# Synthesizing Multiple Boolean Functions using Interpolation on a Single Proof

Georg Hofferek[1] Ashutosh Gupta[2] Bettina Könighofer[1] Jie-Hong Roland Jiang[3] Roderick Bloem[1]
[1]Graz University of Technology, Austria [2]IST Austria [3]National Taiwan University

*Abstract*—**It is often difficult to correctly implement a Boolean controller for a complex system, especially when concurrency is involved. Yet, it may be easy to formally specify a controller. For instance, for a pipelined processor it suffices to state that the visible behavior of the pipelined system should be identical to a non-pipelined reference system (Burch-Dill paradigm). We present a novel procedure to efficiently synthesize multiple Boolean control signals from a specification given as a quantified first-order formula (with a specific quantifier structure). Our approach uses uninterpreted functions to abstract details of the design. We construct an unsatisfiable SMT formula from the given specification. Then, from just one proof of unsatisfiability, we use a variant of Craig interpolation to compute multiple coordinated interpolants that implement the Boolean control signals. Our method avoids iterative learning and back-substitution of the control functions. We applied our approach to synthesize a controller for a simple two-stage pipelined processor, and present first experimental results.**

## I. Introduction

Some program parts are easier to write than others. Freedom of deadlocks, for instance, is trivial to specify but not to implement. These parts lend themselves to synthesis, in which a difficult part of the program is written automatically. This approach has been followed in program sketching [20], [22], [21], in lock synthesis [25], and in synthesis using templates [9], [23], [24].

In this paper, we consider systems that have multiple unimplemented Boolean control signals. The systems that we will consider may not be entirely Boolean. We will consider systems with uninterpreted functions, but our method extends to systems with linear arithmetic. For example, consider a microprocessor. Following Burch and Dill [5], we assume that a reference implementation of the datapath is available. Constructing a pipelined processor is not trivial, as it involves implementing control logic signals that control the hazards arising from concurrency in the pipeline. Correctness of the pipelined processor is stated as equivalence with the reference implementation. In this setting, we can avoid the complexity of the datapath (which is the same in the two implementations) by abstracting it away using uninterpreted functions. Where Burch and Dill verify that the implementation of the control signals is correct, we construct a correct implementation automatically. This problem was previously addressed in [12]. We improve over that paper by directly encoding the problem into SMT, thus avoiding BDDs, and by avoiding backsubstitution in case multiple functions are synthesized.

Our approach is also applicable to synthesis of conditions in (loop-free) programs. As noted in [9], synthesizing loop-free programs can be a building block of full program synthesis. Prior work [20] presented various techniques to deal with finite loops. Those techniques are also applicable in our framework.

To synthesize a single missing signal, we can introduce a fresh uninitialized Boolean variable $c$. We can express the specification as a logical formula $\forall I \exists c \forall O. \Phi(I, c, O)$, which states that, for each input $I$, there exists a value of $c$ such that each output $O$ of the function is correct. Here, $I$ and $O$ can come from non-Boolean domains. If an implementation is possible, the formula is valid and a witness function for $c$ is an implementation of the missing signal.

Following [14], we can generate a witness using interpolation. In this paper, we generalize this approach by allowing $n \geq 1$ missing components to be synthesized simultaneously. This leads us to a formula of the form $\forall I \exists c_1 \dots c_n \forall O. \Phi(I, c_1 \dots c_n, O)$. We use an SMT solver to prove a related formula unsatisfiable and use interpolation [18] to obtain the desired witness functions. The first contribution of this paper is to extend prior work [14] beyond the propositional level, and consider formulas expressed in the theory of uninterpreted functions and equality. As a second contribution, we propose a new technique, called *$n$-interpolation*, which corresponds to simultaneously computing $n$ *coordinated* interpolants from just one proof of unsatisfiability. Like the interpolation procedures of [11], [15], we need a "colorable" proof, which we produce by transforming a standard proof from an SMT solver.

Our algorithm avoids the iterative interpolant computation described in [14], where interpolants are iteratively substituted into the formula. As the iterative approach needs one SMT solver call per witness function, and interpolants may grow dramatically over the iterations, this computation may be costly and may yield large interpolants. A similar back-substitution method is also used in [2] for GR(1) synthesis and in [16] for functional synthesis. Our new method requires the expansion of the the (Boolean) existential quantifier, increasing the size of the formula exponentially (w.r.t. the number of control signals). Note, however, that previous approaches [14] have the same limitation.

## II. Illustration

In this Section we illustrate our approach using a simple controller synthesis problem. Figure 1 shows an incomplete

This research was supported by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), the Austrian Science Fund (FWF) through projects RiSE (S11406-N23) and QUAINT (I774-N23), and ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

ISBN 978-0-9835678-3-7/13. 

hardware design. There are 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 $c_2$. The controller of $c_1$ and $c_2$ is not implemented. Suppose the specification of the incomplete design states that the signs of the two outputs must be different. Formally, the specification is

$$\forall i_1, i_2.\exists c_1, c_2.\forall o_1, o_2.((c_1 \wedge o_1 = i_1 \vee \neg c_1 \wedge o_1 = neg(i_1))\wedge (c_2 \wedge o_2 = i_2 \vee \neg c_2 \wedge o_2 = neg(i_2))) \rightarrow (pos(o_1) \leftrightarrow \neg pos(o_2)),$$

where the predicate $pos$ returns T iff its parameter is positive. We can compute witness functions for $c_1$ and $c_2$ using $n$-interpolation.[1] Our method returns witness functions $c_1 = pos(i_1)$ and $c_2 = \neg pos(i_2)$. (Other functions are also possible.)

