# A Formalism and a Verification Method for Bus-Based Systems

Felice Balarin

Cadence Berkeley Laboratories, Berkeley, USA

Bus-based systems consist of many similar components which communicate through a fixed channel. We propose a formal model for such systems, using finite-state automata to model behavior of components, and logic formulas to model bus arbitration. For this purpose, we define the *indexed propositional logic*, show that it has a small model property, and use that property to construct an abstraction of bus-based systems which does not depend on the actual number of components. We show that the proposed formalism can be used to model buses at various levels of abstraction.

*Keywords:* formal verification, automata, iterative arrays, abstraction

## 1. Introduction

Verification is an important component of any design methodology. A design error that is discovered late in the design process can significantly increase the cost of a project. A design error discovered after a system is put in use is usually even costlier, either in the form of expensive product recalls and customer dissatisfaction, or in the worst case, in form of failure of life-critical systems such as flight controllers.

Presently, simulation is the prevailing method for verifying digital systems. But as technology advances and systems become more complex, verification by simulation is becoming increasingly insufficient. An illustrative example was reported by Chen, Yamazaki and Fujita [12]. A data-switching chip (assumed to be correct by the designers) was exhibiting incorrect behavior, but only sporadically, and only after several seconds of operation at 156MHz. Simulating such long input patterns is clearly very expensive, and the probability of selecting one that exhibits faulty behavior is quite low. However, by using a formal verification tool Chen et al. were able to identify the fault. The tool also generated a short (50 cycles) input pattern that exhibits the fault.

In the future, simulation is likely to remain an important verification method, but it is also likely that it will be more and more supplemented by formal verification methods. Formal verification is particularly effective in early design phases, when the system is described at a high level of abstraction, hence is less complex than later refinements, and must satisfy properties that are typically simple and easy to specify formally.

The phrase "formal verification" is broadly used to indicate any technique where one, with a rigor of a mathematical proof, establishes ( or disproves) a *satisfaction* relation between a *design* and a *specification*. Formal verification paradigms vary with a choice of formalisms for the design and specification, and a choice of a satisfaction relation (see [17] for a comprehensive survey and exhaustive list of references). The basic tradeoff is between expressiveness and efficiency. On one side of the spectrum are general theorem proving techniques (e.g. [9, 15, 2]), which are very expressive, but are provably hard, and can be only partially automated. On the other side are finite-state techniques that suffer from somewhat limited expressiveness, but can be completely automated, and precisely for that reason have attracted significant interest recently.

## 1.1. Formal verification of finite-state systems

Two dominating approaches to automatic verification of finite-state systems have emerged: *model checking* based on temporal logics, and *language containment* based on automata theory. In the model checking approach a system is verified if it represents a model of a given formula in some temporal logic. Most of the logics proposed for specification and verification of finite-state systems can be classified as either *branching time* or *linear time* temporal logics (see [14] for excellent survey). The precise formulation of the model checking problem is somewhat different for each class.

Models of branching time temporal logics are trees, but their subformulas reason about paths in a tree. Typically, a subformula may postulate that another formula holds in every (or some) state along a path. A formula is then closed by existentially (or universally) quantifying over paths. In other words, a formula is true of the tree rooted in a state, if the subformula is true on some (or all) paths from that state. With every finite-state system we associate an infinite tree with the initial state as its root, and children of every state being its possible next states. A system is verified, if a formula is true of the associated tree. The first efficient automatic model checking procedure for any temporal logic was given Clarke, Emerson and Sistla [13] for the branching time logic CTL, which still remains the most widely used logic in verification. For example, verification systems SMV [25], and HSIS [4] are based on it.

Formulas of linear time temporal logics reason about paths, and the system is verified if the formula is true for all paths from the initial states. The use of linear time temporal logics for specification and verification of finite-state systems dates back to late seventies and early eighties when in a trail-blazing contribution Pnueli and Manna [27, 24] use such a logic to specify many interesting properties of computer programs. Linear time temporal logics are still an interesting research topic, but their inherent complexity [32] has limited their use in practice.

Linear temporal logics are closely related to another popular verification paradigm known as *language containment* [33, 23]. Here, systems are model as *automata*, and the *language* accepted by an automaton is assumed to be its behavior. In other words, the behavior of a finite state system is a set of input-output sequences that can be observed from the outside. The verification problem is to check whether every sequence in the language is acceptable, i.e. we need to check that the language of a system is contained in the language consisting of all acceptable sequences. That language is specified by another automaton, so the verification problem reduces to checking language containment between two automata.

For example, assume that the system being designed is a traffic light controller. The language of the system consists of all possible sequences of lights generated by the controller. There may be more than one such sequence if the system is designed to adapt to the traffic flow. One property we may want to check is that at no times the light is green in both directions. To verify this property we first construct an automaton whose language contains all sequences *except* those where lights are green in both directions at some point in time (incidentally, this is a very simple, two-state automaton), and then check if the language of the system is contained in the language of this automaton.

Model checking and language containment are not as different as they may appear at first. Indeed, model checking of linear temporal logics can be reduced to language containment [34, 33]. First, we construct an automaton that accepts all the sequences that satisfy the formula. Then, we check whether the language of that automaton contains the language of the system. Similarly, one can use automata on trees to formulate automata-theoretic approach to branching-time model checking [8].

## 1.2. Problems with formal verification

Even though the verification of finite-state systems can be completely automated, it is still not widely used in practice, mainly due to issues related to specification of correct behavior, design methodology, and complexity of the associated algorithms.

Before using a formal verification tool, a user must first specify acceptable behaviors (in case of language containment, the acceptable behaviors are represented by a language, and in case of model checking, it is represented by a formula). In the worst case, specifying acceptable behaviors is as hard and as error-prone as actually designing a system. Fortunately, specification is often much simpler than design for at least the following reasons:

- It is often possible to express *what* a system must do (a specification) much more concisely than *how* it is doing it (a design).
- Acceptable behaviors can usually be specified as a list of *properties* that a system must satisfy. Even though properties are typically simple, designing a system that satisfies all of them is not.

