A Theory of Abstraction for Arrays

Steven M. German
IBM T.J. Watson Research Center

Abstract— **We develop a theory for reasoning about temporal safety properties of systems with arrays. The theory leads to an automatic algorithm for constructing sound and complete abstractions. Our approach has advantages over previous approaches for important classes of digital designs, including designs with clock gating. We define a function that gives, in a certain sense, the size of the smallest sound and complete array abstraction of a system. This function is difficult to compute. However, we present a static analysis algorithm that efficiently computes a safe size of a sound and complete abstraction by overapproximating the minimum size. Our algorithm can often construct abstractions with small arrays for complex industrial designs.**

I. Introduction

Because of their large state spaces, arrays create a special difficulty for formal verification of hardware designs. In this paper we develop a theory that gives conditions under which a system with large arrays can be replaced by a system having smaller arrays, such that safety properies hold in the larger system iff they hold in the smaller one. The size of the arrays in the smaller system is determined by an efficient static analysis algorithm. An advantage of our approach is that it transforms a sequential system into another sequential system, allowing any verification method for sequential systems to be used after running our algorithm.

Previous researchers have developed several methods for reasoning about hardware systems with arrays, especially in the context of model checking. One general approach is to verify systems by considering behavior over a small, bounded number of time steps. Over a small number of time steps, a system performs only a small number of array read and write operations, and accesses only a small number array elements. Thus, bounded time modeling leads naturally to abstractions for arrays [1]–[4]. Behavior over a bounded number of time steps can be used to prove bounded correctness properties [5], or to prove unbounded properties by induction [6].

Abstracting arrays over bounded time intervals has the disadvantage that as the length of the time interval increases, the size of the array abstraction must be increased as well. For example, when proving properties by induction based on k-step unwindings, it is necessary to increase the length of the bounded interval until the property becomes inductive, which may make it necessary to consider large arrays [7].

A second general approach is to transform a sequential system with arrays into another sequential system with smaller arrays. Bjesse [7] developed an approach in which a sequential model with small arrays is constructed by abstraction refinement. One difference between our approach and [7] is that we use a static analysis algorithm instead of abstraction refinement. In our approach, the static analysis algorithm produces a single abstract model that is sound and complete for safety properties. Thus our approach eliminates the need for iterative abstraction refinement.

Another advantage of our approach is that we can build sound and complete abstractions for designs where a value read from an array can take an unbounded length of time before affecting an output signal. In [7], every abstract model of an array is characterized by a set of clock-cycle delays. Specifically, a small model of an array represents the results of reading the array at a finite set of fixed times prior to the clock cycle at which the value read from the array propagates to the system output. However, in many systems, the results of reading an array are stored for an unbounded amount of time in hardware registers or other arrays before being used to produce a system output. Hardware clock-gating [8] is one important design technique that leads to unbounded delays between the time an array is read and the time the array's value affects a system output. Our approach builds abstract models with unbounded delays, while the approach in [7] cannot reduce such models.

The outline of this paper is as follows. In Section II we define the mathematical framework for our theory. We define an operational semantics for executing systems that is appropriate when arrays have been replaced with smaller abstract arrays. In Section III we prove theorems showing the existence of sound and complete abstractions of systems with arrays. In Section IV we present an algorithm for analyzing a system to determine the size of a sound and complete array abstraction. In Section V we show how to build an abstracted version of a system, using the sizes for arrays determined by our theory. Section VI presents initial results of using our algorithm on industrial examples. Due to space limitations most proofs are omitted. A version of the paper with proofs will be published as a technical report, and will be available from the author and the IBM Research Division Libraries before FMCAD 2011.

II. Preliminaries

We begin by defining the syntax and semantics of a term-level logic with arrays. In our logic, there are two kinds of variables: signal variables, and array variables. Let X_s be the set of signal variables and X_a be the set of array variables. We define signal expressions and array expressions to be the smallest sets of expressions satisfying the following definitions. A variable (resp., expression) is either a signal variable (resp., expression) or an array variable (resp., expression).

1) A signal variable is a signal expression.
2) If op is a k-ary operator symbol and $e_1, \ldots, e_k$ are signal expressions, then $op(e_1, \ldots, e_k)$ is a signal expression.
3) If e_1, e_2, e_3 are signal expressions, then $mux(e_1, e_2, e_3)$ is a signal expression.

4) An array variable is an array expression.
5) If a is an array expression and e is a signal expression, then $a[e]$ is a signal expression.
6) If a is an array expression and e_1, e_2 are signal expressions, then $write(a, e_1, e_2)$ is an array expression.

For the semantics, we assume the existence of a set V of at least two *signal values*. We assume that V is finite, but much of the theory is true even if V is infinite. Let $0, 1$ be distinct signal values. We assume that for each domain value $i \in V$, there is a constant symbol c_i such that c_i evaluates to i. For $k \geq 0$, a k-ary operator op is a symbol whose interpretation is a function $OP : V^k \rightarrow V$.

An array will be abstracted by replacing it with an array having a smaller domain. We will need to give meaning to array access expressions $a[i]$, where the value of i is not in the domain of a. For this purpose, we introduce a bottom value $\perp \notin V$. Let $V^+ = V \cup \{\perp\}$. We will define a semantics that propagates the bottom value onward, starting from an array access that is outside the domain of the array. In the case of mux expressions, an expression will have a value in V provided the first argument has a value in $\{0, 1\}$ and the selected argument of the mux has a value in V.

In the semantics of our logic, a signal variable is a name that can be assigned signal values, and an array variable is a name that can be assigned array values. An array value is a function in $V \rightarrow V^+$. We explicitly allow array values to be partial functions whose domains are not equal to the entire set of signal values V. We will use array values that are partial functions on V to reason about systems containing an array without representing all of the array's elements. An array value v is said to be *pure* if $\forall x \in \text{dom}(v) : v(x) \in V$.

A *state* assigns values to variables. A state assigns a signal value to each signal variable, and assigns a pure array value to each array variable. Note that states only contain the values in V, not the bottom value. In sequential systems, which will be defined shortly, we will use states only to represent the initial values of state variables and the values of input signals. The reason that states do not contain values with $\perp$ is that these values do not represent initial states or input values; the bottom value is only produced during evaluation of certain expressions.

The *domain* of an array expression a in a state σ, $\text{D}(a, \sigma)$, is a set of index values for the array expression. For an array variable a, $\text{D}(a, \sigma)$ is defined as $\text{dom}(\sigma(a))$, the domain of the function value assigned to a. Array write operations do not change the domain of an array. The domain of $write(a_1, e_1, e_2)$ in state σ is inductively defined by $\text{D}(write(a_1, e_1, e_2), \sigma) = \text{D}(a_1, \sigma)$. We will need the notion of the *root* of an array expression. For an array variable a, $\text{root}(a) = a$. The root of a write expression is $\text{root}(write(b, e_1, e_2)) = \text{root}(b)$.

We define the value of an expression exp with respect to a state σ, written $\sigma[\![exp]\!]$, as follows. In the following definition, $e, e_1, e_2, \ldots,$ are signal expressions and a is an array expression.

1) $\sigma[\![v]\!] = \sigma(v)$, where v is a signal variable.

2) $\sigma[\![op(e_1, \ldots, e_n)]\!] =$
$$\begin{cases} OP(\sigma[\![e_1]\!], \ldots, \sigma[\![e_n]\!]), & \text{if } \sigma[\![e_i]\!] \neq \perp, \text{ for } i = 1, \ldots, n, \\ & \text{where } OP \text{ is the interpretation of the operator} \\ & \text{symbol } op \\ \perp & \text{if for some } i,\ \sigma[\![e_i]\!] = \perp \end{cases}$$

3) $\sigma[\![mux(e_1, e_2, e_3)]\!] = \begin{cases} \sigma[\![e_2]\!] & \text{if } \sigma[\![e_1]\!] = 0 \\ \sigma[\![e_3]\!] & \text{if } \sigma[\![e_1]\!] = 1 \\ \perp & \text{if } \sigma[\![e_1]\!] \notin \{0, 1\} \end{cases}$

4) $\sigma[\![a[e]]\!] = \begin{cases} (\sigma[\![a]\!])(\sigma[\![e]\!]) & \text{if } \sigma[\![e]\!] \in \text{D}(a, \sigma) \\ \perp & \text{if } \sigma[\![e]\!] \notin \text{D}(a, \sigma) \end{cases}$

5) $\sigma[\![a]\!] = \sigma(a)$, where a is an array variable.

