have that $\left\{[\![\overline{x}]\!], [\![\overline{c}]\!], [\![\overline{x}']\!]\right\} \vDash \Phi$. We have to show that this implies that $\left\{[\![\overline{x}_U]\!], \left[\!\left[\overline{f}\right]\!\right], [\![\overline{c}]\!], [\![\overline{x}'_U]\!], \left[\!\left[\overline{f}'\right]\!\right]\right\} \vDash \big(AC(\Phi) \rightarrow \mathit{no\_array}(\Phi)\big)$. It is easy to see that this is the case. Φ features universal quantification over indices, where $\mathit{no\_array}(\Phi)$ only features finite conjunctions. Any model that satisfies a universal quantification surely also satisfies a finite conjunction over the same variable. Note that since the right-hand side of the implication $\big(AC(\Phi) \rightarrow \mathit{no\_array}(\Phi)\big)$ is satisfied by $\left\{[\![\overline{x}_U]\!], \left[\!\left[\overline{f}\right]\!\right], [\![\overline{c}]\!], [\![\overline{x}'_U]\!], \left[\!\left[\overline{f}'\right]\!\right]\right\}$, it is irrelevant whether or not $\left\{[\![\overline{x}_U]\!], \left[\!\left[\overline{f}\right]\!\right], [\![\overline{c}]\!], [\![\overline{x}'_U]\!], \left[\!\left[\overline{f}'\right]\!\right]\right\} \vDash AC(\Phi)$. This concludes the proof in "$\Rightarrow$" direction.

"$\Leftarrow$": Without loss of generality, we assume that Φ contains no array-write expressions. If this is not the case we can easily obtain an equivalent formula without write expressions by applying the write axioms. Let $[\![\overline{x}]\!]$, $[\![\overline{x}']\!]$ be arbitrary interpretations for $\overline{x}$ and $\overline{x}'$ in Φ^{Q}. Let α and γ be mappings inverse to those of the "$\Rightarrow$" case. For each $[\![f]\!] \in \left[\!\left[\overline{f}\right]\!\right]$ and each $[\![f']\!] \in \left[\!\left[\overline{f}'\right]\!\right]$ let $[\![f]\!] = \alpha([\![\mathsf{R}]\!])$, and $[\![f']\!] = \gamma([\![\mathsf{R}']\!])$, where R and R' are the array variables replaced by the functions symbols f and f', respectively. For each $[\![x]\!] \in [\![\overline{x}]\!]$ and each $[\![x']\!] \in [\![\overline{x}']\!]$ let $[\![x_U]\!] = [\![x]\!]$, and $[\![x'_U]\!] = [\![x']\!]$). Let $[\![\overline{c}]\!]$ be an interpretation for $\overline{c}$ such that for all possible interpretations $\left[\!\left[\overline{f}'\right]\!\right]$ for $\overline{f}'$, $[\![\overline{x}'_U]\!]$ for $\overline{x}'_U$, and $[\![\lambda]\!]$ for λ we have $\mathcal{M} = \left\{[\![\overline{x}_U]\!], \left[\!\left[\overline{f}\right]\!\right], [\![\overline{c}]\!], [\![\overline{x}'_U]\!], \left[\!\left[\overline{f}'\right]\!\right], [\![\lambda]\!]\right\} \vDash \mathit{no\_array}(\Phi)$.

Let $\varphi_\lambda = \bigwedge_{i\,\in\,\mathcal{I}\setminus\{\lambda\}} i \neq \lambda$. For a sufficiently large domain it is always possible to choose $[\![\lambda]\!]$ in such a way that $\mathcal{M} \vDash \varphi_\lambda$. The remaining constraints in $AC(\Phi)$ are always satisfied, because they correspond to instances of the write axioms of $\mathcal{T}^{\mathrm{P}}_A$. Thus, $\mathcal{M} \vDash AC(\Phi)$. The assumption of validity of Φ^{Q}_U implies that in this case $\mathcal{M} \vDash \mathit{no\_array}(\Phi)$. Bradley et al. [BM07] have proven that any model $\mathcal{M}$ that satisfies φ_λ and $\mathit{no\_array}(\Phi)$ also satisfies Φ (when applying the proper mapping between function symbols and array variables). Thus, the model $\mathcal{M}$, where the interpretations for the universally quantified variables in Φ^{Q} have been chosen arbitrarily, satisfies Φ, under the assumption that Φ^{Q}_U is valid. **Q. E. D.**

4.2.3 Reduction from $\mathcal{T}_U$ to $\mathcal{T}_E$

To reduce the formula $\Phi_U^{\mathrm{Q}} = \forall \overline{x}_U \,.\, \forall \overline{f} \,.\, \exists \overline{c} \,.\, \forall \overline{x}'_U \,.\, \forall \overline{f}' \,.\, \Phi_U$ — which we obtained from the previous reduction step — to $\mathcal{T}_E$ we use Ackermann's reduction [Ack54]. As outlined in Section 2.2.1, Ackermann's reduction introduces fresh variables to replace all instances of uninterpreted functions and predicates. Variables that replace instances of functions whose symbol is in the set $\overline{f}'$ will be added to $\overline{x}'_U$. Variables that replace other function or predicate instances will be added to $\overline{x}_U$. We will refer to the updated sets with $\overline{x}'_E$ and $\overline{x}_E$, respectively.

Example 11
Consider Equation 4.7 in Example 10. After replacing the array reads with function calls, this equation has the following function instances: $REG(s), REG(w)$, etc. For these instances, new domain variables x^s_{REG}, x^w_{REG}, etc. are introduced. One of the constraints of $CC(\Phi_U)$ is then

$$(s = w) \rightarrow \left(x^s_{REG} = x^w_{REG}\right).$$

Both x^s_{REG} and x^w_{REG} are added to $\overline{x}_E$.

Theorem 6 — Reduction to $\mathcal{T}_E$
Let Φ_U be a $\mathcal{T}_U^{qf}$-formula. Then the formula

$$\Phi_U^{Q} = \forall \overline{x}_U \,.\, \forall \overline{f} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \forall \overline{f}' \,.\, \Phi_U \tag{4.9}$$

is valid if and only if the formula

$$\Phi_E^{Q} = \forall \overline{x}_E \,.\, \exists \overline{c} \,.\, \forall \overline{x}'_E \,.\, \big(CC(\Phi_U) \rightarrow no\_func(\Phi_U)\big) \tag{4.10}$$

is valid.

Proof
"$\Rightarrow$": We assume validity of Φ_U^{Q} and prove validity of Φ_E^{Q}. Let $\left[\!\left[\overline{x}^a_f\right]\!\right]$, $\left[\!\left[\overline{x}^a_{f'}\right]\!\right]$, $[\![\overline{x}]\!]$, and $[\![\overline{x}']\!]$ be arbitrary interpretations for the variables in $\overline{x}_E$ and $\overline{x}'_E$, where $\overline{x}$ and $\overline{x}'$ are the variables in the intersection of $\overline{x}_U \cap \overline{x}_E$ and $\overline{x}'_U \cap \overline{x}'_E$, respectively. Let $D = \left[\!\left[\overline{x}^a_f\right]\!\right] \cup \left[\!\left[\overline{x}^a_{f'}\right]\!\right] \cup [\![\overline{x}]\!] \cup [\![\overline{x}']\!]$. Let α be a mapping from an interpretation D for domain variables to an interpretation $\left[\!\left[\overline{f}\right]\!\right]$ for function symbols in $\overline{f}$, such that for $\left[\!\left[\overline{f}\right]\!\right] = \alpha(D)$ each $[\![f]\!] \in \left[\!\left[\overline{f}\right]\!\right]$ satisfies $\forall\, [\![a]\!] \in D \,.\, \forall\, [\![f]\!] \in \left[\!\left[\overline{f}\right]\!\right] \,.\, [\![f]\!]\,([\![a]\!]) = \left[\!\left[x^a_f\right]\!\right]$. In case such interpretations $\left[\!\left[\overline{f}\right]\!\right]$ do not exist due to functional inconsistencies, α returns an arbitrary interpretation $\left[\!\left[\overline{f}\right]\!\right]$. Let γ be a mapping from D to an interpretation $\left[\!\left[\overline{f'}\right]\!\right]$ for function symbols in $\overline{f}'$, defined analogously to α. Let $\left[\!\left[\overline{f}\right]\!\right] = \alpha(D)$ and $\left[\!\left[\overline{f}'\right]\!\right] = \gamma(D)$. Let $[\![\overline{c}]\!]$ be an interpretation for $\overline{c}$ such that for all possible interpretations $\left[\!\left[\overline{f}'\right]\!\right]$ for $\overline{f}'$ and $[\![\overline{x}']\!]$ for $\overline{x}'_E$, we have

that $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{f}\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{f}'\right]\!\!\right]\right\} \vDash \Phi_U$. We have to show that this implies that $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{x}^a_f\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{x}^a_{f'}\right]\!\!\right]\right\} \vDash \Phi_E$. Models that do not satisfy $CC(\Phi_U)$ trivially satisfy Φ_E. We only need to consider models that do satisfy $CC(\Phi_U)$. Thus, for any function instance $f(a)$ in Φ_U, we have $[\![f]\!]([\![a]\!]) = \left[\!\!\left[x^a_f\right]\!\!\right]$, where $\left[\!\!\left[x^a_f\right]\!\!\right]$ is the interpretation for the variable x^a_f with which $f(a)$ has been replaced. Thus, we conclude that if $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{f}\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{f}'\right]\!\!\right]\right\} \vDash \Phi_U$, we have $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{x}^a_f\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{x}^a_{f'}\right]\!\!\right]\right\} \vDash \Phi_E$. This concludes the proof in "$\Rightarrow$" direction.

"$\Leftarrow$": Let $[\![\overline{x}]\!]$, $\left[\!\!\left[\overline{f}\right]\!\!\right]$, $[\![\overline{x}']\!]$, and $\left[\!\!\left[\overline{f}'\right]\!\!\right]$ be arbitrary interpretations for $\overline{x}$, $\overline{f}$, $\overline{x}'$, and $\overline{f}'$ in Φ^{Q}_U. Let α be a mapping from an interpretation for function symbols to an interpretation for domain variables such that $\left[\!\!\left[\overline{x}^a_f\right]\!\!\right] = \alpha([\![f]\!])$ satisfies $\left[\!\!\left[x^a_f\right]\!\!\right] = [\![f]\!]([\![a]\!])$ for all $\left[\!\!\left[x^a_f\right]\!\!\right] \in \left[\!\!\left[\overline{x}^a_f\right]\!\!\right]$. Let γ be a mapping analogous to α. Let $\left[\!\!\left[\overline{x}^a_f\right]\!\!\right] = \alpha([\![f]\!])$ and $\left[\!\!\left[\overline{x}^a_{f'}\right]\!\!\right] = \gamma([\![f']\!])$. Let $[\![\overline{c}]\!]$ be an interpretation for $\overline{c}$ such that for all possible interpretations $[\![\overline{x}']\!]$ and $\left[\!\!\left[\overline{x}^a_{f'}\right]\!\!\right]$ for $\overline{x}'_E$ we have that $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{x}^a_f\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{x}^a_{f'}\right]\!\!\right]\right\} \vDash \Phi_E$.

Due to the definition of α and γ, we know that $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{x}^a_f\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{x}^a_{f'}\right]\!\!\right]\right\}$ $\vDash CC(\Phi_U)$. Due to the assumption of validity of Φ^{Q}_E, we therefore know that $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{x}^a_f\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{x}^a_{f'}\right]\!\!\right]\right\} \vDash \mathit{no\_func}(\Phi_U)$. We have to show that this implies $\left\{[\![\overline{x}]\!], \left[\!\!\left[\overline{f}\right]\!\!\right], [\![\overline{c}]\!], [\![\overline{x}']\!], \left[\!\!\left[\overline{f}'\right]\!\!\right]\right\} \vDash \Phi_U$. This follows trivially from the definitions of $\mathit{no\_func}(\Phi_U)$, α, and γ. **Q. E. D.**

4.2.4 Reduction from $\mathcal{T}_E$ to Propositional Logic

The previous reduction step resulted in a formula $\Phi^{\mathrm{Q}}_E = \forall\overline{x}_E\,.\,\exists\overline{c}\,.\forall\overline{x}'_E\,.\,\Phi_E$. We use the graph-based method by Bryant et al. [BV00], which we discussed in Section 2.2.1, to reduce Φ^{Q}_E to the propositional level. The result of this reduction will be a quantified Boolean formula Φ^{Q}_{prop}. The set of fresh variables introduced for equalities between two terms from $\overline{x}_E$ will be denoted $\overline{b}_x$. The set of fresh variables introduced for equalities between one term from $\overline{x}_E$ and one term from $\overline{x}'_E$ and equalities between two terms from $\overline{x}'_E$ will be denoted $\overline{b}_{x'}$.

Theorem 7 — Reduction to Propositional Logic
Let Φ_E be a $\mathcal{T}_E^{qf}$-formula. Then, for a sufficiently large domain, the formula

$$\Phi_E^Q = \forall \overline{x}_E \,.\, \exists \overline{c} \,.\, \forall \overline{x}'_E \,.\, \Phi_E \tag{4.11}$$

is valid if and only if the formula

$$\Phi_{prop}^Q = \forall \overline{b}_x \,.\, \exists \overline{c} \,.\, \forall \overline{b}_{x'} \,.\, \Big(TC(\Phi_E) \rightarrow skel(\Phi_E) \Big) \tag{4.12}$$

is valid.

Proof
"$\Rightarrow$": Let $\left[\!\left[\overline{b}_x\right]\!\right]$, and $\left[\!\left[\overline{b}_{x'}\right]\!\right]$ be arbitrary interpretations for $\overline{b}_x$ and $\overline{b}_{x'}$ in Φ_{prop}^{Q}. Let $E = \left[\!\left[\overline{b}_x\right]\!\right] \cup \left[\!\left[\overline{b}_{x'}\right]\!\right]$. Let α be a mapping from an interpretation E to an interpretation for domain variables in $\overline{x}_E$ as follows. Let $[\![\overline{x}_E]\!] = \alpha(E)$ such that for all $[\![x_1]\!], [\![x_2]\!] \in [\![\overline{x}_E]\!]$ we have that $[\![x_1]\!] = [\![x_2]\!]$ if and only if $[\![b_{x_1=x_2}]\!] = \top$. In case such interpretations $[\![\overline{x}_E]\!]$ do not exist due to transitivity violations, α returns arbitrary interpretations. Let γ be a mapping from E to an interpretation for domain variables in $\overline{x}'$ defined analogously to α. Let $[\![\overline{x}_E]\!] = \alpha(E)$ and $[\![\overline{x}'_E]\!] = \gamma(E)$. Let $[\![\overline{c}]\!]$ be an interpretation for $\overline{c}$ such that for all possible interpretations $[\![\overline{x}'_E]\!]$ for $\overline{x}'_E$, we have $\Big\{ [\![\overline{x}_E]\!], [\![\overline{c}]\!], [\![\overline{x}'_E]\!] \Big\} \models \Phi_E$. We have to show that this implies that $\left\{ \left[\!\left[\overline{b}_x\right]\!\right] [\![\overline{c}]\!], \left[\!\left[\overline{b}_{x'}\right]\!\right] \right\} \models \big(\Phi_{prop} = TC(\Phi_E) \rightarrow skel(\Phi_E)\big)$. Models that do not satisfy $TC(\Phi_E)$ trivially satisfy Φ_{prop}. We only need to consider models that do satisfy $TC(\Phi_E)$. For any equality atom $x_1 = x_2$ in Φ_E, we have $([\![x_1]\!] = [\![x_2]\!]) \Leftrightarrow [\![b_{x_1=x_2}]\!]$. Due to the definition of $skel(\Phi_E)$, we conclude that if $\Big\{ [\![\overline{x}_E]\!], [\![\overline{c}]\!], [\![\overline{x}'_E]\!] \Big\} \models \Phi_E$, we have $\left\{ \left[\!\left[\overline{b}_x\right]\!\right], [\![\overline{c}]\!], \left[\!\left[\overline{b}_{x'}\right]\!\right] \right\} \models \Phi_{prop}$. This concludes the proof in "$\Rightarrow$" direction.

"$\Leftarrow$": Let $[\![\overline{x}_E]\!]$, and $[\![\overline{x}'_E]\!]$ be arbitrary interpretations for $\overline{x}_E$ and $\overline{x}'_E$ in Φ_E^{Q}. Let α be a mapping from an interpretation for domain variables in $\overline{x}_E$ to an interpretation for propositional variables $\overline{b}_x$ such that $\left[\!\left[\overline{b}_x\right]\!\right] = \alpha([\![\overline{x}_E]\!])$ satisfies $[\![b_{x_1=x_2}]\!] \leftrightarrow ([\![x_1]\!] = [\![x_2]\!])$ for each $[\![b_{x_1=x_2}]\!] \in \left[\!\left[\overline{b}_x\right]\!\right]$. Let γ be a mapping from an interpretation for domain variables in $\overline{x}'_E$ to an interpretation for propositional variables $\overline{b}_{x'}$ analogous to α. Let $\left[\!\left[\overline{b}_x\right]\!\right] = \alpha([\![\overline{x}_E]\!])$ and $\left[\!\left[\overline{b}_{x'}\right]\!\right] = \gamma([\![\overline{x}_E]\!] \cup [\![\overline{x}'_E]\!])$. Let $[\![\overline{c}]\!]$ be an interpretation for $\overline{c}$ such that for all possible interpretations $\left[\!\left[\overline{b}_{x'}\right]\!\right]$ for $\overline{b}_{x'}$ we have $\left\{ \left[\!\left[\overline{b}_x\right]\!\right], [\![\overline{c}]\!], \left[\!\left[\overline{b}_{x'}\right]\!\right] \right\} \models \big(\Phi_{prop} = TC(\Phi_E) \rightarrow skel(\Phi_E)\big)$. Due to the definition of α and γ, we know that $\left\{ \left[\!\left[\overline{b}_x\right]\!\right], [\![\overline{c}]\!], \left[\!\left[\overline{b}_{x'}\right]\!\right] \right\} \models TC(\Phi_E)$. Due to the assumption of validity of Φ_{prop}^{Q}, we therefore know that $\left\{ \left[\!\left[\overline{b}_x\right]\!\right], [\![\overline{c}]\!], \left[\!\left[\overline{b}_{x'}\right]\!\right] \right\} \models skel(\Phi_E)$. We have to show that this implies that $\left\{ [\![\overline{x}_E]\!], [\![\overline{c}]\!], [\![\overline{x}'_E]\!] \right\} \models \Phi_E$. This follows trivially from the definitions of $skel(\Phi_E)$, α, and γ. **Q. E. D.**

4.2.5 Extracting Certificates

In the previous sections we have shown how to reduce a formula in Φ^{Q} to an equivalent propositional formula $\Phi^{\mathrm{Q}}_{prop} = \forall \overline{b}_x . \exists \overline{c} . \forall \overline{b}_{x'} . \Phi_{prop}$. Now we want to compute functions (in terms of the variables in $\overline{b}_x$) for the control signals in $\overline{c}$. One naive possibility to do so is to use Binary Decision Diagrams (BDDs). We compute a BDD for Φ_{prop}, and subsequently perform the inner universal quantifications to obtain $\hat{\Phi}_{prop} = \forall \overline{b}_{x'} . \Phi_{prop}$. Although — in the worst case — this might blow up the size of the BDD exponentially (with respect to $\left|\overline{b}_{x'}\right|$), the average case may be much more space-efficient.

The formula $\hat{\Phi}_{prop}$ can be viewed as the characteristic function of a multi-output Boolean relation, with inputs $\overline{b}_x$ and outputs $\overline{c}$. There are many different (symbolic) ways to compute functions compatible with a given relation. An overview of several such methods in the scope of synthesis is given in Section 4.2.6. One of the first methods that has been proposed is presented in [WB93]. (See also [BGJ$^+$07b, JLH09].) It proceeds as follows. For every output c_i we first perform the existential quantification of all other outputs. From the remaining formula, we compute the positive and the negative cofactor of c_i, to which we will refer by Φ_{c_i} and $\Phi_{\neg c_i}$, respectively. These can be used to determine the on-set, off-set, and don't-care-set of the function for c_i in the following way:

$$\begin{array}{lcrcr} ON & = & \Phi_{c_i} & \wedge & \neg\Phi_{\neg c_i} \\ OFF & = & \neg\Phi_{c_i} & \wedge & \Phi_{\neg c_i} \\ DC & = & \Phi_{c_i} & \wedge & \Phi_{\neg c_i} \end{array}$$

Any minimization algorithm for incompletely specified Boolean functions can be used to compute an actual implementation f_{c_i} for c_i, using the sets *ON*, *OFF*, and *DC*. Once this is done, we resubstitute f_{c_i} for c_i in $\hat{\Phi}_{prop}$. This is necessary, because values of other outputs might depend on the actual choice of c_i. After that, we can proceed with computing a function for the next output.

The control functions can easily be integrated into the original circuit. For a variable b_x that represents the equality $x_1 = x_2$, we build a comparator for the terms x_1 and x_2 and add it to the design. The outputs of all such comparators are then used as inputs for the control functions f_{c_i}. That is, the control signals c_i are Boolean combinations of the comparator outputs. It should be noted that only terms that appear in the original $\mathcal{S}^{\mathrm{Q}}$ formula are used as inputs to the comparators. That is, no new array reads or instances of functions or predicates are introduced. However, not every comparator necessarily corresponds to an equality of the original formula. This is due to the fact that during the creation of the transitivity constraints, new edges are added to the equality graph to make it chordal.

Example 12
For the design in Figure 3.1(b), one possible solution is $c := (s = w)$. Thus, we insert a comparator into the design, whose inputs are connected to the primary input s and the pipeline register w. The output of the comparator is then the desired control signal c.

4.2.6 Alternative Methods for Certificate Extraction

Declaration of Sources

This section is based on and reuses material from the following source, previously published by the author:

- [EKH12] Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 91–100. IEEE, 2012.

References to this source are not always made explicit.

Many approaches for extracting certificates from propositional formulas of the form $\forall \overline{b}_x . \exists \overline{c} . \Phi_{prop}$ have been proposed. In synthesis settings, such formulas can be referred to as *general strategies* [EKH12]. In this section, we describe some approaches to construct implementations from such general strategies and state our experiences with them. These experiences are based on experiments with benchmarks from temporal logic synthesis. An evaluation of how well these experiences correlate with controller synthesis benchmarks — as presented in this thesis — remains for future work.

