12th IFAC/IEEE Workshop on Discrete Event Systems
Cachan, France. May 14-16, 2014

Symbolic Interpretation and Execution of Extended Finite Automata ⋆

Mohammad Reza Shoaei * **Bengt Lennartson** *

* *Department of Signals and Systems, Chalmers University of Technology, SE-412 96, Gothenburg, Sweden (email: {shoaei, bengt.lennartson}@chalmers.se).*

Abstract: We introduce a symbolic interpretation and execution technique for Extended Finite Automata (EFAs) and provide an interpreter that symbolically interprets and executes EFAs w.r.t. their (internal) variables. More specifically, the interpreter iterates over the EFA transitions, and by passing each transition, it symbolically interprets and evaluates the condition on the transition w.r.t. the known values of variables, and leaves other variables intact, and when it terminates, it returns the residual model. It is shown that the behavior of the residual system with respect to the original system is left unchanged. Finally, we demonstrate the effectiveness and necessity of the symbolic interpretation and execution combined with abstractions for the nonblocking supervisory control of two manufacturing systems.

Keywords: Discrete-event systems; symbolic interpretation; supervisory control theory.

1. INTRODUCTION

Traditionally, finite-state automata have been used for the supervisory control of discrete-event systems (DES), Cassandras and Lafortune [2008] and Wonham [2013], which has been found to be non-trivial for complex systems with data. Modeling using *Extended Finite Automata* (EFAs), i.e., an ordinary finite automaton whose transitions are augmented with *variable updates*, makes it possible to, efficiently and in a compact form, model DES that involve non-trivial data manipulation, see Skoldstam et al. [2007].

A challenge with this new control framework is to symbolically interpret and optimize the models before synthesizing the controller in order to be able to exploit various abstraction methods, such as Shoaei et al. [2012] and Mohajerani et al. [2013]; reducing the complexity and more often avoiding state-space explosion. To this end, a naive attempt would be to expand the domain of "internal" variables on every transition of the system. This is, however, not efficient (in particular, for variables with large domain) as it requires to "blindly" expand the domain, not only those particular values which are required.

To overcome this problem, we introduce a *symbolic interpretation and execution* (or just interpretation) technique for EFAs. The interpretation process is performed by an interpreter $[\![.]\!]$ that iterates over the EFA transitions and, instead of blind expansion of the domain of variables, it symbolically interprets and executes, or more specifically, partially evaluates the condition on that transitions w.r.t. the known variables value in the context. When $[\![.]\!]$ terminates, it returns the "residual" EFA model.

The overall motivation for interpretation of EFAs is that analyzing the residual models is often more efficient than analyzing the original ones, since the interpreter $[\![.]\!]$ has already pre-executed the portions of system that depend on the internal variables without computing the global (explicit) model. This pays off when, e.g, one seeks for abstraction possibilities to further reduce the complexity of the system before constructing the global model. Another application of EFA interpretation can be seen in the process of synthesizing a supervisor for EFAs using BDDs, see Miremadi et al. [2012]. In this, one can, instead of directly convert the EFA models to BDDs, first interpret and execute the (internal) variables and simplify the models, then convert the residual models to BDDs. This can, sometimes significantly, help to decrease the number of BDD variables and avoid (possible) out of memory errors.

In this paper, we provide an algorithm that implements the interpreter $[\![.]\!]$. Further, we formulate the partial evaluation (execution) process by a proof calculus, of which we show its soundness. Furthermore, for the purpose of supervisory control, we provide sufficient conditions to guarantee that the behavior of the residual system is left unchanged compared to the original system, hence resulting in maximally permissive and nonblocking control to the entire system by using the interpreted models.

We note that the proposed technique is conceptually similar to that of program execution, cf. Jones et al. [1993] and Hatcliff [2003]. In this paper, however, we provide a basic starting point to bring the advantages of the symbolic interpretation and execution to DES with data and to the best of our knowledge, it is the first attempt to use such a technique for the purpose of supervisory synthesis. This paper also demonstrates the importance of using not only abstractions, but also to include the symbolic interpretation to obtain significant state reduction before ordinary synthesis.