6) $\sigma[\![write(a, e_1, e_2)]\!] =$
$$\begin{cases} (\sigma[\![a]\!])\,[\sigma[\![e_1]\!] \leftarrow \sigma[\![e_2]\!]] & \text{if } \sigma[\![e_1]\!] \in \text{D}(a, \sigma) \\ \sigma[\![a]\!] & \text{if } \sigma[\![e_1]\!] \in V - \text{D}(a, \sigma) \\ \text{bottom}(a, \sigma) & \text{if } \sigma[\![e_1]\!] = \perp \end{cases}$$

Expressions of the form $op(e_1, \ldots, e_n)$ can be used to represent blocks of combinational logic containing many gates. In the semantics, an op expression has value $\perp$ whenever any of the input signals has value $\perp$. The advantage of this semantics is that it allows us to define a circuit that computes $\sigma[\![op(e_1, \ldots, e_n)]\!]$, without having to add signals to express whether the output of each gate in a large block has value $\perp$. We will use such circuits in building our abstract models. The abstract model can have fewer gates because only the inputs and output of a large block need to consider $\perp$.

In the array write expression $write(a, e_1, e_2)$, e_1 is the address and e_2 is the value written. There are three cases in the semantics of array writes. The first case updates a single element of an array when e_1 has a value in the domain of the array. In the second case, e_1 has a value in V, but the value is outside the domain of the array. In this case, the value of the array is not changed by the write operation. Note that the operation of writing to an array does not extend the domain of the array. In the third case, the index e_1 has the value $\perp$. For an array expression a and a state σ, we define $\text{bottom}(a, \sigma)$ to be an array value, the function that maps all elements of $\text{D}(a, \sigma)$ to $\perp$. The intuition is that if we write to an address e_1 that has value $\perp$, then it cannot be determined which element of the array is changed, so all elements are marked as having value $\perp$.

Electronic designs sometimes have arrays where writing is conditional. For example, writing can be controlled by an enable signal, with value 1 to indicate writing a new value. Conditional writing can be modelled by write expressions such as $write(a, address, mux(enable, a[address], new\_value))$. This expression produces an unchanged array value when $enable = 0$ and $address$ is a value in the domain of the array, and writes a new value when $enable = 1$.

A signal expression e is said to be *satisfied* by a state σ if $\sigma[\![e]\!] = 1$. We write $\sigma \models e$ to denote that σ satisfies e. A signal expression e is said to be satisfiable if there exists a state that satisfies e, and e is said to be valid if it is satisfied by all states.

A *system* $\mathcal{M}$ has the form $(\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$. $\mathcal{S}$ is a set of signal and array variables forming the state variables of the system. $\mathcal{I}$ is a set of input signal variables. $\mathcal{N}$ defines the next-state function of the system. $\mathcal{N}$ is a function from the variable names in $\mathcal{S}$ to expressions, such that if e is a signal variable, then $\mathcal{N}(e)$ is a signal expression, and if a is an array variable, then $\mathcal{N}(a)$ is an array expression. For an array variable a, we require that $\mathrm{root}(\mathcal{N}(a)) = a$; that is, the next state expression for an array variable a must be formed by a sequence of writes to a. $\mathcal{O}$ is a set of signal variables that are the outputs of the system, and $\mathcal{E}$ is a function mapping variables in $\mathcal{O}$ to signal expressions. The sets $\mathcal{S}, \mathcal{I}, \mathcal{O}$ must be pairwise disjoint.

We define the executions of a system by giving an operational semantics based on expansions of the next-state functions for state variables. Given a system $\mathcal{M}$ and a state variable $s \in \mathcal{S}$, we define

$$\begin{array}{lcl} s^0 & = & s, \\ s^{k+1} & = & \mathcal{N}(s)[\mathcal{S}/\mathcal{S}^k,\ \mathcal{I}/\mathcal{I}^k],\ \text{for } k = 0, 1, \ldots, \end{array}$$

where $\mathcal{S}/\mathcal{S}^k$ replaces each variable s_j in $\mathcal{S}$ with the expression $(s_j)^k$, and $\mathcal{I}/\mathcal{I}^k$ replaces each input variable u in $\mathcal{I}$ with a fresh signal variable u^k. For an output variable $v \in \mathcal{O}$, the output value at step k depends on the state and input variables at step k,

$$v^k \ = \ \mathcal{E}(v)[\mathcal{S}/\mathcal{S}^k,\ \mathcal{I}/\mathcal{I}^k],\ \text{for } k = 0, 1, \ldots$$

For an arbitrary expression e over the variables of $\mathcal{M}$, we define $e^k = e[\mathcal{S}/\mathcal{S}^k, \mathcal{I}/\mathcal{I}^k, \mathcal{O}/\mathcal{O}^k]$.

The safety properties of a system are specified by its output signals. For an output signal v of a system $(\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$, v is said to be valid with respect to the system iff for all states σ, $\sigma \models v^k$, for all $k \geq 0$.

Our theory does not prescribe an approach to variable typing. Instead, we allow the set of initial states of a system to be a parameter of the theory. It will be useful to make two considerations about the set of initial states of a system. First, it is conventional to assume that an array variable has the same set of indices in any system state. We formalize this by saying that a *type* $\mathcal{T}$ is a set of states such that for any array variable a and $\sigma_1, \sigma_2 \in \mathcal{T}$, $\mathrm{dom}(\sigma_1(a)) = \mathrm{dom}(\sigma_2(a))$. We will evaluate correctness of systems over sets of initial states that are types. As mentioned previously, the dimension of an array does not change dynamically.

Second, because the value $\perp$ has a special meaning in our theory, care is needed when translating a design from a hardware language into our theory. We need to make sure that a $\perp$ value cannot be generated accidentally because the original design subscripts an array outside of the declared domain. One way is to define the design in a language where array subscripts can be checked statically, and then translate a checked design into our theory. Another approach is to translate an array read $a[i]$ in the hardware design into an expression $\mathit{mux}(\mathit{lower} \leq i \wedge i \leq \mathit{upper}, a[i], \mathit{nondet})$, in our theory. Here, the value of $a[i]$ is used if i is in the declared domain, and otherwise a nondeterminsitic input value nondet is used.

In our theory, we can express the notion that a design is well-typed by introducing an assumption that the set of initial states has the property that all the expressions generated by the operational semantics of a system produce values in V. We say that a state σ is *safe* with respect to an expression e iff $\sigma[\![e]\!] \in V$ and for every subexpression e' of e, $\sigma[\![e']\!] \in V$. For a state to be safe is a stronger condition than just saying the output has a value in V. We say that a type $\mathcal{T}$ is safe for a system $\mathcal{M}$ iff for all state or output variables v of $\mathcal{M}$, for all states $\sigma \in \mathcal{T}$, and for all $k \geq 0$, σ is safe with respect to v^k. We define the notion of correctness for safety properties by evaluating variables of a system in all states of a safe type. We say that a variable v is valid in a system $\mathcal{M}$ and initial state set $\mathcal{T}$, written $\mathcal{M}, \mathcal{T} \models v$, iff for all states $\sigma \in \mathcal{T}$, for all $k \geq 0$, $\sigma[\![v^k]\!] = 1$.

III. Existence of Array Abstractions

In this section we show that under certain conditions, there are small abstract models for systems with arrays. The abstract models are sound and complete for safety properties.