Kukula and Shiple [KS00] described a simple technique to compute a circuit from a general strategy in BDD form. The main idea is to take the graph structure of the BDD and instantiate an 8-gate building block for all nodes to obtain an implementation. The resulting circuits have a very high depth (more than two times the number of state and input variables) and experience shows that they are often huge [BGJ+07b].

The temporal logic synthesis tool Anzu [JGWB07] uses a simple, cofactor-based approach [BGJ+07b], which we already described in Section 4.2.5. There is also a simple but effective optimization, mentioned by Bloem et al. [BGJ+07b]. For each output, they remove unnecessary input variables by existential quantification. We will subsequently refer to this method as the *cofactor approach*.

Baneres et al. [BCK04] present a recursive paradigm for extracting completely specified Boolean functions from general strategies. Their approach is based on first computing the single output functions independently, without resubstitution. In a second stage they recursively resolve inconsistencies resulting from uncoordinated choices during the first stage. They also introduce a recursion-depth limit. If the limit is reached, their algorithm falls back to an arbitrary other relation-solving method. We reimplemented their approach within the temporal logic synthesis tool RATSY [BCG$^+$10] and applied it to its general strategies. Unfortunately, first experimental results were rather discouraging. Without any recursion limit, the approach timed out even for rather small benchmarks. However, using a recursion limit, we (almost) always hit the fall-back mechanism. The result of the fall-back mechanism is in almost all cases the same as if the recursive approach of [BCK04] had not been used at all. Therefore, this approach does not provide any improvement concerning circuit size, but only increases computation time significantly. We believe that this is due to the fact that general strategies in temporal logic synthesis settings are highly non-deterministic, and in particular have many vertices that [BCK04] calls "non-don't-care extendable".

Another approach that we implemented in RATSY is the Minato-Morreale algorithm for computing an *irredundant sum-of-products* [Min92, Mor70]. It is a recursive procedure that takes a general strategy as an input and computes a sum-of-products form for a compatible completely specified function. The final result is *irredundant* in the sense that no single literal or cube can be deleted without changing the function. We use the recursive structure of the algorithm to build a multi-level Boolean circuit along the way. The resulting circuits are comparable in size to the ones obtained through the cofactor approach. Computation times, however, are significantly higher. To further improve these results, we also tried using a "cache". In each step, the algorithm first checks whether a function lying in the desired interval of functions has already been built as a circuit in previous steps. If so, this function (and the corresponding wire in the circuit) is reused. To keep the memory footprint of the cache small and to speed up the process of a cache look-up, we did not store the BDDs of the functions, but rather used a signature-based approach as in [MCB05]. We only store the function's output for some random input vectors. These outputs are called a *signature*. Signatures have a very low memory footprint. When doing a look-up, we can use the signature to perform a fast pre-test. This pre-test may, however, create false positives. Thus, whenever the pre-test yields a positive result, we (recursively) reconstruct a BDD for the function in question from the structure of the circuit generated so far. We subsequently use this BDD to perform a sound comparison to check whether or not the function really lies within the desired interval. Experimental results have shown that, unfortunately, we get almost no cache hits. The hits we do get are mostly very small, almost trivial functions, consisting of only a handful of gates. Thus, the gain due to sharing is negligible. On the other hand, computation time rises significantly due to the many look-up checks that have to be performed. We also noticed that — when extended from completely specified functions [MCB05] to intervals — the signature-based pre-test gives too many false positives to be of use.

Jiang et al. [JLH09] presented a SAT-solver-based approach to compute functions from a general strategy. Their method is based on Craig interpolation. This is the basis for our own interpolation-based approaches, which we will discuss in

Chapter 5.

A method for computing circuits based on computational learning is presented in [EKH12]. It starts with simple candidate functions and refines them based on the counterexamples that are returned by a teacher oracle. This is the only approach we are aware of that clearly outperforms the cofactor approach with respect to resulting circuit sizes. Computation times are, however, longer than with the cofactor approach. An adaption of this learning approach to our controller synthesis problems will be discussed in Section 6.4.

4.3 Computational Complexity

Let us briefly discuss the computational complexity of the reduction steps presented in the previous section. Let n be the size of the original $\mathcal{S}^{\mathrm{Q}}$ formula. Consider the reduction to uninterpreted functions and equality (Section 4.2.2). There can be at most n array-write expressions to remove. Moreover, the constraints for λ can have n conjuncts at most, as the cardinality of the index set $\mathcal{I}$ is bound by n. Thus, the size of the constraints $AC(\Phi)$ is bound by $\mathcal{O}(n)$. The universal quantifiers in array properties are all replaced by $\mathcal{O}(n)$ conjuncts. The number of array properties is bound by n. Thus, the total size of the reduced formula Φ_U^{Q} is $\mathcal{O}(n^2)$. The reduction to pure equality logic (Section 4.2.3) causes another polynomial increase in size, whose details depend on the number of different functions, the number of function instances, and the arity of the functions. Ackermann's reduction also introduces a linear number of new variables (one per function instance). The reduction to propositional logic (Section 4.2.4) causes a cubic increase in the number of variables and the formula's size, in the worst case [BV00]. Concerning computation time, all steps so far can be done in polynomial time.

Let us now consider the generalized specification language $\mathcal{S}^{\mathrm{Q}^+}$ (see Definition 22 on page 35), which allows an arbitrary number of quantifier alternations, as long as existential quantification is over propositional variables only. The reductions shown in Section 4.2 (and the proofs for their correctness) can be generalized in a straightforward way.

Theorem 8 — Computational Complexity of $\mathcal{S}^{\mathbf{Q}^+}$
Deciding whether or not a formula from $\mathcal{S}^{\mathrm{Q}^+}$ is valid is a PSPACE-complete decision problem.

Proof
The QSAT problem, that is, deciding whether or not a quantified Boolean formula without free variables is valid,[13] is known to be PSPACE-complete. We will show PSPACE-completeness of our problem in two steps. First, we reduce our problem to QSAT, thereby showing that it is contained in PSPACE. Second, we will reduce QSAT to our problem, thereby showing that it is PSPACE-hard.

[13]Note that without free variables "satisfiability" and "validity" actually mean the same thing, as the formula in question can only be either $\top$ or $\bot$.

1. **Containment.**
   Given a formula Φ^{Q^+} from $\mathcal{S}^{Q^+}$, we apply the reductions described in Section 4.2. As mentioned above, this can be done in polynomial time and with a polynomial increase in the number of variables and the length of the formula. The resulting formula is a QBF. Determining its validity is an instance of the QSAT problem. As the QSAT problem is contained in PSPACE, deciding validity for $\mathcal{S}^{Q^+}$-formulas is also contained in PSPACE.

2. **Hardness.**
   The QBF of a QSAT instance is actually a special case of a $\mathcal{S}^{Q^+}$ formula, where *all* variables — not just the existentially quantified ones — are Boolean. Thus, any algorithm for deciding validity of $\mathcal{S}^{Q^+}$ formulas can be used directly to decide QSAT instances. As QSAT is PSPACE-hard, deciding validity for $\mathcal{S}^{Q^+}$-formulas is also PSPACE-hard.

We have shown that deciding validity for $\mathcal{S}^{Q^+}$-formulas is both contained in PSPACE and PSPACE-hard. Thus, it is PSPACE-complete. **Q. E. D.**

Obviously, the PSPACE-completeness of deciding $\mathcal{S}^{Q^+}$ does *not* imply that also the (potentially easier) problem of deciding $\mathcal{S}^{Q}$ is PSPACE-complete. For the QSAT problem, fixing the number of quantifier alternations results in complexities from the *polynomial hierarchy*. Thus, for $\mathcal{S}^{Q}$ — with its fixed quantifier prefix — we obtain the following complexity:

Theorem 9 — Computational Complexity of $\mathcal{S}^{Q}$
Deciding whether or not a formula from $\mathcal{S}^{Q}$ is valid is a Π_2^P-complete (also known as coNP^{NP}-complete) decision problem.

Proof
The QSAT$_2^{\forall}$ problem, that is, the problem of deciding whether a QBF with exactly 2 quantifier alternations and an outermost universal quantifier is valid, is Π_2^P-complete [Wra76]. The rest of the proof proceeds analogously to the proof of Theorem 8. **Q. E. D.**

5
Interpolation-based Synthesis

Declaration of Sources

This chapter is based on and reuses material from the following source, previously published by the author:

- [HGK$^+$13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In Jobstmann and Ray [JR13], pages 77–84.

References to this source are not always made explicit. In particular, Sections 5.2 and 5.3 are heavily based on the paper cited above. Section 5.4, however, contains material that has not been published before.

In Chapter 4 we have shown how to transform specifications in $\mathcal{S}^{\mathrm{Q}}$ into propositional formulas, from which we can extract certificates. In this chapter, we will focus on interpolation-based techniques for certificate computation. Interpolation has been used for certificate computation in Boolean settings by Jiang et al. [JLH09]. We improve over their work in two ways. First, we perform interpolation on the theory-level, instead of on the propositional level. Thus, we avoid the cumbersome and sometimes even infeasible translations to propositional logic that we described in Chapter 4. Details will be presented in Section 5.1. Second, we show how to compute multiple coordinated interpolants from a single refutation proof. This will be discussed in Section 5.2. The price to pay for multiple interpolants from a single proof is that the proof needs to satisfy certain properties. In addition to being colorable (which is also required for iterative single interpolation), it must be local-first. We will define local-first proofs and show how to obtain them from arbitrary proofs in Section 5.3.

In Section 5.4, we present an alternative to transforming an ordinary refutation proof: By using a *modular SMT solver*, we can directly generate a colorable, local-first proof. Using such a solver we can even solve the generalized problem of computing certificates for existentially quantified propositional variables in formulas in $\mathcal{S}^{\mathrm{Q}^+}$.

The interpolation-based certificate-computation methods are also the basis for our prototype synthesis tool SURAQ, which will be presented in Chapter 7 in more detail.

5.1 Iterative Interpolation

A first naive way to use interpolation for certificate extraction would be to use the method proposed by Jiang et al. [JLH09]. To do so, we take a formula Φ^{Q} in $\mathcal{S}^{\mathrm{Q}}$ and transform it to propositional logic, as outlined in Section 4.2. We then eliminate the inner universal quantifiers, by expanding them.[14] This gives us a formula of the form $\forall \overline{b} \,.\, \exists \overline{c} \,.\, \Phi_{prop}$. We then take the part Φ_{prop} without the quantifier prefix. As mentioned in Section 4.2.5, the formula Φ_{prop} corresponds to the characteristic function of a Boolean relation whose outputs are the control signals $\overline{c}$ we want to synthesize. Thus, by applying the method of [JLH09] to Φ_{prop}, we obtain the implementations for the signals in $\overline{c}$.

Interpolation, however, is not just possible on the propositional level. Craig's interpolation theorem holds for full first-order logic. Unfortunately, it does not guarantee quantifier-free interpolants, in the general case. It can even be shown that in some theories a quantifier-free interpolant does not exist for certain formulas. Fortunately, the theory of uninterpreted functions and equality is not one of those. In fact, it has been shown that for unsatisfiable formulas in $\mathcal{T}_U^{\mathrm{qf}}$ a quantifier-free interpolant always exists and can be computed easily from a refutation proof [McM05, FGG$^+$12]. For the rest of this Chapter we will thus only consider formulas in $\mathcal{T}_U^{\mathrm{qf}}$, unless explicitly stated otherwise. For specifications Φ^{Q} in $\mathcal{S}^{\mathrm{Q}}$ that also include array terms, we perform the reduction described in Section 4.2.2 to obtain a formula Φ_U^{Q} in $\mathcal{T}_U$, whose subformula behind the quantifier prefix is in $\mathcal{T}_U^{\mathrm{qf}}$. Thus, we can compute quantifier-free interpolants for $\mathcal{S}^{\mathrm{Q}}$.

5.1.1 Single Control Signal

Let us first consider the case where we want to synthesize just one single control signal c from a specification $\Phi^{\mathrm{Q}} = \forall \overline{x} \,.\, \exists c \,.\, \forall \overline{x}' \,.\, \Phi$, where Φ is in $\mathcal{T}_U^{\mathrm{qf}}$. We perform

[14]One possible way to do this is to use BDDs. In many cases, BDDs avoid the worst-case exponential blowup of the quantifier expansion.

the expansion-negation procedure, described in Section 4.1 to obtain

$$\varphi = \mathit{expand\_negate}(\Phi) = \varphi_0(\overline{x}, 0, \overline{x}'_0) \wedge \varphi_1(\overline{x}, 1, \overline{x}'_1) \qquad (5.1)$$

Note that we have made explicit which variables occur in which part of φ, because this is important for interpolation. For a valid specification Φ^{Q}, the formula φ is unsatisfiable (see Lemma 4 on page 45). We compute an interpolant that only depends on the shared symbols $\overline{x}$. This interpolant is a certificate for c as shown in the following theorem.

Theorem 10
The interpolant between $\varphi_0(\overline{x}, 0, \overline{x}'_0)$ and $\varphi_1(\overline{x}, 1, \overline{x}'_1)$ is a certificate for c in Φ^{Q}.

Proof
Let χ be an interpolant between φ_0 and φ_1. The variables in $\overline{x}'_0$ and $\overline{x}'_1$ occur only in either φ_0 or φ_1, but not in both. Thus, they cannot occur in the interpolant χ. The only variables that can occur in χ are $\mathit{symb}(\chi) = \mathit{symb}(\varphi_0) \cap \mathit{symb}(\varphi_1) = \overline{x}$. Furthermore, we know that $\varphi_0 \to \chi$ and $\varphi_1 \to \neg\chi$, due to the definition of an interpolant. By reversing the implications we get $\neg\chi \to \neg\varphi_0$ and $\chi \to \neg\varphi_1$. Based on the definition of $\mathit{expand\_negate}()$ we also know that $\varphi_i(\overline{x}, i, \overline{x}'_i) = \neg\Phi(\overline{x}, i, \overline{x}'_i)$. By reintroducing the universal quantifiers for $\overline{x}'_i$, we get $\neg\chi(\overline{x}) \to \forall \overline{x}' \,.\, \Phi(\overline{x}, 0, \overline{x}')$ and $\chi(\overline{x}) \to \forall \overline{x}' \,.\, \Phi(\overline{x}, 1, \overline{x}')$. Therefore, $\forall \overline{x} \,.\, \forall \overline{x}' \Phi(\overline{x}, \chi(\overline{x}), \overline{x}')$ is valid. Hence, $\chi(\overline{x})$ is a certificate for c in Φ^{Q}. **Q. E. D.**

Based on Theorem 10, we can synthesize a single control signal. Note that we performed interpolation in $\mathcal{T}_U^{\mathrm{qf}}$ and thus did not have to perform a reduction to propositional logic.

5.1.2 Multiple Control Signals

Based on finding a single certificate via interpolation we will now show how to find certificates for multiple control signals in an iterative way. Note that we cannot straightforwardly use the methods proposed in [JLH09], because we have an inner universal quantifier (over domain variables) that we cannot simply eliminate as it can be done in the propositional case. We will discuss this in more detail towards the end of this section.

Our approach to iterative certificate computation is shown in Algorithm 5.1. It is different to the method presented in [JLH09] with respect to the way expansion of the existential quantifier is done. The algorithm takes as input a formula $\Phi^{\mathrm{Q}} = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$, where Φ is in $\mathcal{T}_U^{\mathrm{qf}}$, and proceeds as follows. We loop over all control signals c in $\overline{c}$ and compute one certificate for c per iteration. To do so, we first expand and negate Φ (line 3), as explained in Definition 23 on page 45. Next, we divide the conjuncts of φ into two groups: those where the index that corresponds to the current c is 0 — which we call φ_A, and those where it is 1 — which we call φ_B. This is done in Lines 6–10. Based on these groups, we compute an interpolant χ (line 11), which we add to the list of certificates computed so far (line 12). In line 13 we resubstitute the certificate we just computed back into Φ. This is imperative for overall correctness, because the choice for a value

Algorithm 5.1: Iterative certificate computation.

```
Input  : A formula $\Phi^{\mathrm{Q}} = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$, where $\Phi$ is in $\mathcal{T}_U^{\mathrm{qf}}$.
Output: A list of certificates $\chi(\overline{x})$ for each $c \in \overline{c}$ in $\Phi^{\mathrm{Q}}$.
result = [ ] // empty list
foreach $c$ in $\overline{c}$ do
    $\varphi = \mathit{expand\_negate}(\Phi) = \varphi_{00\ldots00} \wedge \ldots \wedge \varphi_{11\ldots11}$
    $\varphi_A = \top$
    $\varphi_B = \top$
    foreach $\varphi_{i_1 i_2 \ldots i_n}$ in $\varphi$ do
        if index $i_k$ of $\varphi_{i_1 i_2 \ldots i_n}$ corresponding to $c$ is $0$ then
            $\varphi_A = \varphi_A \wedge \varphi_{i_1 i_2 \ldots i_n}$
        else
            $\varphi_B = \varphi_B \wedge \varphi_{i_1 i_2 \ldots i_n}$
    $\chi$ = interpolant($\varphi_A, \varphi_B$)
    result = result + [$\chi$]
    $\Phi = \Phi$[substitute $\chi$ for $c$]
    $\overline{c} = \overline{c} \setminus c$
return result
```

for c_1 can influence the possible choices for another variable c_2. Simply imagine a specification that, for example, says that the choice for c_1 is arbitrary, but c_2 must always be the same as c_1. Thus, once we fix a function for one c, we have to ensure that all subsequent choices respect this decision. Finally, we remove c from $\overline{c}$ in line 14. Since c no longer occurs in Φ after substitution, we do not need to expand over it in the next iteration.

Theorem 11
Algorithm 5.1 always terminates and computes correct certificates for the existentially quantified variables.

Proof
Termination. All loops in Algorithm 5.1 are just iterations over finite data structures, which are thus guaranteed to end. Furthermore, all calls to functions (computing an interpolant, substitution in a formula, etc.) return a result after finitely many steps. Thus, the algorithm always terminates.

Correctness. See proof of Theorem 10. **Q. E. D.**

The Need for Complete Expansion

The expansion performed in line 3 of algorithm 5.1 creates $2^{|\overline{c}|}$ conjuncts. It may seem odd that the computation of one single certificate requires an expansion into an exponential number of conjuncts. Indeed, for propositional logic, Jiang et al. [JLH09] show an iterative procedure that requires expansion over one $c \in \overline{c}$

in each step only. Kuncak et al. [KMPS10] use a very similar technique for formulas in first-order logic. Unfortunately, this technique cannot be used for our problem, because of one fundamental difference. The formulas considered in [JLH09] and [KMPS10] have a $\forall$-$\exists$-quantifier structure, while our formulas have a $\forall$-$\exists$-$\forall$-quantifier structure. In the remainder of this section, we will first briefly recapitulate how iterative certificate computation for formulas with an $\forall$-$\exists$-quantifier structure can avoid the exponential blow-up. Subsequently, we will demonstrate why this approach is not applicable to our $\forall$-$\exists$-$\forall$-formulas.

Consider a formula $\forall \overline{x} \,.\, \exists \overline{c} \,.\, \Phi$. We pick one c from $\overline{c}$, for which we want to compute a certificate first. Let $\tilde{c} = \overline{c} \setminus c$. The trick is to (implicitly) treat all variables in $\tilde{c}$ as additional "inputs" for the witness function for c. Note, however, that $\forall \overline{x} \,.\, \forall \tilde{c} \,.\, \exists c \,.\, \Phi$ is not necessarily valid. For some values of $\overline{x}$ and $\tilde{c}$ there might be no value for c that makes Φ true. Another way of seeing this is that the relation from $\overline{x} \cup \tilde{c}$ to c, whose characteristic function is Φ, is not *total*. Jiang et al. [JLH09] show how to remedy the problem by *completing* the relation. They suggest to replace Φ with $\Psi = \Phi(\overline{x}, \tilde{c}, c) \vee \forall c \,.\, \neg\Phi(\overline{x}, \tilde{c}, c)$. The intuitive meaning of this is that for values for $\overline{x}$ and $\tilde{c}$ for which no correct value of c existed, the new relation will allow an arbitrary value for c. This is justified by the fact that a correct choice of value for all the variables in $\tilde{c}$ can never be one for which there is no legal choice for c. Due to the fact that Ψ now represents a total relation, the formula $\forall \overline{x} \,.\, \forall \tilde{c} \,.\, \exists c \,.\, \Psi$ is valid. We compute *expand_negate*$(\Psi) = \psi_0 \wedge \psi_1$, where

$$\psi_0 = \neg\Phi(\overline{x}, \tilde{c}, 0) \wedge \Phi(\overline{x}, \tilde{c}, 1) \tag{5.2}$$
$$\psi_1 = \neg\Phi(\overline{x}, \tilde{c}, 1) \wedge \Phi(\overline{x}, \tilde{c}, 0). \tag{5.3}$$

Note that the conjunction of ψ_0 and ψ_1 is trivially unsatisfiable. An interpolant χ between ψ_0 and ψ_1 is a correct certificate for c, according to [JLH09]. We can resubstitute χ for c in Φ and proceed to compute a certificate for the next variable in $\overline{c}$. Note that in the first step the size of the formula passed to the SAT/SMT solver is only linear in the size of the original quantified formula. For subsequent steps, the size depends on the size of the interpolants computed in previous steps. Moreover, n calls to the solver will be necessary to compute n certificates.

Let us see what happens when we try to apply this procedure to $\forall$-$\exists$-$\forall$-formulas. As outlined in Section 4.1, we rename the variables $\overline{x}'$, which are bound by the inner universal quantifier, for each of the conjuncts of the (negated) expansion. That would lead to formulas

$$\psi_0 = \neg\Phi(\overline{x}, \tilde{c}, 0, \overline{x}'_0) \wedge \Phi(\overline{x}, \tilde{c}, 1, \overline{x}'_0) \tag{5.4}$$
$$\psi_1 = \neg\Phi(\overline{x}, \tilde{c}, 1, \overline{x}'_1) \wedge \Phi(\overline{x}, \tilde{c}, 0, \overline{x}'_1). \tag{5.5}$$

However, the conjunction of Equations 5.4 and 5.5 is not necessarily unsatisfiable, as shown by the following example.