Note that computing two interpolants independently may not work. For instance, we may choose $c_1 = \texttt{T}$ or we can take $c_2 = \texttt{T}$, but we cannot choose $c_1 = c_2 = \texttt{T}$. This problem is normally solved by substituting one solution before the next is computed, but our method computes both interpolants simultaneously and in a coordinated way.

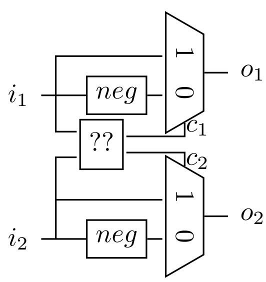



Fig. 1. Example of controller synthesis.

## III. Preliminaries

### A. Uninterpreted Functions and Arrays

We consider the *Theory of Uninterpreted Functions and Equality* $\mathcal{T}_U$. We have variables $x \in \mathcal{X}$ from an uninterpreted domain, Boolean variables $b \in \mathcal{B}$, uninterpreted function symbols $f \in \mathcal{F}$, and uninterpreted predicate symbols $P \in \mathcal{P}$. The following grammar defines the syntax of the formulas in $\mathcal{T}_u$.

$$\begin{aligned} \text{terms} \ni t &::= x \mid f(t, \dots, t), \\ \text{atoms} \ni a &::= b \mid P(t, \dots, t) \mid t = t, \\ \text{formulas} \ni \phi &::= a \mid \neg\phi \mid \phi \vee \phi. \end{aligned}$$

Let $\phi_1 \wedge \phi_2$ be short for $\neg(\neg\phi_1 \vee \neg\phi_2)$. Let $a \neq b$ be short for $\neg(a = b)$. Let $\texttt{T} = \phi \vee \neg\phi$, let $\texttt{F} = \neg\texttt{T}$, and let $\mathbb{B} = \{\texttt{T}, \texttt{F}\}$.

A *literal* is an atom or its negation. Let $l$ be a literal. If $l = \neg a$ then let $\neg l = a$. A *clause* is a set of literals, interpreted as the disjunction. The empty clause $\emptyset$ denotes F. A *conjunctive formula* is the negation of a clause. A *CNF formula* is a set of clauses. A CNF formula is interpreted as the conjunction of its clauses. Since any formula can be converted into a CNF formula, we will assume that all the formulas in this paper are CNF formulas. Let $\phi$ and $\psi$ be CNF formulas/clauses/literals. Let $Symb(\phi)$ be the set of variables, functions, and predicates occurring in $\phi$. Let $\phi \preceq \psi$ iff $Symb(\phi) \subseteq Symb(\psi)$. Let $Lits(\phi) = \{a, \neg a \mid a \text{ is an atom in } \phi\}$. For a clause $C$, let $C|_\phi = \{s \in C \mid s \preceq \phi\}$.

[1]We must add the axiom $(pos(i_1) \oplus pos(neg(i_1))) \wedge (pos(i_2) \oplus pos(neg(i_2)))$.

$$\textsc{Hyp}\ \frac{}{C}\ C \in \phi,\ \phi \in \text{CNF} \qquad \textsc{Axi}\ \frac{}{C} \vdash_{\mathcal{T}_U} C$$

$$\textsc{Res}\ \frac{a \vee C \quad \neg a \vee D}{C \vee D}$$

Fig. 2. Sound and complete proof rules for the theory $\mathcal{T}_U$.

Arrays are useful for modeling memory whose size is not known a priori. We will use a decidable fragment, known as the *Array Property Fragment* with *uninterpreted indices* to create specifications from which we synthesize controllers. Bradley et al. [4] present an algorithm to reduce formulas with array properties to equisatisfiable formulas over the theory of uninterpreted functions. Hofferek and Bloem [12] show that this algorithm generalizes to the quantified formulas that occur in controller synthesis problems. For the rest of this paper, we assume that specifications and formulas containing array properties have been reduced to formulas over the theory of uninterpreted functions.

### B. Proofs of Unsatisfiability

We consider the usual semantics of formulas in $\mathcal{T}_u$. The problem of proving unsatisfiability of formulas is decidable. Many *Satisfiability Modulo Theories (SMT) Solvers* exist that can decide the satisfiability of CNF-$\mathcal{T}_U$ formulas, and, in case the formula is not satisfiable, produce a *proof of unsatisfiability*.

A (named) *proof rule* is a template for a logic entailment between a (possibly empty) list 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 proofs we consider will be based on the rules given in Fig. 2. They form a sound and complete proof system for proving unsatisfiability of a CNF-$\mathcal{T}_U$ formula $\phi$. The Hyp rule is used to introduce clauses from $\phi$ into the proof. The Axi rule is used to introduce theory-tautology clauses. In their simplest form, these clauses represent concrete instances of theory axioms (reflexivity, symmetry, transitivity and congruence). However, as our proof transformation algorithms will produce theory tautologies that are based on several axioms, we use the following, less restrictive, definition.

**Definition 1** (Theory-Tautology Clause)**.** *A* theory-tautology clause *is a clause of the form* $(\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_k \vee b)$ *that is tautologically true within the theory* $\mathcal{T}_U$*. The literals* $\neg a_i$*, for* $0 < i \leq k$*, are called the* implying literals *and the (positive) literal* $b$ *is called the* implied literal*.*

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* or the *pivot*.

