Controller Synthesis for Pipelined Circuits Using Uninterpreted Functions

Georg Hofferek and Roderick Bloem
Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Austria
{*georg.hofferek, roderick.bloem*}*@iaik.tugraz.at*

***Abstract*—We present a novel abstraction-based approach to controller synthesis based on the use of a logic with uninterpreted functions, arrays, equality, and limited quantification. Extending the Burch-Dill paradigm for the verification of pipelined processors, we show how to use this logic to synthesize the Boolean control of a pipelined circuit, using a sequential version as the specification. Thus, we tackle the main difficulty in constructing concurrent systems, that of constructing a control that prevents conflicts due to concurrency. At the same time, we avoid the complexity of the datapath, taking advantage of the fact that it must mirror the operations in the sequential variant.**

We start with the controller's specification, an equivalence criterion written in a fragment of second-order logic, stating that for all possible inputs/states, there exist Boolean control values such that the outcome is correct. We show how to decide such formulas by a reduction to propositional logic. From this formula, we can then extract the controller. We show preliminary results for a simple pipelined system.

***Keywords*-controller synthesis; pipelined circuits; uninterpreted functions; array property fragment; equality logic; first-order logic**

I. Introduction

In concurrent systems, we can distinguish the computation of the results from the logic that ensures that concurrent operations are performed properly. In hardware, these roles are taken by the datapath and the control logic. Although the implementation effort for the datapath does not increase much when moving from a sequential to a concurrent setting, the control logic becomes much more intricate. At the same time, datapaths are hard to specify formally, but the requirements for the control logic are often very simple. For instance, the requirement may merely be that the sequential and parallel version of the system produce the same results. Thus, the control logic is a perfect target for formal verification.

This observation underlies the Burch-Dill paradigm of pipelined processor verification [9]. In this paradigm, the datapath of a processor is abstracted away almost completely using uninterpreted functions, and verification focusses on the pipeline control. A non-pipelined version of the same processor (which is significantly simpler to construct) is used as a specification.

This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), and by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23).

In this paper, we propose to extend the *verification* of pipelined processor control to its automatic *synthesis*. Thus, we aim at the automatic construction of that part of the system that is easiest to specify, yet hardest to implement and test. The synthesis of such logic is an instance of controller synthesis [30]. It could be performed using standard bit-level temporal logic synthesis tools, but that approach scales badly with the width and complexity of the datapath, which includes the Arithmetic Logic Unit (ALU).

Following Burch and Dill, we use commutativity of a diagram relating the behavior of the ISA model and the pipelined processor as a sufficient criterion for correctness. The correctness criterion is equivalent to the validity of a formula in the quantifier-free logic of equality, uninterpreted functions, and arrays, where uninterpreted functions are used to model such elements as the ALU or decoding logic, and arrays are used to model memory and the register file. For synthesis, we obtain a formula with limited quantification. We will assume that all control signals are Boolean, which means that decidability is easily established using an expansion that is exponential in the number of control signals.

As the first contribution of this paper, we will show how to decide the formula using reductions that roughly follow those used in the quantifier-free case. Our decision procedure has the benefit that we do not necessarily incur the exponential explosion. As a second contribution, we will show how to automatically extract an implementation of the controller from the formula. We will present two such ways. One is based on Binary Decision Diagrams (BDDs) and cofactors, the other one utilizes interpolating SAT solvers. We show a proof-of-concept implementation (based on BDDs) of our approach that has not yet been optimized for efficiency.

The pipelined circuits considered in this paper are still preliminary and do not consider liveness aspects. Ensuring progress makes the correctness criteria a little more complicated [24]. At the moment, we only consider properties that are captured by a Burch-Dill-style verification condition.

Recently, there has been a proliferation of approaches for temporal logic synthesis [10], [23], [29], [19], [31], [14], [32], [4], [27]. Although quite successful, these approaches appear less applicable to controller synthesis because of the lack of abstraction. In program repair [20], predicate abstraction has been used [16], but uninterpreted functions

978-1-4577-0118-4/11/$26.00 ©2011 IEEE

have not. Synthesis of synchronization skeletons has been described by Clarke and Emerson [11]. More recent work on the issue was done by Vechev et al. [36]. They tackle the problem of inserting synchronization statements into concurrent programs. Like us, they assume that the basic computation has been implemented and only the concurrency aspect must be synthesized. Their approach is based on predicate abstraction, but the standard option of refining the abstraction is complemented by the option of modifying the program to assure atomicity of a sequence of statements. There are of course other approaches to abstraction in game-based settings (e.g., [12]), but to our knowledge none that use uninterpreted functions. Solar-Lezama [33] presents program sketching, an approach where the user only sketches the high-level idea of a program, while low-level details are handled by the synthesis procedure. Srivastava et al. [35] describe how program verification can, in some circumstances, be generalized to program synthesis. Kuncak et al. [22] deal with functional synthesis, which is more data-oriented.

The work which has the most similar goals to ours is, however, by Nurvitadhi et al. [28]. They also present a method to automatically construct pipeline control. Their approach is quite different from ours. They perform data-hazard analysis and resolution, while we start from a logic specification, a Burch-Dill-style verification condition, and perform correct-by-construction synthesis. We present a formal proof that our synthesis results fulfil the initial specification. Besides the fundamental internal differences, there are also some differences from a user's point of view. E.g., our approach does not need manually implemented "read-enable" and "write-enable" signals, nor do we impose the restrictions of [28] on the structure of the write interface of the pipeline. On the other hand, we do require a complete datapath of the pipeline, including potential forwarding paths.

The rest of this paper is organized as follows. In Section II we will revisit necessary preliminaries for our method. Section III introduces how to formulate equivalence criteria for pipelined circuits, illustrated by a running example. In Section IV will formally define these equivalence criteria and show that they are decidable. In Section V we present several reduction steps that reduce equivalence criteria to equivalent propositional formulas. Subsequently, we show in Section VI how to extract functions for control signals from a reduced equivalence criterion. In Section VII we will present first results obtained with our method, and Section VIII concludes the paper.

II. Preliminaries

A. Pipelined Circuits

Pipelining is a design technique that is used to shorten the longest combinational path within a circuit and thus facilitates operation at higher clock frequencies. It is used in particular in many modern microprocessors. Execution of a single instruction is broken down into steps like fetching the instruction from memory, decoding the instruction, fetching necessary operands (if any), actually executing the instruction, and writing back the result. The concrete division into pipeline steps as well as the number of stages may of course vary.

However, pipelining also comes with some difficulties. For example, one stage of the pipeline might need results that have not yet been written back to memory, but are still being processed in some later pipeline stage. In such a case it must be possible to *forward* the data directly from one stage to another. The decision whether to use forwarded data or data read from memory must be taken by the pipeline controller, based on the sequence of instructions and operands that is currently being processed. Sometimes it might also be necessary to *stall* the pipeline, to wait for results to become available.

B. Array Property Fragment

The *Theory of Arrays* T_A is a first-order theory, axiomatized by McCarthy [25]. It allows formal reasoning about arrays, by specifying axioms that describe how reading from and writing to an array works. To establish notation, we will write $A[i]$ to denote the value of array A at index i. By $A\{j \leftarrow x\}$, we denote an array identical to A, except that the value at index j equals x. In other words,

$$A\{j \leftarrow x\}[j] = x \quad \text{and} \quad \forall i \neq j \,.\, A\{j \leftarrow x\}[i] = A[i].$$

This is called the *write axiom*. For convenience, we use the shorthand notation $A = B$ when we mean $\forall i \,.\, A[i] = B[i]$.

Bradley et al. [6] have identified a decidable fragment of T_A which they call the *Array Property Fragment*. In the following, we will focus on properties of arrays with *uninterpreted indices*, as presented in [7].

An array property is a formula of the following 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:

$$\begin{array}{rcl} \text{iguard} & \rightarrow & \text{iguard} \wedge \text{iguard} \mid \text{iguard} \vee \text{iguard} \mid \text{atom} \\ \text{atom} & \rightarrow & \text{var} = \text{var} \mid \mathit{evar} \neq \text{var} \mid \text{var} \neq \mathit{evar} \mid \top \\ \text{var} & \rightarrow & \mathit{evar} \mid \mathit{uvar} \end{array}$$

where uvar is any of the universally quantified variables from $\bar{i}$, and evar is an unquantified variable or a constant.