Example 13
Let $\Phi = x' \leftrightarrow (c_1 \leftrightarrow c_2)$, where x' is the only (Boolean) variable in $\overline{x}'$, c_1 and c_2

are the only two variables in $\overline{c}$, and $\overline{x}$ is empty. We try to compute a certificate for c_2, while treating c_1 as an input. Performing the expansion and negation outlined above, we obtain

$$\psi_0 = \neg\Phi(c_1, 0, {x'}_0) \wedge \Phi(c_1, 1, {x'}_0) \tag{5.6}$$
$$\psi_1 = \neg\Phi(c_1, 1, {x'}_1) \wedge \Phi(c_1, 0, {x'}_1). \tag{5.7}$$

Consider the model $\mathcal{M}$, where $[\![c_1]\!] = \top$, $[\![{x'}_0]\!] = \top$, and $[\![{x'}_1]\!] = \bot$. Clearly, we have that

$$\mathcal{M} \not\models \Phi(c_1, 0, {x'}_0) \tag{5.8}$$
$$\mathcal{M} \models \Phi(c_1, 1, {x'}_0) \tag{5.9}$$
$$\mathcal{M} \not\models \Phi(c_1, 1, {x'}_1) \tag{5.10}$$
$$\mathcal{M} \models \Phi(c_1, 0, {x'}_1). \tag{5.11}$$

It follows that $\mathcal{M}$ satisfies both ψ_0 (Equation 5.6) and ψ_1 (Equation 5.7). Thus, $\psi_0 \wedge \psi_1$ is satisfiable.

The reason for the potential satisfiability of $\psi_0 \wedge \psi_1$ lies in the renaming of the variables in $\overline{x}'$. Without such renaming, the formula would be trivially unsatisfiable. However, if we do not perform renaming, we face another problem. Without renaming, the variables in $\overline{x}'$ are no longer local to either ψ_0 or ψ_1; they can now occur in both simultaneously. Thus, an interpolant χ could also depend on those variables. This could lead to a combinational loop: The value of c_1 could depend on the value of c_2, and the value of c_2 could depend on the value of c_1. We thus conclude that the procedure of [JLH09] and [KMPS10] — which expands only over one of the variables in $\overline{c}$ in each step — is not applicable to $\forall$-$\exists$-$\forall$-formulas.

5.2 n-Interpolation

The iterative approach presented in the previous section requires n calls to an SMT solver to compute n certificates from n refutation proofs. SMT solving is a costly operation. Thus, we would like to reduce the number of solver invocations. In this section, we will show how to generalize the notion of an interpolant to what we call an n-interpolant, computing n (coordinated) interpolants simultaneously, from a single refutation proof. Again, we consider formulas $\Phi^{\mathrm{Q}} = \forall\overline{x}\,.\,\exists\overline{c}\,.\,\forall\overline{x}'\,.\,\Phi$, where Φ is in $\mathcal{T}_U^{\mathrm{qf}}$. To obtain an unsatisfiable formula for interpolation, we compute $\varphi = \mathit{expand\_negate}(\Phi) = \varphi_{0\ldots0} \wedge \ldots \wedge \varphi_{1\ldots1}$. Generalizing the notions presented in Section 2.3, we will call the conjuncts φ_i the 2^n *partitions* of φ, where $n = |\overline{c}|$, and associate a *color* with the local symbols of each partition. Note that since the φ_is are obtained by only renaming variables, the shared non-logical symbols between any two partitions are the same.

Definition 24 — Global and Local Symbols
Symbols in the set $G = \bigcap_{i\,\in\,\mathbb{B}^n} \mathit{symb}(\varphi_i)$ are called global *symbols. All other*

symbols are called local *(with respect to the one partition in which they occur). For $\varphi = expand\_negate(\Phi)$, we have $G = \overline{x}$, and in each partition i, we have local variables $\overline{x}'_i$.*

Let $\overline{\chi}$ be a vector of formulas $(\chi_1, \ldots, \chi_n)$. Let $\oplus$ be the exclusive-or (xor) operator. For a word $i \in \mathbb{B}^n$, let $\overline{\chi}' = \overline{\chi} \oplus i$ if for each $j \in \{1, \ldots, n\}$, $\chi'_j = \chi_j \oplus i_j$. Let $\bigvee \overline{\chi}$ be short for $\chi_1 \vee \ldots \vee \chi_n$. Let $C|_i = C|_{\varphi_i}$, that is, $C|_i$ is C with all literals that contain symbols that do not occur in φ_i removed. The following definition generalizes the notion of interpolant and partial interpolant from two formulas to 2^n formulas.

Definition 25 — n-(Partial) Interpolant
*Let $\bigwedge_{i \in \mathbb{B}^n} \varphi_i$ be an unsatisfiable CNF formula. Let n be a node in the refutation proof of $\bigwedge_{i \in \mathbb{B}^n} \varphi_i$. Let $C = clause(n)$. An n-*partial interpolant *$\overline{\chi}$ for C with respect to the partitions φ_i is a vector of formulas with length n, such that $\forall i \in \mathbb{B}^n \,.\, \varphi_i \rightarrow \big(C|_i \vee \bigvee(\overline{\chi} \oplus i)\big)$ and $\overline{\chi} \preceq G$. If $C \equiv \bot$ then $\overline{\chi}$ is an n-*interpolant *with respect to the partitions φ_i.*

Based on this definition, the following theorem is a generalization of Theorem 10, showing that n-interpolants are a mean of certificate computation.

Theorem 12
The components of an n-interpolant $\overline{\chi}$ with respect to the partitions φ_i constitute certificates for the variables $\overline{c}$ in the formula Φ^Q.

Proof
Since for each i, the variables $\overline{x}'_i$ only appear in φ_i, they cannot be in G; only $\overline{x}$ are in G and thus, and n-interpolant can only depend on $\overline{x}$. Let $\overline{\chi}(\overline{x}) = (\chi_1(\overline{x}), \ldots, \chi_n(\overline{x}))$ be an n-interpolant with respect to the partitions φ_i. Due to the definition of n-interpolants, we have that for each $i \in \mathbb{B}^n$

$$\varphi_i \rightarrow \bigvee \overline{\chi}(\overline{x}) \oplus i \tag{5.12}$$

holds. After reversing the implications and expanding the definition of φ_i, we obtain

$$\bigwedge \neg\overline{\chi}(\overline{x}) \oplus i \rightarrow \Phi(\overline{x}, i, \overline{x}'_i). \tag{5.13}$$

After reintroducing the quantifiers for $\overline{x}'$, we obtain

$$\bigwedge \neg\overline{\chi}(\overline{x}) \oplus i \rightarrow \forall \overline{x}' \,.\, \Phi(\overline{x}, i, \overline{x}'). \tag{5.14}$$

Therefore, $\forall \overline{x} \,.\; i = \overline{\chi}(\overline{x}) \rightarrow \forall \overline{x}' \,.\, \Phi(\overline{x}, i, \overline{x}')$. Therefore,

$$\forall \overline{x} \,.\, \forall \overline{x}' \,.\, \Phi(\overline{x}, \overline{\chi}(\overline{x}), \overline{x}'). \tag{5.15}$$

Hence for each $j \in \{1, \ldots, n\}$, we have that $\chi_j(\overline{x})$ is a witness function for c_j in Φ^Q. **Q. E. D.**

$$\textsc{mHyp}\,\frac{}{C\ [i]}\,C \in \varphi_i \qquad \textsc{mAxi}\,\frac{}{C\ [i]}\,C \preceq \varphi_i$$

$$\textsc{mRes}\,\frac{a \vee C\ [i] \qquad \neg a \vee D\ [i]}{C \vee D\ [i]}\,i \in \mathbb{B}^n, a \vee C \vee D \preceq \varphi_i$$

$$\textsc{mRes-G}\,\frac{a \vee C\ [\overline{\chi}^C] \qquad \neg a \vee D\ [\overline{\chi}^D]}{\begin{array}{l} C \vee D \quad [(\ (a \vee \chi_1^C) \wedge (\neg a \vee \chi_1^D), \\ \qquad\qquad\qquad \ldots, \\ \qquad\quad (a \vee \chi_n^C) \wedge (\neg a \vee \chi_n^D)\ )] \end{array}}\,a \preceq G$$

Figure 5.1: n-Interpolating proof rules for an unsatisfiable $\varphi = \bigwedge_{i \in \mathbb{B}^n} \varphi_i$. These rules can only annotate proofs that are colorable and local-first.

5.2.1 Computing n-interpolants

As we have just seen, n-interpolants can serve as certificates. Let us thus now see how we can compute them. In Figure 5.1, we present annotating proof rules for n-interpolants, similar to the ones shown in Section 2.3. Note that due to the way we have defined refutation proofs (see Definition 15), we do not require any theory-specific proof rules. Theory-reasoning is done via theory lemmata that are introduced as leaves of the proof by the mAxi-rule. The proof rules annotate each conclusion of a proof step with an n-partial interpolant for the conclusion with respect to the partitions. These annotation rules require two properties of the proof. First, it needs to be colorable.[15] That is, every leaf of the proof contains only global symbols and/or symbols that are local to exactly one partition. Second, it needs to be local-first.

Definition 26 — Local-first Proof
A refutation proof is local-first*, if for every resolution node with a resolving literal that contains local symbols, both its premises are derived from the same partition.*

In Figure 5.1, the rule mHyp annotates the derived clause C with i if C appears in partition φ_i. Similarly, the rule mAxi annotates theory lemma C with i if $C \preceq \varphi_i$. Rules mRes and mRes-G annotate resolution steps. mRes-G, which is only applicable if the resolving literal is global, follows Pudlák's interpolation system [Pud97] n times, once for each of the components of the partial interpolant. mRes is only applicable if both premises are annotated with the same n-partial interpolant and this n-partial interpolant is an element of $\mathbb{B}^n$. Note that despite these restrictions, these rules will always be able to annotate a proof that is colorable and local-first.

The intuition behind the annotation rules is as follows. Every partition φ_i corresponds to a vector i of values for the variables in $\overline{c}$. Moreover, the partition represents a set of values for $\overline{x}$, for which the values i *cannot* be used for $\overline{c}$ in order to satisfy Φ. Remember that $\varphi_i = \neg\Phi_i$. The rule mRes-G basically works

[15]We extend Definition 18 (page 30) from two partitions to 2^n partitions in the obvious way.

like a multiplexer. Based on the truth value of a literal over variables from $\overline{x}$ only, it selects the child node in which the literal appeared in polarity opposite to the present value. This makes sure that we will *not* end up in a partition that corresponds to values for $\overline{c}$ that are not allowed for the values of $\overline{x}$ that led us there. In other words, the partition we end up in will correspond to values for $\overline{c}$ that will satisfy Φ for the values of $\overline{x}$ that led us there. We will now provide a more formal argument for the correctness of the annotation rules in the proof of the following theorem.

Theorem 13
Annotations in the rules in Figure 5.1 are n-partial interpolants for the respective conclusions with respect to the partitions φ_i.

Proof
We prove the theorem using induction over the proof structure. All annotations must satisfy the conditions of an n-partial interpolant. Note that $\bigvee i \oplus i = \bot$ and if $i' \neq i$ then $\bigvee i' \oplus i = \top$.
Base cases:

- **mHyp:** Let $i' = i$. Since $C \in \varphi_i$, $C|_{i'} = C$. Therefore, $C|_{i'} \vee \bigvee i \oplus i' = C$. Since $C \in \varphi_i$, $\varphi_{i'} \rightarrow C$. Therefore, $\varphi_{i'} \rightarrow C|_{i'} \vee \bigvee i \oplus i'$. Now, let $i' \neq i$. Therefore, $C|_{i'} \vee \bigvee i \oplus i' = \top$. Therefore, $\varphi_{i'} \rightarrow C|_{i'} \vee \bigvee i \oplus i'$. Furthermore, since i consists of propositional constants only, we have $i \preceq G$.

- **mAxi:** Let $i' = i$. Since $C \preceq \varphi_i$, $C|_{i'} = C$. Therefore, $C|_{i'} \vee \bigvee i \oplus i' = C$. Since C is a theory lemma, $\varphi_i \rightarrow C|_{i'} \vee \bigvee i \oplus i'$. For $i' \neq i$, we again have $C|_{i'} \vee \bigvee i \oplus i' = \top$ and thus proceed as in the previous rule. Also, $i \preceq G$, just as reasoned above.

Inductive cases:
As induction hypothesis, we assume that the annotations of the premises are correct n-partial interpolants.

- **mRes:** Let $i' = i$. From the induction hypothesis it follows that

$$\varphi_{i'} \rightarrow a \vee C \vee \bigvee i \oplus i' \text{ and } \varphi_{i'} \rightarrow \neg a \vee D \vee \bigvee i \oplus i'. \tag{5.16}$$

  Following a "resolution-like" reasoning, it follows that $\varphi_{i'} \rightarrow C \vee D \vee \bigvee i \oplus i'$. For $i' \neq i$, the disjunct $\bigvee i \oplus i'$ becomes $\top$ again, and we can reason as in the base cases. $i \preceq G$ also follows from the same argument as in the base cases.

- **mRes-G:** Due to the definition of n-partial interpolants, for each $i \in \mathbb{B}^n$ we have

$$\varphi_i \rightarrow a \vee C|_i \vee \bigvee \overline{\chi}^C \oplus i \text{ and } \varphi_i \rightarrow \neg a \vee D|_i \vee \bigvee \overline{\chi}^D \oplus i. \tag{5.17}$$

  After taking conjunction of the two implications, we obtain

$$\varphi_i \rightarrow (a \vee C|_i \vee \bigvee \overline{\chi}^C \oplus i) \wedge (\neg a \vee D|_i \vee \bigvee \overline{\chi}^D \oplus i). \tag{5.18}$$

After expanding the definition of or-of-xor of vectors and moving a and $\neg a$ inside the vector disjunction, we obtain

$$\varphi_i \to (C|_i \vee \bigvee_{j=0}^{n} (a \vee \chi_j^C \oplus i_j)) \wedge (D|_i \vee \bigvee_{j=0}^{n} (\neg a \vee \chi_j^D \oplus i_j)). \tag{5.19}$$

Using $(a \vee b) \wedge (c \vee d) \vdash (a \vee c) \vee (b \wedge d)$, we obtain

$$\varphi_i \to (C \vee D)|_i \vee (\bigvee_{j=0}^{n} (a \vee \chi_j^C \oplus i_j) \wedge \bigvee_{j=0}^{n} (\neg a \vee \chi_j^D \oplus i_j)). \tag{5.20}$$

After moving out the disjunctions, we obtain

$$\varphi_i \to (C \vee D)|_i \vee \bigvee_{j=0}^{n} ((a \vee \chi_j^C \oplus i_j) \wedge (\neg a \vee \chi_j^D \oplus i_j)). \tag{5.21}$$

After moving out the $\oplus$ operator, we obtain

$$\varphi_i \to (C \vee D)|_i \vee \bigvee_{j=0}^{n} ((a \vee \chi_j^C) \wedge (\neg a \vee \chi_j^D)) \oplus i_j. \tag{5.22}$$

Since $\{a, \overline{\chi}^C, \overline{\chi}^D\} \preceq G$, symbols in the annotation of the conclusion are also within G. **Q. E. D.**

5.2.2 The Need for Local-first Proofs

The local-first property is actually needed to compute coordinated interpolants. Using an example, we will illustrate that without this property we could potentially get incorrect results.

Example 14
Consider the formula $\forall a, b \,.\, \exists c_1, c_2 \,.\, \forall l \,.\; \Phi(a, b, c_1, c_2, l)$, *where*

$$\begin{aligned}\Phi(a, b, c_1, c_2, l) =& (c_1 \wedge \neg c_2 \wedge \neg a) \vee (\neg c_1 \wedge c_2 \wedge \neg b) \vee \\ & (c_1 \wedge c_2 \wedge ((\neg l \wedge a) \vee (l \wedge b)))\end{aligned} \tag{5.23}$$

and — for the sake of simplicity — all variables are Boolean. We will compute certificates for c_1 *and* c_2 *in terms of* a *and* b*. After instantiating for all values of* c_1 *and* c_2*, and negating the instantiated formulas, we obtain*

$$\varphi_{00} = \top \tag{5.24}$$
$$\varphi_{01} = b \tag{5.25}$$
$$\varphi_{10} = a \tag{5.26}$$
$$\varphi_{11} = (l \vee \neg a) \wedge (\neg l \vee \neg b) \tag{5.27}$$
$$\varphi = \varphi_{00} \wedge \varphi_{01} \wedge \varphi_{10} \wedge \varphi_{11}. \tag{5.28}$$

The set of global variables is $G = \{a, b\}$. Since l appears only in φ_{11}, we do not need to rename it. Suppose the following is the proof of unsatisfiability of φ produced by an SMT solver.

$$\text{Res}\,\dfrac{\text{Res}\,\dfrac{a \quad l \vee \neg a}{l} \qquad \text{Res}\,\dfrac{b \quad \neg l \vee \neg b}{\neg l}}{\bot}$$

In the above proof, l is used as a resolving literal in the very last proof step. Thus, the proof violates the local-first property. If we compute an interpolant between $\varphi_{00} \wedge \varphi_{01}$ and $\varphi_{10} \wedge \varphi_{11}$ by applying Pudlák's original interpolation rules [Pud97] on the proof, then we obtain the following annotated proof.

$$\text{Res}\,\dfrac{\text{Res}\,\dfrac{a\,[\top] \quad l \vee \neg a\,[\top]}{l\,[\top]} \qquad \text{Res}\,\dfrac{b\,[\bot] \quad \neg l \vee \neg b\,[\top]}{\neg l\,[b]}}{\bot\,[b]}$$

Similarly, if we compute an interpolant between $\varphi_{00} \wedge \varphi_{10}$ and $\varphi_{01} \wedge \varphi_{11}$ by applying the original interpolation rules on the proof, then we obtain the following annotated proof.

$$\text{Res}\,\dfrac{\text{Res}\,\dfrac{a\,[\bot] \quad l \vee \neg a\,[\top]}{l\,[a]} \qquad \text{Res}\,\dfrac{b\,[\top] \quad \neg l \vee \neg b\,[\top]}{\neg l\,[\top]}}{\bot\,[a]}$$

From the above annotations, we learn $\chi_{c_1} = a$ and $\chi_{c_1} = b$, which are not valid certificates if they are used together because $\Phi(a, b, a, b, l)$ is not a valid formula. Lets consider the following unsatisfiability proof, which satisfies the local-first property.

$$\text{Res}\,\dfrac{\text{Res}\,\dfrac{\text{Res}\,\dfrac{l \vee \neg a \quad \neg l \vee \neg b}{\neg a \vee \neg b} \quad a}{\neg b} \quad b}{\bot}$$

If we annotate this proof with our annotation rules, we obtain

$$\text{Res}\,\dfrac{\text{Res}\,\dfrac{\text{Res}\,\dfrac{\vee \neg a\,[(\top, \top)] \quad \neg l \vee \neg b\,[(\top, \top)]}{\neg a \vee \neg b\,[(\top, \top)]} \quad a\,[(\top, \bot)]}{\neg b\,[(\top, a)]} \quad b\,[(\bot, \top)]}{\bot\,[(b, \neg b \vee a)]}.$$

From the above annotations we learn $\chi_{c_1} = b$ and $\chi_{c_1} = \neg b \vee a$, which are valid certificates.

It is also noteworthy that McMillan's interpolation rules [McM05] do not produce coordinated interpolants, not even with a local-first proof. The reason for that is that McMillan's system is asymmetric with respect to the partitions, whereas Pudlák's system is symmetric.

5.2.3 Creating an Implementation from an n-Interpolant

Since an n-interpolant — computed according to the rules in Figure 5.1 — is always quantifier-free, we can easily convert it into an implementation. To create a circuit for one element of the n-interpolant, we create, for every resolution node with a global resolving literal, a multiplexer that has the resolving literal at its selector input. The other inputs connect to the outputs of the multiplexers corresponding to the child nodes. For leaf nodes and resolution nodes with local resolving literals, we use the constants $\top, \bot$, depending on which partition the node belongs to. The output of the multiplexer corresponding to the root node is the final certificate function. Note that, unless we apply logical simplifications, the circuits for all certificates all have the same multiplexer tree and differ only in the constants at the leaves of this tree.

Also note that due to the local-first property, all nodes that are derived from a single partition are annotated with the same n-partial interpolant. Thus, we can disregard such local sub-trees, by iteratively converting nodes that have only descendants from one partition into leaves. This does not affect the outcome of the interpolation procedure.

Now, this gives us a Boolean circuit which has the resolving literals of the refutation proof as its inputs. Resolving literals can either be propositional variables, equalities between two domain terms, or instances of uninterpreted predicates. Propositional variables can be used directly in a Boolean circuit. For equalities between domain terms, we create a comparator. It is interesting to note that, unlike in the certificate computation process described in Section 4.2.5, interpolation can introduce new domain terms — particularly instances of uninterpreted functions — that were not present in the original formula. The reason for this will be explained in Section 5.3.1. This means that, potentially, we have to duplicate the combinational circuits that are represented by uninterpreted functions in order to have the new domain terms available as comparator inputs. The same holds for new instances of uninterpreted predicates.

5.3 Proof Transformations

Our n-interpolation procedure requires refutation proofs to be colorable and local-first. These properties are not guaranteed by efficient modern SMT solvers. In this section we will show how to transform a refutation proof conforming to Definition 15 (page 28) into one that is colorable and local-first. Our proof transformation works in two steps. First, we will split any non-colorable theory lemmata, similar to the technique presented in [FGG^{+}12]. This gives us a colorable proof. In the second step, we will reorder resolution steps to obtain the local-first property. Reordering is based on standard techniques from literature. (See, for example, [DKPW10].)

5.3.1 Obtaining a Colorable Proof

There are two reasons for which a proof may not be colorable. Non-colorable literals (introduced as part of a theory lemma), and non-colorable theory lemmata.

Non-Colorable Literals

Let us first see how non-colorable literals can be introduced into a proof.

Example 15
Let a_0 and a_1 be (domain) variables from two different partitions, and let x be a global (domain) variable. Furthermore, suppose that the partition of a_0 logically entails $a_0 = x$, and the partition of a_1 entails $x = a_1$. Then, based on the transitivity axiom (see Equation 2.3), an SMT solver may introduce the following theory lemma:

$$(a_0 \neq x) \vee (x \neq a_1) \vee (a_0 = a_1). \tag{5.29}$$

Note that this lemma contains the non-colorable literal $a_0 = a_1$.

In [HGK$^+$13], we presented a way to remove non-colorable literals from a proof. This approach was based on the fact that in a refutation proof every literal is resolved eventually, and that the original formula does (per definition) not contain any non-colorable literals. Using this information, the proof can be restructured to remove the non-colorable literal. However, upon closer investigation, it turns out that there is actually a much simpler solution that will be explained in the remainder of this section.