Because arrays and multiplexors propagate values only from the selected input, it is possible for an expression to have a value in V even if some array accesses have value $\perp$. To capture the notion that some, but not all, expressions must have values in V in order to compute the value of a larger expression, we define the set of *essential expressions* of an expression e in a state σ, written $\mathrm{eexp}(e, \sigma)$. See Figure 1. There are four cases for write expressions: Case 1 is when e_3 is not a valid index; Case 2 is when e_1 is not a valid index; Case 3 is when e_1, e_2 are valid indices with different values, so that $\mathit{write}(b, e_1, e_2)[e_3] = b[e_3]$; Case 4 is when $\mathit{write}(b, e_1, e_2)[e_3] = e_2$. Under the assumption that all of the essential expressions in $\mathrm{eexp}(e, \sigma)$, not including e itself, evaluate to values in V, then $\sigma[\![e]\!] \in V$. In reading the definition of $\mathrm{eexp}(e, \sigma)$, it is important to note that e is always an essential expression of itself. Also, the definition of eexp applies a case-splitting rule to array read operations with nested array writes. Because of the case-splitting rule, $\mathrm{eexp}(e, \sigma)$ can contain expressions that are not subexpressions of e.

Lemma 1. Let e be a signal expression and σ be a state. Then $\sigma[\![e]\!] \in V$, iff for all essential expressions f of e in σ, $\sigma[\![f]\!] \in V$. □

The set of *essential indices* of an array variable a with respect to an expression e and a state σ is the set of values in V of signal expressions f such that $b[f]$ is an essential expression, where b is an array expression with $\mathrm{root}(b) = a$.

Formally, we define

$$\begin{array}{l} \mathrm{eindx}(e, \sigma, a) = \\ \{\sigma[\![f]\!] \mid \sigma[\![f]\!] \in \mathrm{D}(a, \sigma) \wedge \exists\ b : b[f] \in \mathrm{eexp}(e, \sigma), \\ \quad \text{where } b \text{ is an array expression and } \mathrm{root}(b) = a\}. \end{array}$$

As an example of essential expressions and indices, consider the expression $\mathit{write}(a, e, a[4])[5]$. In a state σ where $\sigma[\![e]\!] = 5$, the essential expressions are: $\mathit{write}(a, e, a[4])[5]$, e, and $a[4]$; the essential indices of the array a are: 4 and 5. Intuitively, the value of the expression is obtained by evaluating $a[4]$ when the value of $e = 5$. In a state σ where $\sigma[\![e]\!] \neq 5$, the essential expressions are: $\mathit{write}(a, e, a[4])[5]$,

$$\begin{array}{l}
\text{eexp}(e,\sigma) = \\
\quad \text{if } e \text{ is a signal variable } \Rightarrow \{e\} \\[1em]
\quad \text{if } e \text{ is } op(e_1,\ldots,e_n) \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \ldots \cup \text{eexp}(e_n,\sigma) \\[1em]
\quad \text{if } e \text{ is } mux(e_1,e_2,e_3) \Rightarrow \\
\quad \begin{cases}
\text{if } \sigma[\![e_1]\!] \notin \{0,1\} \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \\
\text{if } \sigma[\![e_1]\!] = 0 \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_2,\sigma) \\
\text{if } \sigma[\![e_1]\!] = 1 \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_3,\sigma)
\end{cases} \\[1em]
\quad \text{if } e \text{ is } b[e_1] \text{, where } b \text{ is an array variable } \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \\[1em]
\quad \text{if } e \text{ is } write(b,e_1,e_2)[e_3] \Rightarrow \\
\quad \begin{cases}
\text{if } \sigma[\![e_3]\!] \notin \text{D}(b,\sigma) \Rightarrow \{e\} \cup \text{eexp}(e_3,\sigma) \\
\text{if } \sigma[\![e_3]\!] \in \text{D}(b,\sigma) \wedge \sigma[\![e_1]\!] = \bot \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_3,\sigma) \\
\text{if } \sigma[\![e_3]\!] \in \text{D}(b,\sigma) \wedge \sigma[\![e_1]\!] \in V \wedge \sigma[\![e_1]\!] \neq \sigma[\![e_3]\!] \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_3,\sigma) \cup \text{eexp}(b[e_3],\sigma) \\
\text{if } \sigma[\![e_3]\!] \in \text{D}(b,\sigma) \wedge \sigma[\![e_1]\!] = \sigma[\![e_3]\!] \Rightarrow \{e\} \cup \text{eexp}(e_1,\sigma) \cup \text{eexp}(e_2,\sigma) \cup \text{eexp}(e_3,\sigma)
\end{cases}
\end{array}$$

Fig. 1. Definition of essential expressions eexp

e, and $a[5]$; the essential indices of a are: 5. In this case, the value of the expression is obtained by evaluating $a[5]$.

For states σ, σ', we say that σ' is a *substate* of σ, written $\sigma' \leq \sigma$ iff

1) For all signal variables v, $\sigma'(v) = \sigma(v)$, and
2) For all array variables a, $\text{dom}(\sigma'(a)) \subseteq \text{dom}(\sigma(a)) \wedge \forall i \in \text{dom}(\sigma'(a)) : \sigma'(a)(i) = \sigma(a)(i)$.

The following lemma says that if an expression e evaluates to a value in V in a state σ, then there is a substate $\sigma' \leq \sigma$, that gives e the same value and such that each array variable is only defined over the essential indices of the variable in σ. That is, for all array variables a, $\text{dom}(\sigma'(a)) = \text{eindx}(e, \sigma, a)$. This lemma will allow us to replace arrays by smaller abstractions.

Lemma 2. Let e be a signal expression and σ be a state such that $\sigma[\![e]\!] \in V$. Then there exists a state σ' such that $\sigma' \leq \sigma$, for all array variables a, $\text{dom}(\sigma'(a)) = \text{eindx}(e, \sigma, a)$, and $\sigma[\![e]\!] = \sigma'[\![e]\!]$. □

IV. Size of Array Abstractions

We define the size of a state σ, written $|\sigma|$, to be the function mapping each array variable a to the size of the domain of a: for all array variables a, $|\sigma|(a) = |\text{dom}(\sigma(a))|$. Similarly, we define the size of a type $\mathcal{T}$ to be $|\sigma|$, for any state $\sigma \in \mathcal{T}$.

From Lemma 2, we see that if $\sigma[\![e]\!] = v$, for $v \in V$, then there is a state σ' such that for each array variable a, the size of array a is $|\text{eindx}(e, \sigma, a)|$, and $\sigma'[\![e]\!] = v$. Now, suppose $\mathcal{U}$ is any set of states, and we want to evaluate an expression e over all states in $\mathcal{U}$. It is sufficient to evaluate e over all states where the size of each array a is the maximum size needed for any state. This value is given by the function $\Sigma_{\mathcal{U}}$: *expressions* $\rightarrow (X_a \rightarrow \mathbb{N})$, where

$$\Sigma_{\mathcal{U}}(e)(a) = \max_{\sigma \in \mathcal{U}} |\text{eindx}(e, \sigma, a)|.$$

With this definition, $\Sigma_{\mathcal{U}}(e) : X_a \rightarrow \mathbb{N}$ is a function that encapsulates all of the sizes of arrays needed to evaluate the expression e. The function $\Sigma_{\mathcal{U}}(e)(a)$ always has a defined value when V is finite, because the value of eindx is a subset of V. When $\mathcal{U}$ is the set of all states over V, we drop the subscript and write $\Sigma(e)(a)$. $\Sigma(e)(a)$ gives an upper bound on the size of arrays needed to test if an expression is satisfiable in any state.

Proposition 1. Let σ', σ be states such that $\sigma' \leq \sigma$. Let e be a signal expression and let $i \in V$. Then the following three conditions hold:

1) $\sigma[\![e]\!] = i \Rightarrow (\sigma'[\![e]\!] = i \vee \sigma'[\![e]\!] = \bot)$
2) $\sigma[\![e]\!] = \bot \Rightarrow \sigma'[\![e]\!] = \bot$
3) $\sigma'[\![e]\!] = i \Rightarrow \sigma[\![e]\!] = i$ □

Proposition 2. Let e be an expression, σ be a state, and a be an array variable. Then $|\text{eindx}(e, \sigma, a)| \leq |\text{dom}(\sigma(a))|$. This is true because in the definition of eindx, each element of $\text{eindx}(e, \sigma, a)$ must be an element of $\text{dom}(\sigma(a))$. □

Theorem 1: Small Model Theorem. If a signal expression e is satisfiable, there is a state σ that satisfies e such that $|\sigma| = \Sigma(e)$.