**Definition 2** (Unsatisfiability Proof)**.** *An* unsatisfiability proof *for a CNF-*$\mathcal{T}_U$ *formula* $\phi$ *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) *$clause(r) = \emptyset$.*
(2) *For all $n \in N_L$, $clause(n)$ is either a clause from $\phi$ (if $rule(n) =$* HYP*) or a theory-tautology clause (if $rule(n) =$* AXI*).*
(3) *The nodes in $N_L$ whose clauses are theory-tautology clauses can be ordered in such a way that for each such node each implying literal either occurs in $\phi$, or is an implied literals of the tautology clause of a a preceding node (according to the order).*[2]
(4) *The root has no incoming edges, the leaves have no outgoing edges, and all nodes in $n \in N \setminus N_L$ have exactly 2 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 valid instance of proof rule $rule(n)$.*

We used the VERIT SMT solver [3], which provides proofs that conform to these requirements.

### C. Transitivity-Congruence Chains

Given a set $A$ of atoms, we can use the well-known congruence-closure algorithm to construct a *congruence graph* [8] according to the following definition.

**Definition 3** (Congruence Graph)**.** *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 $f(a_1, \ldots, a_k)$ and $f(b_1, \ldots, b_k)$ of the same uninterpreted function $f$. In this case, the congruence justification is a set of $k$ paths in the graph connecting the $a_i$ and $b_i$ respectively, not using the edge which they label.*

**Definition 4** (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 $Lits(\pi)$ 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 $\pi_i$ is recursively defined as $\bigcup_i Lits(\pi_i)$.*

**Theorem 1.** *The conjunction of the literals in a transitivity-congruence chain $(a \rightsquigarrow b)$ implies $a = b$ within $\mathcal{T}_U$. I.e., $(\bigvee_{l \in Lits(a \rightsquigarrow b)} \neg l) \vee (a = b)$ is a theory-tautology clause.*

### D. Craig Interpolation

Let $\phi$ and $\psi$ be CNF formulas such that $\phi \wedge \psi$ is unsatisfiable. The algorithm presented in [18] for computing an interpolant between $\phi$ and $\psi$ needs a proof of unsatisfiability of $\phi \wedge \psi$. By annotating this proof with the partial interpolants, the algorithm computes the interpolant. In this paper, we present slightly different annotation rules to compute interpolants, which are results of mixing ideas from [15], [19].

[2] This means that every (new) literal is defined only in terms of previously known literals. The order corresponds to the order in which the solver introduced the new literals.

$$\textsc{iHyp-}\phi\ \frac{}{C[\mathtt{F}]}\ C \in \phi \qquad \textsc{iHyp-}\psi\ \frac{}{C[\mathtt{T}]}\ C \in \psi$$

$$\textsc{iAxi-}\phi\ \frac{}{C[\mathtt{F}]}\ C \preceq \phi, \vdash_{\mathcal{T}_U} C \qquad \textsc{iAxi-}\psi\ \frac{}{C[\mathtt{T}]}\ C \preceq \psi, \vdash_{\mathcal{T}_U} C$$

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

$$\textsc{iRes-}\phi\ \frac{a \vee C[I_C] \qquad \neg a \vee D[ID]}{C \vee D[I_C \vee I_D]}\ a \preceq \phi, a \npreceq \psi$$

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

Fig. 3. Interpolating proof rules

**Definition 5** (Partial interpolant)**.** *Let $C$ be a clause such that $\phi \wedge \psi \rightarrow C$. A formula $I$ is a* partial interpolant *for $C$ between $\phi$ and $\psi$ if $\phi \rightarrow C|_\phi \vee I$, and $\psi \rightarrow C|_\psi \vee \neg I$, and $I \preceq \phi$, and $I \preceq \psi$. If $I$ is a partial interpolant for $C = \emptyset$ between $\phi$ and $\psi$, then $I$ is an interpolant between $\phi$ and $\psi$.*

In Figure 3, we present interpolating proof rules. In an unsatisfiability proof of $\phi \wedge \psi$, these rules annotate (in square brackets) each conclusion with a partial interpolant for the conclusion. Rules IHYP-$\phi$ and IHYP-$\psi$ are used at leaf nodes that have clauses from $\phi$ and $\psi$ respectively. Rules IAXI-$\phi$ and IAXI-$\psi$ are used for leaves with theory-tautology clauses, whose symbols are a subset of the symbols in $\phi$ and $\psi$ respectively. Note that these rules assume that the unsatisfiability proof of $\phi \wedge \psi$ is *colorable*.

**Definition 6** (Colorable Proof)**.** *A proof of unsatisfiability of $\phi \wedge \psi$ is* colorable *if for every leaf $n_L$ of the proof $Symb(clause(n_L)) \subseteq Symb(\phi)$ or $Symb(clause(n_L)) \subseteq Symb(\psi)$.*

In Section V, we will present an algorithm that transforms a proof into a colorable proof. Due to this assumption we can easily find corresponding partial interpolants for theory-tautology clauses, which are either `T` or `F`. For internal proof nodes, we follow Pudlák's interpolation system [19]. The annotation of the root node (with the empty clause) is the interpolant between $\phi$ and $\psi$. See [7] for a proof of correctness of the annotating proof rules.

## IV. CONTROLLER SYNTHESIS

### A. Overview

Following [12], we assume that synthesis problems are given as formulas of the form

$$\forall \bar{i}\ \exists \bar{c}\ \forall \bar{o}.\ \Phi(\bar{i}, \bar{c}, \bar{o}), \tag{1}$$

where $\bar{c}$ is a vector of Boolean variables and $\Phi$ is a formula over theory $\mathcal{T}_U$. Let $\bar{c} = (c_1, \ldots, c_n)$. Each $c_i$ represents a missing if-condition in a program or a one-bit control signal in a hardware design. Witness functions for the existentially quantified variables in Eq. (1) are implementations of the missing components. Therefore, the synthesis problem is equivalent to finding such witness functions. I.e., find $(f_1(\bar{i}), \ldots, f_n(\bar{i}))$ such that $\forall \bar{i}\ \forall \bar{o}.\ \Phi(\bar{i}, (f_1(\bar{i}), \ldots, f_n(\bar{i})), \bar{o})$ holds true.