To reduce the possibility of errors, the properties must not only be simple, but also expressed in a way that is natural to designers and consistent with other system documentation. Unfortunately, neither finite-state automata, nor CTL formulas satisfy these conditions. A promising approach to this problem is to develop translators that provide formal interpretation (in terms of automata or temporal logic formulas) of specification methods used by designers, such as HDL annotations [26, 3, 6] or timing diagrams [30].

Another issue that has to be addressed before automatic formal verification is widely accepted is the development of a verification based design methodology. This includes engineering issues like providing a common design representation to be used by synthesis, simulation and verification tools, theoretical issues like identifying properties that are preserved under a given set of synthesis operations, as well as organizational issues such as deciding at which points in the design cycle simulation or verification tools should be used to optimize the overall design process. Some of these issues have been addressed in the literature [4, 6, 23], but many practical questions have yet to be resolved.

Last but not least, a formal verification method can be successful only if accompanied with elaborate complexity management techniques. At first glance, it might seem that complexity is not a significant issue because algorithms polynomial in the number of states exist both for CTL model checking and for language containment. However, the problem is that in practice, systems to be verified are never specified by explicitly enumerating all states. Typically, they are specified as a composition of several components which are specified either as software (a program in some language), or as hardware (combinational logic plus latches). In this case, the number of states can be exponential in the size of the description, and in fact both CTL model checking and language containment of systems consisting of interacting components are PSPACE-complete [5]. This is the well known *state explosion problem*.

Attacking the state explosion problem is the focus of a wide range of research. Two approaches dominate: *symbolic computation* where one tries to manipulate a large number of states efficiently by representing sets of states symbolically (rather than by enumeration), and *simplification* where one argues about the correctness of complex systems by verifying their simplified versions. Those two approaches are independent, and in fact, several researcher have proposed methods that combine both approaches.

By far the most widely used approach to symbolic computation is to represent sets of states with their characteristic functions. If states are encoded with binary variables, then the characteristic function of a set is just a Boolean function, and is typically represented by a *binary decision diagram* (BDD) [10]. This approach is successful because in many practical cases large sets of states have quite small BDD representations, and thus it is possible to manipulate sets of states that are too large to enumerate with existing computing resources. BDD-based algorithms (and verification systems) are available both for model checking [25] and language containment [4]. The success of BDD-based approaches have made them almost synonymous to symbolic computation. Still, not all symbolic approaches are BDD-based. For example, in the verification of real-time and hybrid systems, sets of linear inequalities are used to represent convex polyhedra bounded by them [19, 1].

Recent advancements in symbolic computation techniques have significantly increased the capabilities of automatic formal verification, but simplifications are still necessary to handle most real-life systems. We make a distinction between two kinds of simplifications: *exact* and *conservative*. Exact simplifications (or *reductions*) preserve all aspects of system behavior, therefore the original system is verified *if and only if* the simplified system is. They can be applied to virtually any verification formalism, but it is often hard to find an exact simplification of reasonable size.

On the other hand, conservative simplification ( or *abstractions*) preserve enough behavior of a system to guarantee that the original system is verified *if* the simplified system is. However, if the simplified system is not verified, the original system might or might not satisfy the required property. Conservative approximations can often lead to much larger savings than exact simplifications, but they exist only for some formalisms. In particular, if the set of properties expressible in a formalism is closed under complementation (i.e. if for every property $P$ there exists a property $\overline{P}$ such that a system $S$ satisfies $P$ if and only if it does not satisfy $\overline{P}$), then every abstraction is also a reduction. To see this, assume towards a contradiction that $S'$ is an abstraction but not a reduction of a system $S$, i.e. assume that there exists a property $P$ such that $S$ satisfies $P$ but $S'$ does not. This implies that $S'$ satisfies $\overline{P}$ even though $S$ does not, contradicting the assumption that $S'$ is an abstraction of $S$.

Most of branching time temporal logics (including CTL) are closed under complementation, hence they allow only exact simplification. However, if the logic is restricted to universal path quantifiers (and of course no negation), then conservative simplifications are possible. Roughly speaking, any system with more paths is an abstraction of the original system. For example, Grümberg and Long [16] have studied such a restriction of CTL (called ACTL), and showed that one system is an abstraction of another, if the so-called *simulation relation* holds between them.

The language containment (hence also model checking of linear time temporal logics) clearly allows conservative approximations: an automaton is an abstraction if its language contains the language of the original system. But that means that checking whether one system is an abstraction of the other is just another instance of the original verification problem (and hence just as hard). Still, a careful use of abstractions can be beneficial. For example, one might check abstractions of (usually small) components, and deduce from that an abstraction of the (usually large) complete system. These and other strategies for simplification of communicating automata were studied by Kurshan [21, 23].

## 1.3. Bus-based systems

Another approach to managing the complexity of verification is to define a procedure, and prove (usually manually) once and for all, that it always generates abstraction. In this paper we define such a procedure for a class of systems called *bus-based systems*. In general, systems consisting of many identical components are called *networks* or *iterative systems*. If, in addition, all the communication data is available to all the components, and the width of the communication path does not change with the number of components, such networks are called *bus-based*.

Typically, iterative arrays are designed to work correctly regardless of the actual number of components. For example, mutual exclusion algorithm, token passing and bus protocols should all work for any number of participating processes. Ideally, an abstraction of such a system should not depend on the actual number of components. Such an abstraction is called a *network invariant*. Once an invariant of manageable size is found it allows:

- verification of a large system with a fixed number of components; and at the same time also
- verification of the entire class of systems with the same structure but with different number of components.

A network invariant is generally a conservative approximation. If an invariant happens to be an exact approximation, we say that it is *tight*. If a tight invariant is verified, then so is every system in the class, but if a tight invariant is not verified, then there exists at least one system in the class which exhibits undesirable behavior.

### 1.4. Contributions and related work

In this paper, we propose a new formalism to model bus-based systems, based on use of automata to model basic components, and the use of logic formulas to model bus arbitration. For this purpose, we define a logic called *indexed propositional logic* (IPL). We show that IPL has a small model property, and use that property to construct size-independent abstraction of bus-based systems. We show that the proposed formalism can be used to model buses at various levels of abstraction.