In the value constraint $G_{\bar{i}}$, universally quantified variables may only occur inside array reads $A[i]$. Furthermore, nested reads like $A[B[i]]$ are disallowed.

Bradley et al. [7] present an algorithm to reduce formulas with array properties to equisatisfiable formulas over the theory of uninterpreted functions. The main step of the

algorithm is the reduction of universal quantification to a finite conjunction over a 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 $evar$ during the parsing of the index guards, and the special term λ (representing *"any other index"*). Bradley et al. prove that it suffices to consider these finitely many indices to prove or disprove satisfiability. More details on the reduction algorithm and the aforementioned proof of correctness can be found in [7].

C. Uninterpreted Functions

Uninterpreted Functions are function symbols within first-order formulas, which are not axiomatized beyond functional consistency [21]. I.e., the only thing that is known about such a function is that it returns the same output value, when given equal input values. More formally, for an n-ary function F the following holds:

$$\forall t_1, \ldots, t_n, t'_1, \ldots, t'_n \,.\, \bigwedge_i (t_i = t'_i) \rightarrow F(t_1, \ldots, t_n) = F(t'_1, \ldots, t'_n).$$

This axiom holds the key to Ackermann's reduction [1] to equality logic. In short, all instances of function calls are replaced with fresh domain variables. The formula is then amended with functional consistency constraints which enforce equality between the fresh variables when there is equality between all the arguments of their respective function calls. A detailed explanation of Ackermann's reduction can be found in [21].

D. Equality Logic

Equality logic is a first-order logic with one special (interpreted) predicate symbol "=", representing equality (with the expected semantics). It allows reasoning about elements of a *domain*, e.g., integers, reals, or (finite or infinite) subsets of them.

Bryant and Velev [8] show how a formula in equality logic can be reduced to one in propositional logic. Their method is based on constructing a non-polar equality graph and inferring *transitivity constraints* from it. These constraints are then added to the *propositional skeleton* of the formula. The propositional skeleton is the propositional formula which is obtained by replacing every equality literal with a fresh Boolean variable. A good introduction to this reduction algorithm can be found in [21].

III. Formulating the Equivalence Criterion

In this section we show how to obtain an equivalence criterion for a pipelined circuit that serves as the formal specification for the pipeline controller. We use a simple example to illustrate the necessary steps. Performing this procedure automatically, when given a graphical or net-list representation of the model(s) is trivial.

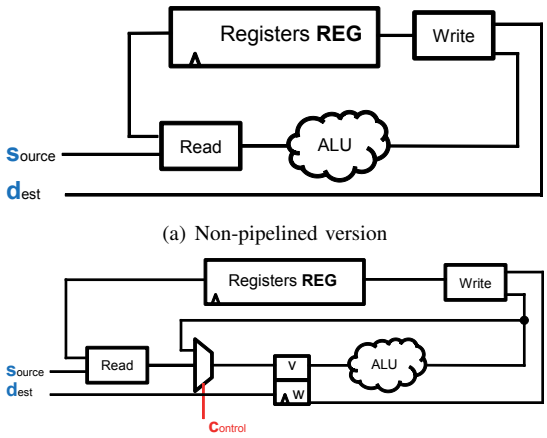

Figure 1. A simple example of a microprocessor-like circuit, in non-pipelined and pipelined version.

Example 1. *In Fig. 1, we present a sketch of a simple microprocessor-like circuit, to which all the examples in this paper refer. Let us first examine the non-pipelined version in Fig. 1(a). The circuit has two inputs s and d, representing a source and a destination address, respectively. The design contains an array of registers (register file) REG. Data words can be read from and written to the register file, using an address to index one specific register. This is symbolized by the* Read *and* Write *blocks in Fig. 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.*

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 approach.

Fig. 1(b) shows a pipelined version of the circuit in Fig. 1(a), with one stage of pipeline registers v and w. In this circuit, 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.

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)

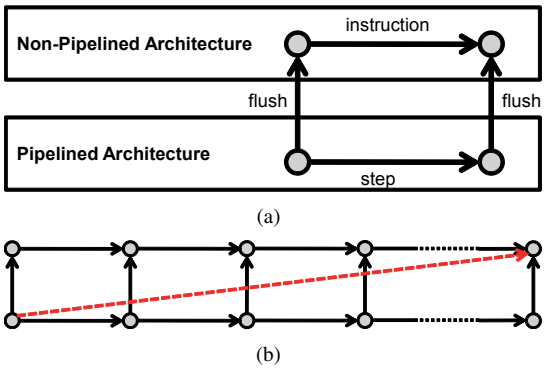

Figure 2. How to show equivalence between pipelined and non-pipelined version, according to the Burch-Dill paradigm [9]

control signal c.

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

Informally speaking, the specification of a pipeline controller is that it should ensure that the pipelined circuit's behavior is (functionally) equivalent to that of a non-pipelined reference implementation. In this section we will show how to formalize this informal criterion. We will follow the Burch-Dill approach [9], which we will briefly recapitulate.

Fig. 2 illustrates what is meant by equivalence between a pipelined and a non-pipelined version of a processor. Suppose we start with an empty pipeline in the initial state and apply a sequence of inputs, resulting in a sequence of states of the circuit (cf. Fig. 2(b), lower line). After that, we perform a flush of the pipeline (we will detail how this works below) to obtain a final state. The flush is indicated by the rightmost upwards arrow in Fig. 2(b). If we apply the same sequence of inputs to the non-pipelined implementation (starting from the same initial state of the register file, of course), we want to obtain a sequence of states which ends in the same final state that we had before (cf. Fig. 2(b), upper line).

To show that it does not matter whether we perform the sequence of instructions on the pipelined or the non-pipelined circuit, and that in both cases we reach the same final state in the upper right corner of Fig. 2(b) (as illustrated by the dashed red line), it suffices to show that flushing the pipeline and performing one step are commutative operations (cf. Fig. 2(a)), for arbitrary steps and states of the pipeline. If commutativity holds in this general case, then, by repeated application of commutativity, we can conclude that equivalence also holds for the entire sequence of inputs. We will thus derive our equivalence criterion from the commutativity outlined by Fig. 2(a).

The basic structure of the equivalence criterion consists of three parts. We assume an (arbitrary) initial state (Fig. 2(a), lower left corner). The first part of the equivalence criterion is a formula, describing the behavior of the circuit from this initial state along the *"flush-instruction"* path (upwards arrow, then rightwards arrow). Analogously, the second part describes the behavior along the *"step-flush"* (rightwards arrow, then upwards arrow). The third part asserts that the two final states resulting from the two other parts are in fact identical.

We will now show how to obtain these three parts. First, we need to discuss how to model the different parts of a circuit in a formula. To model address-based memory (e.g. 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 (e.g. pipeline registers) are modeled by simple variables ranging over an uninterpreted (infinite or sufficiently large[1]) 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. We use primes to denote time steps. I.e., 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 the equivalence criterion, we have to model the flushing of the pipeline from an arbitrary current state. We use an approach resembling the *completion functions* presented by Hosabettu et al. [17]. Instead of actually flushing the pipeline, which would require a sequence of special inputs, we model the effect that completing an unfinished pipeline stage would have on the observable parts of the circuit. After completing one pipeline stage, we consider this stage removed from the circuit and continue completing remaining stages, if any.

Example 2. *For the circuit in Fig. 1(b), completion is achieved by updating $REG[w]$ to the value $ALU(v)$. In our example, there are no more stages to complete, thus flushing this circuit is modeled by the following equation, where the "ci" subscript symbolizes that we are in the* "complete, then instruction" *path of the equivalence diagram (Fig. 2(a)).*

$$REG'_{ci} = REG\{w \leftarrow ALU(v)\} \tag{1}$$

After completing the pipeline, we model one step in the non-pipelined version of the circuit (corresponding to

[1]We will discuss the precise meaning of *sufficiently large* in Section V.

the upper rightwards arrow in Fig. 2(a)). Thus, 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 instruction of the non-pipelined circuit.

Example 3. *One step of the non-pipelined circuit is modeled by the following equation:*

$$REG''_{ci} = REG'_{ci}\{d \leftarrow ALU(REG'_{ci}[s])\} \tag{2}$$

The conjunction of Equations 1 and 2 forms φ_{ci}.