Proof. Let σ be a state that satisfies e. By Lemma 2 there is a state σ' such that σ' satisfies e, and for all a, $|\text{dom}(\sigma'(a))| \leq \Sigma(e)(a)$. From Propositions 1.3 and 2, it follows that σ' can be expanded to a state σ'' such that $\sigma' \leq \sigma''$, $|\sigma''| = \Sigma(e)$, and σ'' satisfies e. □

We define an upper bound for a system $\mathcal{M}$ and a set of states $\mathcal{U}$ by a function $\Sigma^*_{\mathcal{M},\mathcal{U}} : X_s \rightarrow (X_a \rightarrow \mathbb{N})$,

$$\Sigma^*_{\mathcal{M},\mathcal{U}}(v)(a) = \max_{k=0,1,\ldots} \Sigma_{\mathcal{U}}(v^k)(a),$$

where v is a signal variable and a is an array variable. The value of $\Sigma^*_{\mathcal{M},\mathcal{U}}(v)(a)$ is an upper bound on the number of index values of the array a needed to evaluate all of the expansions of the signal variable v over all states in $\mathcal{U}$. Like $\Sigma_{\mathcal{U}}(e)(a)$, the function $\Sigma^*_{\mathcal{M},\mathcal{U}}(v)(a)$ has a defined value when V is finite. When $\mathcal{U}$ is the set of all states, we drop the second subscript and write $\Sigma^*_{\mathcal{M}}(v)(a)$.

The following theorem says that it is sound and complete to reason about an output variable v as a safety property of a system, by evaluating v in all states of size $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)$. In the statement of the theorem, $\mathcal{T}$ is a set of states for the unabstracted model. We assume that $\mathcal{T}$ is safe, so that all expressions in the executions of the system can be evaluated. The theorem says that if we evaluate the truth of a safety property v over all states σ', such that σ' is a substate of some state in $\mathcal{T}$, and the size of σ' is $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)$, then we can determine the result of evaluating v over all states in the unabstracted model over the set of states $\mathcal{T}$. The theorem provides a sound and complete method for reasoning about safety properties while reducing the size of arrays.

Theorem 2.1. Let $\mathcal{M}$ be a system with output variable v and let $\mathcal{T}$ be a safe type for $\mathcal{M}$. Let

$$\mathcal{T}' = \{\sigma' \mid \exists \sigma \in \mathcal{T} : \sigma' \leq \sigma \ \wedge \ |\sigma'| = \Sigma^*_{\mathcal{M},\mathcal{T}}(v)\}$$

Then $\mathcal{M}, \mathcal{T} \models v$ iff $\forall k \geq 0$, $\forall \sigma' \in \mathcal{T}' : \sigma'[\![v^k]\!] = 1 \vee \sigma'[\![v^k]\!] = \bot$.

Proof. ($\Rightarrow$) $\mathcal{M}, \mathcal{T} \models v$ means $\forall k \geq 0$, $\forall \sigma \in \mathcal{T}$, $\sigma[\![v^k]\!] = 1$. If $k \geq 0$, $\sigma \in \mathcal{T}$, and $\sigma' \leq \sigma$, then by Proposition 1.1, $\sigma'[\![v^k]\!] = 1$ or $\sigma'[\![v^k]\!] = \bot$.

($\Leftarrow$) What we need to show is that if there is a counterexample for some state in $\mathcal{T}$ in the unabstracted system, then there is a counterexample in a state in $\mathcal{T}'$ in the abstracted system. Suppose there is a counterexample: let $k \geq 0$ and $\sigma \in \mathcal{T}$ be such that $i \in V$, $i \neq 1$ and $\sigma[\![v^k]\!] = i$. By Lemma 2, we know there is a state σ_1, such that $\sigma_1 \leq \sigma$, for all array variables a, $\mathrm{dom}(\sigma_1(a)) = \mathrm{eindx}(v^k, \sigma, a)$, and $\sigma_1[\![v^k]\!] = i$. By definition, $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a)$ takes the maximum value of $|\mathrm{eindx}(v^k, \sigma, a)|$ over all $k \geq 0$ and $\sigma \in \mathcal{T}$, so that for all a, $|\mathrm{dom}(\sigma_1(a))| \leq \Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a)$. By Proposition 2, for all a, $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a) \leq |\mathrm{dom}(\sigma(a))|$, since $\sigma \in \mathcal{T}$. Therefore, there is a state σ_2, such that $\sigma_1 \leq \sigma_2 \leq \sigma$ and $|\sigma_2| = \Sigma^*_{\mathcal{M},\mathcal{T}}(v)$. Note that $\sigma_2 \in \mathcal{T}'$. By Proposition 1.3, $\sigma_2[\![v^k]\!] = i$. Therefore, σ_2 is a counterexample in $\mathcal{T}'$.

We are now ready to complete the proof. If it is the case that for all $k \geq 0$ and $\sigma' \in \mathcal{T}'$, $\sigma'[\![v^k]\!] = 1 \vee \sigma'[\![v^k]\!] = \bot$, then there cannot be a counterexample to the truth of v over all states in $\mathcal{T}$. Since $\mathcal{T}$ is a safe type, it must be the case that $\forall k \geq 0$, $\forall \sigma \in \mathcal{T}$, $\sigma[\![v^k]\!] = 1$. □

In practice, it is difficult to evaluate $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a)$, since this involves finding a maximum value over all states in $\mathcal{T}$ and over all computation steps. However, we show later in the paper that there are ways to compute an upper bound on $\Sigma^*_{\mathcal{M}}(v)(a)$. Computing an upper bound is easier than computing the exact value, and also it is easier to compute an upper bound over all states instead of over a set $\mathcal{T}$. Since $\Sigma^*_{\mathcal{M}}(v)(a)$ is a maximum over all states, $\Sigma^*_{\mathcal{M},\mathcal{T}}(v)(a) \leq \Sigma^*_{\mathcal{M}}(v)(a)$ for all a. Note that it is possible for $\Sigma^*_{\mathcal{M}}(v)(a)$ to be larger than the size of a in $\mathcal{T}$. To prove properties over the set of states $\mathcal{T}$, we would not want to make the size each array variable a equal to $\Sigma^*_{\mathcal{M}}(v)(a)$. Instead we make the size of each array variable a equal to the minimum of $\Sigma^*_{\mathcal{M}}(v)(a)$ and the size of a in $\mathcal{T}$. Let v be a fixed variable name, and define a function $\mu : X_a \to \mathbb{N}$,

$$\mu(a) = \min(|\mathcal{T}|(a),\ \Sigma^*_{\mathcal{M}}(v)(a))$$

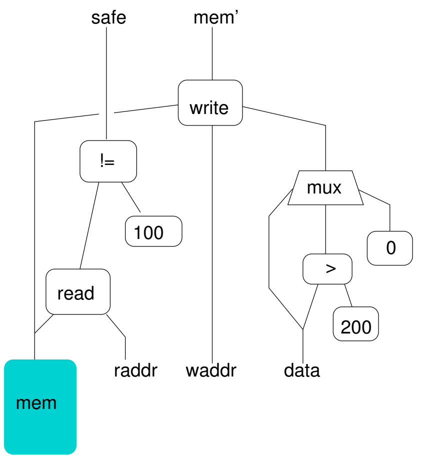


Fig. 2. Example 1

Let $\mathcal{T}''$ be the set of states that are smaller than states in $\mathcal{T}$ and of size μ,

$$\mathcal{T}'' = \{\sigma' \mid \exists \sigma \in \mathcal{T} : \sigma' \leq \sigma \ \wedge \ |\sigma'| = \mu\}$$

Since the size of arrays in $\mathcal{T}''$ are at least as large as in $\mathcal{T}'$ defined in Theorem 2.1, it follows from Proposition 1 that Theorem 2.1 holds if we use $\mathcal{T}''$ in place of $\mathcal{T}'$.