The question one should ask is *why* a DPLL(T)-based SMT solver introduces new literals in the first place. As outlined in Section 2.2.3, the DPLL(T) algorithm checks partial assignments (which are conjunctions of literals) for theory-consistency, and introduces blocking clauses (which are the negation of inconsistent assignments). By construction, the blocking clauses can only contain literals that occur in the formula to solve. Also, blocking clauses are, by definition, theory lemmata. If the solver would just introduce these blocking clauses, the final refutation proof would not have any non-colorable literals. However, one important reason for producing refutation proofs is that a third party should be able to easily verify that the formula in question is indeed unsatisfiable. Theory lemmata resulting from blocking clauses can, however, be rather larger, and it might not at all be obvious that they are indeed theory lemmata. To remedy this situation, many good SMT solvers provide proofs that long and complex theory lemmata are a logical consequence of simpler and easier to check theory lemmata. These proofs can contain literals that did not occur in the original formula to solve and thus can potentially be non-colorable.

Example 16
Let a_0 and a_1 be two local variables, as in Example 15. Let x and y be two global variables. Furthermore, let $f(\cdot)$ be a unary uninterpreted function. Suppose

that during a run of DPLL(T), an SMT solver has made the following (partial) assignment to theory literals:

$$(x \neq a_0) \lor (a_0 \neq y) \lor (y \neq a_1) \lor (f(x) = f(a_1)) \tag{5.30}$$

This assignment is not consistent with $\mathcal{T}_U$. However, to make this easily verifiable, a solver might want to prove this fact based on simple theory lemmata. Such a proof could look like this:

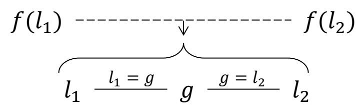
$$
\textsc{Res}\;\dfrac{
 \textsc{Res}\;\dfrac{
   \text{Trans.}\dfrac{}{\begin{array}{c}(x \neq a_0)\lor\\ (a_0\neq a_1)\lor\\ (x=a_1)\end{array}}
   \quad
   \text{Trans.}\dfrac{}{\begin{array}{c}(a_0\neq y)\lor\\(y\neq a_1)\lor\\(a_0=a_1)\end{array}}
 }{\begin{array}{c}(x\neq a_0)\lor(a_0\neq y)\lor\\(y\neq a_1)\lor(x=a_1)\end{array}}
 \quad
 \text{Congr.}\dfrac{}{\begin{array}{c}(x\neq a_1)\lor\\(f(x)=f(a_1))\end{array}}
}{(x\neq a_0)\lor(a_0\neq y)\lor(y\neq a_1)\lor(f(x)=f(a_1))}
$$

In this proof, all leaves are simple instances of the theory axioms transitivity (Equation 2.3) and function congruence (Equation 2.4). Thus, they are more easily verifiable than the final theory lemma. Note, however, that the proof for the theory lemma contains the non-colorable literal $(a_0 = a_1)$ that did not occur in the original formula to solve (and thus also does not occur in the final theory lemma).

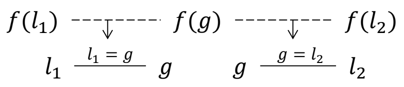
Since we are just interested in computing (n-) interpolants, and not in (easily) checking proofs, we can simply disregard the proofs of (complex) theory lemmata. That is, if a proof node is derived solely from theory lemmata — and thus is a theory lemmata itself — we make this node a leaf of the proof by removing its children. This way, the proof will only contain literals that occur in the original formula to solve. By definition, all those literals are colorable. Therefore, we will henceforth assume that all literals occurring in proofs are colorable.

Splitting Non-Colorable Theory Lemmata

Even if all literals occurring in a proof are colorable, the proof can still contain non-colorable theory lemmata. For example, the theory lemma in Example 16 contains the literals $x \neq a_0$ and $y \neq a_1$, which have different color. We will show how to split such non-colorable theory lemmata into colorable ones. Our splitting approach is a generalization of the one used in [FGG$^+$12] to more than two colors. We proceed as follows. First, we convert the theory lemma back into an assignment that is not $\mathcal{T}$-satisfiable by inverting all literals and converting the disjunctions to conjunctions. Next, we compute the congruence graph over all positive literals in the unsatisfiable assignment. For each of the negative literals of the assignment, we then check whether its positive form is implied by the congruence graph, and compute the corresponding transitivity-congruence chain. Note that since the assignment is unsatisfiable, at least one such literal (and the corresponding transitivity-congruence chain) must exist.



(a) Non-Colorable Transitivity-Congruence Chain for $(f(l_1) \rightsquigarrow f(l_2))$



(b) Colorable Transitivity-Congruence Chain for $(f(l_1) \rightsquigarrow f(l_2))$

Figure 5.2: Splitting a non-colorable transitivity-congruence chain by introducing global intermediate terms.

The transitivity-congruence chain will be the basis for splitting the non-colorable theory lemma. As a prerequisite, we need to make all edges in the chain colorable. A *colorable edge* is an edge for which there is a color w such that all the edge's literals are w-colorable. Edges with an equality justification already are colorable, as we assumed that no non-colorable literals occur in the theory lemma. Edges with congruence justifications, however, may still be non-colorable. That is, the two terms they connect might belong to different partitions, and/or some of the paths that prove equality for the function/predicate parameters might span over more than one partition. Fuchs et al. [FGG+12] have shown how to recursively make all edges in a chain colorable by introducing global intermediate terms for non-colorable edges. We will illustrate this procedure with a simple example, and refer to [FGG+12] for details.

Example 17
Suppose we have the two local terms $f(l_1)$ and $f(l_2)$, where l_1, l_2 are from two different partitions, and a global term g. (See Figure 5.2.) A possible (non-colorable) congruence justification for $f(l_1) = f(l_2)$ could be given as $(l_1 = g, g = l_2)$. The edge between $f(l_1)$ and $f(l_2)$ is now split into two (colorable) parts: $f(l_1) = f(g)$, with justification $l_1 = g$, and $f(g) = f(l_2)$, with justification $g = l_2$. Note that $f(g)$ is a new term that (possibly) did not appear in the congruence graph before. Since we assumed that there are no non-colorable equality justifications in our graph, such a global intermediate term must always exist. It should be clear how to extend this procedure to n-ary functions.

Note that in a colorable chain, every edge either connects two terms of the same partition, or a global term and a local term. In other words, terms from different partitions are separated by at least one global term between them. We now divide the whole chain into (overlapping) segments, such that each segment uses only w-colorable symbols. The global terms that separate symbols with different colors are part of both segments.[16] Assume for the moment that the chain starts and ends with a global term. We will show how to deal with local terms at the beginning/end of the chain later. For ease of presentation, also assume that the chain consists of only two segments. An extension to chains with more segments can be done by recursion. We take the first segment of the chain (from start to the global term that is at the border to the next segment),

[16] If there is more than one consecutive global term, we arbitrarily choose the last one.

$$\textsc{Res}\dfrac{\textsc{Res}\dfrac{n_1 : [c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq k_g \vee c_g = k_g] \quad n_2 : [f_g \neq h_3 \vee h_3 \neq k_g \vee f_g = k_g]}{n_3 : [c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq h_3 \vee h_3 \neq k_g \vee c_g = k_g]} \quad n_4 : [a_1 \neq b_1 \vee b_1 \neq c_g \vee c_g \neq k_g \vee k_g \neq l_1 \vee a_1 = l_1]}{n_5 : [a_1 \neq b_1 \vee b_1 \neq c_g \vee c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq h_3 \vee h_3 \neq k_g \vee k_g \neq l_1 \vee a_1 = l_1]}$$

Figure 5.3: *Splitting theory lemmata.* Suppose we have created the transitivity-congruence chain $(a_1 \rightsquigarrow b_1 \rightsquigarrow c_g \rightsquigarrow d_2 \rightsquigarrow e_2 \rightsquigarrow f_g \rightsquigarrow h_3 \rightsquigarrow k_g \rightsquigarrow l_1)$ from a theory lemma, where all the edges are colorable. The number in the index indicates the partition of the respective term, with g being used for global terms. First, we consider only the part from the first to the last global term (c_g and k_g, respectively). We "split" this sub-chain into the chains $(c_g \rightsquigarrow d_2 \rightsquigarrow e_2 \rightsquigarrow f_g \rightsquigarrow k_g)$ and $(f_g \rightsquigarrow h_3 \rightsquigarrow k_g)$ and convert them into (colorable) theory lemmata (nodes n_1 and n_2, respectively). By resolution, we obtain n_3. Now, we create the theory lemma in node n_4, which corresponds to all links of the original chain which we have not dealt with already, and a "shortcut" over the part we have already considered: $(a_1 \rightsquigarrow b_1 \rightsquigarrow c_g \rightsquigarrow k_g \rightsquigarrow l_1)$. Note that this is also a colorable theory lemma. By resolution over n_3 and n_4, we obtain n_5, whose clause is identical to the theory lemma from which we started.

plus a new "shortcut" literal that states equality between the last term of the first segment and the last term of the entire chain, and use them as implying literals for a new theory lemma clause. The implied literal of this theory lemma will be an equality between the first and the last term of the entire chain. Next, we create a theory lemma with the literals of the second segment of the chain. Note that the implied literal of this theory lemma (which occurs in positive phase) is the same as the shortcut literal used in the theory lemma corresponding to the first segment. There, however, it occurred in negative phase. Thus, we can use this literal for resolution between the two theory lemmata. We obtain a new internal proof node whose clause has all the literals of the entire chain as implying literals, and an equality between start term and end term of the chain as the implied literal. That is, the clause of this new internal node has the same conclusion as the non-colorable theory lemma from which we started.

In case the start/end of the chain is not a local term, we first deal with the sub-chain from the first to the last global term, as described above. Note that if both start and end of the chain are local terms, they have to belong to the same partition, because otherwise the implied literal would be non-colorable. That would violate the assumption that the proof is free of non-colorable literals. We create a theory lemma with the local literals from the start/end of the chain, and one shortcut literal that equates the first and last global term. This literal can be used for resolution with the implied literal of the node obtained in the previous step.

In summary, this procedure replaces all leaves that have non-colorable theory lemmata with subtrees whose leaves are all colorable theory lemmata, and whose root is labeled with the same clause as the original non-colorable leaf.

Example 18
Fig. 5.3 shows how to split the non-colorable theory lemma $(a_1 \neq b_1 \vee b_1 \neq c_g \vee c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq h_3 \vee h_3 \neq k_g \vee k_g \neq l_1 \vee a_1 = l_1)$.

$$\textsc{Res}\,\dfrac{\textsc{Res}\,\dfrac{g \lor l \lor D \quad \neg g \lor E}{l \lor D \lor E} \quad \neg l \lor C}{C \lor D \lor E} \quad \rightsquigarrow \quad \textsc{Res}\,\dfrac{\textsc{Res}\,\dfrac{g \lor l \lor D \quad \neg l \lor C}{g \lor C \lor D} \quad \neg g \lor E}{C \lor D \lor E}$$

$$\textsc{Res}\,\dfrac{\textsc{Res}\,\dfrac{g \lor l \lor D \quad \neg g \lor l \lor E}{l \lor D \lor E} \quad \neg l \lor C}{C \lor D \lor E} \quad \rightsquigarrow \quad \textsc{Res}\,\dfrac{\textsc{Res}\,\dfrac{g \lor l \lor D \quad \neg l \lor C}{g \lor C \lor D} \quad \textsc{Res}\,\dfrac{\neg g \lor l \lor E \quad \neg l \lor C}{\neg g \lor C \lor E}}{C \lor D \lor E}$$

Figure 5.4: If a local resolving literal l resolves premises that are derived from two different partitions, we can push this resolution step towards the leaves using one of the above transformation rules. After the transformation, the proof first resolves l then g.

5.3.2 Reordering Resolution Steps

The last transformation we need to perform is making the proof local-first. To do this, we use the standard reordering techniques from [DKPW10], which are shown in Figure 5.4.[17] Depending on the matching pattern we can apply one of those two rules to reorder resolution steps. By repeated application we can push resolutions over local resolving literals towards the leaves of the proof, until all their respective premises are derived from just a single partition.

5.3.3 Summarizing the Transformation Steps

The final result of the proof transformation steps outlined in the Sections above is a colorable, local-first proof, from which we can compute an n-interpolant. This is summarized in the following theorem.

Theorem 14
After splitting all non-colorable theory lemmata and reordering the resolution steps, we obtain a colorable, local-first proof.

Proof
Colorability. A proof is colorable, iff all its leaves are colorable (see Definition 18, page 30). There are two kinds of leaves in the proof. The first kind has clauses that are part of the original formula to solve. Since such a clause must belong to exactly one partition, it is colorable. The second kind has clauses that are theory lemmata. All non-colorable theory lemmata have been split into colorable ones by the procedure outlined in Section 5.3.1. Thus, all theory lemmata at leaf nodes are colorable.

Local-first. Resolutions over local resolving literals have been pushed towards the leaves as far as necessary by the reordering procedure outlined in Section 5.3.2. Thus, the proof is local-first. **Q. E. D.**

[17] Note that these rules assume that the proof is redundancy free, which can be achieved by the algorithms presented in [Gup12].

5.4 Modular SMT Solving

To compute n-interpolants, refutation proofs need to be colorable and local-first. In the previous section, we have shown how to transform an arbitrary refutation proof that conforms to Definition 15 (page 28) in such a way that it satisfies these properties. In some instances, however, these transformations (especially reordering to obtain the local-first property) can be too expensive with respect to computational resources. Thus, it is desirable to directly obtain a proof that already satisfies these properties. To achieve this goal, we propose *modular SMT solving*, which is a result of mixing and extending ideas from [McM11] and [BVB$^+$13]. We extend the approach of [BVB$^+$13] from propositional satisfiability to SMT problems. Furthermore, we are interested in tree-like dependencies, whereas [BVB$^+$13] focuses on linear dependencies. The technique we are going to present extends to the generalized version of our synthesis problem, as stated in Section 4.3, where we allow multiple quantifier alternations, as long as all existential quantifiers are over propositional variables only.

5.4.1 Tree-like Modular SMT Problems

A *tree-like modular SMT problem* is an SMT problem where the formula to decide is a conjunction whose conjuncts are distributed over the nodes of a tree. That is, every node in the tree is associated with a formula. Moreover, every node is also associated with a set of literals. The initial[18] set of literals of a node is defined below.

Definition 27 — $ch(v)$, $pa(v)$, $anc(v)$, $desc(v)$, $\phi(v)$
Let v be a node in a tree-like modular SMT problem. Then $ch(v)$ is the set of v's children, or $\emptyset$, if v is a leaf; $pa(v)$ is the parent of v, or $\emptyset$, if v is the root; $anc(v) = \{pa(v)\} \cup anc(pa(v))$ is the set of ancestors of v; $desc(v) = \bigcup_{i \in ch(v)} \{i\} \cup desc(i)$ is the set of descendants of v; and $\phi(v)$ is the formula associated with v.

Definition 28 — Literals of Nodes
Let v be a node in a tree-like modular SMT problem. Let $Lits(\phi(v))$ be the set of literals occurring in $\phi(v)$. Then the set of literals associated with a node is

$$L(v) = \left(\bigcap_{i \in ch(v)} L_a(i) \right) \setminus L(pa(v)), \text{ where} \tag{5.31}$$

$$L_a(v) = Lits(\phi(v)) \cup \bigcup_{i \in ch(v)} L_a(i). \tag{5.32}$$

Intuitively, $L_a(v)$ is the set of *all* literals "reachable" from node v while traversing the tree towards the leaves. On the other hand $L(v)$ is the set of literals which are reachable from all children of node v, but not from any siblings of v — as

[18] Due to interpolation-based splitting of theory lemmata, these sets may grow during the solving process. Details will follow in Section 5.4.2.

otherwise they would be in $L(pa(v))$. The sets $L_a(v)$ can be computed bottom-up (that is, from the leaves to the root), while the sets $L(v)$ can be computed top-down (that is, from the root to the leaves).

The semantics of a tree-like modular SMT problem is the conjunction of the formulas of all nodes. That is, by forming this conjunction, the tree-like modular SMT problem is converted into an equivalent regular SMT problem. More formally, we define a function $solve(v, \alpha)$ that takes a node v and a (partial) assignment α over the literals of v's ancestors such that calling *solve* on the root node and with an empty assignment returns the overall solution of the tree-like modular SMT problem.

Definition 29 — $\Gamma(v)$
Let v be a node in a tree-like modular SMT problem. Then $\Gamma(v) = \bigwedge_{i \,\in\, desc(v)} \phi(i)$ is the conjunction of the formulas associated with the descendants of v.

Definition 30 — $\phi[\alpha]$
Let ϕ be a first-order formula. Let α be a (partial) assignment to the literals occurring in ϕ. Then

$$\phi[\alpha] = \begin{cases} \top & \textit{if } \phi \textit{ evaluates to true under } \alpha, \\ \bot & \textit{if } \phi \textit{ evaluates to false under } \alpha, \\ ? & \textit{otherwise.} \end{cases} \tag{5.33}$$

Note that the "?" case can occur because α is not necessarily a full assignment.

Definition 31 — $A(v, \alpha, \beta)$
Let v be a node in a tree-like modular SMT problem. Let α be a (partial) assignment of the literals in $\bigcup_{i \,\in\, anc(v)} L(i)$. Let β be a full assignment of the literals in $L(v)$. In the following, γ represents a full assignment of the literals in $\bigcup_{i \,\in\, desc(v)} L_a(i)$. We define

$$A(v, \alpha, \beta) = \begin{cases} \gamma & \textit{if there exists a } \gamma \textit{ such that } \alpha \cup \beta \cup \gamma \models \mathcal{T} \textit{ and} \\ & \Gamma(v)[\alpha \cup \beta \cup \gamma] = \top, \\ \bot & \textit{if for all possible } \gamma \textit{ it holds that } \Gamma(v)[\alpha \cup \beta \cup \gamma] = \bot \textit{ or} \\ & \alpha \cup \beta \cup \gamma \not\models \mathcal{T}, \\ ? & \textit{otherwise.} \end{cases}$$

Definition 32 — $solve(v, \alpha)$
Let v be a node in a tree-like modular SMT problem. Let α be a (partial) assignment of the literals in $\bigcup_{i \,\in\, anc(v)} L(i)$. In the following, β represents a full assignment of the literals in $L(v)$, and γ represents a full assignment of the literals in $\bigcup_{i \,\in\, desc(v)} L_a(i)$. We define

$$solve(v, \alpha) = \begin{cases} \beta \cup A(v, \alpha, \beta) & \textit{if there exists a } \beta \textit{ such that } A(v, \alpha, \beta) \neq \bot \textit{ and} \\ & A(v, \alpha, \beta) \neq ? \textit{ and } \phi(v)[\alpha \cup \beta \cup A(v, \alpha, \beta)] = \top, \\ \bot & \textit{if for all possible } \beta \textit{ it holds that } A(v, \alpha, \beta) = \bot \\ & \textit{or } \phi(v)[\alpha \cup \beta \cup A(v, \alpha, \beta)] = \bot, \\ ? & \textit{otherwise.} \end{cases}$$

If $solve(r, \emptyset)$, where r is the root node of a tree-like modular SMT problem, returns $\bot$, the problem is unsatisfiable. If it returns an assignment, the problem is satisfiable. Note that $solve(r, \emptyset)$ cannot return "?", because the root has no ancestors and thus there are no unassigned literals left during the evaluation of the formulas of the nodes.

Henceforth, we will focus on tree-like modular SMT problems for which the formula in all internal nodes is (initially[19]) empty (corresponding to $\top$). The generalization to non-empty internal nodes is straightforward but not relevant for the synthesis problems we consider.

From Synthesis Problems to Modular SMT Problems

We consider formulas of the form

$$\Phi^{Q^+} = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \exists \overline{c}' \,.\, \forall \overline{x}'' \,.\, \dots\ \Phi, \tag{5.34}$$

and wish to synthesize certificates for the existentially quantified Boolean variables $\overline{c}, \overline{c}', \dots$. To do so, we first expand the innermost existential quantifier, as shown in the proof of Theorem 4 in Section 4.1. We then proceed to expand the next existential quantifier (which is now the innermost existential quantifier), and continue this expansion, until all existential quantifiers have been expanded. The partitions of the resulting formula are then assigned to the leaves of a tree-like modular SMT problem, as shown in the following example.

Example 19
Consider the following formula with two levels of existential quantifiers:

$$\Phi^{Q^+} = \forall \overline{x} \,.\, \exists c_1, c_2 \,.\, \forall \overline{x}' \,.\, \exists c_1', c_2' \,.\, \forall \overline{x}'' \,.\, \Phi. \tag{5.35}$$

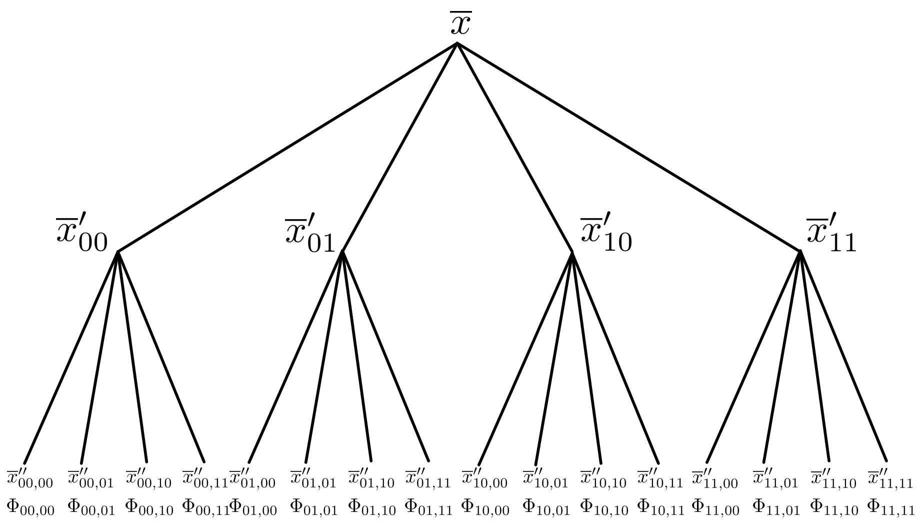
When we expand the innermost existential quantifier and rename the inner universal variables, we obtain

$$\Phi^{Q^+} = \forall \overline{x} \,.\, \exists c_1, c_2 \,.\, \forall \overline{x}', \overline{x}''_{00}, \overline{x}''_{01}, \overline{x}''_{10}, \overline{x}''_{11} \,.\, \Phi_{00} \vee \Phi_{01} \vee \Phi_{10} \vee \Phi_{11}. \tag{5.36}$$

When we now expand the remaining existential quantifier, we obtain

$$\begin{aligned}\Phi^{Q^+} = \forall \overline{x},\ & \overline{x}'_{00},\ \overline{x}''_{00,00},\ \overline{x}''_{00,01},\ \overline{x}''_{00,10},\ \overline{x}''_{00,11}, \\ & \overline{x}'_{01},\ \overline{x}''_{01,00},\ \overline{x}''_{01,01},\ \overline{x}''_{01,10},\ \overline{x}''_{01,11}, \\ & \overline{x}'_{10},\ \overline{x}''_{10,00},\ \overline{x}''_{10,01},\ \overline{x}''_{10,10},\ \overline{x}''_{10,11}, \\ & \overline{x}'_{11},\ \overline{x}''_{11,00},\ \overline{x}''_{11,01},\ \overline{x}''_{11,10},\ \overline{x}''_{11,11} \,.\ && \Phi_{00,00} \vee \Phi_{00,01} \vee \Phi_{00,10} \vee \Phi_{00,11} \vee \\ &&& \Phi_{01,00} \vee \Phi_{01,01} \vee \Phi_{01,10} \vee \Phi_{01,11} \vee \\ &&& \Phi_{10,00} \vee \Phi_{10,01} \vee \Phi_{10,10} \vee \Phi_{10,11} \vee \\ &&& \Phi_{11,00} \vee \Phi_{11,01} \vee \Phi_{11,10} \vee \Phi_{11,11}.\end{aligned} \tag{5.37}$$

[19]Due to conflict-driven clause learning and the addition of theory lemmata that serve as blocking clauses for theory-inconsistent (partial) assignments, the formula of a node may change over time — while always preserving the overall semantics of the tree-like modular SMT problem.