The second part of the equivalence criterion, φ_{sc}, is obtained by modeling the *"step, then complete"* part of the equivalence diagram. This is done similarly to the "ci" part. Since φ_{ci} and φ_{sc} describe how the elements of the circuit are *updated* during execution, we define $\varphi_{upd} := \varphi_{ci} \wedge \varphi_{sc}$.

Example 4. *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 equation (corresponding to the lower rightwards arrow in Fig. 2(a)).*

$$REG'_{sc} = REG\{w \leftarrow ALU(v)\} \wedge (w' = d) \wedge \Big((c \wedge v' = ALU(v)) \vee (\neg c \wedge v' = REG[s])\Big) \tag{3}$$

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

$$REG''_{sc} = REG'_{sc}\{w' \leftarrow v'\} \tag{4}$$

We form the conjunction of Equations 3 and 4 to obtain φ_{sc}.

The third part of the equivalence criterion, φ_{equiv} is supposed to ensure that the states obtained by the update rules of φ_{ci} and φ_{sc} are identical.

Example 5. *In our example, the observable state of the circuit are the values of the register file. Thus, we obtain*

$$\varphi_{equiv} := (REG''_{ci} = REG''_{sc}) \tag{5}$$

The steps so far have more or less followed the standard Burch-Dill approach for verification of pipelined circuits. From here on, we will present our extension of this approach to synthesis. What we want to state is that for any arbitrary initial state, and for any arbitrary inputs, as well as for any arbitrary interpretation of the functions in the circuit, it is possible to set values for the control signals, such that the final states of the ci- and the sc-path are always identical, as long as the update rules are followed.

Example 6. *For the circuit in Fig. 1, the equivalence criterion is given by the following equation:*

$$\forall REG\,.\,\forall ALU\,.\,\forall\, v, w, s, d\,.\,\exists\, c\,.\,\forall REG'_{ci}, REG''_{ci}, REG'_{sc}, REG''_{sc}\,.\,\forall\, v', w'\,.\,\Big(\varphi_{upd} \rightarrow \varphi_{equiv}\Big) \tag{6}$$

There are some noteworthy facts about Equation 6. First of all, it is a closed formula; all variables are bound by a quantifier. Second, the existential quantification is over Boolean control signals only. Only the universal quantification is over elements of a (possibly) infinite domain. Third, due to the quantifier structure, we observe that the control values may depend on the initial state (register file and pipeline registers), inputs, and the concrete interpretation of the functions. However, control values are independent of any "future state" of the circuit. Furthermore, we note that, since there is also quantification over array variables and function symbols, Equation 6 is a second-order formula. However, this will not hinder us to develop a decision procedure for this kind of formulas, due to the special quantifier structure.

We are interested in deciding the *validity* of equivalence criteria such as Eq. 6. In case such an equivalence criterion is not valid, it is not possible to find control values that ensure correctness for all possible states/inputs; the control problem is *unrealizable*. A possible reason for unrealizability is that the datapath does not feature enough options to ensure correct behavior. As a consequence, the number and position of control signals must be known a priori.

Example 7. *If we remove the multiplexer for data forwarding from the pipelined circuit in Fig. 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 control problem becomes unrealizable.*

In situations like the one in Example 7, it is easy to obtain a *counterexample*; i.e., a sequence of inputs for which it is impossible to find control values so that the specification is fulfilled. Such a sequence is helpful to find out what needs to be added to the datapath in order to make the synthesis problem realizable.

In addition to the realizability question, we want to extract from equations like Eq. 6 functions for the control signals, in terms of the inputs, current state, and function interpretation.

IV. Equivalence Criteria and Decidability

In the previous section we have shown, using a simple example, how one can obtain an equivalence criterion which can be viewed as a formal specification for a controller to be synthesized. We will now formally define the class of equivalence criteria that can be handled by our approach, and the fragment of first-order logic on which they are based. Subsequently we will show that they are decidable.

Definition 1. *$\mathcal{L}^{\mathbb{D}}_{AUE}$ is the subset of first-order formulas based on the theories of arrays, uninterpreted functions, and*

equality, which adheres to the following BNF-style grammar.

$$
\begin{array}{rcl}
\text{formula} & \rightarrow & \text{array\_property} \mid \text{term} = \text{term} \mid \\
& & \text{array\_term} = \text{array\_term} \mid \\
& & \texttt{propositional\_var} \mid \\
& & <\textit{Bool. combin. of } \text{formula}> \\
\text{array\_term} & \rightarrow & \texttt{array\_var} \mid \\
& & \texttt{array\_var}\{\text{term} \leftarrow \text{term}\} \\
\text{term} & \rightarrow & \texttt{domain\_var} \mid \texttt{domain\_const} \mid \\
& & \texttt{function\_symbol}(\text{term}*) \mid \\
& & \texttt{array\_var}[\text{term}] \\
\text{array\_property} & \rightarrow & \forall \bar{i} \,.\, \text{iguard}_{\bar{i}} \rightarrow \text{valconstr}_{\bar{i}} \\
\text{iguard}_{\bar{i}} & \rightarrow & \text{iguard}_{\bar{i}} \wedge \text{iguard}_{\bar{i}} \mid \\
& & \text{iguard}_{\bar{i}} \vee \text{iguard}_{\bar{i}} \mid \text{atom}_{\bar{i}} \\
\text{atom}_{\bar{i}} & \rightarrow & \text{true} \mid \text{var}_{\bar{i}} = \text{var}_{\bar{i}} \mid \\
& & \text{evar}_{\bar{i}} \neq \text{var}_{\bar{i}} \mid \text{var}_{\bar{i}} \neq \text{evar}_{\bar{i}} \\
\text{var}_{\bar{i}} & \rightarrow & \text{evar}_{\bar{i}} \mid \text{uvar}_{\bar{i}} \\
\text{uvar}_{\bar{i}} & \rightarrow & <\textit{any } i \in \bar{i}> \\
\text{evar}_{\bar{i}} & \rightarrow & \texttt{domain\_var} <\textit{except any } i \in \bar{i}> \mid \\
& & \texttt{domain\_const} \mid \\
& & \texttt{function\_symbol}(\text{evar}_{\bar{i}}*) \\
\text{valconstr}_{\bar{i}} & \rightarrow & \texttt{array\_var} = \texttt{array\_var} \mid \\
& & \texttt{array\_var}[\text{var}_{\bar{i}}] = \text{evar}_{\bar{i}} \mid \\
& & <\textit{Bool. combin. of } \text{valconstr}_{\bar{i}}>
\end{array}
$$

The symbols `domain_var` *represent variables ranging over* $\mathbb{D}$*, and* `domain_const` *are constants chosen from* $\mathbb{D}$*. The symbols* `array_var` *are array variables, as axiomatized by the theory of (extensional) arrays, and the* `function_symbols` *represent uninterpreted functions.*

Definition 2 (Equivalence Criterion)**.** *Let* $\bar{R}$ *and* $\bar{R}'$ *be finite sets of array variables* $R_1, \ldots, R_i$ *and* $R'_1, \ldots, R'_j$ *respectively, let* $\bar{f}$ *be a finite set of function symbols* $f_1, \ldots, f_k$*, let* $\bar{s}$ *and* $\bar{s}'$ *be finite sets of variables* $s_1, \ldots, s_l$ *and* $s'_1, \ldots, s'_m$ *ranging over* $\mathbb{D}$ *respectively, and let* $\bar{c}$ *be a finite set* $c_1, \ldots, c_n$ *of propositional variables. Let* $\forall \bar{R}$ *be a shorthand notation for* $\forall R_1 \,.\, \forall R_2 \ldots . \forall R_i$*, let* $\exists \bar{c}$ *be shorthand for* $\exists c_1 \,.\, \exists c_2 \ldots . \exists c_m$*, and let* $\forall \bar{R}'$*,* $\forall \bar{s}$*, and* $\forall \bar{s}'$ *be analogous shorthand notations.*

Let φ_{AUE} *be a formula from* $\mathcal{L}^{\mathbb{D}}_{AUE}$*. Let* $vars_A(\varphi_{AUE})$ *be the set of all array variables in* φ_{AUE}*. Similarly, let* $vars_F(\varphi_{AUE})$*,* $vars_D(\varphi_{AUE})$*, and* $vars_P(\varphi_{AUE})$ *be the sets of all function symbols, domain variables, and propositional variables in* φ_{AUE}*, respectively. If* $vars_A(\varphi_{AUE}) = \bar{R} \cup \bar{R}'$*,* $vars_F(\varphi_{AUE}) = \bar{f}$*,* $vars_D(\varphi_{AUE}) = \bar{s} \cup \bar{s}'$*, and* $vars_P(\varphi_{AUE}) = \bar{c}$*, then the (closed) formula*

$$\forall \bar{R} \,.\, \forall \bar{f} \,.\, \forall \bar{s} \,.\, \exists \bar{c} \,.\, \forall \bar{R}' \,.\, \forall \bar{s}' \,.\, \varphi_{AUE}$$

is an equivalence criterion.