Although iterative systems have been studied for a long time [18], only recently there has been a significant interest in the formal verification of such systems. Browne, Clarke and Grümberg [11], and Shtadler and Grümberg [31] have studied conditions under which the satisfaction of formulas of certain temporal logics is independent of the size of the system. In [31] the conditions seem to be quite restrictive, while in [11] the conditions cannot in general be checked automatically. Wolper and Lovinfosse [35] have studied formal verification of iterative systems generated by interconnecting identical processes in certain regular fashion. They also present some decidability results for related problems. Kurshan and McMillan [22] present slightly more general results which can be applied both to process algebra and automata-based approaches. In both cases, automatic tools are used only to verify that a finite state system suggested by the user is indeed an invariant. Automatic invariant generation is considered in [7, 28, 29], but the proposed procedure may not always terminate.

Our work is unique, because it provides for automatic generation of a network invariant in finite time. Unfortunately, the invariant is not tight, in the sense that it is possible for the generated invariant not to satisfy some property, even though all the instances do. However, this is the best we can hope for, because a tight finite-sate invariant does not always exist (see Theorem 2).

In the rest of this paper, we first review pertinent elements of the automata theory in section 2. Then, we introduced the logic IPL in section 3. In section 4 we present a complete formal model for bus-based systems, and a procedure for automatic generation of an invariant. The application of this formalism to modeling of buses is described in section 5. Final remarks are given in section 6.

## 2. Automata theory

In this section we review only the most pertinent elements of the automata theory. The complete treatment can be found in numerous references including [20].

### 2.1. Automata

A *finite-state automaton* over some alphabet $\Sigma$ is a 4-tuple $(S, I, T, F)$, where $S$ is some finite and non-empty set of *states*, $I \subseteq S$ is the set of *initial states*, $T \subseteq S \times \Sigma \times S$ is the *transition relation*, and $F \subseteq S$ is the set of *final states*. Automata are often visualized as graphs with one node for every state, and an edge from node $s$ to node $q$ labeled $x$ for every $(s, x, q) \in T$. Consistent with this interpretation, we will use $s \xrightarrow{x} q$ to denote the element $(s, x, q)$ of a transition relation. An automaton is said to be *deterministic* if it has a unique initial state, and for every $s \in S$ and every $x \in \Sigma$ there exists <u>at most</u> one $q$ such that $s \xrightarrow{x} q \in T$. An automaton is said to be *complete* if for every $s \in S$ and every $x \in \Sigma$ there exists <u>at least</u> one $q$ such that $s \xrightarrow{x} q \in T$.

### 2.2. Language

Given some finite-state automaton $A$, the string $(x_1 \ldots x_n) \in \Sigma^*$ is said to be in the *language* of $A$ (denoted by $\mathcal{L}(A)$) if there exists a sequence $s_0, \ldots, s_n$ of states (called an *accepting run*) such that $s_0$ is an initial state, $s_n$ is a final state, and $s_{i-1} \xrightarrow{x_i} s_i$ is in the transition relation for all $i = 1, \ldots, n$. Given any automaton $A$, it is easy to construct an automaton such that it is complete and has the same language as $A$. Such an automaton is basically the same as $A$, except possibly for one extra state. It is also possible to construct an automaton that is deterministic and has the same language as $A$ (through the procedure known as subset construction [20]), but this automaton may be exponentially larger than $A$.

### 2.3. Composition, union, complement

Given two automata $A = (S_A, I_A, T_A, F_A)$ and $B = (S_B, I_B, T_B, F_B)$ their *composition* is defined by:

$$A \| B = (S_A \times S_B,\ I_A \times I_B,\ T_{A\|B},\ S_A \times S_B) \ ,$$

where:

$$T_{A\|B} = \{(s,s') \xrightarrow{x} (q,q') \mid s \xrightarrow{x} q \in T_A$$
$$\text{and } s' \xrightarrow{x} q' \in T_B\} \ .$$

The *union* of two automata is defined by:

$$A \cup B = (S_A \cup S_B,\ I_A \cup I_B,\ T_A \cup T_B,\ S_A \cup S_B) \ .$$

The complement of an automaton $A=(S,I,T,F)$ is defined by:

$$\overline{A} = (S,\ I,\ T,\ \{s \in S \mid s \notin F\}) \ .$$

**Proposition 1.** $\mathcal{L}(A\|B) = \mathcal{L}(A) \cap \mathcal{L}(B)$.

**Proposition 2.** $\mathcal{L}(A \cup B) = \mathcal{L}(A) \cup \mathcal{L}(B)$.

**Proposition 3.** *If $A$ is deterministic and complete, then:*

$$\mathcal{L}(\overline{A}) = \overline{\mathcal{L}(A)} = \{(x_1 \dots x_n) \in \Sigma^* \mid$$
$$(x_1 \dots x_n) \notin \mathcal{L}(A)\} \ .$$

Typically, components of digital systems are modeled by automata where states correspond to internal states of the system, and the alphabet corresponds to values of externally visible signals. The observable behavior is then represented by the language of the automaton. The interaction of two components is modeled by their composition: the global transition of the system consists of the transitions in all the components, and all the components must agree on the values of common signals.

To verify an automaton $A$ means to check whether its language is contained in the language of some other *"property"* automaton $P$. It follows from propositions 1 and 3 that if $P$ is deterministic and complete, then checking whether $\mathcal{L}(A) \subseteq \mathcal{L}(P)$ is equivalent to checking whether $\mathcal{L}(A\|\overline{P})$ is empty. To check the emptiness of the language of some automaton, it is enough to check for existence of a path from some of its initial states to some of its final states. This can be done by standard graph searching techniques, in time proportional to the size of transition relation.