We compute the witness functions through the following steps:

(1) Expand the existential quantifier and negate the formula $\Phi$ to obtain an unsatisfiable formula $\phi$ (Sec. IV-B).

(2) Obtain a proof of unsatisfiability from an SMT solver.
(3) Transform the proof into a colorable, local-first proof (Sec. V).
(4) Perform $n$-interpolation on the transformed proof. The elements of the $n$-interpolant correspond to the witness functions (Sec. IV-B).

We will first introduce the notion of $n$-interpolation and show how it is used to find witness functions in Section IV-B. Subsequently, we will show how to transform a proof of unsatisfiability so that it is suitable for $n$-interpolation in Section V.

### *B. Finding Witness Functions through Interpolation*

Jiang et al. [14] show how to compute a witness function in Eq. (1) using interpolation if $\bar{c}$ contains a single Boolean $c$. In this case, Eq. (1) reduces to $\forall \bar{i}\ \exists c\ \forall \bar{o}.\ \Phi(\bar{i}, c, \bar{o})$. After expanding the existential quantifier by instantiating the above formula for both Boolean values of $c$ and renaming $\bar{o}$ in each instantiation, we obtain the equivalent formula $\forall \bar{i}\ \forall \bar{o}_{\mathtt{F}}, \bar{o}_{\mathtt{T}}.\ \Phi(\bar{i}, \mathtt{F}, \bar{o}_{\mathtt{F}}) \vee \Phi(\bar{i}, \mathtt{T}, \bar{o}_{\mathtt{T}})$. Since all the quantifiers are universal, the disjunction is valid. Therefore, its negation $\neg\Phi(\bar{i}, \mathtt{F}, \bar{o}_{\mathtt{F}}) \wedge \neg\Phi(\bar{i}, \mathtt{T}, \bar{o}_{\mathtt{T}})$ is unsatisfiable. The interpolant between the two conjuncts is the witness function for variable $c$.

**Theorem 2.** *The interpolant between $\neg\Phi(\bar{i}, \mathtt{F}, \bar{o}_{\mathtt{F}})$ and $\neg\Phi(\bar{i}, \mathtt{T}, \bar{o}_{\mathtt{T}})$ is the witness function for $c$. (For a proof, see [13].)*

We now extend this idea to compute witness functions when $\bar{c}$ is a vector of Booleans $(c_1, \ldots, c_n)$. Let $\mathbb{B}^n$ denote the set of vectors of length $n$ containing $\mathtt{F}$s and $\mathtt{T}$s. For vector $w \in \mathbb{B}^n$, let $w_j$ be the Boolean value in $w$ at index $j$. Since $\bar{c}$ is a Boolean vector, we can expand the existential quantifier for $\bar{c}$ in Eq. (1) by enumerating the finitely many possible values of $\bar{c}$ to obtain $\forall \bar{i} \bigvee_{w \in \mathbb{B}^n} \forall \bar{o}\ .\ \Phi(\bar{i}, w, \bar{o})$. By dropping the quantifiers and renaming $\bar{o}$ accordingly, we obtain $\bigvee_{w \in \mathbb{B}^n} \Phi(\bar{i}, w, \bar{o}_w)$. It is valid iff Eq. (1) is valid. Let $\phi$ denote its negation $\bigwedge_{w \in \mathbb{B}^n} \neg\Phi(\bar{i}, w, \bar{o}_w)$, which is unsatisfiable. Let $\phi_w$ denote $\neg\Phi(\bar{i}, w, \bar{o}_w)$. We will call the $\phi_w$s the $2^n$ *partitions* of $\phi$. We will learn a vector of *coordinated* interpolants from an unsatisfiability proof of $\phi$. These interpolant formulas will be witness functions for $\bar{c}$. Since $\phi_w$s are obtained by only renaming variables, the shared symbols between any two partitions are equal.

**Definition 7** (Global and Local Symbols)**.** *Symbols in the set $G = \bigcap_{w \in \mathbb{B}^n} Symb(\phi_w)$ are called* global *symbols. All other symbols are called* local *(w.r.t. the one partition in which they occur).*

Let $\overline{I}$ be a vector of formulas $(I_1, \ldots, I_n)$. Let $\oplus$ be the exclusive-or (xor) operator. For a word $w \in \mathbb{B}^n$, let $\overline{I}' = \overline{I} \oplus w$ if for each $j \in 1..n$, $I'_j = I_j \oplus w_j$. Let $\bigvee \overline{I}$ be short for $I_1 \vee \cdots \vee I_n$. Let $C|_w = C|_{\phi_w}$. The following definition generalizes the notion of interpolant and partial interpolant from two formulas to $2^n$ formulas.

**Definition 8** ($n$-Partial Interpolant)**.** *Let $C$ be a clause such that $(\bigwedge_{w \in \mathbb{B}^n} \phi_w) \rightarrow C$. An n-partial interpolant $\overline{I}$ for $C$ w.r.t. the $\phi_w$s is a vector of formulas with length $n$ such that $\forall w \in \mathbb{B}^n.\ \phi_w \rightarrow (C|_w \vee \bigvee(\overline{I} \oplus w))$ and $\overline{I} \preceq G$. If $C = \emptyset$ then $\overline{I}$ is an* n-interpolant *w.r.t. the $\phi_w$s.*

$$\textsc{mHyp}\ \frac{}{C[w]}\ C \in \phi_w \qquad \textsc{mAxi}\ \frac{}{C[w]}\ C \preceq \phi_w$$

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

$$\textsc{mRes-G}\ \frac{a \vee C[\overline{I}^C] \qquad \neg a \vee D[\overline{I}^D]}{C \vee D\ \ [(\ (a \vee I_1^C) \wedge (\neg a \vee I_1^D), \ \ldots, \ (a \vee I_n^C) \wedge (\neg a \vee I_n^D)\ )]}\ a \preceq G$$

Fig. 4. $n$-Interpolating proof rules for an unsatisfiable $\phi = \bigwedge_{w \in \mathbb{B}^n} \phi_w$. These rules can only annotate proofs that are colorable and local-first.