Theorem 2.2. Let $\mathcal{M}$ be a system with output variable v, let $\mathcal{T}$ be a safe type for $\mathcal{M}$ and let $\mathcal{T}''$ be as defined above. Then $\mathcal{M}, \mathcal{T} \models v$ iff $\forall k \geq 0$, $\forall \sigma' \in \mathcal{T}'' : \sigma'[\![v^k]\!] = 1 \vee \sigma'[\![v^k]\!] = \bot$. □

In an implementation of Theorem 2.2, a model would be constructed with each array variable a having a domain of size $\mu(a)$. As an example, if the original model has an array a with domain $[1..100]$, and the size of the domain of a is 2 in the abstract model, then we need to evaluate the abstract model over all states where the domain of a is any two elements of $[1..100]$. As in [7], a value for the address of each row in the abstract array is chosen nondeterministically at the start of the run. The read and write operations in the abstract model use the address chosen for each of the rows, instead indexing into the array. The implementation then uses model checking to evaluate for all $\sigma \in \mathcal{T}''$, $\sigma[\![v^0]\!], \sigma[\![v^1]\!]$, and so on. If there is a state $\sigma \in \mathcal{T}$ such that for some k, $\sigma[\![v^k]\!]$ evaluates to a value $x \in V$ other than 1, then there is a state σ' such that $\sigma' \leq \sigma$, $|\sigma'| = \mu$, and $\sigma'[\![v^k]\!] = x$.

Example 1. This example appeared in [7]. On each clock cycle, the design inputs values for the signals `raddr`, `waddr`, and `data`. The array `mem` is an array state variable. A network with a multiplexor produces an output value that depends on the value of the input `data`. If `data` > 200, the multiplexor outputs the value of `data`; otherwise it outputs the value 0. On each clock cycle, the array `mem` is written at address `waddr` and with the value output from the multiplexor. On each clock cycle, the array `mem` is read at address `raddr`. The correctness property asserts that the value read from the array is never equal to 100.

Here, we render the example in our formalism. The system is $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$, where the set of state variables is $\mathcal{S} = \{\mathtt{mem}\}$, the set of input variables is $\mathcal{I} = \{\mathtt{raddr}, \mathtt{waddr}, \mathtt{data}\}$, and the set of output variables is $\mathcal{O} = \{\mathtt{safe}\}$. The next state function $\mathcal{N}$ and output function $\mathcal{E}$ are

$$\begin{array}{l} \mathcal{N}(\mathtt{mem}) = \mathit{write}(\mathtt{mem},\ \mathtt{waddr},\ \mathit{mux}(\mathtt{data} > 200,\ \mathtt{data},\ 0)) \\ \mathcal{E}(\mathtt{safe}) = \mathtt{mem}[\mathtt{raddr}] \neq 100. \end{array}$$

For this system, the first few expansions of mem and safe are given by

$$\begin{array}{l} \mathtt{safe}^0 = \mathtt{mem}^0[\mathtt{raddr}^0] \neq 100 \\ \mathtt{mem}^1 = \mathit{write}(\mathtt{mem}^0, \mathtt{waddr}^0, \mathit{mux}(\mathtt{data}^0 > 200, \mathtt{data}^0, 0)) \\ \mathtt{safe}^1 = \mathit{write}(\mathtt{mem}^0, \mathtt{waddr}^0, \mathit{mux}(\mathtt{data}^0 > 200, \mathtt{data}^0, 0)) \\ \qquad [\mathtt{raddr}^1] \neq 100 \\ \mathtt{mem}^2 = \mathit{write}(\mathit{write}(\mathtt{mem}^0, \mathtt{waddr}^0, \\ \qquad\qquad \mathit{mux}(\mathtt{data}^0 > 200, \mathtt{data}^0, 0)), \\ \qquad \mathtt{waddr}^1, \\ \qquad \mathit{mux}(\mathtt{data}^1 > 200, \mathtt{data}^1, 0)) \\ \mathtt{safe}^2 = \mathtt{mem}^2[\mathtt{raddr}^2] \neq 100. \end{array}$$

One can easily see by inspection that exactly one array index expression appears in $\mathtt{safe}^k$, for any value of k. Thus the maximum number of indexes into the array mem needed to evaluate safe is $\Sigma^*_{\mathcal{M}}(\mathtt{safe})(\mathtt{mem}) = 1$. It follows that it is sound and complete to verify the output signal safe as a safety property by evalating $\mathtt{safe}^k$, for all k, in states where mem is modelled as a single-element array.

If we change the example so that the result of reading mem is stored in a register for an unbounded number of cycles before the value is used to produce the output, then the method of [7] would not be able to reduce the size of the array. Our method does reduce the array to one element in this modified example. □

Example 2. This example is similar to Example 1, please see Figure 3. Here, the result of reading the memory is routed to a multiplexor. The multiplexor sends a value to the state variable read1. On each cycle, the variable read1 either holds its previous value or stores the output of the array read, depending on the value of the input signal hold. Because the value read from the array can be held for an unbounded amount of time in read1 beforefore reaching the output, the method of [7] cannot reduce the size of the array. In contrast, our method can reduce the array to one entry. □

The following definition of a function $\mathrm{ub}(e, a)$ computes a simple upper bound on $\Sigma(e)(a)$. For $b[e]$, if b is a write expression with root a, then e is counted as an array index of a. For *write* expressions, the definition says that the expression e_1 must always be evaluated, and there are two cases for b and $e2$. If the array is read at index e_1, then e_2 must be evaluated; otherwise b must be evaluated. To cover both cases, we take the maximum value.

$$\begin{array}{l} \mathrm{ub}(\mathit{var}, a) = 0,\ \text{if } \mathit{var} \text{ is a signal variable or an array variable} \\ \mathrm{ub}(c, a) = 0,\ \text{if } c \text{ is a constant} \\ \mathrm{ub}(\mathit{op}(e_1, \ldots, e_n), a) = \mathrm{ub}(e_1, a) + \ldots + \mathrm{ub}(e_n, a) \\ \mathrm{ub}(\mathit{mux}(e_1, e_2, e_3), a) = \mathrm{ub}(e_1, a) + \max(\mathrm{ub}(e_2, a), \mathrm{ub}(e_3, a)) \\ \mathrm{ub}(b[e], a) = \left\{ \begin{array}{ll} \mathrm{ub}(b, a) + \mathrm{ub}(e, a) + 1 & \text{if root}(b) = a \\ \mathrm{ub}(b, a) + \mathrm{ub}(e, a) & \text{otherwise} \end{array} \right. \\ \mathrm{ub}(\mathit{write}(b, e_1, e_2), a) = \mathrm{ub}(e_1, a) + \max(\mathrm{ub}(e_2, a), \mathrm{ub}(b, a)) \end{array}$$

Theorem 3. For any signal expression e, state σ, and array variable a, $|\mathrm{eindx}(e, \sigma, a)| \leq \mathrm{ub}(e, a)$. □

Proof. The theorem can be proved by induction on expressions. □

In the function $\mathrm{ub}(e, a)$, multiple instances of the same expression are added. A more accurate estimate of $\Sigma(e)(a)$ can be obtained by a function that computes the sets of possible index expressions.

A better estimate of $\Sigma(e)(a)$ can be obtained by a function that computes the sets of possible index expressions. For any non-empty set X whose elements are a finite number of finite sets, let $\|X\|$ be the largest size of an element of X, $\max_{x \in X} |x|$. The function $\phi(a, e)$ defined below computes a set of sets of index expressions with the property that $\Sigma(e)(a) \leq \|\phi(e, a)\|$. Each element of $\phi(a, e)$ is a set of index expressions; the elements of $\phi(a, e)$ cover all the possible values of $\mathrm{eindx}(e, \sigma, a)$.