## 2.4. Quotient

Given some (not necessarily finite-state) automaton $A = (S,I,T,F)$ over alphabet $\Sigma$, and some equivalence relation $\sim$ on $S$, the *quotient of $A$ with respect to $\sim$* (denoted by $A/\sim$) is the automaton $(S',I',T',F')$ over the same alphabet, where:

- $S' \subseteq S$ is some set of representatives of equivalence classes induced by $\sim$, i.e. $S'$ is such that:
  1. for every $s \in S$ there exists $q \in S'$ such that $s \sim q$, and
  2. if $s$ and $q$ are two distinct elements of $S'$, then $s \sim q$ does not hold,
- $I' = \{s' \in S' | s' \sim s \text{ for some } s \in I\}$,
- $T' = \{(s' \xrightarrow{x} q') \in S' \times \Sigma \times S' \mid s' \sim s$ and $q' \sim q$ for some $(s \xrightarrow{x} q) \in T\}$,
- $F' = \{s' \in S' | s' \sim s \text{ for some } s \in F\}$.

Note that even though there are many possible choices of $S'$, the corresponding automata differ only in state names.

**Proposition 4.** $\mathcal{L}(A) \subseteq \mathcal{L}(A/\sim)$.

*Proof.* Let $s_0, \dots, s_n$ be an accepting run in $A$ of some string $x_1 \dots x_n$, and for all $i = 0, \dots, n$ let $\tilde{s}_i$ denote a representative of $s_i$ in $S'$, i.e. a unique element of $S'$ satisfying $\tilde{s}_i \sim s_i$. It follows from the definition that $\tilde{s}_0, \dots, \tilde{s}_n$ is an accepting run of $x_1 \dots x_n$ in $A/\sim$. □

## 2.5. Iterative arrays

An iterative array can be thought of as an infinite sequence $A_1, A_2, \dots$ of finite states automata, where each $A_i$ represents the behavior of the system consisting of exactly $i$ instances of the basic component. A network invariant is any automaton whose language contains languages of all $A_i$'s. The language of a tight invariant must be exactly $\bigcup_{i=1}^{\infty} \mathcal{L}(A_i)$. If $\bigcup_{i=1}^{\infty} \mathcal{L}(A_i)$ is not regular, then no finite-state automaton can have such a language [20]. However, one may try to find a "tightest regular invariant", i.e. an invariant that is not tight, but that has the language that is contained in the languages of all other invariants. Unfortunately, if $\bigcup_{i=1}^{\infty} \mathcal{L}(A_i)$ is not regular, such an invariant does not exist, as shown by the following result:

**Proposition 5.** *Let $\mathcal{L}_1$ be some non-regular language, and let a finite-state automaton $A$ be such that $\mathcal{L}_1 \subset \mathcal{L}(A)$. Then, there exists a finite-state automaton $A'$ such that:*

$$\mathcal{L}_1 \subset \mathcal{L}(A') \subset \mathcal{L}(A) \ .$$

*Proof.* Let $x_1 \ldots x_n$ be some string in $\mathcal{L}(A)$ which is not in $\mathcal{L}_1$ (since $\mathcal{L}(A)$ is regular and $\mathcal{L}_1$ is not, such a string always exists), and let $X$ be a finite-state automaton such that $\mathcal{L}(X) = \{(x_1 \ldots x_n)\}$ (any language containing a single string is regular, thus it is the language of some finite state automaton). It follows from propositions 1 and 3 that

$$\mathcal{L}_1 \subset \mathcal{L}(A \| \overline{X}) \subset \mathcal{L}(A) \ .$$

□

## 3. Indexed propositional logic

Given a set of *index variables* $Ind = \{i, j, .....\}$, and a finite set of *states* $S$ (which act as monadic predicates in our logic), *well formed formulas* (WFF's) are defined recursively as follows:

- $s(i)$ and $i = j$ are WFF's for any $s \in S$, and any $i, j \in Ind$.
- if $f$ and $g$ are WFF's, then so are also $f \wedge g$ (conjunction), $\overline{f}$ (negation), and $\exists i.f$ (existential quantification), where $i$ is some index variable.

We will also freely use standard abbreviations $\forall i.f$, $f \vee g$, and $f \Longrightarrow g$, with their usual meanings: $\overline{\exists i.\overline{f}}$, $\overline{\overline{f} \wedge \overline{g}}$, and $\overline{f} \vee g$, respectively. A WFF is closed if all variables appearing in it are quantified. IPL formulas are closed WFF's.

Models of WFF's are $n$-tuples of states. Whether a $n$-tuple satisfies a WFF is defined relative to an *interpretation* $\mathcal{I}$ which assigns a number between 1 and $n$ to every indexed variable. Formally, given a WFF $f$, a $n$-tuple $\sigma : \{1, 2, \ldots, n\} \mapsto S$, and an interpretation $\mathcal{I} : Ind \mapsto \{1, 2, \ldots, n\}$, the *satisfaction with respect to* $\mathcal{I}$ (denoted by $\models_{\mathcal{I}}$) is defined recursively as follows:

- $\sigma \models_{\mathcal{I}} s(i)$ iff $\sigma(\mathcal{I}(i)) = s$,
- $\sigma \models_{\mathcal{I}} i = j$ iff $\mathcal{I}(i) = \mathcal{I}(j)$,
- $\sigma \models_{\mathcal{I}} f \wedge g$ iff $\sigma \models_{\mathcal{I}} f$ and $\sigma \models_{\mathcal{I}} g$,
- $\sigma \models_{\mathcal{I}} \overline{f}$ iff it is not the case that $\sigma \models_{\mathcal{I}} f$,
- $\sigma \models_{\mathcal{I}} \exists i.f$ iff $\sigma \models_{\mathcal{I}'} f$, for some interpretation $\mathcal{I}' : Ind \mapsto \{1, 2, \ldots, n\}$ satisfying $\mathcal{I}'(j) = \mathcal{I}(j)$ for all $j \neq i$.

It is straightforward to show that if $\sigma \models_{\mathcal{I}} f$, and $f$ is a closed WFF, then $\sigma \models_{\mathcal{I}'} f$ for any interpretation $\mathcal{I}'$, so without a loss of generality, we can write $\sigma \models f$.