$\overline{x}$
$\overline{x}'_{00}$
$\overline{x}'_{01}$
$\overline{x}'_{10}$
$\overline{x}'_{11}$
$\overline{x}''_{00,00}$ $\overline{x}''_{00,01}$ $\overline{x}''_{00,10}$ $\overline{x}''_{00,11}$ $\overline{x}''_{01,00}$ $\overline{x}''_{01,01}$ $\overline{x}''_{01,10}$ $\overline{x}''_{01,11}$ $\overline{x}''_{10,00}$ $\overline{x}''_{10,01}$ $\overline{x}''_{10,10}$ $\overline{x}''_{10,11}$ $\overline{x}''_{11,00}$ $\overline{x}''_{11,01}$ $\overline{x}''_{11,10}$ $\overline{x}''_{11,11}$
$\Phi_{00,00}$ $\Phi_{00,01}$ $\Phi_{00,10}$ $\Phi_{00,11}$ $\Phi_{01,00}$ $\Phi_{01,01}$ $\Phi_{01,10}$ $\Phi_{01,11}$ $\Phi_{10,00}$ $\Phi_{10,01}$ $\Phi_{10,10}$ $\Phi_{10,11}$ $\Phi_{11,00}$ $\Phi_{11,01}$ $\Phi_{11,10}$ $\Phi_{11,11}$

Figure 5.5: Tree-Like Modular SMT Problem. Each leaf of the tree is associated with one formula $\Phi_{i,j}$. Each node is associated with a set of literals.

We now assign the 16 partitions $\Phi_{i,j}$ we obtained to the leaves of a tree-like modular SMT problem, as shown in Figure 5.5. Note that the number of partitions we obtain is independent of the number of quantifier alternations. The number of partitions solely depends on the number of the existentially quantified variables, regardless of how they are distributed over the quantifier levels. The number of quantifier levels, however, determines the height of the tree of the resulting modular SMT problem.

5.4.2 Solving Algorithm

We propose the following algorithm for solving tree-like modular SMT problems and explain it based on an example where the tree consists of just one root node with two leaves as its direct children. The generalization to larger trees is straightforward.

Every node in the tree runs an instance of the DPLL(T) algorithm to find (partial) assignments for the literals associated with the node that satisfy the formula of the node (modulo the considered theory). Once a (partial) assignment has been found, it is communicated to the child nodes. Each child node now tries to extend the given assignment by finding a (partial) assignment for its own variables so that the conjunction of the parent assignment and its own assignment satisfies the formula of the child node.[20] There are three possible outcomes for each child node. First, the child may find an extension to the given assignment that satisfies its formula. In this case, the extended assignment is

[20]Note that this can be done in parallel for all of the children of a parent node.

communicated back to the parent node. Second, under the given partial assignment from the parent node, the formula of the child may be unsatisfiable ($\perp$). This is communicated back to the parent. Third, a child may not be able to determine whether or not a satisfying extension of the given assignment exists, because the partial assignment obtained from the parent is not complete enough yet. In this case "unknown" ("?") is communicated back to the parent node.

Once a parent node has received the answers from all its children, there are again three possible cases. If one or more children report unsatisfiable, the parent learns a blocking clause corresponding to its current partial assignment to avoid trying the same (partial) assignment again in later steps. If no child reports unsatisfiable, but one or more children report "unknown", the parent makes another decision, extending its own partial assignment and queries all children again, now with the extended partial assignment. In the remaining case, all children have reported satisfiable, along with their respective assignments. In this case, the parent takes the conjunction of all assignments obtained by the children and checks it for theory consistency.[21] If the conjunction is theory consistent (and we are at the root node), we have found a satisfying, theory-consistent assignment for all partitions, and the overall result is satisfiable.

The more interesting case is the one where the conjunction is not theory consistent. In this case, we need to somehow block this conjunction of assignments from occurring again. In a regular SMT setting, we would simply learn and add a blocking clause. However, as this blocking clause would contain (local) literals from more than one partition, there is no node to which we can add it while preserving the modular structure of the SMT problem. Therefore, we proceed as follows. Assume for ease of presentation that there are just two child nodes. If the conjunction of their assignments is unsatisfiable, we compute an interpolant with respect to the two assignments. We then perform Tseitin's encoding on the interpolant to obtain a single literal that represents it. The clauses that result from Tseitin's encoding are added to the current node, as they can (due to the definition of an interpolant) only contain literals that are in the set of literals of this node. Next, we create a clause that will be added to the first child node. We take all the literals of the assignment of the first child, negate them, and add them to the clause. Furthermore, we add the literal that represents the interpolant. The resulting clause basically states that the assignment of the first child implies the interpolant, which is true by definition. Thus, adding this clause does not restrict the formula to solve in any way. For the second child, we do the same thing, except that we add the negation of the literal that represents the interpolant. The literal that represents the interpolant is added to the set of literals of the parent node. Note that no matter what value the parent node assigns to this literal, at least one of the children will compute a different

[21]This is necessary because theory consistency is not a compositional property. That is, even though the assignment of every child may be theory consistent, their conjunction may still be theory inconsistent.

assignment in subsequent steps.

Lemma 5
The tree-like modular SMT problem that results from adding clauses as described above is semantically equivalent to the original tree-like modular SMT problem.

Proof
This follows trivially from the definition of an interpolant and from the fact that blocking clauses are theory lemmata. **Q. E. D.**

Example 20
Consider a node with two child nodes. Let the assignments A_0 and A_1 returned from the children be

$$A_0 := \qquad (x = u) \wedge (u = z) \tag{5.38}$$

$$A_1 := \qquad (x = v) \wedge (v \neq z) \tag{5.39}$$

Obviously, the conjunction $A_0 \wedge A_1$ is not $\mathcal{T}_U$-satisfiable. One possible interpolant between A_0 and A_1 is $\chi := (x = z)$. Since in this example the interpolant already is a single literal, we do not need to perform Tseitin's encoding. Thus, we add the following clause to the first child's formula

$$(x \neq u) \vee (u \neq z) \vee \chi, \tag{5.40}$$

and the following clause to the second's child formula

$$(x \neq v) \vee (v = z) \vee \neg\chi. \tag{5.41}$$

The new literal χ is added to the set of literals of the parent node.

The procedure terminates in the following two cases. If the root node obtained assignments from all its children and the conjunction of these assignments is theory consistent, then the procedure terminates with the overall result “satisfiable”. The second case is that the root node has tried all possible assignments of its literals, and none of them were satisfiable with respect to all children. In this case, the procedure terminates and returns unsatisfiable.

Note that this modular SMT solving procedure is (almost) theory-agnostic. The only prerequisites are that the theory (or fragment, respectively) is decidable, and that there is a procedure to compute interpolants for conjunctions of theory literals that are again in the considered theory (fragment, respectively). For example, $\mathcal{T}_U^{\mathrm{qf}}$ satisfies these prerequisites. It is decidable and quantifier-free interpolants can be computed [FGG+12]. On the other hand, for example the quantifier-free fragment of the theory of linear integer arithmetic does not satisfy these properties. For some unsatisfiable formulas in linear integer arithmetic, no quantifier-free interpolant exists.

Theorem 15
The modular SMT solving procedure outlined above terminates and produces correct results.

Proof
Termination. The number of new literals introduced due to interpolation of child assignments is finite for each node. This is due to the fact that for each theory inconsistent conjunction of child assignments (of which there are only finitely many), interpolation and introduction of new literals is only performed once. The clauses added after interpolation prevent the same inconsistent conjunction of child assignments from occurring again. Thus, every node runs DPLL(T) on a finite number of literals and a finite formula. DPLL(T) is known to terminate, thus every node (including the root node) terminates.

Correctness. The procedure returns satisfiable, if a theory-consistent assignment that satisfies the formula at every (leaf) node has been found. Thus, the conjunction of all these formulas actually was satisfiable and the result is correct. The procedure returns unsatisfiable, if the root node has tried all possible assignments to its literals and none of them could be extended to assignments satisfying the formulas at all (leaf) nodes. Thus, the conjunction of this formulas actually is not satisfiable and the result is correct. **Q. E. D.**

5.4.3 Proof Generation

When solving a formula that corresponds to a synthesis problem — such as the expanded version of Equation 5.34 — all non-leaf nodes of the modular SMT solver are initialized with empty formulas. At the end of the procedure — once unsatisfiability has been established — the non-leaf nodes will contain clauses learned from conflicts and theory lemmata. In particular, the root node will contain a set of clauses whose conjunction is unsatisfiable and a refutation proof can be computed in the standard way. This corresponds to the green part of the proof in Figure 5.6. Note that, by construction, the clauses in the root node only use literals that are in the set of literals associated with the root node. In terms of our synthesis problem, that would be global literals, corresponding to variables in the outermost universal quantifier. Thus, also the refutation proof contains only such literals. Every clause in the root node is either a theory lemma, or a logical consequence of one of its children; namely from the one that reported unsatisfiable at the time the clause was learned. From the conflict that lead the child to reporting unsatisfiable, we can construct a proof for the learned clause. This corresponds to the red and the blue part in Figure 5.6. This proof only uses literals that are associated with the respective child node, or literals from any node on a path to the root. We can continue to construct proofs for learned clauses in this way, until we reach the leaves and thus clauses that are part of our original formula to solve. Concerning theory lemmata, there are two cases. Theory lemmata that were learned directly at a node only contain literals from this node and nodes on the path to the root. Thus, they cause no problems with respect to obtaining a colorable, local-first proof. Theory lemmata that were learned while checking the conjunction of several child nodes are not used directly, but split into colorable theory lemmata using interpolation, as outlined in the previous section. Thus, they also cause no problems. Thus, we can obtain

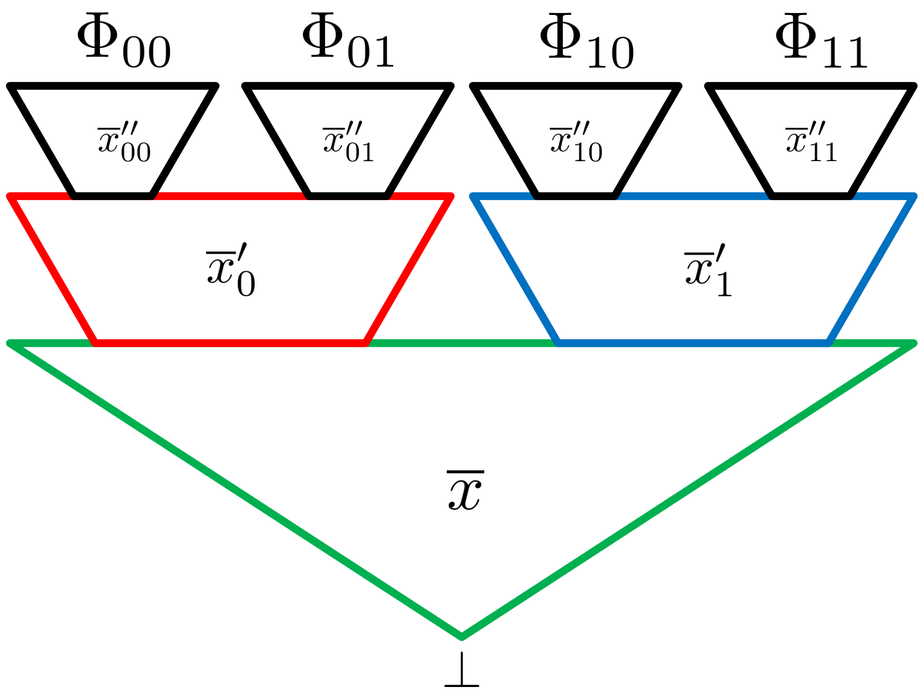


Figure 5.6: Modular Proof. Each part consists only of resolution steps over the variables from the label of the respective part or a label of any part that is on a path to the root.

a proof with a structure as shown in Figure 5.6. This proof is colorable and local-first by construction. It even has a generalized local-first property, that is, it is local-first with respect to the expansion of each level of existential quantification.

Interpolation

We can use one single proof obtained from the procedure outlined above to compute certificates for all existentially quantified variables of all quantifier levels, using n-interpolation. When computing certificates for the variables of the innermost existential quantifier, we proceed as outlined in Section 5.2. We remove all the local derivations (black parts in Figure 5.6) from the proof and compute an n-interpolant. To compute certificates for the variables of the next existential quantifier, we can remove one more level of (now) local derivations (black, red, and blue parts in Figure 5.6) and compute another n-interpolant on this truncated proof. In other words the difference when computing n-interpolants for different levels of existential quantification is just how much of the proof can be regarded as local.

6 Other Synthesis Approaches

The previous chapter presented interpolation-based synthesis approaches. In this chapter, we will discuss some alternatives that are based on different decision procedures. These have not been investigated to the same level of detail as interpolation-based synthesis. We just want to sketch the basic ideas of how to base the synthesis problem on different decision procedures. A more detailed analysis and comparison to interpolation-based synthesis remains for future work.

First, we briefly recapitulate the BDD-based cofactor approach which we introduced in [HB11]. Next, we discuss two alternatives to reduce our synthesis problem to the QSAT problem. The first one is inspired by eager SMT encoding techniques, the second one mimics lazy encoding. The last alternative approach that we will briefly discuss is a template-instantiating approach.

6.1 Binary Decision Diagrams

Declaration of Sources

This section is based on and reuses material from the following source, previously published by the author:

- [HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42.

References to this source are not always made explicit.

The BDD-based approach — originally suggested in [HB11] — has been described in Chapter 4. We briefly repeat the main steps and state some experiences. As we have shown in Chapter 4, a formula Φ^{Q} in $\mathcal{S}^{\mathrm{Q}}$ can be reduced to a (quantified) propositional formula Φ^{Q}_{prop}. In [HB11], we suggested to use BDDs to extract certificates from Φ^{Q}_{prop}. This is done by creating a BDD for the quantifier-free suffix of Φ^{Q}_{prop}, using it to perform the inner universal quantification of Φ^{Q}_{prop}, and using a cofactor-based approach to compute certificates.

Our experience (see [HB11]) shows, however, that BDDs seem to be a suboptimal data structure for this kind of problem. The BDD for transitivity constraints can become exponentially large, irrespective of the variable ordering [BV00]. In our experiments, most of the computation time was indeed spent on computing a BDD for the transitivity constraints. Without dynamic reordering, we run out of memory quickly; with dynamic reordering enabled, most of the computation time is spent for reordering. This is illustrated by the following experimental results: In a first experiment, with dynamic reordering enabled, it took approximately 14 hours to compute the BDD for the very simply toy example that was used for illustration in Chapter 3. We stored the variable order that had resulted from dynamic reordering at this point and used it as a fixed order for subsequent experiments, thereby reducing the computation time to roughly 10 minutes. It should also be mentioned that this rather simple example resulted in 151 propositional variables, after the reduction steps.

6.2 Eager Encoding to QSAT

The idea behind eager encoding to QSAT has already been mentioned in the proof of Theorem 8 (see page 55). For a formula $\Phi^{\mathrm{Q}} = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$, we perform the reductions described in Section 4.2 on Φ. As a result, we obtain a QBF $\Phi^{\mathrm{Q}}_{prop} = \forall \overline{b}_x \,.\, \exists \overline{c} \,.\, \forall \overline{b}_{x'} \,.\, \Phi_{prop}$. We can pass this QBF to a certificate-producing QBF solver. The certificates returned are a solution for our synthesis problem. Note that with respect to QBF solvers, there are two types of certificates: syntactic certificates (Q-resolution proofs or Q-consensus proofs), and semantic certificates (Skolem functions or Herbrand functions) [BJ12]. We are interested in the semantic certificates, that is, Skolem functions for the variables in $\overline{c}$.

Preliminary experiments with this approach, however, suggest that for synthesis problems of reasonable size, the number of constraints introduced by the reductions can be very large. In particular, the number of transitivity constraints tends to become so large, that we ran out of (several hundred GB of) disc space while trying to write the input file for a QBF solver. For smaller instances, where the input file for the QBF solver could be generated, the preliminary experiments suggest that the resulting QSAT instances are rather hard to solve for current certificate-producing QBF solvers. As we mentioned in [HGK⁺13], we tried to run DepQBF [LB10] on an example that the interpolation-based synthesis approach can synthesize within a few seconds, and it exhausted all 192 GB of main memory within approximately one hour and did not produce a result. After this rather discouraging first results, this approach was not investigated further.

6.3 Lazy Encoding to QSAT

As we have seen in the previous section, eager encoding to QSAT can produce very large QSAT instances. One obvious way of trying to cope with this problem is to try a more lazy encoding approach instead. Consider again a formula $\Phi^{\mathrm{Q}} = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$. Now instead of eagerly computing all constraints to obtain an equivalent QSAT instance, we simply consider the propositional skeleton $\mathit{skel}(\Phi)$ (see Definition 3, page 20) and pass the QSAT instance $\Phi^{\mathrm{Q}}_{prop} = \forall \overline{b}_x \,.\, \exists \overline{c} \,.\, \forall \overline{b}_{x'} \,.\, \mathit{skel}(\Phi)$ to a QBF solver. If this instance happens to be satisfiable, we can compute certificates and are done. Note that such a certificate would be correct for all possible truth values of equalities between variables in $\overline{x}$ — even those that do not correspond to any actual values of $\overline{x}$ due to, for example, transitivity violations. Thus, such a certificate would clearly be a valid solution for our synthesis problem. It is, however, much more likely that the resulting QSAT instance is unsatisfiable. In this case, we can obtain "certificates"[22] for the universally quantified literals — that is, propositional constants for the variables in $\overline{b}_x$ and a function $\overline{c} \mapsto b_{x'}$ for each variable in $\overline{b}_{x'}$ — that demonstrate this unsatisfiability. Assuming that the original formula Φ^{Q} was valid, these certificates must be inconsistent with the theory. In lazy encoding for SMT solving (see Section 2.2.2), inconsistent assignments are removed from consideration through blocking clauses. In an analogous way, we now want to update our QSAT instance with constraints that prevent the solver from returning the same theory-inconsistent certificate again. We will explain one possible way to do this in the next section. Just as in lazy encoding for SMT solving, we repeat this process until enough constraints have been added so that the QSAT

[22] Our definition of certificates (Definition 16, page 28) considers only existentially quantified variables. However, a generalization to universally quantified variables (in case of an unsatisfiable QSAT instance) is straightforward.

instance becomes satisfiable and we can compute a certificate for the signals to synthesize.

Finding Constraints from an Unsatisfiable QSAT Instance

There are two different cases to consider. Theory inconsistencies can arise just from literals of the outer universal quantifier, or they can also include literals from the inner universal quantifier. Let us first look at the easier case, where the inconsistency comes solely from literals of the outer universal quantifier. Since this is in fact the outermost quantifier, certificates for literals under this quantifier are Boolean constants. Thus, we can use the congruence closure algorithm to check whether or not the assignment of constants to literals is theory consistent. If the assignment is theory consistent, the inconsistency includes literals of the inner universal quantifier. We will deal with this second case in a moment. If, however, the assignment is inconsistent with the theory, we can compute constraints to block this assignment in future iterations. Let A be an unsatisfiable core of the inconsistent assignment. Then we refine the formula of our QSAT instance as follows:

$$\neg A \to \mathit{skel}(\Phi). \tag{6.1}$$

This makes it easier to satisfy the QSAT instance, because any assignment that is a superset of A now trivially satisfies the formula. We iterate this constraint learning procedure until either the QSAT instance becomes satisfiable, or the certificate we obtain does not have an inconsistent assignment for the literals of the outermost universal quantifier. Note that we can potentially learn more than one constraint per iteration, if the inconsistent assignment contains more than one unsatisfiable core.

If there are no inconsistencies within the literals of the outer universal quantifier, yet the QSAT instance is still unsatisfiable, we need to consider the literals of the inner universal quantifier. The certificates for these literals are, however, not constants but functions of the existentially quantified variables in $\overline{c}$. What we need now is an (efficient) way to prevent the solver from returning the same functions again in subsequent calls. One possible approach is as follows: We iterate over all (finitely many) possible values for $\overline{c}$ and for each set of values, we compute the values of the literals of the inner universal quantifier. For at least one value of $\overline{c}$, the resulting values of the literals of the universal quantifiers must be theory inconsistent. Once we have identified such an inconsistency, we can proceed as above and update the formula of the QSAT instance. This is done in the same way as in the first case (see Equation 6.1). The only difference is that the assignment A is now over variables from $\overline{b}_x \cup \overline{c} \cup \overline{b}_{x'}$.

Eventually, we will have learned enough constraints to make the QSAT instance satisfiable and obtain the certificates for the signals to synthesize. However, one potential drawback of this approach is that it might need a very large number of iterations before the QSAT instance becomes satisfiable. Moreover, it is not clear how long each run of the QBF solver will take. We expect that for non-trivial benchmarks the number of iterations is rather high, and also the solving time per iteration will increase steadily. While the first few solver calls (which return “unsatisfiable”) might be rather fast, it should take more and more time, as more constraints are added. At this point, however, this is just an intuition and not backed by a thorough experimental investigation.