For the remainder of this paper, we will use capital letter R for array variables, letter f for function symbols, letters d, i, s, v, w, x, y, z for domain variables, and letters c, e for propositional variables.

Theorem 1. *The validity of an equivalence criterion (as defined in Definition 2) is decidable.*

Proof: The existential quantification in an equivalence criterion is, per definition, over Boolean variables only. Thus, we can rewrite the existential quantification as 2^n disjunctions, for n Boolean variables $c_1, \ldots, c_n$ in $\bar{c}$.

$$
\begin{array}{c}
\forall \bar{R} \,.\, \forall \bar{f} \,.\, \forall \bar{s} \,.\, \exists \bar{c} \,.\, \forall \bar{R}' \,.\, \forall \bar{s}' \,.\varphi_{AUE} \\
\Updownarrow \\
\forall \bar{R} \,.\, \forall \bar{f} \,.\, \forall \bar{s} \,.\, \Big(\forall \bar{R}' \,.\, \forall \bar{s}' \,.\varphi_{AUE} \mid_{\bar{c}=00\ldots00} \vee \\
\qquad \forall \bar{R}' \,.\, \forall \bar{s}' \,.\varphi_{AUE} \mid_{\bar{c}=00\ldots01} \vee \ldots \vee \\
\qquad \forall \bar{R}' \,.\, \forall \bar{s}' \,.\varphi_{AUE} \mid_{\bar{c}=11\ldots11} \Big)
\end{array}
$$

where $\varphi_{AUE} \mid_{\bar{c}=\ldots}$ stands for φ_{AUE} in which the variables $\bar{c}$ have been replaced by true or false, as indicated by the given bit string. We rename all variables in $\bar{R}'$ and $\bar{s}'$ so that variables do not occur outside the scope of their corresponding quantifier. Thus, we can switch the order of quantification and disjunction. We obtain

$$
\begin{array}{l}
\forall \bar{R} \,.\, \forall \bar{f} \,.\, \forall \bar{s} \,.\, \forall \hat{R}' \,.\, \forall \hat{s}' \,.\, \Big(\varphi_{AUE} \mid_{\bar{c}=00\ldots00} \vee \\
\qquad \varphi_{AUE} \mid_{\bar{c}=00\ldots01} \vee \ldots \vee \\
\qquad \varphi_{AUE} \mid_{\bar{c}=11\ldots11} \Big)
\end{array} \tag{7}
$$

Clearly, Equation 7 is valid if and only if

$$\neg \varphi_{AUE} \mid_{\bar{c}=00\ldots00} \wedge \ldots \wedge \neg \varphi_{AUE} \mid_{\bar{c}=11\ldots11} \tag{8}$$

is unsatisfiable.

Note that Equation 8 is in $\mathcal{L}^{\mathbb{D}}_{AUE}$, for which satisfiability is decidable [21]. ■

V. Reduction Steps

In order to be able to extract functions for control values, we perform three validity-preserving reduction steps on the equivalence criterion, in order to obtain an equivalent propositional formula. The first reduction [7] will remove arrays and reduce the equivalence criterion to a formula with uninterpreted functions and equalities only. Second, we use Ackermann's reduction [1] to remove uninterpreted functions and reduce the formula to one with equalities only. Third, we further reduce it to propositional logic, using the method presented in [8]. Finally, we show how to extract functions for control values from the reduced formula, using

a cofactor approach. In the subsequent sections, we will present these steps in more detail, and provide a formal proof that each of the reduction steps is validity preserving.

A. 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: Non-Boolean universal quantification, followed by Boolean existential quantification, followed again by non-Boolean universal quantification. Let the two formulas be $\forall \bar{a} \,.\, \exists \bar{b} \,.\, \forall \bar{c} \,.\, \varphi$ and $\forall \bar{x} \,.\, \exists \bar{y} \,.\, \forall \bar{z} \,.\, \psi$. The proof that validity of the first formula implies validity of the second proceeds as sketched by the following equation.

$$\begin{array}{cccc} \forall \bar{a} \,. & \exists \bar{b} \,. & \forall \bar{c} \,. & \varphi \\ \alpha \uparrow & \beta \downarrow & \gamma \uparrow & \\ \forall \bar{x} \,. & \exists \bar{y} \,. & \forall \bar{z} \,. & \psi \end{array} \tag{9}$$

We start with an arbitrary interpretation $\bar{\mathsf{x}}$ for $\bar{x}$ in ψ. Next, we map these values to corresponding values $\bar{\mathsf{a}}$ for $\bar{a}$ in φ, according to a mapping $\alpha : \bar{\mathsf{x}} \mapsto \bar{\mathsf{a}}$. We then use the assumption of the validity of the first formula to find values $\bar{\mathsf{b}}$ for $\bar{b}$ such that for any arbitrary interpretation $\bar{\mathsf{c}}$ for $\bar{c}$ we have $\{\bar{\mathsf{a}}, \bar{\mathsf{b}}, \bar{\mathsf{c}}\} \models \varphi$. We use another mapping $\beta : \bar{\mathsf{b}} \mapsto \bar{\mathsf{y}}$ to find values $\bar{\mathsf{y}}$. Now we still need to prove that $\forall \bar{z} \,.\, \psi[\bar{\mathsf{x}}/\bar{x},\ \bar{\mathsf{y}}/\bar{y}]$ is valid. To do so, we arbitrarily choose an interpretation $\bar{\mathsf{z}}$ for $\bar{z}$ and use a mapping $\gamma : \bar{\mathsf{z}} \mapsto \bar{\mathsf{c}}$ to find corresponding values $\bar{\mathsf{c}}$. From the assumption of validity of the first formula we know that $\{\bar{\mathsf{a}}, \bar{\mathsf{b}}, \bar{\mathsf{c}}\} \models \varphi$. What remains to be shown is that this implies $\{\bar{\mathsf{x}}, \bar{\mathsf{y}}, \bar{\mathsf{z}}\} \models \psi$. 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 $\bar{b}$ and $\bar{y}$ 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 equivalence criterion. Within each proof, we will only present the mappings α and γ, and we will show that $\varphi[\bar{\mathsf{a}}/\bar{a},\ \bar{\mathsf{b}}/\bar{b},\ \bar{\mathsf{c}}/\bar{c}]$ implies $\psi[\bar{\mathsf{x}}/\bar{x},\ \bar{\mathsf{y}}/\bar{y},\ \bar{\mathsf{z}}/\bar{z}]$.

Some of the proofs will require that the domain $\mathbb{D}$ 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 $|\mathbb{D}| \geq \#var(\varphi)$, where $\#var(\varphi)$ is the number of first-order variables and constants in φ. Note that in particular any infinite domain is obviously sufficiently large with respect to any formula.

B. Reduction to Uninterpreted Functions and Equality

We start with an equivalence criterion, as defined by Definition 2. For the reduction to uninterpreted functions and equality we proceed as in [7]. We take the part φ_{AUE} (i.e., the part without the quantifier prefix) of the equivalence criterion and apply several transformations to it. First, all array writes are removed by introducing new variables and applying the write axiom.

Example 8. *Consider the term $REG\{w \leftarrow ALU(v)\}$ from Example 2. A new variable REG' is introduced, the term is replaced by REG', and the conjunct*

$$REG'[w] = ALU(v) \wedge \forall i \,.\, i \neq w \rightarrow REG'[i] = REG[i]$$

is added.