First, we define $X \uplus Y$ for sets X, Y.

$$X \uplus Y = \{x \cup y \mid x \in X, y \in Y\}$$

Then we define $\phi(e, a)$, where e is an expression and a is an array variable, as follows.

$$\phi(v, a) = \{\emptyset\},\ \text{if } v \text{ is a signal variable or an array variable}$$

$$\phi(c, a) = \{\emptyset\},\ \text{if } c \text{ is a constant}$$

$$\phi(\mathit{op}(e_1, \ldots, e_n), a) = \phi(e_1, a) \uplus \ldots \uplus \phi(e_n, a)$$

$$\phi(\mathit{mux}(e_1, e_2, e_3), a) = (\phi(e_1, a) \uplus \phi(e_2, a)) \cup (\phi(e_1, a) \uplus \phi(e_3, a))$$

$$\phi(b[e], a) = \left\{ \begin{array}{ll} \phi(b, a) \uplus \phi(e, a) \uplus \{\{e\}\} & \text{if root}(b) = a \\ \phi(b, a) \uplus \phi(e, a) & \text{otherwise} \end{array} \right.$$

$$\phi(\mathit{write}(b, e_1, e_2), a) = (\phi(e_1, a) \uplus \phi(e_2, a)) \cup (\phi(e_1, a) \uplus \phi(b, a))$$

Theorem 4. For any signal expression e, state σ, and array variable a, $|\mathrm{eindx}(e, \sigma, a)| \leq \|\phi(e, a)\|$. □

In order to create sound abstractions for model checking, we want to compute an upper bound on the sequence $\phi(v^0, a), \phi(v^1, a), \ldots$, when an upper bound exists. One way of computing an upper bound uses a generalization of function $\phi(e, a)$ to a function $\Phi(e, a, \theta, \theta_A)$, where e is an expression, a is an array variable, and θ is a function mapping variables to sets of expressions. The idea is that the algorithm will iterate, at each step setting $\theta(v)$ to a set that is at least as general as each element of $\phi(v^k, a)$. The θ functions will provide a

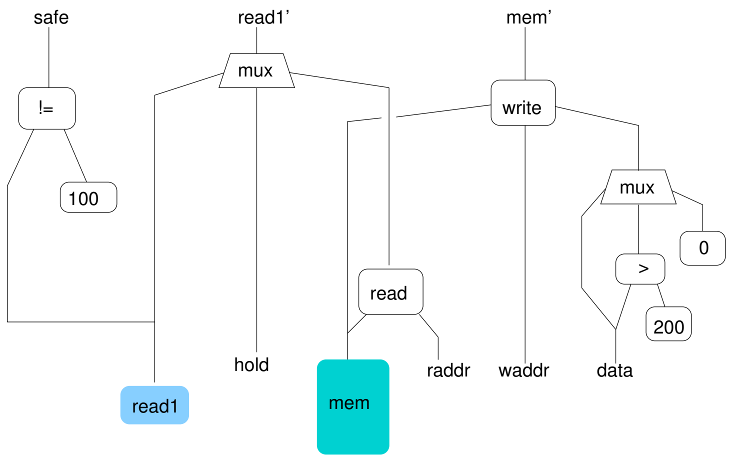


Fig. 3. Example 2

simple way to detect when a fixed point is reached in the iteration.

For a signal variable s, $\theta(s)$ will have the form $\{x_1, \ldots, x_n\}$, for some n, where the x_i are fresh distinct signal variables. Intuitively, we set $\theta(s)$ to a set of fresh variables larger than the set of essential indices needed to evaluate s. For array variables, we define $\theta_A(b, e)$ to be a function taking an array variable b and a signal expression e, and returning a set of signal variables, $\{x_{e,1}, \ldots, x_{e,n}\}$, for some n. The subscript on e simply indicates that $x_{e,i}$ is a fresh distinct signal variable related to e.

The definitions of $\phi(v, a)$ and $\Phi(v, a, \theta, \theta_A)$ differ in the case when v is a signal or array variable; in these cases Φ applies the function θ or θ_A. The functions θ, θ_A will be assigned increasingly large sets as the algorithm iterates. We define $\Phi(e, a, \theta, \theta_A)$ and $\Phi_A(b, e, a, \theta, \theta_A)$ by mutual recursion in the full paper .

Given a system $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{N}, \mathcal{O}, \mathcal{E})$, we define a sequence of approximations to $\phi(s^k, a)$, for each signal state variable s, and approximations to $\phi(b^k, a)$, for each array variable b. In the following equations, s is a signal variable, b is an array variable, e is a signal expression, and k is a natural number. The definitions use $\mathcal{N}(v)$, the next-state expression for the state variable v. See Figure 4.

For $k \geq 0$, let $\text{size}^k(v, a) = \|\text{approx}^k(v, a)\|$, be the size of the kth approximation. For an output variable v, we will define $\text{size}^k(v, a)$ using the approximations for the state variables:

$$\text{size}^k(v, a) = \|\Phi(\mathcal{E}(v), a, \theta^{a,k}, \theta_A^{a,k})\|,$$

The idea behind the definition of $\text{size}^k(s, a)$ is to make successive overapproximations to the value of $\Sigma(s^k)(a)$. We begin by setting the first approximation, $\text{size}^0(s, a)$, to 0. At each successive step, we make $\theta^k(s)$ be a set of n distinct signal variables if the value of $\text{size}^k(s, a)$ is n. This gives an overapproximation because, for example, there is no sharing of common expressions in $\theta^{a,k}(s_i), \theta^{a,k}(s_j)$, when s_i, s_j are different state variables. At each step we use the next-state expression $\mathcal{N}(s)$ to compute the maximal sets of indices needed for the array variable a in the next expansion of s.

$$\text{approx}^0(s, a) = \{\emptyset\}$$
$$\text{approx}^{k+1}(s, a) = \Phi(\mathcal{N}(s), a, \theta^{a,k}, \theta_A^{a,k})$$
$$\text{approx}_A^0(b, e, a) = \{\emptyset\}$$
$$\text{approx}_A^{k+1}(b, e, a) = \Phi_A(\mathcal{N}(b), e, a, \theta^{a,k}, \theta_A^{a,k})$$
$$\theta^{a,0}(s) = \emptyset$$
$$\theta^{a,k+1}(s) = \{s_1, \ldots, s_n\},$$
$$\quad \text{where } n = \|\text{approx}^{k+1}(s, a)\|$$
$$\quad \text{and } s_1, \ldots, s_n \text{ are distinct fresh signal variables}$$
$$\theta_A^{a,0}(b, e) = \emptyset$$
$$\theta_A^{a,k+1}(b, e) = \{b_{e,1}, \ldots, b_{e,n}\},$$
$$\quad \text{where } n = \|\text{approx}_A^{k+1}(b, 0, a)\|$$
$$\quad \text{and } b_{e,1}, \ldots, b_{e,n} \text{ are distinct fresh signal variables}$$

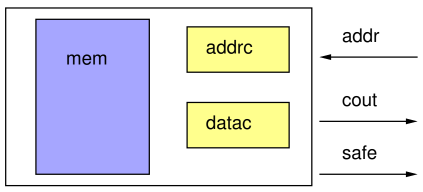
Fig. 4. Definitions of approx^k and θ

The following theorem says that $\text{size}^k(v, a)$ overapproximates $\Sigma(v^k)(a)$.

Theorem 6. For any array variable a, signal variable v, and $k \geq 0$, $\Sigma(v^k)(a) \leq \text{size}^k(v, a)$. □

We can now present an algorithm compute_size for computing a size for an array variable a that makes a sound and complete model for checking a property output variable v of a system. The inputs of compute_size are a system $\mathcal{M}$, a signal variable v of $\mathcal{M}$, an array variable a, and a natural number *OriginalSize*. The value of *OriginalSize* is the size of the array a in the original model. The algorithm computes $\text{size}^k(v, a)$ for increasing values of k, until either a fixed point is reached or $\text{size}^k(v, a) > \textit{OriginalSize}$.