6.4 Template Instantiating

The idea of template instantiating stems from the fact that for many synthesis problems, especially for pipeline controllers, very small solutions exist. Thus, trying to “guess” them and check if the guess was correct might be feasible. In its simplest form, this approach can be based on an observation that we mentioned in Section 4.2.5 and formalize in the following lemma.

Lemma 6
Given a formula $\Phi^Q = \forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \Phi$ in S^Q, if there exists a certificate for the variables in $\overline{c}$, there exists one that can be constructed solely by Boolean combinations of propositional variables occurring in $\overline{x}$, uninterpreted predicate instances which occur in Φ and do not contain variables from $\overline{x}'$, and equalities over terms which occur in Φ and do not contain variables from $\overline{x}'$.

Proof
In Section 4.2.5 we have shown a method to transform Φ^Q into an equivalent (quantified) propositional formula Φ^Q_{prop}, from which we can extract such a certificate. This method can be applied to all formulas Φ^Q for which a solution exists. **Q. E. D.**

Based on this lemma, we can find a solution by searching the space of all possible functions over these literals. There are many possible ways to perform this search. In the following, we describe a simple approach that starts with small functions. We first construct the set of literals, as outlined in Lemma 6. Next, we iterate over all literals in this set and check whether the formula consisting of just this one literal (or its negation, respectively) is a valid solution to our synthesis problem. The check can be done by substituting the alleged solution for the corresponding c in Φ, expand over all other variables in $\overline{c} \setminus c$, negate, and check for satisfiability. If the solution is not correct, we try the next literal. If no solution consisting of just one literal is correct, we try Boolean combinations of two literals, then three literals, etc.

Alternatively, we can also enlarge the set of literals by constructing equalities over new terms (which are obtained by applying an uninterpreted function to an existing term), or adding new uninterpreted predicate instances (whose parameters are existing terms). Introducing new literals can facilitate finding short solutions, as illustrated by the following example.

Example 21
Suppose one possible solution for one $c \in \overline{c}$ in a synthesis problem is $P(a)$. Suppose further that the predicate instance $P(a)$ does not occur in the original formula, but the instance $P(x)$ does. Moreover, suppose that also the terms a

and x appear in the original formula. We know that there exists a solution that is expressible solely by the literals listed in Lemma 6. For example, suppose that $a = x \wedge P(x)$ is also a correct solution.[23] *Using only literals listed in Lemma 6, the solution is (at least) two literals long. By introducing the (new) literal $P(a)$, the solution can be shortened to just one single literal.*[24]

An obvious disadvantage of the template instantiation method is that a large number of SMT solver calls is necessary to check potential solutions. On the other hand, most of the checks might be rather fast. Experience suggests that finding a satisfying assignment is usually much faster than proving unsatisfiability. Moreover, an incremental SMT solver could be used to potentially further decrease the time required to perform multiple subsequent checks.

The simplest form of template instantiation that we just described just uses the satisfiable/unsatisfiable answer from the SMT solver. The solver can, however, provide more than that. In case the answer is "satisfiable" — that is, the guessed solution is not correct — the solver can provide a concrete counterexample. That is, an assignment to the variables in the formula such that the current guess computes the opposite of what it should compute. We can use such counterexamples to guide the refinement of the guesses. Such a guided refinement is also called *learning* [Ang87]. In our setting, we can use learning techniques similar to the ones described in [EKH12].

[23]Note that $a = x \wedge P(x)$ implies $P(a)$ (in $\mathcal{T}_U$), but not vice versa.

[24]Note that we disregard the "cost" associated with creating a new predicate instance $P(a)$. If P has a complex combinational implementation, the solution $a = x \wedge P(x)$ might be preferable, if $P(x)$ is already available in the circuit.

7 Implementation and Experimental Results

We have implemented the interpolation-based synthesis approach (Sections 5.1 and 5.2) in a prototype tool called SURAQ. It is available as open source (under LGPL v3); all details can be found on the SURAQ website.[25] Preliminary tests with eager encoding to QSAT (see Section 6.2) were also done with this prototype. In this chapter, we will first describe SURAQ and its development history in more detail. To some extent this will repeat what we described in previous chapters (mostly Chapter 5), but from a more procedural and implementation-specific point of view. We will also include some of the "lessons learned" that are imperative for turning the theoretical concept into an efficient synthesis tool. After that we will describe the benchmarks that we used for the experimental evaluation. Finally, in the last section of this chapter, we are going to present the results of these experiments.

7.1 SURAQ — A Prototype Implementation

SURAQ is short for **S**ynthesizer using **U**ninterpreted Functions, A**r**rays **a**nd E**q**uality.[26] It is implemented in Java and consists of over 130 classes (distributed in roughly 13 sub-packages) and more then 35,000 lines of code. The decision to implement the tool in Java is mostly based on our familiarity with

[25]`http://www.iaik.tugraz.at/content/research/design_verification/suraq/`

[26]Any similarities between the name SURAQ and Surak, the "legendary Vulcan philosopher, scientist, and logician" (see `http://en.memory-alpha.org/wiki/Surak`) are of course purely coincidental. ;-)

the language and the Eclipse IDE. Moreover, we thought that a strongly typed language would ease handling of complex formulas of different (derived) types. We also considered automated garbage collection to be an advantage, mostly taking away the burden of debugging memory leaks. As the original plan was to outsource most of the computational complexity to an external SMT solver anyway, the performance penalties of using the Java virtual machine were expected to be insignificant.

In the remainder of this section, we will explain all the steps the tool performs during a normal run. Along the way, we will give some information on the "history" of the development process and the rationales for the most important design decisions.

7.1.1 Input Format

The first step in a run of SURAQ is to read the specification Φ^{Q} from a file. The input format is based on the SMTLIB (version 2) format [BST10]. In short, SMTLIB is used to state a formula Φ in $\mathcal{S}$ (see Definition 20, page 34). The $\forall$-$\exists$-$\forall$-quantifier prefix that we need to form a formula Φ^{Q} in our specification language $\mathcal{S}^{\mathrm{Q}}$ (Definition 21, page 35) is given implicitly. Annotations and variable types determine which of the variables in Φ is bound by which of the quantifiers.

In more detail, a SURAQ input file is composed of S-expressions and possibly comments, following the basic syntax of SMTLIB. The first expression in a SURAQ input file must be `(set-logic Suraq)`. This implicitly declares an uninterpreted sort `Value` and a Boolean sort `Control`. Following the `set-logic` command, one can use the SMTLIB command `declare-fun` to declare the variables, uninterpreted functions, and uninterpreted predicates occurring in Φ. Variables must be of sort `Value`, `(Array Value Value)` (that is, arrays mapping from indices of sort `Value` to elements of sort `Value`), `Bool`, or `Control`. Variables of type `Control` represent the (Boolean) control signals for which an implementation should be synthesized. That is, these variables are (implicitly) bound by the existential quantifier in Φ^{Q}. All variable declarations (except those of sort `Control`) can also be annotated with the attribute `:no_dependence`. This means that the synthesized control signals may not depend on the value of such a variable. In other words, variables that have the `:no_dependence` attribute are bound by the inner universal quantifier of Φ^{Q}; all other variables are bound by the outer universal quantifier. In addition to declaring variables, one can also use the SMTLIB command `define-fun` to define macros that are abbreviations for an expression. These follow the syntactic rules of SMTLIB with the additional constraint that the sort `Control` may not be used as a parameter type.[27] Finally, the `assert` command can be used to state the actual formula Φ, using the variables, functions, predicates, and macros defined before. To state the formula, the standard SMTLIB operators `not`, `and`, `or`, `xor`, `=>` (implication), `ite` (if-then-else), `=` (pairwise equality), and `distinct` (pairwise inequality) can

[27]Instead, one should use the type `Bool`. Variables of type `Control` can be used in any place where a `Bool` expression is allowed.

be used. In addition to that, the standard array-read function `select` and the standard array-write function `store` can be used. Within the formula Φ, variables of sort `Control` can be used in any place where a term of sort `Bool` would be allowed. If more than one `assert` command is given, Φ is the conjunction of the formulas of the individual `assert` commands. Other SMTLIB commands are not supported by SURAQ.

SURAQ parses the input files in two steps. First, it just parses the tree of S-expressions, without interpreting them. In a second step, this tree is recursively processed and interpreted. The result of this parsing phase is a list of variables (per sort), a list of variables that have the `:no_dependence` attribute, and the actual formula Φ.

To store formulas internally, SURAQ uses its `formula` package. This package provides one class for each type of formula: And-formula, Or-formula, Not-formula, etc. All these classes implement the `Formula` interface, which provides all the methods that can be executed on any arbitrary formula. This representation basically corresponds to the syntax tree of the original formula. Objects of type `Formula` are immutable. This provides one important advantage. If a formula contains two identical subformulas, they can be represented by the same object (and not just two equal objects). This effectively turns the syntax tree into a DAG. We achieved this property by making all the class constructors private, and just provide static `create` methods with the same signature(s). These methods first check if an object equal to the requested ones already exists, and if so return the existing object. Weak references to all existing objects are stored in a `HashMap`, which allow for fast and efficient look-ups. Weak references are necessary to allow for the garbage collection of objects that are no longer in real use.

7.1.2 Formula Processing

After parsing the input file and creating the internal representation of the main formula, SURAQ performs a series of transformations to the formula. First, all instances of `define-fun` macros are "flattened". That is, any call to a macro is replaced with the actual body of the macro that it abbreviates. The result is a formula that does not contain any instances of `define-fun` macros any more. This step is necessary because macros could have array variables as parameters. Array variables will be replaced by uninterpreted function instances in one of the next steps. Unfortunately, uninterpreted functions cannot be parameters of `define-fun` macros in SMTLIB. As a secondary reason, several of the later steps are a little easier to perform and check, if the formula does not contain macros.

Second, the formula is reduced to $\mathcal{T}_U^{\mathrm{qf}}$ using the procedure suggested by Bradley et al. [BM07], which we outlined in Section 2.2.1. SURAQ also removes all (non-Boolean) `ite` expressions by introducing auxiliary variables.[28] Next, the formula is expanded (with respect to the existentially quantified variables) and

[28]This can be seen as being part of Tseitin's encoding to obtain a CNF.

negated (see Definition 23, page 45). As derivations that are done purely within one partition are irrelevant for interpolation, we can perform simplification (for example by Boolean constant propagation) on each partition formula. Since the expansion step introduced Boolean constants into the formula, simplification at this stage makes sense even if the original formula was already simplified maximally. As a last formula processing step, SURAQ performs Tseitin's encoding of each partition formula, to obtain a CNF. The conjunction of all partition CNFs can then be passed to an SMT solver.

7.1.3 SMT Solver Interaction

SURAQ's interaction with SMT solvers works as follows. The input for the solver is written to a file (in SMTLIB format). The solver is then started as an external process, whose output is also written to a file. SURAQ waits for the external process to terminate and the reads the solver's output from the respective file. SURAQ currently uses two SMT solvers: Z3 [dMB08] and VERIT [BdODF09]. Z3 is used to simplify the partitions formulas before Tseitin's encoding, as it supports a `simplify` command that does just that. Originally, the plan was to use Z3 also as the main underlying SMT solver. However, it turns out that the refutation proofs produced by Z3 do not conform to the format described in Definition 15 (Section 2.2.4), and it is not trivial to transform them into this format either. VERIT, on the other hand, produces proofs (almost) conforming to Definition 15. Thus, SURAQ relies on VERIT for refutation proof production.

7.1.4 Proof Processing

After VERIT has produced a refutation proof, SURAQ parses and processes this proof. Processing involves the following steps. In VERIT's proof format, several resolution steps can be combined into one. During parsing, SURAQ splits these into binary resolutions. Furthermore, SURAQ discards all subproofs of theory lemmata, as outlined in Section 5.3.1. This way, no non-colorable literals will occur in the remaining proof. SURAQ is also capable of checking the proofs produced by VERIT,[29] although this is deactivated by default. To check theory lemmata, SURAQ has its own implementation of the congruence closure algorithm.

In the next step, SURAQ splits all non-colorable theory lemmata into colorable ones, as outlined in Section 5.3.1. Since there is no dependence between different theory lemmata, this can be done in parallel. SURAQ supports a command line option to specify how many threads should be used to split theory lemmata. There are only very few things that need to be synchronized between these threads. Basically, synchronization is only needed for creating new formula objects (to keep them unique), finding fresh IDs for new proof nodes, and adding proof nodes to the proof. The computation time consumption of these operations is negligible, compared to the actual splitting operations. Thus, a very high

[29]It should be noted that this feature was added mainly to find bugs in SURAQ. No errors in VERIT's proofs were discovered during this research.

degree of parallelism and a correspondingly high speed-up can be achieved. This has been confirmed experimentally.

Another noteworthy fact is that, in many cases, Suraq found much shorter (and thus stronger) theory lemmata during splitting. Shorter theory lemmata make some resolution steps unnecessary and thus lead to shorter proofs. Since rewriting the proofs internally did take an extensive amount of time — due to some limitations in Suraq's internal data structures — we took the following work-around approach. After splitting all theory lemmata, the resulting leaves of the proof (theory lemmata and clauses from the actual formula) were given to a propositional SAT solver. A SAT solver was used instead of an SMT solver, because an SMT solver might potentially have introduced new non-colorable theory lemmata. A SAT solver, however, does not introduce new leaves into the proof. The SAT solver is, however, still able to produce a refutation proof, because the set of theory lemmata that is added to the propositional skeleton of the original formula obviously suffices to proof unsatisfiability, as we already have a refutation proof that uses only these theory lemmata. This can be seen as a special case of eager encoding, where not an exhaustive but a sufficient set of constraints is added to the propositional skeleton. For simplicity, the SAT solver used by Suraq is also veriT, just in propositional mode. It turns out that the proof produced by this second run is up to 40–50% smaller than the first one.

The last proof-processing step is to reorder the resolution steps such that the proof becomes local-first. This is done by repeated, recursive application of the rewrite rules presented in Section 5.3.2. After this last step, the proof is ready for n-interpolation.

7.1.5 Interpolation

Suraq is capable of performing n-interpolation as well as "standard" interpolation for iterative synthesis (see Section 5.1). In case of n-interpolation, the first step is to discard all parts of the proof that are derived just from a single partition, as the n-partial interpolant for all these nodes would be the same anyway. After that, a recursive procedure annotates every node of the proof with its corresponding n-partial interpolant.

For iterative synthesis, we do not need to split the theory lemmata, nor do we need to reorder resolution steps. After parsing the proof (which includes splitting multi-resolution steps into binary resolutions), we can immediately compute an interpolant by recursively annotating proof nodes with partial interpolants. To find partial interpolants for (potentially non-colorable) theory lemmata, we have implemented the $\mathcal{T}_U^{\mathrm{qf}}$-interpolation procedure by Fuchs et al. [FGG$^+$12].

Once we have computed the n-interpolant (or all standard interpolants iteratively), the following final steps have to be performed. First, we simplify the interpolant(s), using the logic synthesis tool abc.[30] This is particularly important for iterative interpolation, in order to avoid creating an unnecessarily large

[30]http://www.eecs.berkeley.edu/~alanmi/abc/

SMT solver input file in the next iteration. Next, all occurrences of Tseitin variables in the interpolant(s) are replaced with the formulas they represent. Finally, all instances of uninterpreted functions that actually represent array variables are converted back into proper array reads. This is necessary to ensure that the final result is syntactically compatible to the original specification.

7.1.6 Output Format and Checking Results

The output format of Suraq is also based on SMTLIB. Basically, the output file of Suraq is a copy of the input file, with some important changes. First, the formula that was asserted in the input file will be asserted in negated form in the output file. Second, for each control signal `c_i` that was synthesized, a statement of the following form is added:

```
(assert (= c_i <synthesis result>))                    (7.1)
```

Furthermore, the initial command `(set-logic Suraq)` is replaced with the standard SMTLIB command `(set-logic Arrays)`, an explicit declaration of the uninterpreted sort `Value` is added, all occurrences of sort `Control` are changed to sort `Bool`, and all occurrences of the attribute `:no_dependence` are removed. Finally, a `(check-sat)` command is added at the very end. This yields a file that can be given to any SMTLIB-compatible SMT solver that can decide $\mathcal{T}_A^{\mathrm{P}}$. The solver should report "unsatisfiable", iff the synthesis result is correct. Suraq even has a command-line argument `--check_result` that instructs it to directly call Z3 on its output file to verify correctness of the synthesis result.

7.1.7 Lessons Learned

This section will briefly summarize the most important lessons learned during the implementation of Suraq. Without the optimizations mentioned here, the tool would not have been able to deal with benchmarks beyond toy size.

Reusing Immutable Formula Objects

Originally, objects of classes from the `formula` package were regular, mutable, non-unique objects. For larger formulas, this turned out to be a huge waste of memory. In particular, when parsing (large) proofs, the same literals occur repeatedly, in the conclusions of different proof nodes. Creating a new object every time was an enormous waste of time (for object allocation and initialization) and memory. Thus — as mentioned in Section 7.1.1 — we switched to immutable formula objects that are reused.

DAG-Sharing

It turned out that the proofs obtained from veriT have a very high degree of DAG-sharing. That is, the size of the tree resulting from unrolling the DAG is significantly larger than the DAG itself. To give an example, for a proof with approximately 1.8 million nodes, unrolling the DAG resulted in more than 10^{18} nodes.[31]

Since interpolants are computed by annotating proof nodes with partial interpolants, the syntactic structure of an interpolant is the same as the structure of the proof it was computed from. That means that proofs with a high degree of DAG-sharing lead to interpolants with an equally high degree of sharing. Thus, without the formula-reusing mechanism outlined above, it would have been impossible to even compute interpolants.

As a consequence of this high degree of sharing and the extremely large sizes of unrolled DAGs, all recursive methods on proof and formulas — even the most simple ones — need to be implemented "DAG-aware". That is, intermediate results for each node should be stored, and when a node is visited again at a later time, the intermediate result should be reused instead of recursively descending to the nodes children again.

Recursive Methods With Set Results

Many of the recursive methods of proofs and formulas have sets as results. For example, there are methods to compute the set of all variables occurring in a given formula. A naive way of implementing such methods would be as follows. In each step, a new `Set` object is allocated. All elements of the sets obtained from recursive calls are added to this new set. The set is then returned as a result. The obvious disadvantage of this method is that for each step, a new object has to be allocated and elements need to be copied. To alleviate the problem, the signature of the method can be changed so that a reference to a `Set` object can be passed as a parameter. Now, instead of allocating a new `Set` object, elements can, in each step, simply be added to the given set.

Memory Footprint of HashSets

The `HashSet` class is a widely used implementation of Java's `Set` interface. One of its major drawbacks is that, in its default configuration, it wastes quite some memory. Whenever the size of the set becomes larger than 0.75 times the current size of the hash table, the size of the hash table is doubled. Thus, for example, a set with 7 elements actually uses memory for 16 elements. This can become a problem when the number of such sets gets large. For example, if every node in a (large) proof stores one such set (even when it is small), the total amount of memory wasted can become an issue. To alleviate the problem, different `Set` implementations, such as for example `TreeSet` can be used. For large sets, however, `HashSet` is still good alternative, because of the fast access to elements.

[31] Note that $2^{63} \approx 10^{19}$. Thus, computing the tree-size of proofs had to be implemented using the `BigInteger` class, as even a `long` would overflow for some of our examples.

Assert Statements

Development and debugging of SURAQ made extensive use of `assert` statements. An `assert` statement takes a Boolean condition which represents some invariant/assumption that is supposed to hold at a particular place in the program. If this place is ever reached with the condition being false, an exception is thrown and the program terminates. If such assertions are used extensively, they contribute to an *early-fail characteristic* of the program. That is, the program fails much earlier than it would without the assertion. Thus, the point of failure is usually much closer to the cause of the failure. This is particularly important when dealing with very large data structures. This is illustrated by the following example. Before we extensively used `assert` statements, we encountered a bug where an object representing one node in a large proof was in an inconsistent state. Using a debugger, we discovered, however, that at creation time the object was constructed in a consistent state. It was not trivial to find out which of several recursive operations on the large proof left the object in question in an inconsistent state. By adding assertions about the consistency of the object's state to each method, localizing the problem became significantly easier.

Another big advantage of `assert` statements is that they can be activated or deactivated through a command-line option of the Java virtual machine. Thus, once enough confidence in the correctness of the implementation has been achieved, they can be deactivated and no longer impose a performance overhead.

Java Programs with Large Heaps

During the development of SURAQ we noticed that many Java-related tools (debugger, profiler, memory analyzer, libraries for serialization/deserialization of objects, etc.) encounter various performance problems when confronted with a program that requires a large amount of heap space. This is one issue that, in hindsight, is an argument against the decision to use Java for this project. On the other hand, it is unclear, if the corresponding tools for a different programming language would have worked any better.

7.2 Benchmarks

Declaration of Sources

This section is based on and reuses material from the following source, previously published by the author:

- [HGK$^+$13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In Jobstmann and Ray [JR13], pages 77–84.

References to this source are not always made explicit. In particular, Sections 7.2.1 and 7.2.2 are heavily based on the paper cited above. Section 7.2.3, however, contains material that has not been published before.

In this section, we will present the benchmarks that are the basis for our experimental evaluation in Section 7.3. In addition to the three benchmark families presented here, we also used the simple pipeline example that we used to illustrate our modeling approach (see Example 2) in Section 3.2. All the benchmarks are part of the Suraq distribution.[32] Some important characteristics of the benchmarks are summarized in Table 7.1.