Lemma 1. *Let φ be an equivalence criterion. Let φ' be the equivalence criterion obtained by removing all array write expressions according to the write axiom, as outlined above. Then φ is valid if and only if φ' is valid.*

For the remainder of this paper, we will only consider equivalence criteria without array-write expressions. For equivalence criteria with array-write expressions we apply Lemma 1 to obtain one without.

Next, we find the index set $\mathcal{I}$, as outlined in Section II-B. Then, we replace all array reads by uninterpreted function instances. Similar to the short-hand notation introduced in Definition 2, we will write $\bar{f}_R$ and $\bar{f}_{R'}$ for the function symbols corresponding to the array variables in $\bar{R}$ and $\bar{R}'$ respectively. Finally, all universal quantifications over array indices are replaced by finite conjunctions over the index set $\mathcal{I}$.

Definition 3. *Let φ_{AUE} be a formula from $\mathcal{L}^{\mathbb{D}}_{AUE}$. $no\_array(\varphi_{AUE})$ is the first-order formula over the theories of uninterpreted functions and equality obtained from φ_{AUE} after applying the aforementioned transformations.*

Example 9. *Consider the equivalence criterion from Example 6. 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. E.g., the term $\forall i \,.\, i \neq w \rightarrow REG'[i] = REG[i]$ from Example 8 is replaced by*

$$\bigwedge_{i \in \mathcal{I}} i \neq w \rightarrow REG'[i] = REG[i]. \tag{10}$$

Next, array reads are replaced by function calls. E.g., $REG'_{ci}[w]$ becomes $REG'_{ci}(w)$. Note that for simplicity and readability we name these functions exactly the same as the corresponding arrays. The distinction is made by using square brackets $[\cdot]$ with arrays, and parenthesis $(\cdot)$ with functions.

Furthermore, we construct φ_λ, which we will call the λ-*constraints*, in the following way:

$$\varphi_\lambda := \bigwedge_{i \in \mathcal{I} \setminus \{\lambda\}} i \neq \lambda \tag{11}$$

Theorem 2 (Reduction to Uninterpreted Functions and Equality)**.** *For a sufficiently large domain, the equivalence criterion*

$$\forall \bar{R} \,.\, \forall \bar{f} \,.\, \forall \bar{s} \,.\, \exists \bar{c} \,.\, \forall \bar{R}' \,.\, \forall \bar{s}' \,.\, \varphi_{AUE}$$

is valid if and only if the formula

$$\forall \bar{f} \,.\, \forall \bar{f}_R \,.\, \forall \bar{s} \,.\, \exists \bar{c} \,.\, \forall \bar{f}_{R'} \,.\, \forall \bar{s}' \,.\, \forall \lambda \,.(\varphi_\lambda \rightarrow no\_array(\varphi_{AUE})) \tag{12}$$

is valid.

Proof: "⇒": We assume validity of the equivalence criterion, and proof validity of Equation 12. Let $\bar{\mathsf{f}}$, $\bar{\mathsf{f}}_\mathsf{R}$, $\bar{\mathsf{f}}_{\mathsf{R}'}$, $\bar{\mathsf{s}}$, $\bar{\mathsf{s}}'$, and Λ be arbitrary interpretations for $\bar{f}$, $\bar{f}_R$, $\bar{f}_{R'}$, $\bar{s}$, $\bar{s}'$, and λ in Equation 12 respectively. Let α be a mapping from an interpretation of function symbols in $\bar{\mathsf{f}}_\mathsf{R}$ to an interpretation of array variables in $\bar{\mathsf{R}}$ as follows. For all $\mathsf{f}_\mathsf{R} \in \bar{\mathsf{f}}_\mathsf{R}$ let $\alpha(\mathsf{f}_\mathsf{R})$ be an interpretation R for an array variable in $\bar{R}$ such that $\forall i \,.\, \mathsf{f}_\mathsf{R}(i) = \mathsf{R}[i]$. Let γ be a mapping from $\bar{\mathsf{f}}_{\mathsf{R}'}$ to $\bar{\mathsf{R}}'$, defined analogously to α. Let $\mathsf{R} = \alpha(\mathsf{f}_\mathsf{R})$ for each $\mathsf{R} \in \bar{\mathsf{R}}$ and $\mathsf{R}' = \gamma(\mathsf{f}_{\mathsf{R}'})$ for each $\mathsf{R}' \in \bar{\mathsf{R}}'$. Let $\bar{\mathsf{c}}$ be an interpretation for $\bar{c}$ such that $\forall \bar{R}' \,.\, \forall \bar{s}' \,.\, \varphi_{AUE}[\bar{\mathsf{R}}/\bar{R},\ \bar{\mathsf{f}}/\bar{f},\ \bar{\mathsf{s}}/\bar{s},\ \bar{\mathsf{c}}/\bar{c}]$.

We have to show that $\{\bar{\mathsf{R}}, \bar{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{R}}', \bar{\mathsf{s}}'\} \models \varphi_{AUE}$ implies that $\{\bar{\mathsf{f}}_\mathsf{R}, \bar{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{f}}_{\mathsf{R}'}, \bar{\mathsf{s}}', \Lambda\} \models (\varphi_\lambda \rightarrow no\_array(\varphi_{AUE}))$. It is easy to see that this is the case. φ_{AUE} features universal quantification over indices, where $no\_array(\varphi_{AUE})$ 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 $(\varphi_\lambda \rightarrow no\_array(\varphi_{AUE}))$ is satisfied by $\{\bar{\mathsf{f}}_\mathsf{R}, \bar{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{f}}_{\mathsf{R}'}, \bar{\mathsf{s}}', \Lambda\}$, it is irrelevant whether or not $\{\bar{\mathsf{f}}_\mathsf{R}, \bar{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{f}}_{\mathsf{R}'}, \bar{\mathsf{s}}', \Lambda\} \models \varphi_\lambda$. This concludes the proof in "⇒" direction.

"⇐": Let $\bar{\mathsf{R}}$, $\bar{\mathsf{f}}$, $\bar{\mathsf{s}}$, $\bar{\mathsf{R}}'$, and $\bar{\mathsf{s}}'$ be arbitrary interpretations for $\bar{R}$, $\bar{f}$, $\bar{s}$, $\bar{R}'$, and $\bar{s}'$ in the equivalence criterion. Let α and γ be mappings inverse to those of the "⇒" case. For each $\mathsf{f}_\mathsf{R} \in \bar{\mathsf{f}}_\mathsf{R}$ and each $\mathsf{f}_{\mathsf{R}'} \in \bar{\mathsf{f}}_{\mathsf{R}'}$ let $\mathsf{f}_\mathsf{R} = \alpha(\mathsf{R})$ and $\mathsf{f}_{\mathsf{R}'} = \gamma(\mathsf{R}')$. Let $\bar{\mathsf{c}}$ be an interpretation for $\bar{c}$ such that $\forall \bar{f}_{R'} \,.\, \forall \bar{s}' \,.\, \forall \lambda \,.(\varphi_\lambda \rightarrow no\_array(\varphi_{AUE})[\bar{\mathsf{f}}_\mathsf{R}/\bar{f}_R,\ \bar{\mathsf{f}}/\bar{f},\ \bar{\mathsf{s}}/\bar{s},\ \bar{\mathsf{c}}/\bar{c}]$.

For a sufficiently large domain it is always possible to choose Λ in such a way that $\mathcal{M} = \{\bar{\mathsf{f}}, \bar{\mathsf{f}}_\mathsf{R}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{f}}_{\mathsf{R}'}, \bar{\mathsf{s}}', \Lambda\} \models \varphi_\lambda$. The assumption of validity of Eq. 12 implies that in this case $\mathcal{M} \models no\_array(\varphi_{AUE})$. Bradley et al. [7] have proven that any model $\mathcal{M}$ that satisfies φ_λ and $no\_array(\varphi_{AUE})$ also satisfies φ_{AUE} (when applying the proper mapping between function symbols and array variables). ■

C. Reduction to Equality

We use Ackermann's reduction [1] to reduce the formula φ_{UE} over the theories of uninterpreted functions and equality, which we obtained in the previous reduction step, to pure equality logic [21]. To ease presentation, we will only consider unary functions. Note that the extension of Ackermann's reduction to n-ary functions is straightforward [21].