Given the set of states $\{a, b\}$, some examples of WFF's are:

$$a(i) \ , \tag{1}$$
$$\exists i.(i = j) \wedge (i = k) \ , \tag{2}$$
$$\forall i.\forall j.(b(i) \wedge b(j)) \Longrightarrow (i = j) \ , \tag{3}$$
$$\exists i.\exists j.\exists k.\exists l.\Big((a(i) \wedge a(j) \wedge \overline{(i = j)}) \vee (b(k) \wedge b(l) \wedge \overline{(k = l)})\Big) \ . \tag{4}$$

WFF's (3) and (4) are also IPL formulas. Their informal meanings are that *"there exists at most one b in a tuple"*, and that *"there are at least two a's or two b's in a tuple"*, respectively. For example, they are both satisfied by the tuple $(a, b, a, a)$. The same tuple also satisfies (1) and (2) with respect to the interpretation $i = j = k = 3$, but does not satisfy either of the two with respect to the interpretation $i = j = 2$, $k = 1$.

For a given WFF $f$, define a relation $R_f$ on the set of index variables, as follows:

$$R_f \ = \ \{(i,j) \mid i = j \text{ is a subformula of } f\} \ ,$$

and let $R_f^*$ be reflexive, symmetric and transitive closure of $R_f$. Obviously, $R_f^*$ is an equivalence relation, and thus it induces a partition of $Ind$ into equivalence classes. Let the *order* of a WFF $f$ (denoted by $ord(f)$) be the cardinality of the largest equivalence class of $R_f^*$. Finally, let $[R_f^*]_i$ denote the set $\{j \neq i | (i,j) \in R_f^*\}$. Intuitively, if $j$ is in $[R_f^*]_i$, then the truth value of $f$ depends on whether or not $i = j$, either explicitly, because $i = j$ is a subformula of $f$, or implicitly, by transitivity of $=$. If $j$ is not in $[R_f^*]_i$, then we do not care whether a specific interpretation assigns the same value to $i$ and $j$ or not, because the truth value of $f$ is the same in both cases.

It is easy to see that the $ord(f)$ is at most one larger than the number of subformulas of the form $i = j$, and at least one larger than the size of $[R^*_f]_i$ for any $i$.

For example, for formulas (1)–(3) relations $R^*_f$ induce a single equivalence class containing all the index variable, while for formula (4) $R^*_f$ induces two classes: $\{i, j\}$ and $\{k, l\}$. Consequently, the orders of formulas (1)–(4) are 1, 3, 2, and 2, respectively.

Let $\sigma$ and $\sigma'$ be two tuples of states (not necessarily of the same length). We say that $\sigma$ is *k-equivalent* to $\sigma'$ (denoted $\sigma \sim_k \sigma'$) if one of the following two conditions holds for every state $s \in S$:

- $s$ appears in $\sigma$ and $\sigma'$ the same number of times, or
- $s$ appears at least $k$ times in both $\sigma$ and $\sigma'$.

For example, $(a, b, c, c)$ and $(c, a, c, b, c)$ are 2-equivalent, but not 3-equivalent.

Let $\sigma$ and $\sigma'$ be two tuples of states, and let $\mathcal{I}$ and $\mathcal{I}'$ be two interpretations of index variables in a WFF $f$, such that $\sigma(\mathcal{I}(i))$ and $\sigma'(\mathcal{I}'(i))$ are well defined for all $i \in Ind$. We say that $(\sigma, \mathcal{I})$ is $f$-equivalent to $(\sigma', \mathcal{I}')$ if $\sigma \sim_{ord(f)} \sigma'$ and the following two conditions hold:

$$\sigma(\mathcal{I}(i)) = \sigma'(\mathcal{I}'(i)) \qquad \forall i \in Ind \quad (5)$$

$$\mathcal{I}(i) = \mathcal{I}(j) \text{ iff } \mathcal{I}'(i) = \mathcal{I}'(j) \qquad \forall (i, j) \in R^*_f \quad (6)$$

This terminology is justified by the following theorem:

**Theorem 1.** *If $(\sigma, \mathcal{I})$ is $f$-equivalent to $(\sigma', \mathcal{I}')$, then $\sigma \models_{\mathcal{I}} f$ iff $\sigma' \models_{\mathcal{I}'} f$.*

*Proof.* By induction on the length of a formula. Base cases ($s(i)$ and $i = j$) follow directly from (5) and (6) respectively.

Let the inductive assumption be that the theorem holds for all proper subformulas of $f$. Then, the inductive steps for cases $f = g \wedge h$, and $f = \overline{g}$ follow trivially from the inductive assumption.

Consider now the case $f = \exists i.g$. By the inductive assumption and the symmetry of $f$-equivalence, it suffices to show that for any interpretation $\mathcal{J}$ such that $\mathcal{J}(j) = \mathcal{I}(j)$ for any $j \neq i$, there exists an interpretation $\mathcal{J}'$ such that $\mathcal{J}'(j) = \mathcal{I}'(j)$ for any $j \neq i$, and $\mathcal{J}$ and $\mathcal{J}'$ satisfy (5) and (6). Towards that, assume $\mathcal{J}$ is such that $\mathcal{J}(j) = \mathcal{I}(j)$ for all $j \neq i$. Two cases are possible (by assumption $\sigma \sim_{ord(f)} \sigma'$):

1. $\mathcal{J}(j) = \mathcal{J}(i)$ for some $j \in [R^*_f]_i$, or $\sigma(\mathcal{J}(i))$ appears the same number of times both in $\sigma$ and $\sigma'$.
   It follows from (5) and (6) that there exists $c$ such that $\sigma(\mathcal{J}(i)) = \sigma'(c)$, and for any $j \in [R^*_f]_i$:
   $$\mathcal{I}'(j) = c \text{ iff } \mathcal{I}(j) = \mathcal{J}(i) \ .$$
   Setting $\mathcal{J}'(i) = c$ and $\mathcal{J}'(j) = \mathcal{I}'(j)$ for all $j \neq i$ satisfies all the requirements.
2. $\mathcal{J}(j) \neq \mathcal{J}(i)$ for all $j \in [R^*_f]_i$, and $\sigma(\mathcal{J}(i))$ appears at least $ord(f)$ times both in $\sigma$ and $\sigma'$.
   Since there are fewer than $ord(f)$ elements of $[R^*_f]_i$, there must exist a $c$ such that $\sigma(\mathcal{J}(i)) = \sigma'(c)$ and $\mathcal{I}'(j) \neq c$ for all $j \in [R^*_f]_i$. Again, setting $\mathcal{J}'(i) = c$ and $\mathcal{J}'(j) = \mathcal{I}'(j)$ for all $j \neq i$ satisfies all the requirements.