7.2.1 Scalable, Illustrative Example

The first family of benchmarks we want to discuss was originally presented in [HGK$^+$13]. It is a very simple, yet nicely scalable circuit, whose smallest version is shown in Figure 7.1. It has two input bit-vectors i_1 and i_2, carrying non-zero signed integers, and also two output bit-vectors o_1 and o_2 carrying signed integers. The block *neg* flips the sign of its input. The outputs are controlled by two bits, c_1 and c_2. These are the signals we wish to synthesize, based on the following specification: The signs of the two outputs must be different. Formally, this specification can be stated in $\mathcal{S}^{\mathrm{Q}}$ as follows

$$\begin{aligned}\forall i_1, i_2 \,.\, \exists c_1, c_2 \,.\, \forall o_1, o_2 \,.((c_1 \wedge o_1 = i_1 \vee \neg c_1 \wedge o_1 = \mathit{neg}(i_1))\wedge \\ (c_2 \wedge o_2 = i_2 \vee \neg c_2 \wedge o_2 = \mathit{neg}(i_2))) \rightarrow (\mathit{pos}(o_1) \oplus \mathit{pos}(o_2)),\end{aligned} \tag{7.2}$$

where the predicate *pos* returns $\top$ iff its parameter is positive. To compute certificates for c_1 and c_2, we must add the axiom

$$(\mathit{pos}(i_1) \oplus \mathit{pos}(\mathit{neg}(i_1))) \wedge (\mathit{pos}(i_2) \oplus \mathit{pos}(\mathit{neg}(i_2))). \tag{7.3}$$

[32] `http://www.iaik.tugraz.at/content/research/design_verification/suraq/`

Table 7.1: Benchmark Characteristics. The first column lists the name of the benchmark. The second column states the number of control signals to be synthesized. The third column gives the number of variables on which the synthesized control signals may depend. Conversely, the fourth column gives the number of auxiliary variables, on which the control signals may *not* depend. Note that the numbers in columns 3 and 4 refer to the number of *all* variables (propositional, domain, and array variables). The fifth column gives the number of uninterpreted functions, the sixth column the number of uninterpreted predicates that occur in the benchmark. Note that these refer to the number of different functions/predicates declared, and not to the number of function/predicate instances within the formula. The seventh column gives the number of array variables (including auxiliary array variables, that is, variables on which the control signals may *not* depend). The eighth and ninth column give the size of the main formula, that is, the number of nodes in the syntax tree, after instantiation of all `define-fun` macros. In column 8, the size is computed based on a DAG with sharing, the data in column 9 is based on unrolling the DAG into a tree. All data, except for columns 8 and 9, is based on the original input file.

1	2	3	4	5	6	7	8	9
Name	$\lvert\overline{c}\rvert$	$\lvert\overline{x}\rvert$	$\lvert\overline{x}'\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\mathsf{R}\rvert$	Size (DAG)	Size (tree)
simple_pipeline	1	5	5	1	0	5	8	8
illus_02	2	2	2	1	1	0	21	21
illus_03	3	3	3	1	1	0	29	29
illus_04	4	4	4	1	1	0	37	37
illus_05	5	5	5	1	1	0	45	45
illus_06	6	6	6	1	1	0	53	53
illus_07	7	7	7	1	1	0	61	61
illus_08	8	8	8	1	1	0	69	69
illus_09	9	9	9	1	1	0	76	77
simple_processor	2	6	11	6	1	5	36	38
dlx_stall	1	23	87	9	5	21	639	1214
dlx_f-a-ex	1	23	86	9	5	21	636	1219
dlx_f-b-wb	1	23	87	9	5	21	635	1205
dlx_stall_f-a-ex	2	23	86	9	5	21	638	1213

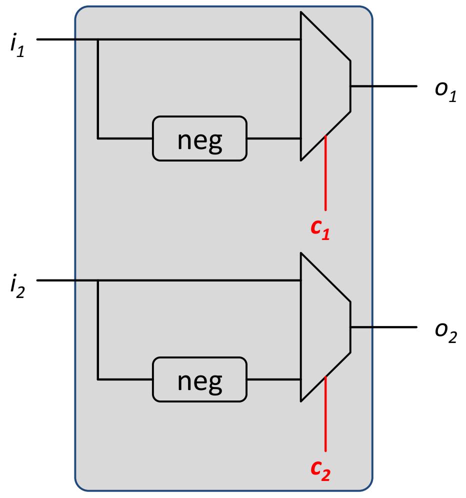


Figure 7.1: Scalable, Illustrative Example. The control signals c_1 and c_2 control whether an output equals the corresponding input, or the negation of the corresponding input. The specification states that the signs of the output must be different.

Note that this example also illustrates how the certificates for different control signals can depend on each other. Thus, computing them independently may not work. For instance, we may choose $c_1 = \top$ or we can take $c_2 = \top$, but we cannot choose $c_1 = c_2 = \top$.

We scale this example up in the following way: For a natural number $n \geq 2$, the circuit has n inputs i_j, n outputs o_j, and n control signals c_j. For each j, we have that

$$((c_j \wedge o_j = i_j \vee \neg c_j \wedge o_j = neg(i_j)). \tag{7.4}$$

Furthermore, the specification states that the chained xor of all output signs is true. That is,

$$\left(\bigoplus_{1 \leq j \leq n} pos(o_j) \right) = \top. \tag{7.5}$$

7.2.2 Simple Processor

In Figure 7.2, we show a simple (fictitious) microprocessor with a 2-stage pipeline. To keep the design simple, we deliberately left out some features most microprocessor usually have: for example, a register file, or the ability to directly access arbitrary memory locations.[33] MEM represents the main memory. We assume that the value at address 0 is hardwired to 0. That is, reading from address 0

[33]Reading from arbitrary memory locations can, however, be achieved by using self-modifying code.

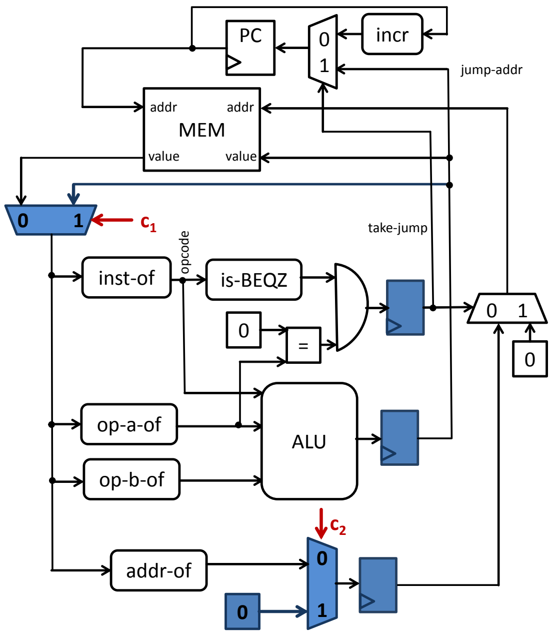


Figure 7.2: A simple microprocessor with a 2-stage pipeline. (Source: [HGK$^+$13])

always yields value 0, and writing to address 0 has no effect.[34] The blocks inst-of, op-a-of, op-b-of, and addr-of represent combinational functions that decode a memory word. The block incr increments the program counter (PC). The block is-BEQZ is a predicate that checks whether an instruction is a branch instruction. The design has two pipeline-related control signals for which we would like to synthesize an implementation. Signal c_1 causes a value in the pipeline to be forwarded and signal c_2 squashes the instruction that is currently decoded and executed in the first pipeline stage. This might be necessary due to speculative execution based on a “branch-not-taken assumption”. The implementation of these control signals is not as simple as it might seem at first glance. For example, the seemingly trivial solution of setting $c_1 = \top$ whenever PC equals the address register is not correct. For example, if both the address register and PC are equal to 0, forwarding should not be done, because writing to address 0 has no effect on MEM.[35] By taking out the blue parts in Figure 7.2, we obtain the non-pipelined reference implementation which we used to formulate a Burch-Dill style equivalence criterion, as outlined in Section 3.2. The resulting formula was used as a specification for synthesis.

[34] Writing to address 0 can be seen as a “no-operation” (NOP) instruction.

[35] We actually made this mistake while trying to create and model-check a manual implementation for the control signals, and it took some time to locate and understand the problem.

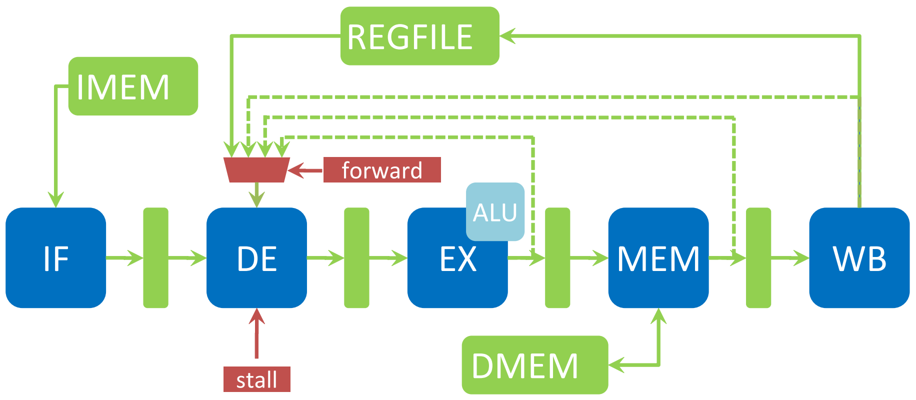


Figure 7.3: Pipelined DLX Processor. The design consists of five pipeline stages (shown in blue) with pipeline registers (shown in green) in between. The dashed lines represent potential data-forwarding paths. Since there are two operands that could potentially be forwarded from three different pipeline stages, six Boolean signals are required to control forwarding. The seventh control signal can stall the pipeline if necessary.

7.2.3 DLX Processor

The most complex benchmark we used is a five-stage pipelined DLX processor, as introduced by Hennessy and Patterson [HP96]. An abstract sketch, showing the five stages of the pipeline is depicted in Figure 7.3. The DLX processor features three different addressable memories, the register file REGFILE, the data memory DMEM, and the instruction memory IMEM. We have identified the following seven Boolean control signals:

- `forward-a-from-ex`,
- `forward-a-from-mem`,
- `forward-a-from-wb`,
- `forward-b-from-ex`,
- `forward-b-from-mem`,
- `forward-b-from-wb`, and
- `stall`.

The names of these signals indicate what each of them controls: For signals whose names start with `forward`, setting `forward-X-from-Y` to $\top$ means that operand `X` is forwarded from pipeline stage `Y`. Signal `stall` stalls the pipeline, if set to $\top$.

By adding manual implementations for some of the control signals, we have created several different variants of this benchmark, with different numbers of signals to synthesize.

7.3 Experimental Results

For our experimental evaluation, we used the benchmarks described in the previous section. From the DLX benchmark-family, we picked three representatives: “dlx_stall”, which has just one control signal (`stall`); “dlx_f-a-ex” and “dlx_f-b-wb”, which also have one control signal each (`forward-a-from-ex` and `forward-b-from-wb`, respectively); and “dlx_stall_f-a-ex”, which has two control signals (`stall` and `forward-a-from-ex`). Unfortunately, Suraq was unable to handle DLX benchmarks with more control signals.

The experiments were performed on a machine with 3 quad-core Intel Nehalem CPUs with hyperthreading, giving a total of 24 logical cores.[36] The machine has 192 GB of main memory available. The Java virtual machine was invoked with a heap size limit of 150 GB and a maximum stack size of 2000 MB. Assertions (see Section 7.1.7) were disabled.

Suraq keeps track of many interesting numbers and statistics. In the remainder of this section, we will focus on analyzing the most interesting and important results and draw some conclusions. We will first look at results from n-interpolation mode, before also giving some results for the iterative interpolation mode. Finally, in Section 7.3.4, we will summarize our key experimental results.

7.3.1 Runtime Results (n-Interpolation Mode)

The first thing we analyzed were the runtimes of the different benchmarks, and how the total time is distributed over the major tasks performed by Suraq. These results are shown in Table 7.2. We can see that for the simple benchmarks, in particular the ones of the illus_XX family, most of the time is spent in creating the input file for the SMT solver. One can also see that this time increases roughly exponentially with the number of control signals. This is no surprise, as each additional control signal doubles to number of partitions that need to be created, encoded, and simplified. All other tasks only need negligible time, for these benchmarks.

Another interesting fact is that the simple_processor benchmark can also be solved in a very short time, despite the fact that this benchmark models a complete microprocessor — even if it is a simple one. This shows that using uninterpreted functions is a good mean of abstraction in such a case, turning a non-trivial problem into an easily solvable instance.

Since the aforementioned benchmarks can all be solved so fast that no interesting conclusions can be drawn about the distribution of the runtime over

[36]Note that, except for splitting of non-colorable theory lemmata (where we used all 24 cores available), Suraq is single-threaded.

Table 7.2: Runtime Results (n-Interpolation Mode). Column 1 names the benchmark. Column 2 gives the time for the formula reductions (see Section 7.1.2), that is, the total time required for reading the specification, performing the formula reductions, and creating an input file for veriT. Column 3 gives the time required by veriT to solve this input and create a proof. Column 4 gives the (wall clock) time taken to split all non-colorable theory lemmata, using 24 parallel splitter threads. Column 5 gives the time taken by veriT for propositional SAT solving with the stronger theory lemmata obtained from splitting (see Section 7.1.4). Column 6 gives the time to reorder the proof to make it local-first. Column 7 gives the time spent on proof parsing, including splitting of multi-resolution nodes. This combines the time for parsing the SMT proof and the propositional SAT proof. Column 8 gives the total time required for synthesis. All times are given in seconds, and rounded to integers.

1	2	3	4	5	6	7	8
Name	Formula Reduction	SMT Solving	Splitting Leaves	SAT Solving	Re-ordering	Proof Parsing	Total
simple_pipeline	<1	<1	<1	<1	<1	<1	1
illus_02	<1	<1	<1	<1	<1	<1	<1
illus_03	<1	<1	<1	<1	<1	<1	1
illus_04	1	<1	<1	<1	<1	<1	2
illus_05	2	<1	<1	<1	<1	<1	3
illus_06	4	<1	<1	<1	<1	<1	5
illus_07	7	<1	<1	<1	<1	<1	11
illus_08	14	1	<1	<1	<1	<1	17
illus_09	28	3	<1	<1	<1	<1	34
simple_processor	<1	<1	<1	<1	<1	<1	4
dlx_stall_f-a-ex	6	1718	6	7	n/a	442	n/a

different subtasks, we also included one of the DLX benchmarks in this series of experiments. Unfortunately, it runs out of memory during reordering of the refutation proof to make it local-first. Thus, the time for reordering and the total runtime are not available for this benchmark. We can see, however, that formula reduction is very fast for this benchmark, and most of the time is spent in the SMT solver. Splitting of non-colorable leaves can also be done rather fast. The SAT solver run to obtain a proof based on the (stronger), colorable theory lemmata is also extremely fast, compared to the original SMT solver run. This leads us to conclude that the SMT solver spent a significant part of its runtime on checking the theory-consistency of (partial) assignments and creating theory lemmata to block inconsistent assignments.

There is one more thing to remark about the runtime results. In [HGK$^+$13], we presented runtime results for (a subset of) these benchmarks. The results shown here are, however, significantly better than the ones in [HGK$^+$13]. This discrepancy is due to improvements in SURAQ that have been implemented since the publication of [HGK$^+$13]. In particular, all methods in the `formula` package were rewritten to be DAG-aware (see Section 7.1.7), leading to significant improvements in runtime and enabling solving some benchmarks that could not be solved before.

7.3.2 Proof Sizes (n-Interpolation Mode)

In addition to analyzing runtime, we have also looked at the sizes of the refutation proofs occurring in n-interpolation mode. The results are shown in Table 7.3. Let us first look at the size of the original proof (column 2). For the illus_XX benchmarks, we again see a roughly exponential increase in proof size. Since these benchmarks involve hardly any theory-reasoning, it is not very surprising that the number of non-colorable theory lemmata is rather small. This also explains why the time needed for splitting (see previous section) is so low for these benchmarks.

Another interesting point is that, especially for the larger benchmarks (simple_processor and dlx_stall_f-a-ex), the size of the proof obtained from SAT solving (column 5) is significantly smaller than the original proof. In case of the DLX benchmark, this reduction is over 60%. We conjecture that VERIT's theory solver is not well suited to the characteristics of these benchmarks. We believe that this is caused by the rather long "chains" of transitivity (mixed with function congruence) that arise from the Burch-Dill style equivalence criterion. This would explain why the issue surfaces more prominently for the DLX benchmarks than it does for the simple_processor benchmark. The simple processor has only a two-stage pipeline, whereas DLX has a five-stage pipeline. Thus, the transitivity-chains are potentially significantly longer for DLX. Details about how many (non-colorable) theory lemmata could be made stronger — and by how much — are shown in Table 7.4.

Concerning making the refutation proof local-first, note that for a rather low number of control signals, the increase of the proof size due to reordering is not as bad as for benchmarks with more partitions. This is also to be expected,

Table 7.3: Proof Sizes. The first column gives the name of the benchmark. The second column states the size of the proof, as obtained from veriT, however with subproofs of theory lemmata already removed. The third column gives the number of leaves that are non-colorable and need to be split, and the fourth column gives the total number of leaves. Columns 5 gives the size of the proof obtained by calling a SAT solver on the skeleton of the original formula, together with the colorable theory lemmata and the (stronger) theory lemmata obtained from splitting. This is the proof that is given to the reordering procedure. The size of the proof after reordering is given in column 6. Column 7 gives the size of the proof that is used for n-interpolation, that is, the reordered proof with local subproofs removed. All proof sizes are given as the number of nodes in the DAG.

1	2	3	4	5	6	7
Name	Original Proof	# Leaves to split	# Leaves (total)	Before Reordering	After Reordering	w/o Local Subproofs
simple_pipeline	506	2	178	496	494	12
illus_02	102	2	44	106	106	12
illus_03	179	3	77	198	218	26
illus_04	390	7	133	356	428	46
illus_05	408	9	165	700	971	115
illus_06	669	4	176	758	1 576	320
illus_07	1 006	11	219	916	2 823	785
illus_08	1 101	6	242	2 214	8 082	1 347
illus_09	1 101	7	269	1 388	5 364	1 293
simple_processor	9 576	123	1 503	6 853	7 899	73
dlx_stall_f-a-ex	856 121	2 748	21 349	333 260	n/a	n/a

Table 7.4: Stronger Theory Lemmata. This table shows how many theory lemmata could be made stronger, in relation to the total number of theory lemmata considered. Column 1 gives the name of the benchmark. Column 2 shows how many theory lemmata were made stronger during splitting. Column 3 puts this into context, by showing the total number of theory lemmata that were split. Column 4 shows how many literals were saved in total.

1	2	3	4
Name	Theory lemmata made stronger	Theory lemmata to split	Literals saved
simple_pipeline	2	2	14
illus_02	0	2	0
illus_03	0	3	0
illus_04	0	7	0
illus_05	0	9	0
illus_06	0	4	0
illus_07	0	11	0
illus_08	0	6	0
illus_09	0	7	0
simple_processor	74	123	227
dlx_stall_f-a-ex	1 815	2 748	6 476

as with more partitions, there are more nodes in positions where they do not belong in a local-first proof.

Column 7 in Table 7.3 gives the size of the proof as it is used for the actual n-interpolation. That is, all derivations that are done solely within one partition have been removed, and only the global part of the proof remains. The size of this proof also correlates directly with the size of the resulting n-interpolant and hence the size of the synthesized implementation of control signals. Remember that, in principle, for n-interpolation every node in the proof corresponds to one multiplexer in the interpolant. It is interesting to note that the size of these final proofs is significantly smaller than the total size of the proof, including the local parts. For the simple_processor benchmark, the difference is even more than two orders of magnitude.

7.3.3 Iterative Mode

We now compare the results of n-interpolation mode with iterative mode. Total runtimes for iterative mode are shown in Table 7.5. Table 7.6 details the SMT solving time per iteration. These experiments include three additional DLX benchmarks with one control signal each. These were not tested with n-interpolation mode, as for just one signal n-interpolation mode does not make sense. In n-interpolation mode, SURAQ reorders the proof to make is local-first. This is, however, not necessary if just one single interpolant is computed. Thus, for $n = 1$, iterative mode is always clearly superior to n-interpolation mode.

Note that, in general, iterative mode takes significantly longer for the bench-

Table 7.5: Runtime Results (Iterative Mode). Column 1 gives the name of the benchmark. Column 2 gives the total synthesis time in iterative mode. Times are in seconds, rounded to integers.

1	2
Name	Total Runtime
simple_pipeline	<1
illus_02	1
illus_03	2
illus_04	3
illus_05	6
illus_06	12
illus_07	31
illus_08	66
illus_09	485
simple_processor	4
dlx_stall	537
dlx_f-a-ex	1 358
dlx_f-b-wb	2 174
dlx_stall_f-a-ex	4 528

Table 7.6: SMT Solving Time per Iteration. If a benchmark does not require more than a certain number of iterations, the remaining columns are left empty.

	Iteration								
Name	1	2	3	4	5	6	7	8	9
simple_pipeline	<1								
illus_02	<1	<1							
illus_03	<1	<1	<1						
illus_04	<1	<1	<1	<1					
illus_05	<1	<1	<1	<1	<1				
illus_06	<1	<1	<1	<1	<1	<1			
illus_07	<1	<1	<1	1	<1	<1	<1		
illus_08	1	<1	<1	1	2	2	2	1	
illus_09	3	5	22	45	24	23	10	9	6
simple_processor	<1	<1							
dlx_stall	267								
dlx_f-a-ex	573								
dlx_f-b-wb	590								
dlx_stall_f-a-ex	1 711	923							

Table 7.7: Proof Size per Iteration. If a benchmark does not require more than a certain number of iterations, the remaining columns are left empty.

	Iteration								
Name	1	2	3	4	5	6	7	8	9
simple_pipeline	506								
illus_02	102	166							
illus_03	179	493	508						
illus_04	390	680	724	1 251					
illus_05	408	2 133	3 608	3 298	3 361				
illus_06	669	2 521	1 799	3 906	9 043	10 088			
illus_07	1 006	6 430	7 210	26 072	23 941	26 543	32 009		
illus_08	1 101	7 352	3 332	16 312	32 087	52 782	60 822	73 887	
illus_09	1 101	27 210	60 002	165 636	117 535	243 332	231 789	391 277	281 313
simple_processor	9 576	8 682							
dlx_stall	898 345								
dlx_f-a-ex	1 490 028								
dlx_f-b-wb	2 271 288								
dlx_stall_f-a-ex	856 121	1 460 582							

marks of the illus_XX family. On the positive side, since iterative interpolation does not require reordering the refutation proof — which was too costly for larger DLX benchmarks — the iterative approach is able to finish synthesizing both control signals of the dlx_stall_f-a-ex benchmark.

In Table 7.7, we show the size of the refutation proof in each iteration. These results are particularly interesting: On one hand, one would expect the proof sizes to decrease with each iteration, as the number of partitions is decreasing exponentially. On the other hand, one would expect an increase, as the problem is made more complicated by the resubstitution of each interpolant. It was not clear a priori which of these effects would be dominant. From Table 7.7, we can see that proof size almost always increases with each iteration. Thus, we conclude that — in general — the size reduction cannot counteract the negative effect of resubstitution. Note that in the few exceptional cases where the proof size actually decreases from one iteration to the next, also the solving time (see Table 7.6) decreases, whereas in general, runtime also increases with each iteration.