Ackermann's reduction replaces each instance $f(x)$ of function symbol f with parameter x with a new domain variable d_f^x. Then, *functional constraints* φ_{FC} are constructed in the following way:

$$\varphi_{FC} := \bigwedge_{f \in \bar{f}} \bigwedge_{x,y \in args(f)} \Big(x = y \rightarrow d_f^x = d_f^y \Big) \tag{13}$$

where $\bar{f}$ is the set of function symbols occurring in φ_{UE}, and $args(f)$ are all the terms to which f is applied in φ_{UE}.

Example 10. *Consider Equation 10 in Example 9. After replacing the array reads with function calls, this equation has the following function instances:* $REG(s), REG(w), \ldots$. *For these instances new domain variables* $d^s_{REG}, d^w_{REG}, \ldots$ *are introduced. One of the conjuncts of* φ_{FC} *is then*

$$(s = w) \rightarrow \Big(d^s_{REG} = d^w_{REG} \Big).$$

Definition 4. *$no\_func(\varphi_{UE})$ is the first-order formula over the theory of equality obtained from φ_{UE} by replacing all function instances with fresh domain variables, as outlined above.*

Theorem 3 (Reduction to Equality Logic)**.** *Let φ_{UE} be a quantifier-free formula over the theories of uninterpreted functions and equality. Then the formula*

$$\forall \bar{f} \,.\, \forall \bar{s} \,.\, \exists \bar{c} \,.\, \forall \bar{f}' \,.\, \forall \bar{s}' \,.\, \varphi_{UE} \tag{14}$$

is valid if and only if the formula

$$\forall \bar{d}_f^x \,.\, \forall \bar{s} \,.\, \exists \bar{c} \,.\, \forall \bar{d}_{f'}^x \,.\, \forall \bar{s}' \,.(\varphi_{FC} \rightarrow no\_func(\varphi_{UE})) \tag{15}$$

is valid.[2]

Proof: "⇒": We assume validity of Equation 14 and prove validity of Equation 15. Let $\bar{\mathsf{d}}^\mathsf{x}_\mathsf{f}$, $\bar{\mathsf{d}}^\mathsf{x}_{\mathsf{f}'}$, $\bar{\mathsf{s}}$, and $\bar{\mathsf{s}}'$ be arbitrary interpretations for $\bar{d}_f^x$, $\bar{d}_{f'}^x$, $\bar{s}$, and $\bar{s}'$, in Equation 15 respectively. Let $D = \bar{\mathsf{d}}^\mathsf{x}_\mathsf{f} \cup \bar{\mathsf{d}}^\mathsf{x}_{\mathsf{f}'} \cup \bar{\mathsf{s}} \cup \bar{\mathsf{s}}'$. Let α be a mapping from an interpretation D for domain variables to an interpretation $\bar{\mathsf{f}}$ for function symbols in $\bar{f}$, such that for $\bar{\mathsf{f}} = \alpha(D)$ each $\mathsf{f} \in \bar{\mathsf{f}}$ satisfies $\forall \mathsf{x} \in D \,.\, \forall \mathsf{f} \in \bar{\mathsf{f}} \,.\, \mathsf{f}(\mathsf{x}) = \mathsf{d}^\mathsf{x}_\mathsf{f}$. In case such interpretations $\bar{\mathsf{f}}$ do not exist due to functional inconsistencies, α returns an arbitrary interpretation $\bar{\mathsf{f}}$. Let γ be a mapping from D to an interpretation $\bar{\mathsf{f}}'$ for function symbols in $\bar{f}'$, defined analogously to α. Let $\bar{\mathsf{f}} = \alpha(D)$ and $\bar{\mathsf{f}}' = \gamma(D)$. Let $\bar{\mathsf{c}}$ be an interpretation for $\bar{c}$ such that $\forall \bar{f}' \,.\, \forall \bar{s}' \,.\, \varphi_{UE}[\bar{\mathsf{f}}/\bar{f},\ \bar{\mathsf{s}}/\bar{s},\ \bar{\mathsf{c}}/\bar{c}]$.

We have to show that $\{\bar{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{f}}', \bar{\mathsf{s}}'\} \models \varphi_{UE}$ implies that $\{\bar{\mathsf{d}}^\mathsf{x}_\mathsf{f}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{d}}^\mathsf{x}_{\mathsf{f}'}, \bar{\mathsf{s}}'\} \models (\varphi_{FC} \rightarrow no\_func(\varphi_{UE}))$.

Models that do not satisfy φ_{FC} trivially satisfy $(\varphi_{FC} \rightarrow no\_func(\varphi_{UE}))$. We only need to consider models that do satisfy φ_{FC}. Thus, for any function instance $f(x)$ in

[2]To increase readability and ease notation, we have subsumed all terms of the same "type" that appear under the same quantifier. I.e., instead of writing $\forall \bar{f}, \bar{f}_R$, we simply write $\forall \bar{f}$.

Equation 14, we have $\mathsf{f}(\mathsf{x}) = \mathsf{d}^{\mathsf{x}}_{\mathsf{f}}$, where $\mathsf{d}^{\mathsf{x}}_{\mathsf{f}}$ is the interpretation for the variable d^x_f with which $f(x)$ has been replaced according to the definition of $no\_func(\varphi_{UE})$. Thus, we conclude that if $\{\bar{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{f}}', \bar{\mathsf{s}}'\} \models \varphi_{UE}$, we have $\{\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}'}, \bar{\mathsf{s}}'\} \models no\_func(\varphi_{UE})$. This concludes the proof in "$\Rightarrow$" direction.

"$\Leftarrow$": Let $\bar{\mathsf{f}}$, $\bar{\mathsf{s}}$, $\bar{\mathsf{f}}'$, and $\bar{\mathsf{s}}'$ be arbitrary interpretations for $\bar{f}$, $\bar{s}$, $\bar{f}'$, and $\bar{s}'$ in Equation 14. Let α be a mapping from an interpretation for function symbols to an interpretation for domain variables such that $\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}} = \alpha(\bar{\mathsf{f}})$ satisfies $\mathsf{d}^{\mathsf{x}}_{\mathsf{f}} = \mathsf{f}(\mathsf{x})$ for all $\mathsf{d}^{\mathsf{x}}_{\mathsf{f}} \in \bar{\mathsf{d}}^{\mathsf{x}}_{\bar{\mathsf{f}}}$. Let γ be a mapping analogous to α. Let $\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}} = \alpha(\bar{\mathsf{f}})$ and $\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}'} = \gamma(\bar{\mathsf{f}}')$. Let $\bar{\mathsf{c}}$ be an interpretation for $\bar{c}$ such that $\forall \bar{d}^x_{f'} \,.\, \forall \bar{s}' \,.(\varphi_{FC} \rightarrow no\_func(\varphi_{UE}))[\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}}/\bar{d}^{\mathsf{x}}_{\mathsf{f}}, \ \bar{\mathsf{s}}/\bar{s}, \ \bar{\mathsf{c}}/\bar{c}]$.

Due to the definition of α and γ, we know that $\{\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}'}, \bar{\mathsf{s}}'\} \models \varphi_{FC}$. Due to the assumption of validity of Equation 15, we therefore know that $\{\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}'}, \bar{\mathsf{s}}'\} \models no\_func(\varphi_{UE})$.