**Theorem 3.** *An $n$-interpolant w.r.t. the $\phi_w$s constitutes witness functions for the variables in $\bar{c}$. (For a proof see [13].)*

### *C. Computing $n$-interpolants*

In Figure 4, we present the proof rules for $n$-interpolants. These rules annotate each conclusion of a proof step with an $n$-partial interpolant for the conclusion w.r.t. the $\phi_w$s. These annotation rules require two properties of the proof. First, it needs to be colorable.[3] Second, it needs to be local-first.

**Definition 9** (Local-first Proof)**.** *A proof of unsatisfiability is* local-first*, if for every resolution node with a local pivot both its premises are derived from the same partition.*

The rule mHyp annotates the derived clause $C$ with $w$ if $C$ appears in partition $\phi_w$. Similarly, the rule mAxi annotates theory-tautology clause $C$ with $w$ if $C \preceq \phi_w$. Rules mRes and mRes-G annotate resolution steps. mRes-G is only applicable if the pivot is global and follows Pudlák's interpolation system $n$ times. mRes is only applicable if both premises are annotated with the same $n$-partial interpolant and this $n$-partial interpolant is an element of $\mathbb{B}^n$. Due to the local-first assumption on proofs, these rules will always be able to annotate a proof.

**Theorem 4.** *Annotations in the rules in Figure 4 are $n$-partial interpolants for the respective conclusions w.r.t. the $\phi_w$s. (For a proof see [13].)*

Since the $n$-interpolant 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 pivot, a multiplexer that has the pivot 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 pivots, we use the constants $\mathtt{T}, \mathtt{F}$, depending on which partition the node belongs to. The output of the multiplexer corresponding to the root node is the final witness function. Note that, unless we apply logical simplifications, the circuits for all witness functions 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

[3] We extend Def. 6 from two formulas to $2^n$ partitions in the obvious way.

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.

The local-first property is actually needed to correctly compute witness functions using Pudlák's interpolation system. In [13], we illustrate this observation with an example. Also note that McMillan's interpolation [18] system does not produce correct witness functions even with the local-first property.

## V. Algorithms for Proof Transformation

Our interpolation procedure requires 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 proof conforming to Def. 2 into one that is colorable and local-first. Our proof transformation works in three steps. First, we will remove any non-colorable literals from the proof. Second, we will split any non-colorable theory-tautology clauses. This gives us a colorable proof. In the third step, we will reorder resolution steps to obtain the local-first property [7]. For ease of presentation, we will assume that the proof is a tree (instead of a DAG). The method extends to proofs in DAG form.

### A. Removing Non-Colorable Literals

**Definition 10** (Colorable and Non-Colorable Literals)**.** *A literal $a$ is* colorable *with respect to a partition $\phi_w$ ($w$-*colorable*) iff $a \preceq \phi_w$. A literal that is not $w$-colorable for any partition $w$ is called* non-colorable*.*

Note that global literals are $w$-colorable for every $w$. By definition, the formula $\phi$ is free of non-colorable literals (equalities and predicate instances). Thus, the only way through which non-colorable literals can be introduced into the proof are theory-tautology clauses.

We search the proof for a theory-tautology clause that introduces a non-colorable literal $a$ and has only colorable literals as implying literals. We call this proof node the *defining node* $n_d$. At least one such leaf must exist. We remove this non-colorable literal from the proof as follows. Starting from $n_d$, we traverse the proof towards the root, until we find a node, which we call *resolving node* $n_r$, whose clause does not contain the literal $a$ any more. Since the root node does not contain any literals, such a node always exists. Let $n_a$ and $n_{\neg a}$ be the premises of $n_r$, respectively, depending on which phase of literal $a$ their clause contains. From $n_{\neg a}$, we traverse the proof towards the leaves along nodes that contain the literal $\neg a$. Note that any leaf that we reach in this way must be labeled with a theory-tautology clause, as clauses from $\phi$ cannot contain the non-colorable literal $\neg a$. Note that $\neg a$ is among the implying literals of such a leaf node's clause. I.e., such nodes *use* the literal to imply another one. We will therefore call such a node a *using node* $n_u$. We update $clause(n_u)$, by removing $\neg a$ and adding the implying literals of $clause(n_d)$ instead.

It is easy to see that this does not affect $clause(n_u)$'s property of being a theory-tautology clause. Suppose $clause(n_d)$ is $(\neg x_1 \vee \ldots \vee \neg x_k \vee a)$. Then $\bigwedge_{i=1}^{k} x_i \to a$. By reversing the implication we obtain $\neg a \to \bigvee_{i=1}^{k} \neg x_i$. Therefore, replacing $\neg a$ with the disjunction of the implying literals of $clause(n_d)$ in $clause(n_u)$ is sound.

To keep the proof internally consistent, we have to do the same replacement on all the nodes on the path between $n_u$ and $n_r$. The node $n_r$ itself is not changed, as $clause(n_r)$ does not contain the non-colorable literal $(\neg)a$ any more. I.e., the last node that is updated is the node $n_{\neg a}$.