The rest of the paper is organized as follows. Section 2 briefly recall the predicates, their syntax and semantics, and defines EFAs. In Section 3, we introduce the symbolic interpretation and execution technique for EFAs together with a calculus that mechanizes the partial evaluation process of conditions. In Sections 4 we demonstrate the symbolic interpretation combined with abstractions for nonblocking supervisory control of two industrial manufacturing systems. Finally, we conclude our work in Section 5. The proof details are referred to the appendix.

⋆ This work was carried out at the Wingquist Laboratory VINN Excellence Center within the Area of Advance – Production at Chalmers University of Technology, supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA).

2. PRELIMINARIES

In this section, we recall some basic definitions and concepts to be used later.

2.1 Predicate Logic

Syntax The formulas of our logic are *quantifier-free first-order logic with equality* over a countable set V of *individual variables* $x, y, \ldots$, and a *signature* set Θ consisting of n-ary *function symbols* $f \in \Theta$, where *constants* are denoted by nullary functions, *predicate symbols* $p \in \Theta$ including the *binary equality* symbol $=$, 1, 0, and the propositional connectives $\leftrightarrow, \rightarrow, \wedge, \vee, \neg$. A *term* $t \in T_\Theta(V)$ is a (well formed) expression over symbols in Θ and V. A term is called a *ground term* if it contains no variables. *Formulas* $\phi, \psi, \ldots$ are defined inductively as follows. A formula is either an *atomic formula* $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, a spacial formula $\bot$ (resp. $\top$) which is always false (resp. true), or of the form $\neg\varphi$ or $\varphi \triangleright \psi$ where $\triangleright \in \{\leftrightarrow, \rightarrow, \wedge, \vee\}$ and φ, ψ are formulas.

Semantics Terms and formulas constructed over Θ and V take on meaning when interpreted over a structure called *model*. A model is a pair $\mathcal{M} = (D, \mathcal{I})$ consisting of: A finite and nonempty set D called *domain* (or universe), where we distinguish the values of an individual variable x by a nonempty set D_x; and an *interpreter* function $\mathcal{I}$ that assigns an n-ary function $f^{\mathcal{I}} : D^n \to D$ to each n-ary function symbol $f \in \Theta$ where we regard constants (nullary functions) as just elements of D, and an n-ary relation $p^{\mathcal{I}} \subseteq D^n$ to each n-ary predicate symbol $p \in \Theta$.

Fix $\mathcal{I}$ and let D be the domain of variables. We define a *valuation* map $\eta : T_\Theta(V) \to D$ on terms $T_\Theta(V)$ over variables V. A valuation is uniquely determined by its values on V, since V generates $T_\Theta(V)$. Moreover, any map $\eta : V \to D$ extends uniquely to a valuation $\eta : T_\Theta(V) \to D$ by induction. A *substitution* is a mapping $\eta : T_\Theta(V) \to T_\Theta(V)$. For a term t, $\eta(t) = t[x/\eta(x) | \forall x \in V]$ is a new term obtained by "substituting" all (free) occurrences of x_i in t with t_i $(1 \leq i \leq n)$ and we denote by ϵ the empty substitution such that $\epsilon(t) = t$. The substitution is done for all variables in t simultaneously. Furthermore, we write $\eta[x/t]$ (or $\eta[x \mapsto t]$) to denote a new substitution μ constructed from η such that $\mu(x) = t$ and $\mu(y) = \eta(y)$ for $y \neq x$. We also write $\eta[x \mapsto \epsilon]$ to denote that we drop the substitution x/t from η. In this paper, without loss of generality, we consider valuations as substitutions where a valuation substitutes all variables to their ground terms.

The *satisfaction relation* $\vDash$ (also called semantic entailment) is defined inductively on the structure of formulas as usual [see Gallier, 2003]. If $\eta \vDash \varphi$ holds, we say that φ is true (in $\mathcal{M}$) *under valuation* η, or that η *satisfies* φ (in $\mathcal{M}$). If Γ is a set of formulas, we write $\eta \vDash \Gamma$ if $\eta \vDash \varphi$ for $\varphi \in \Gamma$. If φ is true in all models, then we write $\vDash \varphi$ and say that φ is *valid*. Two formulas ϕ, ψ are said to be *logically equivalent*, denoted $\phi \equiv \psi$, if $\vDash \phi \leftrightarrow \psi$.