We have to show that $\{\bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{d}}^{\mathsf{x}}_{\mathsf{f}'}, \bar{\mathsf{s}}'\} \models no\_func(\varphi_{UE})$ implies $\{\bar{\mathsf{f}}, \bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{f}}', \bar{\mathsf{s}}'\} \models \varphi_{UE}$. This follows trivially from the definitions of $no\_func(\varphi_{UE})$, α, and γ. ■

D. Reduction to Propositional Logic

The reduction to propositional logic is based on the method presented by Bryant and Velev [8]. It proceeds as follows. First, a non-polar equality graph $\mathcal{G}$ for the formula in question is constructed. This graph has a node for every first-order variable, and an edge for every equality atom between such variables present in the formula. The polarity of the equality atom, whether it is e.g. $v = w$ or $v \neq w$, is disregarded. The graph will be used to create *transitivity constraints* φ_{TC}. First, it is made *chordal*. I.e., edges are added to the graph so that it does not contain any chord-free cycles with length greater than 3. Next, each edge in the resulting graph is labeled with a fresh propositional variable, corresponding to the two nodes which are connected by the edge. I.e., an edge between nodes v an w will be labeled with propositional variable $e_{v,w}$. We assume a predefined order on domain variables, in order to avoid introducing duplicate variables $e_{v,w}, e_{w.v}$. The transitivity constraints are then constructed as follows, where $\triangle(\mathcal{G})$ is the set of all triangles in $\mathcal{G}$.

$$\varphi_{TC} \ := \bigwedge_{(x,y,z)\in\triangle(\mathcal{G})} \Big(\begin{array}{l} (e_{x,y} \wedge e_{y,z} \rightarrow e_{x,z}) \wedge \\ (e_{x,y} \wedge e_{x,z} \rightarrow e_{y,z}) \wedge \\ (e_{y,z} \wedge e_{x,z} \rightarrow e_{x,y}) \end{array} \Big) \quad (16)$$

Bryant and Velev [8] prove that for a chordal graph it suffices to consider the transitivity constraints over triangles only.

Definition 5. *The propositional skeleton $skel(\varphi_E)$ of an equality logic formula φ_E is the propositional formula obtained by replacing each equality atom of the form $v = w$ in φ_E with a corresponding (fresh) propositional variable $e_{v,w}$.*

Theorem 4 (Reduction to Propositional Logic)**.** *For a sufficiently large domain, the formula*

$$\forall \bar{s} \,.\, \exists \bar{c} \,.\, \forall \bar{s}' \,.\, \varphi_E \quad (17)$$

is valid if and only if the formula

$$\forall \bar{e}_{s,s} \,.\, \exists \bar{c} \,.\, \forall \bar{e}_{s,s'} \,.\, \forall \bar{e}_{s',s'} \,.\, \Big(\varphi_{TC} \rightarrow skel(\varphi_E) \Big) \quad (18)$$

is valid, where $\bar{e}_{s,s}$, $\bar{e}_{s,s'}$, and $\bar{e}_{s',s'}$ are sets of propositional variables, corresponding to equalities between first-order variables from the sets $\bar{s}$ and $\bar{s}$, $\bar{s}$ and $\bar{s}'$, and $\bar{s}'$ and $\bar{s}'$, respectively.

Proof: "$\Rightarrow$": Let $\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}$, $\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'}$, and $\bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}$ be arbitrary interpretations for $\bar{e}_{s,s}$, $\bar{e}_{s,s'}$, and $\bar{e}_{s',s'}$ in Equation 18 respectively. Let $E = \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}} \cup \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'} \cup \bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}$. Let α be a mapping from an interpretation E to an interpretation for domain variables in $\bar{s}$ as follows. Let $\bar{\mathsf{s}} = \alpha(E)$ such that for all $\mathsf{s}_1, \mathsf{s}_2 \in \bar{\mathsf{s}}$ we have that $\mathsf{s}_1 = \mathsf{s}_2$ if and only if $\mathsf{e}_{\mathsf{s}_1,\mathsf{s}_2} = \mathsf{true}$. In case such interpretations $\bar{\mathsf{s}}$ 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 $\bar{s}'$ defined analogously to α. Let $\bar{\mathsf{s}} = \alpha(E)$ and $\bar{\mathsf{s}}' = \gamma(E)$. Let $\bar{\mathsf{c}}$ be an interpretation for $\bar{c}$ such that $\forall \bar{s}' \,.\, \varphi_E[\bar{\mathsf{s}}/\bar{s}, \ \bar{\mathsf{c}}/\bar{c}]$.

We have to show that $\{\bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{s}}'\} \models \varphi_E$ implies that $\{\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'}, \bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}\} \models (\varphi_{TC} \rightarrow skel(\varphi_E))$.

Models that do not satisfy φ_{TC} trivially satisfy $(\varphi_{TC} \rightarrow skel(\varphi_E))$. We only need to consider models that do satisfy φ_{TC}. Thus, for any equality atom $s_1 = s_2$ in Equation 17, we have $(\mathsf{s}_1 = \mathsf{s}_2) \Leftrightarrow \mathsf{e}_{\mathsf{s}_1,\mathsf{s}_2}$. Due to the definition of $skel(\varphi_E)$, we conclude that if $\{\bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{s}}'\} \models \varphi_E$, we have $\{\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'}, \bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}\} \models (\varphi_{TC} \rightarrow skel(\varphi_E))$. This concludes the proof in "$\Rightarrow$" direction.

"$\Leftarrow$": Let $\bar{\mathsf{s}}$, and $\bar{\mathsf{s}}'$ be arbitrary interpretations for $\bar{s}$ and $\bar{s}'$ in Equation 17. Let α be a mapping from an interpretation for domain variables in $\bar{s}$ to an interpretation for propositional variables $\bar{e}_{s,s}$ such that $\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}} = \alpha(\bar{\mathsf{s}})$ satisfies $\mathsf{e}_{\mathsf{s}_1,\mathsf{s}_2} \Leftrightarrow (\mathsf{s}_1 = \mathsf{s}_2)$ for each $\mathsf{e}_{\mathsf{s}_1,\mathsf{s}_2} \in \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}$. Let γ be a mapping from an interpretation for domain variables in $\bar{s} \cup \bar{s}'$ to an interpretation for propositional variables $\bar{e}_{s,s'}$ and $\bar{e}_{s',s'}$, analogous to α. Let $\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}} = \alpha(\bar{\mathsf{s}})$ and $(\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'}, \bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}) = \gamma(\bar{\mathsf{s}} \cup \bar{\mathsf{s}}')$. Let $\bar{\mathsf{c}}$ be an interpretation for $\bar{c}$ such that $\forall \bar{e}_{s,s'} \,.\, \forall \bar{e}_{s',s'} \,.(\varphi_{TC} \rightarrow skel(\varphi_E))[\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}/\bar{e}_{s,s}, \ \bar{\mathsf{c}}/\bar{c}]$.

Due to the definition of α and γ, we know that $\{\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'}, \bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}\} \models \varphi_{TC}$. Due to the assumption of validity of Equation 18, we therefore know that $\{\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'}, \bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}\} \models skel(\varphi_{UE})$.

We have to show that $\{\bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{e}}_{\mathsf{s},\mathsf{s}'}, \bar{\mathsf{e}}_{\mathsf{s}',\mathsf{s}'}\} \models (\varphi_{TC} \rightarrow skel(\varphi_E))$ implies that $\{\bar{\mathsf{s}}, \bar{\mathsf{c}}, \bar{\mathsf{s}}'\} \models \varphi_E$. This follows trivially from the definitions of $skel(\varphi_E)$, α, and γ. ■

VI. Extracting Control Functions

In the previous sections we have shown how to transform an equivalence criterion into an equivalent propositional formula $\forall \bar{e}_{s,s} \,.\, \exists \bar{c} \,.\, \forall \bar{e}_{s,s'} \,.\, \forall \bar{e}_{s',s'} \,.\, \varphi_{prop}$. Now we want to com-

pute functions (in terms of the variables in $\bar{e}_{s,s}$) for the control signals in $\bar{c}$. This can be done by a QBF solver, capable of computing certificates, e.g. sKizzo [3]. Another possibility is to use Binary Decision Diagrams (BDDs). We compute a BDD for φ_{prop}, and subsequently perform the inner universal quantifications to obtain $\tilde{\varphi}_{prop} := \forall \bar{e}_{s,s'} \,.\, \forall \bar{e}_{s',s'} \,.\, \varphi_{prop}$. Although in the worst case this might blow up the size of the BDD exponentially (w.r.t. $|\bar{e}_{s,s'}| + |\bar{e}_{s',s'}|$), the average case may be much more space-efficient.

The formula $\tilde{\varphi}_{prop}$ can be viewed as the characteristic function of a multi-output Boolean relation, with inputs $\bar{e}_{s,s}$ and outputs $\bar{c}$. An algorithm of how to compute functions compatible with a given relation has been presented in [37]. (See also [5], [18].) 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 $\varphi_{\neg c_i}$, 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:

$$ON = \quad \varphi_{c_i} \quad \wedge \quad \neg\varphi_{\neg c_i} \tag{19}$$

$$OFF = \quad \neg\varphi_{c_i} \quad \wedge \quad \varphi_{\neg c_i} \tag{20}$$

$$DC = \quad \varphi_{c_i} \quad \wedge \quad \varphi_{\neg c_i} \tag{21}$$

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 $\tilde{\varphi}_{prop}$. 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 each variable e_{s_1,s_2} we build a comparator for the terms s_1, s_2 in the circuit. The outputs of all such comparators are then used as inputs for the control functions f_{c_i}. I.e., the control signals c_i are Boolean combinations of the comparator outputs.