Now we have to distinguish two cases. The first case is that node $n_a$ still contains all of the implying literals of $n_d$. In this case, $clause(n_r) = clause(n'_{\neg a})$, where $n'_{\neg a}$ is the updated node $n_{\neg a}$. Thus, we use $n'_{\neg a}$ instead of $n_r$ in $n_r$'s parent node. In the second case, some of the implying literals of $clause(n_d)$ have already been resolved on the path from $n_d$ to $n_r$. In that case $clause(n'_{\neg a})$ contains literals that do not occur in $clause(n_r)$. Let $x_l$ be one such literal. We search the path from $n_d$ to $n_r$ for the node that uses $x_l$ as a pivot. Its premise that is not on the path from $n_d$ to $n_r$ contains $\neg x_l$. We use this node and the node $n'_{\neg a}$ as premises for a new resolution node with $x_l$ as pivot. Note that this resolution may introduce more literals that do not appear in $clause(n_r)$ any more. However, just as with $x_l$, any such literal must have been resolved somewhere on the path between $n_d$ and $n_r$. Thus, we repeat this procedure, replicating the resolution steps that took place between $n_d$ and $n_r$, until we get a node whose clause is identical to $clause(n_r)$. This node can then be used instead of $n_r$ in $n_r$'s parent node. Finally, we remove all nodes that are now unreachable from the proof.

**Example 1.** An illustrative example of this procedure is shown in Figure 5.

We repeat this procedure for all leaves with a non-colorable implied literal and (all) colorable implying literals. Note that one application of this procedure may convert a node where a non-colorable literal was implied by at least one other non-colorable literal into a node where the implied non-colorable literal is now implied only by colorable literals. Nevertheless this procedure terminates, as the number of leaves with non-colorable implied literals decreases with every iteration. Each iteration removes (at least) one such leaf from the proof and no new leaves are introduced.

**Theorem 5.** *Upon termination of this procedure, the proof does not contain any non-colorable literals.*

### B. Splitting Non-Colorable Theory-Tautology Clauses

After removing all non-colorable literals, the proof may still contain non-colorable theory-tautology clauses, i.e., theory-tautology clauses that contain local literals from more than one partition. We split such leaves into several new theory-tautology clauses, each containing only $w$-colorable literals, and, via resolution, obtain a (now internal) node with the same clause as the original non-colorable theory-tautology clause. Note that internal nodes with non-colorable clauses are not a problem for our interpolation procedure, but leaves with non-colorable clauses are. We will show how to split a non-colorable theory-tautology clause with an implied equality

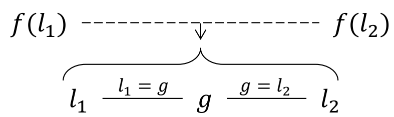
$$\textsc{Res}\,\frac{\textsc{Res}\,\dfrac{n_1 : (l_1 = z_g \vee x_g = y_g) \quad n_d : (l_1 \neq z_g \vee z_g \neq l_2 \vee l_1 = l_2)}{n_a : (x_g = y_g \vee z_g \neq l_2 \vee l_1 = l_2)} \quad \textsc{Res}\,\dfrac{n_u : (l_1 \neq l_2 \vee f(l_1) = f(l_2)) \quad n_3 : (f(l_1) \neq f(l_2) \vee u_g \neq v_g)}{n_{\neg a} : (l_1 \neq l_2 \vee u_g \neq v_g)}}{n_r : (x_g = y_g \vee z_g \neq l_2 \vee u_g \neq v_g)}$$

(a) Proof before removing non-colorable literal $l_1 = l_2$.

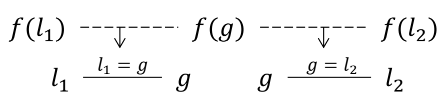
$$\textsc{Res}\,\frac{n_1 : (l_1 = z_g \vee x_g = y_g) \quad \textsc{Res}\,\dfrac{n_3 : (f(l_1) \neq f(l_2) \vee u_g \neq v_g) \quad n'_u : (l_1 \neq z_g \vee z_g \neq l_2 \vee f(l_1) = f(l_2))}{n'_{\neg a} : (l_1 \neq z_g \vee z_g \neq l_2 \vee u_g \neq v_g)}}{n''_{\neg a} : (x_g = y_g \vee z_g \neq l_2 \vee u_g \neq v_g)}$$

(b) Proof after removing non-colorable literal $l_1 = l_2$.

Fig. 5. *Removing a non-colorable literal.* Assume that term indices indicate the number of the partition the term belongs to. Index $g$ is used for global terms. This example shows how the non-colorable literal $l_1 = l_2$, introduced in node $n_d$, is removed from the proof by replacing its negative occurrences with the (colorable) defining literals $(l_1 \neq z_g \vee z_g \neq l_2)$. Note that in the original proof $l_1 \neq z_g$ is already resolved on the path from $n_d$ to $n_r$ using node $n_1$. This resolution step is replicated in the transformed proof by making a resolution step with nodes $n'_{\neg a}$ and $n_1$. Since the literal $x_g = y_g$ introduced into $n''_{\neg a}$ also occurs in the original $n_r$, and also the second defining literal $z_g \neq l_2$ occurs in $n_r$, no further resolution steps are necessary. The conclusions of $n_d$ and $n''_{\neg a}$ are identical and $n''_{\neg a}$ can be used instead of $n_r$ in $n_r$'s parent.




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




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

Fig. 6. Splitting a non-colorable transitivity-congruence chain by introducing global intermediate terms.

literal. This procedure can be trivially extended to implied literals that are uninterpreted predicate instances.