□

**Corollary 1.** *For any IPL formula $f$, and any two tuples $\sigma$ and $\sigma'$ such that $\sigma \sim_{ord(f)} \sigma'$:*

$$\sigma \models f \text{ iff } \sigma' \models f \ .$$

*Proof.* Let $c$ and $c'$ be such that $\sigma(c) = \sigma'(c')$ (by assumption $\sigma \sim_{ord(f)} \sigma'$, such $c$ and $c'$ always exist). Define $\mathcal{I}(i) = c$ and $\mathcal{I}'(i) = c'$ for all index variables $i$. Now, $(\sigma, \mathcal{I})$ is $f$-equivalent to $(\sigma', \mathcal{I}')$, so the claim follows by Theorem 1. □

For example, any IPL formula of order 1 or 2 is either satisfied both by $(a, b, c, c)$ and $(c, a, c, b, c)$, or it is not satisfied by either. However, there exist formulas of order 3 or more which are satisfied by one and not by the other. For example:

$$\exists i.\exists j.\exists k.c(i) \wedge c(j) \wedge c(k) \wedge \overline{i = j} \wedge \overline{i = k} \wedge \overline{j = k}$$

is satisfied by $(c, a, c, b, c)$ but not by $(a, b, c, c)$.

It follows from the Corollary 1 that $\sim_k$ partitions the infinite set of all the tuples into finitely many equivalence classes, all members of which agree on truth values of all the formulas of order less or equal than $k$.

**Proposition 6.** *Given some set of states $S$, the relation $\sim_k$ partitions the set $\bigcup_{n=1}^{\infty} S^n$ of all tuples of states into $(k+1)^{|S|} - 1$ equivalence classes.*

*Proof.* Equivalence classes of $\sim_k$ can be characterized by assigning a number between 0 and $k$ to all $s \in S$. A value $i < k$ indicates that $s$ appears in all elements of the class exactly $i$ times, and the value $k$ indicates that it appears at least $k$ times. All assignments are valid except the one that assigns 0 to all $s$. □

## 4. Iterative systems

An *iterative system* is a pair $(A, f)$ where $A = (S, I, T, F)$ is a finite-state automaton called a *cell*, and $f$ is an IPL formula called an *arbiter*. An *instance* of size $n$ (where $n \geq 1$) of an iterative system $(A, f)$ is the automaton $A_n = (S^n, I_n, T_n, F^n)$, where:

$$I_n = \{(s_1, \ldots, s_n) \mid (s_1, \ldots, s_n) \models f \text{ and } \forall i = 1, \ldots, n : s_i \in I\} , \quad (7)$$

$$T_n = \{(s_1, \ldots, s_n) \xrightarrow{a} (q_1, \ldots, q_n) \mid (q_1, \ldots, q_n) \models f \text{ and } \forall i : s_i \xrightarrow{a} q_i \in T\} \quad (8)$$

In other words, an instance of size $n$ is the composition of $n$ instances of the basic cell restricted to states that satisfy the arbiter.

The language of any invariant of an iterative system $(A, f)$ must contain the language of the infinite-state automaton $\overset{\infty}{A} = \bigcup_{n=1}^{\infty} A_i$. Obviously, the automaton $\overset{\infty}{A} / \sim_{ord(f)}$ has that property, and it is also finite-state, because (by Proposition 6) the number of equivalence classes induced by $\sim_{ord(f)}$ is finite. We can effectively construct such an automaton by constructing $\hat{A} = (\bigcup_{i=1}^{||S|^2 * ord(f)} A_i) / \sim_{ord(f)}$. We claim that $\hat{A}$ has the same language as $\overset{\infty}{A} / \sim_{ord(f)}$. The claim follows from the following lemma:

**Lemma 1.** *Let $S$ be the set of states of the cell of some iterative system, and let $f$ be the arbiter of that system. If $\rho \xrightarrow{a} \sigma$ is a transition of some instance of size larger than $|S|^2 * ord(f)$, then there exists a transition $\rho' \xrightarrow{a} \sigma'$ in the instance of size $|S|^2 * ord(f)$, such that $\rho' \sim_{ord(f)} \rho$ and $\sigma' \sim_{ord(f)} \sigma$.*

*Proof.* Since there are at most $|S|$ distinct states in $\rho$ and $\sigma$, there can be at most $|S|^2$ distinct component transitions in $\rho \xrightarrow{a} \sigma$. Thus, some of the component transitions must appear more than $ord(f)$ times. By eliminating enough appearances of those components, in a way that each one still appears at least $ord(f)$ times, we obtain $\rho' \xrightarrow{a} \sigma'$. Since each component that we have eliminated still appears at least $ord(f)$ times, the same is also true for the number of appearances of the corresponding present (next) states in $\rho'$ (respectively $\sigma'$). Thus, $\rho' \sim_{ord(f)} \rho$ and $\sigma' \sim_{ord(f)} \sigma$ both hold. □

It is straightforward to show that $\mathcal{L}(\hat{A}) \subseteq \mathcal{L}(\overset{\infty}{A} / \sim_{ord(f)})$. To prove the converse, it suffices to invoke Lemma 1. to show that all transitions in $\overset{\infty}{A} / \sim_{ord(f)}$ appear also in $\hat{A}$.