We need to define a set $\text{dep}(v)$ of state variables on which a variable depends. For a state variable v, let $\text{dep}(v)$ be the smallest set of state variables such that 1) $v \in \textit{dep}(v)$, and 2) for all state variables v', if $v' \in \text{dep}(v)$ and v'' is a state variable appearing in $\mathcal{N}(v')$, then $v'' \in \text{dep}(v)$. For an output variable v, we define $\text{dep}(v)$ to be the union of $\text{dep}(v')$ over



VSCM Block Diagram

Fig. 5. Example 3

all state variables v' appearing in $\mathcal{E}(v)$.

Algorithm 1. The algorithm compute_size consists of the following steps.

```
1) $k \leftarrow 0$;  $MaxSize \leftarrow 0$;
2) Until $((\forall v' \in \mathrm{dep}(v) : \mathrm{size}^k(v', a) = \mathrm{size}^{k+1}(v', a)) \vee$
   $\mathrm{size}^k(v, a) > OriginalSize)$ Do
   $\{MaxSize \leftarrow \max(\mathrm{size}^k(v, a), MaxSize); k \leftarrow k+1\}$;
3) If $\mathrm{size}^k(v, a) > OriginalSize$ then return $OriginalSize$;
4) If $\forall v' \in \mathrm{dep}(v) : \mathrm{size}^k(v', a) = \mathrm{size}^{k+1}(v', a)$ then
   return $MaxSize$;
```

The algorithm always terminates, because for $k \geq 0$, $\mathrm{size}^{k+1}(v, a) \geq \mathrm{size}^k(v, a)$. If the algorithm exits with $\forall v' \in \mathrm{dep}(v) : \mathrm{size}^k(v', a) = \mathrm{size}^{k+1}(v', a)$, then it is easy to see that the size computation has reached a fixed point at step k: The value of Φ for each state variable v depends on the previous values of $\theta(v')$ for the variables v' appearing in $\mathcal{N}(v)$. If $\mathrm{size}^k(v', a) = \mathrm{size}^{k+1}(v', a)$ for all v' in the transitive fan-in of v, then the size computation for v is at a fixed point. By Theorem 6, we know that $\forall k \geq 0 : \Sigma(v^k)(a) \leq \mathrm{size}^k(v, a)$. The variable $MaxSize$ is now set to a value that is at least as large as any of the approximate values: $\forall k \geq 0 : \Sigma(v^k)(a) \leq MaxSize$. By Theorem 2 , we can construct a sound and complete abstraction for evaluating v in $\mathcal{M}$ by using compute_size to assign the size of each array variable.

In a more extended presentation, we would show how to improve the accuracy of the size computation by taking shared expressions into account in θ, θ_A.

Example 3. We illustrate Algorithm 1 by showing the analysis of a VSCM (very simple cache memory) unit. The block diagram of the VSCM is shown below. The state variables are $\texttt{mem}, \texttt{addrc}, \texttt{datac}$, the input variable is `addr`, and the output variables are $\texttt{cout}, \texttt{safe}$. The array `mem` represents a large main memory. Arrays `addrc` and `datac` are small arrays that form the cache. The array `addrc` stores the addresses that are cached, while `datac` stores the data for these addresses. The next state function $\mathcal{N}$ and output function $\mathcal{E}$ are shown in Figure 6. A unary operator `key` maps a full address into a value $\texttt{key}(\texttt{addr})$ that is an index into the arrays `addrc` and `datac`. On each clock cycle the cache inputs the signal `addr` and outputs the value of $\texttt{mem}[\texttt{addr}]$. If $\texttt{addrc}[k] = 0$, for some k, then the data at location k in the cache is considered invalid. Initially, $\texttt{addrc}[k] = 0$ for all k in the domain of `addrc`. If $\texttt{addrc}[\texttt{key}(\texttt{addr})] = \texttt{addr} \wedge \texttt{addr} \neq 0$, then the required memory data is provided from the cache by accessing $\texttt{datac}[\texttt{key}(\texttt{addr})]$. Otherwise, the data is fetched from main memory at $\texttt{mem}[\texttt{addr}]$, and $\texttt{addrc}, \texttt{datac}$ are updated. The output signal `cout` is the data output of the cache memory, and the signal `safe` asserts that $\texttt{cout} = \texttt{mem}[\texttt{addr}]$ holds.

The table summarizes the operation of Algorithm 1 for each of the three arrays. The numbers in the table show the values of $\mathrm{size}^k(v, a)$, for each of the state variables. For the array $a = \texttt{mem}$, the algorithm reaches a fixed point at $k = 2$, with the array `datac` using one index value and the other arrays using no indices. For the output signals, `cout` uses one index value and `safe` uses two index values, because the expression for `safe` has subexpressions `cout` and $\texttt{mem}[\texttt{addr}]$. Therefore the abstract model for the output signal safe will reduce `mem` to two entries. For the array $a = \texttt{datac}$, the algorithm reaches a fixed point with one array index. For the array `addrc`, the algorithm is unable to reduce the size of the array, and the original size of the array will be used in the abstract model. At each iteration, $\mathrm{size}^k(\texttt{datac}, \texttt{addrc})$ increases by one, because in the next-state expression for `datac`, the *mux* expression has a control expression (e_1) that uses one array index of `addrc`, and a second input (e_2) that uses k indices of `datac` on iteration k. The approximation computed by Φ therefore uses $k + 1$ array indices at step $k + 1$.

Note that data is read from the array `mem` and stored in `datac` for an unbounded length of time. Our algorithm abstracts `mem`, while the method of [7] cannot.

V. Construction of the Abstract Model

It is straightforward to build a model of a system that uses the semantics with the bottom value. We replace each signal of the original design with a composite signal having two elements: a `value`, which represents the value of the signal in the original design, and a `v` bit, which is true if the signal represents a value in V, and false if the signal represents $\perp$. Each state register is replaced with a register of the composite type. The signal operations of the original circuit are replaced with versions of the operations that recognize the value $\perp$.

The size of each array in the abstract model is determined by running Algorithm 1. An abstract array of size n is implemented using two arrays: an array of n addresses, and an array of n $(\texttt{value}, \texttt{v})$ pairs. The contents of the address array are set nondeterministically in the initial state of the system, and do not change over clock cycles. The array read and write operations are implemented according to the semantics of expressions.

If `p` is an output signal for a safety property $\texttt{p} = \texttt{true}$, then we construct a property expression of the form $\texttt{p.v} \rightarrow \texttt{p.value} = \texttt{true}$. Finally, we use model checking to verify that each of the new output property expressions is always true.

VI. Industrial Examples

In this section, we present initial results of using our abstraction algorithm on industrial hardware designs. The algorithm has been implemented in IBM's model checker.

Many of the arrays used in hardware designs have the property that at each time step, the output signal only depends on a small, bounded number of elements of the array. When an array has this property, our algorithm is often able construct

$\mathcal{N}(\texttt{mem}) = \texttt{mem}$
$\mathcal{N}(\texttt{addrc}) = \mathit{write}(\texttt{addrc}, \texttt{key(addr)}, \mathit{mux}(\texttt{addrc[key(addr)]} = \texttt{addr}, \texttt{addrc[key(addr)]}, \texttt{addr})$
$\mathcal{N}(\texttt{datac}) = \mathit{write}(\texttt{datac}, \texttt{key(addr)}, \mathit{mux}(\texttt{addrc[key(addr)]} = \texttt{addr}, \texttt{datac[key(addr)]}, \texttt{mem[addr]}))$

$\mathcal{E}(\texttt{cout}) = \mathit{mux}(\texttt{addr} \neq 0 \wedge \texttt{addrc[key(addr)]} = \texttt{addr}, \texttt{datac[key(addr)]}, \texttt{mem[addr]})$
$\mathcal{E}(\texttt{safe}) = (\texttt{cout} = \texttt{mem[addr]})$

Computation of $\text{size}^k(v, a)$

array $a = \texttt{mem}$				array $a = \texttt{datac}$				array $a = \texttt{addrc}$			
v	addrc	datac	mem	v	addrc	datac	mem	v	addrc	datac	mem
$k=0$	0	0	0	$k=0$	0	0	0	$k=0$	0	0	0
$k=1$	0	1	0	$k=1$	0	0	0	$k=1$	0	1	0
$k=2$	0	1	0					$k=2$	0	2	0
$\text{size}^2(\texttt{cout}, \texttt{mem}) = 1$				$\text{size}^1(\texttt{cout}, \texttt{datac}) = 1$				$k=3$	0	3	0
$\text{size}^2(\texttt{safe}, \texttt{mem}) = 2$				$\text{size}^1(\texttt{safe}, \texttt{datac}) = 1$				$k=n$	0	n	0

Fig. 6. Very Simple Cache Memory Definitions and Analysis

a small abstract model. On the other hand, there are common uses for arrays that do not have the necessary property. For instance, when an array is used as a content-addressable memory (CAM), each read operation accesses all elements of the array, and the array cannot be abstracted by our current approach.