Using the implying literals of the theory-tautology clause (converted to their positive phase), we create a *congruence graph* (cf. Def. 3). Since the implying literals and the implied literal form a theory tautology, this congruence graph is guaranteed to contain a path between the the two terms equated by the implied literal. We use breadth-first search to find the shortest such transitivity-congruence chain (Def. 4).[4] The chain will be the basis for splitting the non-colorable theory tautology. First, we need to make all edges in the chain colorable. A *colorable edge* is an edge for which there is a $w$ 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-tautology clause. Edges with congruence justifications, however, may still be non-colorable. I.e., the two terms they connect might belong to different partitions, and/or some of the paths that prove equality for the function parameters might span over more than one partition. Fuchs et al. [8] 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 [8] for details.

**Example 2.** 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 Fig. 6.) 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, so that each segment uses only $w$-colorable symbols. The global terms that separate symbols with different colors are part of both segments.[5] Let's 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), 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-tautology clause. The implied literal of this tautology will be an equality between the first and the last term of the entire chain. Next, we create a theory tautology with the literals of the second segment of the chain. Note that the implied literal of this theory-tautology clause (which occurs in positive phase) is the same as the shortcut literal used in the theory-tautology clause corresponding to the first segment. There, however, it occurred in negative phase. Thus, we can use this literal for resolution between the two clauses. We obtain a node that 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. I.e., this new internal node has the same conclusion as the non-colorable theory-tautology clause from which we started.

In case the start/end of the chain is not a local term, we

[4] Note that these graphs are usually relatively small.

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

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

Fig. 7. *Splitting theory tautology clauses.* 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-tautology clause, 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 tautology clauses (nodes $n_1$ and $n_2$, respectively). By resolution we obtain $n_3$. Now, we create the tautology 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-tautology clause. By resolution over $n_3$ and $n_4$ we obtain $n_5$, whose clause is identical to the theory-tautology clause from which we started.

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. We create a theory-tautology clause 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-tautology clauses with subtrees whose leaves are all colorable theory-tautology clauses, and whose root is labeled with the same clause as the original non-colorable leaf.

**Example 3.** Fig. 7 shows how to split the non-colorable theory-tautology clause $(a_1 \neq b_1 \vee b_1 \neq c_g \vee c_g \neq d_2 \vee d_2 \neq e_2 \vee e_2 \neq f_g \vee f_g \neq h_3 \vee h_3 \neq k_g \vee k_g \neq l_1 \vee a_1 = l_1)$.

**Theorem 6.** *After applying the above procedure to all leaves with non-colorable theory-tautology clauses, the proof is colorable.*

### C. Obtaining a local-first proof

To obtain a local-first proof, we traverse the proof in topological order. Each time we encounter a resolution step that has a global pivot and we have seen local pivots among its ancestors then we apply one of the two transformation rules presented in Figure 8 depending on the matching pattern. These two transformation rules are the standard pivot reordering rules from [7]. Note that these rules assume that the proof is redundancy free, which can be achieved by the algorithms presented in [10]. After repeated application of these transformation rules, we can move the resolutions with local pivots towards the leaves of the proof until we don't have any global pivot among its descendants.

**Theorem 7.** *After exhaustive application of this transformation, we obtain a colorable, local-first proof.*

## VI. Experimental Results

We have implemented a prototype to evaluate our interpolation-based synthesis approach. We read the formula $\Phi$ corresponding to our synthesis problem (Eq. 1) from a file in SMT-LIB format [1]. As a first step, our tool performs several transformations on the input formula (reduction of arrays to uninterpreted functions [4], expansion of the existential quantifier to obtain the partitions, renaming of $\bar{o}$-variables in each partition, negation to obtain $\phi$), before giving it to the veriT solver. Second, we apply the proof transformations

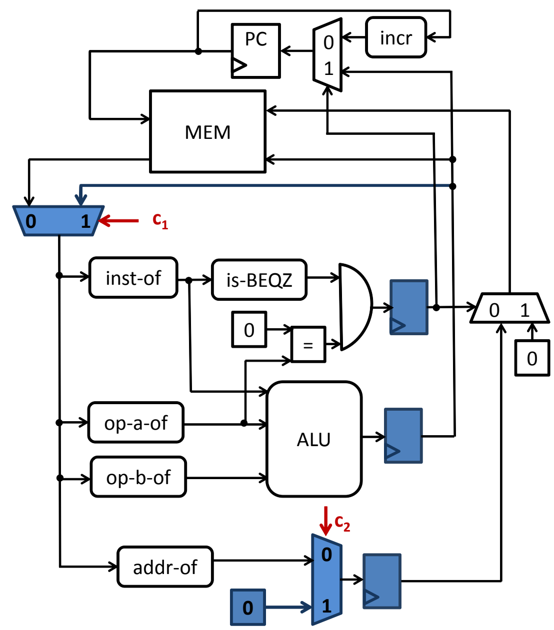



Fig. 9. A simple microprocessor with a 2-stage pipeline.

described in Section V to the proof we obtain from veriT. Third, we compute the witness functions as the $n$-interpolants w.r.t. the partitions of $\phi$.

We have checked all results using Z3 [6], by showing that $\neg\Phi(\bar{i}, (f_1(\bar{i}), \ldots, f_n(\bar{i})), \bar{o})$ is unsatisfiable.

We used our tool on several small examples and also tried one non-trivial example which we explain in more detail. In Fig. 9 we show a simple (fictitious) microprocessor with a 2-stage pipeline. MEM represents the main memory. We assume that the value at address 0 is hardwired to 0. I.e., reading from address 0 always yields value 0. 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 $c_1 = \mathtt{T}$ whenever PC equals the address register is not correct. For example, if PC $= 0$, forwarding

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

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

Fig. 8. If a local pivot $l$ occurs after a global pivot $g$ in a proof then we can rewrite the proof using one of the above transformation rules. After the transformation, the proof first resolves $l$ then $g$.