Example 11. *For the circuit in Fig. 1(b), one possible solution is $c \Leftrightarrow (s = w)$. Thus, we insert a comparator into the circuit, 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.*

An alternative to the aforementioned cofactor-based approach is to employ the relation determinization technique based on propositional interpolation which has been described by Jiang et al. [18].

VII. Results

A. Complexity Discussion

We briefly discuss the computational complexity of the reduction steps. Let n be the size of the original equivalence criterion. Consider the reduction to uninterpreted functions and equality (Section V-B). Clearly, the size of the λ-constraints is bound by $\mathcal{O}(n)$. Moreover, universal quantifications are replaced by $\mathcal{O}(n)$ conjuncts. Thus, the total size of the reduced formula is $\mathcal{O}(n^2)$. The reduction to pure equality logic (Section V-C) 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 V-D) causes at most a cubic increase in the number of variables and the formula's size [8]. Concerning computation time, all steps so far can be done in polynomial time.

Computing the quantifications is (in the worst case) exponential in the number of variables, concerning computation time and resulting formula size. However, this step does not introduce any new variables. The last step, computing the control functions, is worst-case exponential (due to the quantifications).

B. Proof-of-Concept

We have done a proof-of-concept implementation of our method, based on the example presented in Section III. Using a Python script and the CUDD library [34], we created a BDD for the equivalence criterion (reduced to propositional logic). Using the cofactor approach described in Section VI, we computed the interval for the solution. Both interval boundaries were non-trivial, i.e., neither was the lower bound equal to `false`, nor was the upper bound equal to `true`. We also checked that the expected solution $c \Leftrightarrow (s = w)$ is contained in the interval.

We note, however, that BDDs seem to be a suboptimal data structure for this kind of problem. The BDD for transitivity constraints can become exponentially large [8], irrespective of the variable ordering. In our experiments most of the computation time was indeed spent on computing a BDD for the transitivity constraints. Without dynamic reordering, the program quickly runs out of memory; with dynamic reordering enabled, most of the computation time is spent for reordering. In our first experiment, with dynamic reordering enabled, it took approximately 14 hours to compute the BDD for $\tilde{\varphi}_{prop}$. 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 Boolean variables, after the reduction steps.

We have also done a validity analysis of the equivalence criterion of our example using the z3 SMT solver [13]. We modeled the equivalence criterion (without the quantifier prefix) in SMT-LIBv2 [2] format, transformed the validity problem into a satisfiability problem, as outlined in Section IV, and checked it with z3. We also checked whether the expected solution would be a valid solution by asserting

that $c \Leftrightarrow (s = w)$, negating the whole formula and checking for satisfiability (expecting "unsatisfiable" as result). All the checks we performed with z3 terminated within negligible time and all produced the expected results.

VIII. Conclusions and Future Work

We have presented a novel method for correct-by-construction controller synthesis, as well as a proof of its correctness. Complex (datapath) elements are efficiently abstracted by the use of uninterpreted functions. Thus, our method is able to tackle problems that would be prohibitively large, when modeled directly on the Boolean level. To the best of our knowledge this is the first time that uninterpreted functions have been used as the means for abstraction in synthesis. The primary target of our approach is control logic for pipelined circuits, however the method is applicable to all controller synthesis problems, where a suitable correctness criterion can be stated within the class of formulas that we have described. Concerning pipelines, our method covers safety that can be expressed by a Burch-Dill-style [9] verification condition. Extensions to liveness (e.g. that the pipeline does not stall infinitely often) remain for future work.

We have done a proof-of-concept implementation of our method, based on the running example presented in this paper. In the future, we plan to extend this proof-of-concept implementation to a fully functional synthesis tool based on our approach. Furthermore we plan to research several possible improvements. For example, generating the transitivity constraints seems to be a bottleneck of our approach, in particular when working with BDDs. An approach, with a refinement loop were transitivity constraints are added "on demand", i.e., only when they are needed to prove equivalence, could alleviate this problem. Moreover, since BDDs seem to be a suboptimal data structure for this kind of problem, we currently investigate how QBF-solving techniques and interpolation in equality logic with uninterpreted functions [15], [26] can be used as alternative methods to compute control functions. Preliminary experiments suggest that interpolation in equality logic with uninterpreted functions is a very efficient method for finding control functions. A thorough analysis, based on an implementation currently under development, will be done in future work.

References

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

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proc. 8th Int. Workshop on Satisfiability Modulo Theories*, 2010.

[3] M. Benedetti. sKizzo: a suite to evaluate and certify QBFs. In *Proc. of 20th Int. Conf. on Automated Deduction (CADE05)*, 2005. LNCS 3632.

[4] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY — a new requirements analysis tool with synthesis. In *Proc. Computer Aided Verification*, pages 425–429, 2010. LNCS 6174.

[5] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware form PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, 2007. Electronic Notes in Theoretical Computer Science http://www.entcs.org/.

[6] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006.

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

[8] R. E. Bryant and M. Velev. Boolean satisfiability with transitivity constraints. In E. A. Emerson and A. P. Sistla, editors, *12th Conference on Computer Aided Verification (CAV'00)*, pages 85–98. Springer-Verlag, Berlin, 2000. LNCS 1855.

[9] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 68–80. Springer-Verlag, Berlin, 1994. LNCS 818.

[10] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[11] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, pages 52–71, 1982.

[12] L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. In *International Conference on Concurrency Theory (CONCUR'07)*, pages 74–89, 2007.

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

[14] E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *Proc. Computer Aided Verification*, pages 263–277, 2009.

[15] 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, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 413–427. Springer Berlin / Heidelberg, 2009.

[16] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean programs with an application to C. In *18th Conference on Computer Aided Verification (CAV'06)*, pages 358–371, 2006. LNCS 4144.

[17] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Formal verification of a complex pipelined processor. *Formal Methods in System Design*, 23(2):171–213, 2003.

[18] J.H.R. Jiang, H.P. Lin, and W.L. Hung. Interpolating Functions from Large Boolean Relations. In *Proceedings of the International Conference on Computer-Aided Design*, pages 779–784. IEEE/ACM, 2009.

[19] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification*, pages 258–262, 2007.

[20] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S. K. Rajamani, editors, *17th Conference on Computer Aided Verification (CAV'05)*, pages 226–238. Springer-Verlag, 2005. LNCS 3576.

[21] D. Kroening and O. Strichman. *Decision Procedures – An Algorithmic Point of View*. Springer, 2008.

[22] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 316–329, New York, NY, USA, 2010. ACM.

[23] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science*, pages 531–542, Pittsburgh, PA, October 2005.

[24] P. Manolios and S.K. Srinivasan. Automatic verification of safety and liveness for pipelined machines using WEB refinement. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(3):1–19, 2008.

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

[26] K.L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[27] A. Morgenstern and K. Schneider. Exploiting the temporal logic hierarchy and the non-confluence property for efficient LTL synthesis. *CoRR*, abs/1006.1408, 2010.

[28] E. Nurvitadhi, J.C. Hoe, T. Kam, and S.-L.L. Lu. Automatic pipelining from transactional datapath specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(3):441 –454, March 2011.

[29] N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.

[30] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77:81–98, 1989.

[31] Sven Schewe. Bounded synthesis. In *Automated Technology for Verification and Analysis (ATVA'07)*, pages 474–488, 2007.

[32] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for LTL games. In *9th International Conference on Formal Methods in Computer Aided Design (FMCAD'09)*, pages 77–84, 2009.

[33] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, pages 4–13. Springer Berlin / Heidelberg, 2009.

[34] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[35] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.

[36] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *Proc. 37th symposium on Principles of programming languages*, POPL '10, pages 327–338, 2010.

[37] Y. Watanabe and R.K. Brayton. Heuristic minimization of multiple-valued relations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(10):1458 –1472, October 1993.