The following results should be considered preliminary, because the implementation is in development. Data was collected for a set of 401 complex industrial examples. Many of the arrays in these designs have the property needed for our abstraction, but many of the arrays are used as CAMs, and hence are difficult to abstract. Individual designs in the set contain from one array up to several hundred arrays. Overall, our algorithm reduced the size of at least one array in 187 designs, or about 47 per cent of designs. The following table gives the total over all examples of the number of reduced arrays for each original and reduced size.

	Reduced Number of Rows						
Original Rows	1	2	3	4	6	8	> 8
2	144						
8	1	1					
16	14	13	55				
32	37	1	25				
39	24						
48	24						
64	46	29	20	18			
128	4	158	14	23	1	11	
256	3	40	10				
1024	3		10				2

The final version of the paper will compare the performance of other array abstraction algorithms.

One kind of verification problem where our techniques are valuable is proving sequential equivalence of two designs where an array has been reconfigured. In designing complex hardware systems, it is often necessary to reconfigure an array into two or more smaller arrays, due to physical circuit constraints. In simple cases, the reconfiguration consists of dividing an array into two arrays with the same number of index values (rows) as the original array, but narrower data values (fewer columns). For this kind of reconfiguration, it is often possible to prove the designs to be equivalent by automatically discovering a correspondence between the data columns of the original and reconfigured arrays [9].

When reconfiguration involves changing the number of rows in an array, it is harder to prove equivalence, because the two designs have differences in the addressing, data alignment and staging logic.

One real example that highlights the advantage of our techniques over previous approaches is an equivalence check where the original design has an array of 1024 rows by 16 columns, and the reconfigured design has two arrays, each with 128 rows by 64 columns. In this case, the logic near the arrays was substantially redesigned. Because the design uses clock gating, the method of [7] cannot reduce the size of the arrays. Our approach generates an abstract model and verifies equivalence, using four modeled rows from the large array of the original design and one row from each of the smaller arrays in the reconfigured design. The abstract model uses a total of 401 registers, including the three arrays and surrounding control logic. Without using our algorithm, we have found no way to verify this example other than to bit-blast the model into 32912 registers.

VII. Related Work

The work most closely related to our approach is by Bjesse [7]. Both our approach and [7] transform a register transfer level design into an abstract register transfer level design having smaller arrays, and allow any register transfer level verification method to be used on the abstract design. Both approaches use nondeterminism to choose which addresses are modeled in the abstract design. In our approach, the abstract model uses a semantics with a bottom value. The semantics limits evaluation of the correctness property to cases in which the nondeterministically chosen addresses are sufficient to determine the truth of the property. In [7], the correctness property at time t is made conditional on a formula saying that array read operations accessed only modeled addresses at a list of previous time steps. Our approach is effective for reasoning about systems in which a value read from an array can affect the correctness property after an unbounded time delay. In [7], it is inherent in the construction of the abstract models, that reasoning is effective only in cases where there is a bound on the number of time steps after reading a value from an array that the value can affect the correctness property.

Several works [1]–[3] develop approaches for reasoning about systems with arrays by modeling the initial value and data forwarding properties of arrays operations over a bounded number of time steps. BAT [4] is another tool that builds abstractions for arrays over bounded time intervals. BAT uses several term-level techniques to reduce the size of abstract models of arrays before constructing a propositional model. These techniques include term-level uniqueness reductions and memory rewriting.

Model checkers in industry use a diversity of algorithms to analyze hardware designs. Baumgartner *et al* [9] describe enhancements to a number of algorithms in an industrial model checker, to provide more efficient processing by abstracting or simplifying arrays.

McMillan [10] developed a method of compositional model checking in which arrays can be abstracted to a small number of elements by temporal case splitting and symmetry reduction. In [10], the user proves complex designs by manually specifying a set of lemmas; the lemmas are checked automatically. In contrast, our method is directed towards fully automatic verification.

VIII. Discussion

We have introduced a logic of expressions for reasoning about arrays and developed some of its mathematical properties. The semantics of the logic permits reasoning about the value of an expression, when evaluated over states having arrays of different sizes. In Section III, we show that the truth of an expression can be evaluated over a state that may have smaller array sizes than the original model. The existence of adequate model sizes for expressions leads immediately to the existence of adequate model sizes for safety properties of systems. However, to compute the adequate model size directly from the results of Section III could be as difficult as verifying the original design. For this reason, we propose a method of safely overapproximating the minimum adequate size in Section IV. Our algorithm represents approximate sets of array indices using two-level sets of expressions. When using iteration on two-level sets, special care is needed to detect fixed points. Our algorithm constructs a most-general element from each two-level set at each step in the iteration as a way of detecting when a fixed point has been reached.

Although our main focus is on sequential systems, our results could also be useful for checking the satisfiability of formulas in the theory of arrays. Theorem 4 gives a way to overapproximate the number of array indices needed to check satisfiability. Our approximation could lead to improvements to model-based approaches to checking validity.

There exist many possible ways of overapproximating the minimum size of arrays defined in our theory. It should be possible to improve the approximation by using stronger methods to identify common subexpressions for array indices. A further improvement would be to identify expressions that are syntactically distinct but logically equivalent.

Acknowledgments The author gives special thanks to Jason Baumgartner for many helpful discussions, and for implementing the algorithm. Jessie Xu made helpful comments on the paper.

References

[1] M. Velev, R. E. Bryant, and A. Jain, “Efficient modeling of memory arrays in symbolic simulation,” in *Proceedings of Computer Aided Verification 1977*, ser. LNCS, vol. 1254. Springer, pp. 388–399.

[2] M. K. Ganai, A. Gupta, and P. Ashar, “Efficient modeling of embedded memories in bounded model checking,” in *Proc. of Computer Aided Verification, 2004*, ser. LNCS, vol. 3114. Springer, pp. 272–274.

[3] ——, “Verification of embedded memory systems using efficient memory modeling,” in *Proceedings of DATE Europe ’05*. IEEE Computer Society, 2005, pp. 1096–1101.

[4] P. Manolios, S. K. Srinivasan, and D. Vroon, “Automatic memory reductions for rtl model verification,” in *Proceedings of ICCAD ’06*. ACM, pp. 786–793.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic model checking without bdds,” in *TACAS*, 1999, pp. 193–207.

[6] M. Sheeran, S. Singh, and G. Stålmarck, “Checking safety properties using induction and a sat-solver,” in *Formal Methods in Computer-Aided Design 2000*, pp. 108–125.

[7] P. Bjesse, “Word-level sequential memory abstraction for model checking,” in *Formal Methods in Computer-Aided Design ’08*, pp. 16:1–16:9.

[8] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual*. Springer, 2007.

[9] J. Baumgartner, M. Case, and H. Mony, “Coping with Moore’s law (and more): Supporting arrays in state-of-the-art model checkers,” in *Formal Methods in Computer-Aided Design 2010*, pp. 61–69.

[10] K. L. McMillan, “Verification of infinite state systems by compositional model checking,” in *CHARME*, 1999, pp. 219–234.