7.3.4 Key Results

Let us briefly summarize the key results of our experimental evaluation. First, our most important achievement is that we managed to synthesize a controller (consisting of two control signals) for a five-stage pipelined DLX processor [HP96]. The synthesis time is approximately 1 hour and 15 minutes. Since the DLX benchmark is of realistic size and complexity, this proves that our approach for abstraction using uninterpreted functions and interpolation-based certificate computation is scalable enough for real-world problems.

Second, our experiments have revealed that neither iterative interpolation, nor n-interpolation is clearly superior over the other. Instead, it depends on the characteristics of the benchmark which approach performs better. While for some benchmarks n-interpolation clearly outperforms iterative interpolation, in other instances the need for proof reordering makes n-interpolation inapplicable. In the future, we hope to alleviate this problem by using modular SMT solving.

Third, we conclude that in most cases, resubstitution of one solution in iterative mode makes the SMT problem of the following iteration harder, despite that fact that the next iteration will only have half the number of partitions. This underlines the fact that the solutions computed via interpolation are rather large and complex, compared to manual implementations.

8 Conclusion

To conclude this thesis, we will once more summarize the research challenges that we faced at the beginning. After that, we will again highlight the most important goals achieved, and the contributions made to advance the prior state-of-the-art. Finally, we will present new challenges that emerged due to our work, which remain for future investigation.

8.1 Summary in Retrospect

The challenge we were facing when starting this research was the following. How can we synthesize some Boolean control signals for a system that is already partially implemented? In particular, we considered pipelined microprocessors, where pipeline control is easy to formally specify, but hard to implement manually. On the other hand, datapaths of microprocessors are comparatively easy to implement. Using state-of-the-art property synthesis tools of the time did not seem feasible, as this would require to formally specify the datapath's behavior on a bit- and cycle-accurate level. This would have led to specifications of such a large size that they would clearly have been intractable for synthesis tools. Thus, our challenge was to come up with a different specification formalism that would allow us to abstract these complex parts. It soon became clear that uninterpreted functions might provide just what we needed, as they had already been used successfully in a similar verification setting by Burch and Dill [BD94]. The Burch-Dill paradigm also allowed us to transform a temporal problem into a non-temporal domain. While it is not clear how this can be done in the general case, it works fine for pipelined processors. To express our synthesis problems, we extended Burch-Dill equivalence conditions with mixed quantifiers.

After establishing the specification formalism, the next challenge was to find ways to compute certificates for the existentially quantified Boolean control signals. While — in an early stage — several different methods (see Chapter 6) were considered, we decided to focus on interpolation-based approaches, as they seemed to be most promising (see Chapter 5). Thus, our prototype tool Suraq implements these approaches (see Chapter 7).

8.2 Goals Achieved

With this thesis, we have advanced the state-of-the-art in the following ways. First, we have created a specification formalism that allows us to state synthesis problems for Boolean signals in certain settings where parts of a system are already implemented. Existing parts can be efficiently abstracted using uninterpreted functions. The primary example of such a setting is synthesizing controllers for pipelined microprocessors with pre-implemented complex datapath elements.

Second, we have presented several ways how to solve such synthesis problems, by computing certificates for the existentially quantified variables in formulas of the form $\forall \overline{x} \,.\, \exists \overline{c} \,.\, \forall \overline{x}' \,.\, \exists \overline{c}' \,.\, \forall \overline{x}'' \ldots \Phi$. One of these ways is based on n-interpolation, a generalized version of Craig interpolation that we introduced, where multiple coordinated interpolants are computed from a single refutation proof. Refutation proofs for n-interpolation need to be colorable and local-first. We have presented proof transformation procedures to obtain these properties. We have also introduced modular SMT solving, which can directly produce colorable, local-first refutation proofs.

Third, we have created a prototype implementation to demonstrate the feasibility of our synthesis approach. To the best of our knowledge, our synthesis tool Suraq is the first and — as of writing this thesis — currently only tool that supports synthesis of controllers based on specifications with uninterpreted functions as a mean of abstraction. Using Suraq, we were able to synthesize a controller for a five-stage pipelined DLX microprocessor [HP96], which demonstrates the scalability of our approach.

8.3 Future Work

While working on our primary research goals, we have also come across several related aspects, for which further research is clearly indicated. In this section, we will briefly outline these questions, which remain for future work.

8.3.1 Small Certificates

So far, we have focused solely on the logical correctness of the certificates we compute. For practical purposes, it would, however, also be interesting to take their *size*[37] into account. It is interesting to note that — for many examples — very small (manually implemented) solutions exist, yet our synthesis tool only finds comparatively large solutions. This is not unlike the situation with temporal logic synthesis tools [BGJ+07a].

When sticking with interpolation-based approaches, the size of the certificate is clearly correlated to the size of the refutation proof. Thus, existing techniques for proof compression (for example [FMW11] and [BW13]), provided by tools such as Skeptik [BFW14], already work towards this goal. The open question that remains is whether we can find techniques to compress the proof specifically for obtaining small interpolants. Going one step further, it would also be interesting to investigate whether the SMT solver can already be tweaked towards finding a proof that yields small interpolants.

As an orthogonal approach, it would also be interesting to look for different approaches for certificate computation, specifically tailored to respect size metrics. The template instantiation approach that we presented in Section 6.4 is a first step in this direction.

8.3.2 Certificate Strength

In many applications, it might be desirable to obtain either rather weak, or rather strong certificates. That is, certificates with a rather large, or a rather small ON-set. For example, when synthesizing a `stall` signal for a pipelined processor, one usually wishes to have the `stall` signal set to $\top$ only if absolutely necessary. Interpolant strength has been investigated by D'Silva et al. [DKPW10]. It would be interesting to see how and to which extent their results can be ported to n-interpolation. Since n-interpolation requires a local-first refutation proof, some of the freedom used for tweaking interpolant strength in [DKPW10] is already lost. Furthermore, the interdependence of the components of an n-interpolant provides some interesting challenges. For example, strengthening one component might weaken another component.

8.3.3 Modular SMT Solving

While we have introduced the general concept of modular SMT solving in Section 5.4, this approach has not yet been evaluated experimentally. However, irrespective of the results of experimental evaluation, we believe that there are ample possibilities for optimization that should be investigated. A more detailed analysis will, however, have to be postponed until after the first experimental evaluation.

[37]Possible metrics for measuring the size of a certificate include the number of nodes in the syntax tree/DAG, or the number of standard logic gates to implement the certificate in a circuit.

8.3.4 Formulas with Multiple Time-Instances of Control Signals

In Section 3.2, we discussed the problem of modeling control signals in multiple time steps. Our specification language $\mathcal{S}^{\mathrm{Q}}$ does not allow us to express the dependencies and consistency requirements that are required in such a case. (See page 38 for details.) We are currently investigating if Henkin quantifiers [Hen61] provide the expressibility we require for this problem. If so, we could build upon the work of Balabanov et al. [BCJ14] to compute certificates in such a setting.

8.4 Last Words

In the spirit of Surak,[38] we conclude this thesis by saying:

"Live long and prosper."

[38] `http://en.memory-alpha.org/wiki/Surak`

Bibliography

[Ack54] Wilhelm Ackermann. Solvable cases of the decision problem. *Studies in Logic and the Foundations of Mathematics*, 1954.

[AMT13] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In Jobstmann and Ray [JR13], pages 26–33.

[Ang87] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[BCG$^+$10] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY - a new requirements analysis tool with synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer, 2010.

[BCG$^+$14] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.

[BCHJ09] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In Bouajjani and Maler [BM09], pages 140–156.

[BCJ14] Valeriy Balabanov, Hui-Ju Katherine Chiang, and Jie-Hong Roland Jiang. Henkin quantifiers and boolean formulae: A certification perspective of DQBF. *Theor. Comput. Sci.*, 523:86–100, 2014.

[BCK04] David Bañeres, Jordi Cortadella, and Michael Kishinevsky. A recursive paradigm to solve boolean relations. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *DAC*, pages 416–421. ACM, 2004.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1994.

[BDF$^+$12] Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski, and André Sülflow. FoREnSiC — an automatic debugging environment for C programs. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *Haifa Verification Conference*, volume 7857 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2012.

[BdODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In Schmidt [Sch09], pages 151–156.

[BFW14] Joseph Boudou, Andreas Fellner, and Bruno Woltzenlogel Paleo. Skeptik: a proof compression system. In *7th International Joint Conference on Automated Reasoning*, 2014.

[BGH$^+$12] Roderick Bloem, Hans-Jürgen Gamauf, Georg Hofferek, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems with RATSY. In Doron Peled and Sven Schewe, editors, *SYNT*, volume 84 of *EPTCS*, pages 47–53, 2012.

[BGJ$^+$07a] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Automatic hardware synthesis from specifications: a case study. In Rudy Lauwereins and Jan Madsen, editors, *DATE*, pages 1188–1193. ACM, 2007.

[BGJ$^+$07b] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, February 2009.

[BJ12] Valeriy Balabanov and Jie-Hong Roland Jiang. Unified QBF certification and its applications. *Formal Methods in System Design*, 41(1):45–65, 2012.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation.* Springer, 2007.

[BM09] Ahmed Bouajjani and Oded Maler, editors. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science.* Springer, 2009.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In Emerson and Namjoshi [EN06], pages 427–442.

[BP09] Armin Biere and Carl Pixley, editors. *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In Aarti Gupta and Daniel Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[BV00] Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. In Emerson and Sistla [ES00], pages 85–98.

[BVB$^+$13] Sam Bayless, Celina G. Val, Thomas Ball, Holger H. Hoos, and Alan J. Hu. Efficient modular SAT solving for IC3. In Jobstmann and Ray [JR13], pages 149–156.

[BW13] Joseph Boudou and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs by lowering subproofs. In *TABLEAUX*, pages 59–73, 2013.

[Chu62] Alonzo Church. Logic, arithmetic and automata. In *Proceedings of the international congress of mathematicians*, pages 23–35, 1962.

[Cra57] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):pp. 269–285, 1957.

[dAR10] Luca de Alfaro and Pritam Roy. Solving games via three-valued abstraction refinement. *Inf. Comput.*, 208(6):666–676, 2010.

[DKPW10] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[EKH12] Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 91–100. IEEE, 2012.

[EN06] E. Allen Emerson and Kedar S. Namjoshi, editors. *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.

[ES00] E. Allen Emerson and A. Prasad Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[FGG$^+$09] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstic, and Cesare Tinelli. Ground interpolation for the theory of equality. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2009.

[FGG$^+$12] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstic, and Cesare Tinelli. Ground interpolation for the theory of equality. *Logical Methods in Computer Science*, 8(1), 2012.

[FJ12] Bernd Finkbeiner and Swen Jacobs. Lazy synthesis. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012.

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for LTL realizability. In Bouajjani and Maler [BM09], pages 263–277.

[FMW11] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of propositional resolution proofs via partial regularization. In *CADE*, pages 237–251, 2011.

[FS13] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

[GBC06] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to C. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 358–371. Springer, 2006.

[Gup12] Ashutosh Gupta. Improved single pass algorithms for resolution proof reduction. In Supratik Chakraborty and Madhavan Mukund, editors, *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2012.

[HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42.

[Hen61] Leon Henkin. Some remarks on infinitely long formulas. In *Infinistic Methods*, pages 167–183. Pergamon Press, 1961.

[HGK$^+$13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In Jobstmann and Ray [JR13], pages 77–84.

[HGS03] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam K. Srivas. Formal verification of a complex pipelined processor. *Formal Methods in System Design*, 23(2):171–213, 2003.

[HKV12] Krystof Hoder, Laura Kovács, and Andrei Voronkov. Playing in the grey area of proofs. In John Field and Michael Hicks, editors, *POPL*, pages 259–272. ACM, 2012.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition.* Morgan Kaufmann, 1996.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2004.

[HW08] Georg Hofferek and Johannes Wolkerstorfer. Coupon recalculation for the GPS authentication scheme. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[JB12] Swen Jacobs and Roderick Bloem. Parameterized synthesis. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2012.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 226–238. Springer, 2005.

[JGWB07] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 258–262. Springer, 2007.

[JLH09] Jie-Hong Roland Jiang, Hsuan-Po Lin, and Wei-Lun Hung. Interpolating functions from large boolean relations. In *ICCAD*, pages 779–784. IEEE, 2009.

[JR13] Barbara Jobstmann and Sandip Ray, editors. *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013.

[KHB09] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In Biere and Pixley [BP09], pages 152–159.

[KHB10] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging unrealizable specifications with model-based diagnosis. In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Haifa Verification Conference*, volume 6504 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2010.

[KHB13] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT*, 15(5-6):563–583, 2013.

[KJB13] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. PARTY: parameterized synthesis of token rings. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 928–933. Springer, 2013.

[KMPS10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 316–329. ACM, 2010.

[KMPS13] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Functional synthesis for linear arithmetic and sets. *STTT*, 15(5-6):455–474, 2013.

[KS00] James H. Kukula and Thomas R. Shiple. Building circuits from relations. In Emerson and Sistla [ES00], pages 113–123.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Springer, 2008.

[KV09] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Schmidt [Sch09], pages 199–213.

[LB10] Florian Lonsing and Armin Biere. DepQBF: a dependency-aware QBF solver. *JSAT*, 7(2-3):71–76, 2010.

[LDS11] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In Singh et al. [SJKB11], pages 43–50.

[MCB05] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 2005.

[McC63] John McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.

[McM05] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[McM11] Kenneth L. McMillan. Interpolants from Z3 proofs. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 19–27. FMCAD Inc., 2011.

[Min92] Shin-ichi Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *Synthesis and Simulation Meeting and International Interchange (SASIMI'92)*, pages 64–73, 1992.

[Mor70] Eugenio Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Transactions on Computers*, 100(6):504–509, 1970.

[MS96] João P. Marques Silva and Karem A. Sakallah. GRASP — a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[MS99] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[MS10] Andreas Morgenstern and Klaus Schneider. Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. In Angelo Montanari, Margherita Napoli, and Mimmo Parente, editors, *GANDALF*, volume 25 of *EPTCS*, pages 89–102, 2010.

[NHKL10] Eriko Nurvitadhi, James C. Hoe, Timothy Kam, and Shih-Lien Lu. Automatic pipelining from transactional datapath specifications. In *DATE*, pages 1001–1004. IEEE, 2010.

[NHKL11] Eriko Nurvitadhi, James C. Hoe, Timothy Kam, and Shih-Lien Lu. Automatic pipelining from transactional datapath specifications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(3):441–454, 2011.

[NO77] Greg Nelson and Derek C. Oppen. Fast decision algorithms based on union and find. In *FOCS*, pages 114–119. IEEE Computer Society, 1977.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In Emerson and Namjoshi [EN06], pages 364–380.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190. ACM Press, 1989.

[Pud97] Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997.

[Sch09] Renate A. Schmidt, editor. *Automated Deduction — CADE, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*. Springer, 2009.

[SF07] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2007.

[SHB11] Matthias Schlaipfer, Georg Hofferek, and Roderick Bloem. Generalized reactivity(1) synthesis without a monolithic strategy. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Haifa Verification Conference*, volume 7261 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2011.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.

[SJKB11] Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors. *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011.* IEEE, 2011.

[SL13] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[SS09] Saqib Sohail and Fabio Somenzi. Safety first: A two-stage algorithm for LTL games. In Biere and Pixley [BP09], pages 77–84.

[SS13] Saqib Sohail and Fabio Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *STTT*, 15(5-6):433–454, 2013.

[THG$^+$08] Ronald Toegl, Georg Hofferek, Karin Greimel, Adrian Leung, Raphael Chung-Wei Phan, and Roderick Bloem. Formal analysis of a TPM-based secrets distribution and storage scheme. In *ICYCS*, pages 2289–2294. IEEE Computer Society, 2008.

[Tse68] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2:115–125, 1968.

[VYY10] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 327–338. ACM, 2010.

[WB93] Yosinori Watanabe and Robert Brayton. Heuristic minimization of multiple-valued relations. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 12(10):1458–1472, 1993.

[Wra76] Celia Wrathall. Complete sets and the polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):23–33, 1976.

A
List of Publications

According to the Statutes of the Doctoral School of Computer Science at Graz University of Technology, a PhD thesis must contain a list of publications of the candidate, detailing the relationship between the thesis and the (relevant) publications. The reference date for the following list of publications is May 2014.

A.1 Journal Publications

This section lists all journal publications in reverse chronological order.

- [BCG$^+$14] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.

- [KHB13] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *STTT*, 15(5-6):563–583, 2013.

A.2 Publications in Conference and Workshop Proceedings

This section lists all publications in conference and workshop proceedings in reverse chronological order.

- [HGK^{+}13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In Jobstmann and Ray [JR13], pages 77–84

- [EKH12] Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 91–100. IEEE, 2012

- [BDF^{+}12] Roderick Bloem, Rolf Drechsler, Görschwin Fey, Alexander Finder, Georg Hofferek, Robert Könighofer, Jaan Raik, Urmas Repinski, and André Sülflow. FoREnSiC — an automatic debugging environment for C programs. In Armin Biere, Amir Nahir, and Tanja E. J. Vos, editors, *Haifa Verification Conference*, volume 7857 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2012

- [BGH^{+}12] Roderick Bloem, Hans-Jürgen Gamauf, Georg Hofferek, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems with RATSY. In Doron Peled and Sven Schewe, editors, *SYNT*, volume 84 of *EPTCS*, pages 47–53, 2012

- [SHB11] Matthias Schlaipfer, Georg Hofferek, and Roderick Bloem. Generalized reactivity(1) synthesis without a monolithic strategy. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *Haifa Verification Conference*, volume 7261 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2011

- [HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42

- [KHB10] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging unrealizable specifications with model-based diagnosis. In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Haifa Verification Conference*, volume 6504 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2010

- [BCG^{+}10] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY - a new requirements analysis tool with synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer, 2010

- [KHB09] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In Biere and Pixley [BP09], pages 152–159

- [THG^{+}08] Ronald Toegl, Georg Hofferek, Karin Greimel, Adrian Leung, Raphael Chung-Wei Phan, and Roderick Bloem. Formal analysis of a

TPM-based secrets distribution and storage scheme. In *ICYCS*, pages 2289–2294. IEEE Computer Society, 2008

- [HW08] Georg Hofferek and Johannes Wolkerstorfer. Coupon recalculation for the GPS authentication scheme. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008

A.3 Relationship between Publications and Thesis

This thesis is based on the following publications (in chronological order):

- [HB11] Georg Hofferek and Roderick Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In Singh et al. [SJKB11], pages 31–42
- [EKH12] Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 91–100. IEEE, 2012
- [HGK$^+$13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In Jobstmann and Ray [JR13], pages 77–84

Throughout this thesis, sections which reuse material from these publications are marked with grey boxes entitled **Declaration of Sources**. In addition to that, the most important relations between these publications and this thesis are stated below.

In [HB11], we first introduced the concept of synthesizing Boolean signals from specifications with uninterpreted functions. We also provided a proof of decidability, and showed how to reduce the problem to propositional logic. Using an illustrative toy example, we also showed how to model controller synthesis problems for pipelined circuits. Thus, Chapter 3 and Chapter 4 are largely based on [HB11].

In [HGK$^+$13], we have shown how interpolation (on theory level) can be used to solve our synthesis problem. Furthermore, we introduced the concept of n-interpolation and showed how it can be done. Thus, Chapter 5 (with the exception of Section 5.4) is largely based on [HGK$^+$13].

In [EKH12], we show how to use computational learning to obtain functional implementations from non-deterministic relations. The alternative synthesis approach presented in Section 6.4 is based on the same idea. Moreover, in [EKH12], we discussed alternative approaches for extracting certificates from general strategies, and our experiences with them. This discussion is the basis for Section 4.2.6.

B
Cooperations

According to the Statutes of the Doctoral School of Computer Science at Graz University of Technology, a PhD thesis must contain an explanation of cooperations concerning the work described in the thesis. The following list details all such notable cooperations between the author and other persons. The list is in no particular order.

- There have been frequent and ongoing discussions with Roderick Bloem on all parts of this thesis.
- Robert Könighofer participated in many discussions on many different aspects of this thesis, and provided valuable feedback.
- Bettina Könighofer provided assistance in implementing and debugging parts of SURAQ.
- Johannes Winter provided his vast experience with programming and debugging tools to help foster the development of SURAQ.
- Christoph Hillebold implemented SURAQ's parser for VERIT proofs and also refactored the `formula` package of SURAQ such that formulas became immutable, unique objects.
- A first implementation of reordering of resolution proofs within SURAQ was done by Ashutosh Gupta.
- The concept of n-interpolation was developed in close cooperation with Ashutosh Gupta. Jie-Hong Roland Jiang suggested to use Pudlák's symmetric interpolation scheme, instead of the asymmetric scheme by McMillan.

- David Kofler implemented an alternative approach to obtain proofs without non-colorable literals, by replacing the equality predicate with an uninterpreted ternary predicate. This was part of David's Bachelor's thesis, which was supervised by the author of this thesis.
- Pascal Fontaine provided valuable background information on veriT and also contributed to fruitful discussion on how to obtain colorable, local-first proofs.
- Bruno Woltzenlogel Paleo provided some customizations of Skeptik which made interaction with Suraq much easier. He also was part of some very fruitful discussions on proof transformations.
- Georg Weissenbacher contributed to the development of modular SMT solving, by providing his expertise on interpolation in first-order theories.
- Georg Schadler is implementing the modular SMT solving approach as part of his Bachelor's thesis, supervised by the author of this thesis. At the time of writing this list, this was ongoing work.

Index

Note

This index does (in general) not contain any entries that are also listed in the glossary, acronyms, or notation section.

Author Index

Graz University of Technology

Senat

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Signaturwert	rcMKiJvN8RT0panZWXWMdtgx6oUVUwryiLdY9H+8LXXAaU/Ch5c2UW6ylQQq5Bx3iar5ndZz5 7gDmjJcD4SP9w==	
	Unterzeichner	DI Georg Hofferek
	Aussteller-Zertifikat	CN=a-sign-Premium-Sig-02,OU=a-sign-Premium-Sig-02, O=A-Trust Ges. f. Sicherheitssysteme im elektr. Datenverkehr GmbH,C=AT
	Serien-Nr.	946972
	Methode	urn:pdfsigfilter:bka.gv.at:binaer:v1.1.0
	Parameter	etsi-moc-1.1:ecdsa-sha256@173e96df
Prüfinformation	Signaturpruefung unter: http://www.signaturpruefung.gv.at	
Hinweis	Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument ist gemäß § 4 Abs. 1 Signaturgesetz einem handschriftlich unterschriebenen Dokument grundsätzlich rechtlich gleichgestellt.	
Datum/Zeit-UTC	2014-06-17T09:12:10Z	

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)