TABLE I

*Experimental results.* Columns: (1) Name; (2) Number of control signals; (3) Total synthesis time including checking the results; (4) Number of leaves with theory-tautology clauses that define a new non-colorable literal (Number of such leaves at the start of the cleaning procedure + Number of leaves introduced (and subsequently removed) by the procedure); (5) Number of leaves to be split because they contain literals from more than one partition. (Number after "/" is total number of leaves in proof at beginning of split procedure; (6) Time to reorder the proof to be local-first; (7) Number of nodes in proof from VERIT / Size of the transformed proof for interpolation (local sub-trees have been converted to leaves).

| Name | Ctrl | time [s] | # leaves to clean | # leaves to split | Reorder-Time [ms] | Proof size |
|---|---|---|---|---|---|---|
| const | 2 | 0.6 | 0 | 0 / 6 | 42 | 19 / 1 |
| illu02 | 2 | 1.1 | 1 | 1 / 65 | 83 | 205 / 12 |
| illu03 | 3 | 5.0 | 8 | 8 / 138 | 487 | 467 / 22 |
| illu04 | 4 | 8.0 | 3 | 3 / 242 | 532 | 951 / 75 |
| illu05 | 5 | 12.8 | 10 | 10 / 413 | 589 | 1588 / 78 |
| illu06 | 6 | 237.0 | 9 | 9 / 1093 | 1820 | 4691 / 370 |
| illu07 | 7 | 150.0 | 14 | 14 / 1443 | 2860 | 6824 / 555 |
| illu08 | 8 | 1270.0 | 20 | 20 / 3450 | 4980 | 17524 / 1023 |
| pipe | 1 | 1.6 | 6 + 6 | 3 / 70 | 129 | 285 / 22 |
| proc | 2 | 28.1 | 3 + 3 | 61 / 1014 | 1770 | 5221 / 1042 |

should not be done.[6] By taking out the blue parts in Fig. 9 we obtain the non-pipelined reference implementation which we used to formulate a Burch-Dill-style equivalence criterion [5]. The resulting formula was used as a specification for synthesis.

Table I summarizes our experimental results. The benchmark "const" is a simple example with 2 control signals that allows for constants as valid solutions. "illu02" is the example presented in Section II; "illu03" to "illu08" are scaled-up versions of "illu02", with increased numbers of inputs and control signals. "pipe" is the simple pipeline example that was used in [12]. "proc" is the pipelined processor shown in Fig. 9 and described above. All experiments were performed on an Intel Nehalem CPU with 3.4 GHz.

Note that using our new method we have reduced the synthesis time of "pipe" from 14 hours [12] to 1.6 seconds. As a second comparison, we tried to reduce the (quantified) input formula of "proc" to a QBF problem (using the transformations outlined in [12]) and run DEPQBF [17] on it. After approximately one hour, DEPQBF exhausted all 192 GB of main memory and terminated without a result.

## VII. CONCLUSION

Hofferek and Bloem [12] have shown that uninterpreted functions are an efficient way to abstract away unnecessary details in controller synthesis problems. By using interpolation in $\mathcal{T}_U$, we avoid the costly reduction to propositional logic, thus unleashing the full potential of the approach presented in [12]. Furthermore, by introducing the concept of $n$-interpolation, we also avoid the iterative construction which requires several calls to the SMT solver and back-substitution. The $n$-interpolation approach improves synthesis times by several orders of magnitude, compared to previous methods [12], rendering it applicable to real-world problems, such as pipelined microprocessors. We have also shown that a naive transformation to QBF is not a feasible option.

[6] 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.

## REFERENCES


[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Workshop on Satisfiability Modulo Theories*, 2010.

[2] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *DATE*, 2007.

[3] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An open, trustable and efficient SMT-solver. In *CADE*, 2009.

[4] A. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.

[5] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*. Springer, 1994. LNCS 818.

[6] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[7] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *VMCAI*, 2010.

[8] A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground interpolation for the theory of equality. *Logical Methods in Computer Science*, 8(1), 2012.

[9] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*. ACM, 2011.

[10] A. Gupta. Improved single pass algorithms for resolution proof reduction. In *ATVA*. Springer, 2012.

[11] K. Hoder, L. Kovács, and A. Voronkov. Playing in the grey area of proofs. In *POPL*. ACM, 2012.

[12] G. Hofferek and R. Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In *MemoCODE*. IEEE, 2011.

[13] G. Hofferek, A. Gupta, B. Könighofer, J.-H. R. Jiang, and R. Bloem. Synthesizing multiple boolean functions using interpolation on a single proof, 2013. Full version with appendix available at arXiv.org:1308.4767.

[14] J.-H. R. Jiang, H.-P. Lin, and W.-L. Hung. Interpolating functions from large Boolean relations. In *ICCAD*, 2009.

[15] L. Kovács and A. Voronkov. Interpolation and symbol elimination. In *CADE*. Springer, 2009.

[16] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*. ACM, 2010.

[17] F. Lonsing and A. Biere. Integrating dependency schemes in search-based QBF solvers. In *SAT 2010*, 2010.

[18] K. L. McMillan. An interpolating theorem prover. *TCS*, 345(1), 2005.

[19] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 1997.

[20] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*. ACM, 2007.

[21] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*. ACM, 2005.

[22] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*. ACM, 2006.

[23] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*. ACM, 2011.

[24] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*. ACM, 2010.

[25] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.