We have shown how to construct an invariant of an iterative array. The constructed invariant is not necessarily tight. Unfortunately, a tight finite-state invariant does not always exist, as shown by the following theorem:

**Theorem 2.** *There exists an iterative array $(A, f)$ such that $\bigcup_{i=n}^{\infty} \mathcal{L}(A_n)$ is not regular.*

*Proof.* **(sketch)** Consider the iterative array with the basic cell as shown in Figure 1 (initial states are indicated by arrows, final by double circles), and the arbiter:

$$\exists i. \ ((q(i) \vee s(i)) \wedge \forall j. \ ((q(j) \vee s(j)) \Longrightarrow (i = j) \ )) \ , \quad (9)$$

i.e. at all times exactly one cell must be in state $s$ or state $q$. As long as any cell is in state $p$ only 1 can be accepted, and after one cell moves to state $s$ only 0 can be accepted. Therefore, the instance of size $n$ accepts a single string consisting of $n-1$ occurrences of 1, while all cells but one move to state $r$, another 1 while the only cell in state $q$ moves to state $s$, and finally $n-1$ occurrences of 0, while all cells but one move to state $t$, and the last cell moves to $s$. It follows that:

$$\bigcup_{i=n}^{\infty} \mathcal{L}(A_n) = \{1^n 0^{n-1} \mid n \geq 1\} ,$$

which is easily shown not to be regular. □

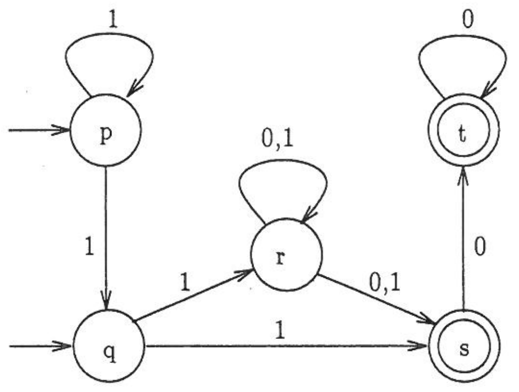

*Fig. 1.* A cell generating a non-regular language.

## 5. Modeling buses

### 5.1. Abstract arbiter

In order to verify a communication protocol, it might be useful to abstract low-level details of bus arbitration, and just postulate that at any given moment only one of the cells can own the bus. Such a requirement can be easily expressed by the IPL formula:

$$\forall i.\ \forall j.\ (master(i) \wedge master(j)) \Longrightarrow (i = j) \quad , \tag{10}$$

where $master(i)$ characterizes a set of states that a cell can move to only if it owns the bus. If, in addition, the bus must be owned by some processor at all times, then (10) must be conjoined with $\exists i.\ master(i)$. The order of either of the formulas is 2.

Consider, for example, an iterative system with the cell shown in Figure 2 (initial states are indicated by an arrow, final by a double circle), and the arbiter (9). Initially, one cell owns the bus, and others wait to receive it. The owner either moves to $s$ asserting 0 on the bus, and retaining the ownership indefinitely, or release the bus (by moving to $r$). If at any time some other cell receives the bus, it will assert 1, and then behave as described. It is not hard to check that the language of the instance of size $n$ is $\{1^i0^+ \mid 0 \leq i < n\}$. Thus, in the language of every instance there exists a string not in the language of any smaller instance ( in particular $1^{n-1}0$), so no finite subset of instances can serve as an invariant. Still, our procedure will find an invariant, which is in this case even tight, in the sense that its language is exactly the (infinite) union of languages of all the instances.

### 5.2. Wired AND

One low-level approach to bus arbitration is to let bus wires be "wired AND's". On these wires only 0's can be asserted. The value of a wire is 1 if no cells assert 0. If a cell wants the ownership, it asserts 0 on one of the dedicated wires. Other cells will not try to transmit until that wire is 1 again. Of course, it is possible that two cell assert 0 at the same moment, so some kind of collision detection and recovery have to be built

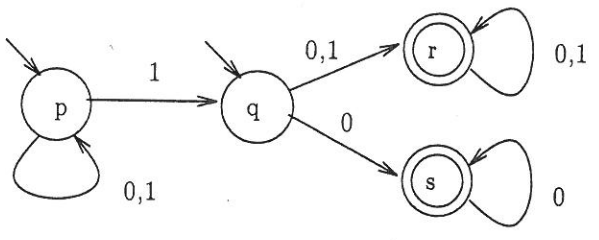

*Fig. 2.* An example of a basic cell.

into the protocol. A single wired AND can be modeled by the following IPL formula:

$$(\exists i.\ sense0(i)) \Longrightarrow (\exists j.\ assert0(j))\,, \qquad (11)$$

where we assume that a cell can move to some state characterized by *sense*0 only if the value of that wire is 0, and similarly, a cell can move to some *assert*0 state only if it is asserting 0. If more wires represent wired AND, we can model them by the conjunction of formulas like (11). It is encouraging to notice that the order of (11) is one, and that it remains one even when many such formulas are conjoined.

## 6. Conclusions

We have proposed a formalism to model bus-based systems, based on the use of automata to model basic components, and the use of logic formulas to model bus arbitration. For this purpose, we have defined the logic IPL. We have shown that IPL has a small model property, and therefore partitions the infinite state space of bus-based systems into a finitely many equivalence classes. We have shown that the quotient with respect to that partition can be constructed in finite time, and propose to use that quotient as a size-independent abstraction of bus-based systems. The abstraction is conservative, and we have shown that it is not generally possible to do any better, because exact simplifications of some bus-based systems have non-regular languages, and therefore cannot be represented by finite-state automata.

## References

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems, Lyngby, Denmark, 10-12 Oct. 1991, Proceedings*, pages 209–229. Springer-Verlag, 1993.

[2] C.M. Angelo, D. Verkest, L. Claesen, and H. De Man. On the comparison of HOL and Boyer-Moore for formal hardware verification. *Formal Methods in System Design*, 2(1):45–72, February 1993.