2.2 Proof Calculus

A proof calculus describes certain syntactic operations to be carried out on formulas. We denote by $\vdash$ a calculus containing "rules", along with some definitions that say how these rules are to be applied. The basic building blocks, to which the rules or our calculus are applied are the *sequents* of the form $\Gamma \Rightarrow \Delta$ (in the literature also denoted as $\Gamma \vdash \Delta$) where Γ and Δ contain formulas. The formulas on the left of the *sequent arrow* $\Rightarrow$ are called *antecedent* and the formulas on the right are called *succedent*. The intuitive meaning of a sequent $\phi_1, \ldots, \phi_m \Rightarrow \psi_1, \ldots, \psi_n$ is as follows: whenever all the ϕ_i of the antecedent are true, then at least one of the succedent is true, informally, $\bigwedge \phi_i \rightarrow \bigvee \psi_j$.

A *rule* (or schema) in the calculus is of the form

$$\frac{\Psi_1, \quad \Psi_2, \quad \ldots, \quad \Psi_n}{\Psi_0}$$

where $\Psi_i := \Gamma_i \Rightarrow \Delta_i$ for $0 \leq i \leq n$ denote sequents. The sequent below the line is the *conclusion* of the rule and the above sequents are its *premises*. A rule with no premises is called a *closing rule*. The meaning of the rule is that if the premises are valid, then the conclusion is also valid. However, we use it in opposite direction, that is to prove the validity of the conclusion, it suffices to prove the premises.

A sequent proof is a tree that is constructed according to a certain set of rules.

Definition 1. A *proof tree* for a formula ϕ is a finite tree where the root sequent (shown at the bottom) is annotated with $\Rightarrow \phi$; each inner node of the tree is annotated at least with a sequent; and a leaf node which may or may not be annotated with a sequent. If it is, it is the (empty) premise of one of the closing rules. A *branch* of a proof tree is a path from the root to one of the leaves. A branch is *closed* if the leaf is annotated with empty sequent. A proof tree is *closed* if all its branches are closed.

We denote by $\Psi_0 \rightsquigarrow \Psi_i$ a branch of a proof tree from the root node Ψ_0 to a node Ψ_i for some $i \in \mathcal{N} := \{0, \ldots, n\}$, where $\mathcal{N}$ is the index set of n nodes. Let $\star$ denote an empty sequent. Then, for a closed branch, we write $\Psi_0 \rightsquigarrow \Psi_i^\star$ instead of $\Psi_0 \rightsquigarrow \star$ where $\Psi_i^\star$ is the conclusion of the rule with empty premise. Further, we denote by $\pi_\phi := \{\Psi_0 \rightsquigarrow \Psi_i\}$ the set of all branches in the tree. Then, we write $\pi_\phi^\star$ when all the branches in π_ϕ are closed, or that the proof tree of ϕ is closed.

For example, consider the following proof for a formula ϕ in some calculus $\vdash$:

$$\dfrac{\dfrac{\Psi_3 \quad \overset{\star}{\Psi_4} \quad \overset{\star}{\Psi_5}}{\Psi_1} \qquad \Psi_2}{\Psi_0}$$

The corresponding proof tree of the above proof has 8 nodes, $\Psi_0, \ldots, \Psi_7$, where Ψ_0 is the root node and Ψ_6, Ψ_7 denote $\star$. Further, $\pi_\phi := \{\Psi_0 \rightsquigarrow \Psi_3, \Psi_0 \rightsquigarrow \Psi_4^\star, \Psi_0 \rightsquigarrow \Psi_5^\star\}$ is the set of all branches. Clearly, π_ϕ is not closed because the branch $\Psi_0 \rightsquigarrow \Psi_3$ is not closed.

A formula ϕ is valid in proof calculus $\vdash$, denoted $\vdash \phi$, iff the proof tree for ϕ (Def. 1), is closed. Then it follows that $\vdash \phi$ iff $\pi_\phi^\star$, i.e., ϕ is valid in $\vdash$ if all branches of its proof tree are closed. If this is the case, then we simply write $\phi \vdash \pi_\phi^\star$ to denote that ϕ is valid in $\vdash$ according to a proof tree with the set of branches $\pi_\phi^\star$.

Definition 2. (Soundness). A calculus system $\vdash$ is said to be sound w.r.t. a semantics $\vDash$ if $\vdash \phi$ implies $\vDash \phi$.

In words, $\vDash \phi$ holds whenever $\vdash \phi$ is valid.