[3] L. M. Augustin, D. C. Luckman, B. A. Gennart, Y. Huh, and A. G. Stanculescu. *Hardware design and simulation in VAL/VHDL*. Kluver Academic Publishers, 1991.

[4] A. Aziz, F. Balarin, R. K. Brayton, S.-T. Cheng, R. Hojati, S. C. Krishnan, R. K. Ranjan, A. L. Sangiovanni-Vincentelli, T. R. Shiple, V. Singhal, S. Tasiran, and H.-Y. Wang. HSIS: A BDD-based environment for formal verification. In *Proceedings of the 31th ACM/IEEE Design Automation Conference*, 1994.

[5] Adnan Aziz and Robert K. Brayton. Verifying interacting finite state machines. Technical Report UCB/ERL M93/52, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, July 1993.

[6] Felice Balarin, Robert K. Brayton, Szu-Tse Cheng, Desmond A. Kirkpatrick, Alberto L. Sangiovanni-Vincentelli, and Ephrem C. Wu. A methodology for formal verification of real-time systems. Technical Report Memorandum No. UCB/ERL M95/11, University of California, Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, February 1995.

[7] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. On the automatic computation of network invariants. In David L. Dill, editor, *Proceedings of Computer Aided Verification: 6th International Conference, CAV'94, Stanford, CA, June 1994*, pages 234–246. Springer-Verlag, 1994. LNCS vol. 818.

[8] Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. In David L. Dill, editor, *Proceedings of the Conference on Computer-Aided Verification*, volume 818 of *LNCS*, pages 142–155, Stanford, CA, June 1994. Springer-Verlag.

[9] W.R. Bevier, W.A. Hunt, J.S. Moore, and W.D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–28, December 1989.

[10] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, June 1990.

[11] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81(1):13–31, 1989.

[12] B. Chen, M. Yamazaki, and M. Fujita. Bug identification of a real chip design by symbolic model checking. In *Proceedings of The European Conference on Design Automation, EDAC-ETC-EUROASIC, Paris, France, 28 Feb.-3 March 1994*, pages 132–136. IEEE Comput. Soc. Press, 1994.

[13] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages Systems*, 2(8):244–263, 1986.

[14] E. ALLAN EMERSON. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 995–1072. Elsvier Science Publishers B. V., 1990.

[15] M. GORDON. Why higher-order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design. Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 153–77. North-Holland, 1986.

[16] ORNA GRUMBERG AND DAVID E. LONG. Model checking and modular verification. In *CONCUR, Amsterdam, The Netherlands*, August 1991. (also submitted to JACM).

[17] AARTI GUPTA. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, October 1992.

[18] FREDERICK C. HENNIE. *Iterative Arrays of Logical Circuits*. MIT Press and John Eiley Sons, Inc., 1961.

[19] THOMAS A. HENZINGER, XAVIER NICOLIN, JOSPEH SIFAKIS, AND SERGIO YOVINE. Symbolic model-checking for real-time systems. In *Proceedings of 7th Symposium on Logics in Computer Science*. IEEE Computer Society Press, 1992.

[20] J.E. HOPCROFT AND J.D. ULLMAN. *Introduction to Automata Theory, languages and Computation*. Addison Wesley, 1979.

[21] R. P. KURSHAN. Analysis of discrete event coordination. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems : Models, Formalisms, Correctness*, pages 414–453. Springer-Verlag, 1990. LNCS vol. 430.

[22] R. P. KURSHAN AND K. L. MCMILLAN. A structural induction theorem for processes. In *Proceedings of the 8th ACM Symp. PODC*, 1989.

[23] ROBERT P. KURSHAN. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[24] ZOHAR MANNA AND ANIR PNUELI. Verification of concurrent programs: The temporal framework. In R. Boyer and J. Moore, editors, *Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1981.

[25] KENNETH L. MCMILLAN. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[26] B. E. NELSON, R. B. JONES, AND D. A. KIRKPATRICK. Simulation event pattern checking with proto. In *Proceedings of the SHDL Conference*, 1994.

[27] A. PNUELI. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, October 1977.

[28] J.K. RHO AND F. SOMENZI. Inductive verification of iterative systems. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 628–33, June 1992.

[29] J.K. RHO AND F. SOMENZI. Automatic generation of network invariants for the verification of iterative sequential systems. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 123–137. Springer-Verlag, 1993. LNCS vol. 697.

[30] R. SCHLÖR AND W. DAMM. Specification and verification of system level hardware designs using timing diagrams. In *Proceedings of The European Conference on Design Automation, Paris, France, February 1993*, 1993.

[31] Z. SHTADLER AND O. GRUMBERG. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop Proceedings, Grenoble, France, 12-14 June 1989*, pages 151–65. Springer-Verlag, 1990. LNCS vol. 407.

[32] ARAVINDA P. SISTLA AND EDMUND M. CLARKE. Complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985.

[33] MOSHE Y. VARDI AND PIERRE WOLPER. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 332–344, Boston, July 1986.

[34] MOSHE Y. VARDI, PIERRE WOLPER, AND ARAVINDA P. SISTLA. Reasoning about infinite computation paths. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science, Tucson, AZ, USA, 7-9 Nov. 1983*. IEEE Comput. Soc. Press, 1983.

[35] P. WOLPER AND V. LOVINFOSSE. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop Proceedings, Grenoble, France, 12-14 June 1989*, pages 68–80. Springer-Verlag, 1990. LNCS vol. 407.

Received: August, 1995
Accepted: November, 1995

*Contact address:*
Felice Balarin
Cadence Berkeley Laboratories
1919 Addison St., Suite 303-304
Berkeley, CA 94704, USA
phone: 1–408–944–7146
fax: 1–510–486–0205
email: felice@cadence.com

FELICE BALARIN received the B.S. degree in electrical engineering and the M.S. degree in computer sciences from the University of Zagreb, Croatia in 1985 and 1989, respectively. He received the Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley in 1994. Since then, he has been a research scientist at the Cadence Berkeley Labs. His research is focused on the analysis and development of formal methods for design, verification and control of digital and real-time systems, and the application of these methods to the hardware/software co-design.