

A Logic for Synchronous Transitions with Dynamic Conflict Resolution

Vanderlei Moraes Rodrigues* Flávio Rech Wagner
 Instituto de Informática, UFRGS†
 {vandi,flavio}@inf.ufrgs.br

Abstract

This paper introduces a formalism named DSYNC aimed at the design and verification of synchronous concurrent systems. The components of this formalism are a transition system and a first-order linear-time temporal logic. The DSYNC transition system adopts a synchronous computation model, includes a method to solve write-conflicts, and represents transitions as possibly non-terminating imperative commands. The conflict resolution method is dynamic because it detects conflicts at run-time. The DSYNC logic allows for formal reasoning about DSYNC transition systems using compositional and modular proofs. Such features are missing in other formalisms based on transition systems and temporal logics, although they are important for the verification of a large class of systems. This paper also discusses some of the pragmatics in verifying systems with DSYNC, and considers some extensions to the formalism. DSYNC is based on the Hoare logic and the UNITY formalism.

Keywords: linear-time temporal logic, Hoare logic, UNITY, synchronous systems.

1 Introduction

The class of formalisms composed of a transition system and a first-order linear-time temporal logic includes the Manna and Pnueli logic [17, 18], TLA [16], UNITY [8, 20], ST [30], and other formalisms [26, 29]. Due to their expressiveness and flexibility, they have been successfully employed in the description and verification of concurrent or reactive systems in several application fields. However, these formalisms deal with asynchronous systems mostly. A distinct class of computational systems is that of synchronous systems. It includes programming languages such as Esterel [2], hardware description languages such as VHDL [22], and specification formalisms such as evolving algebras [12, 13, 14]. Works on the verification of such systems either explore restricted techniques such as finite model-checkers, or depend on idiosyncrasies of a particular notation, or are not mature yet.

*Partially supported by QaP-For/FAPERGS.

†Caixa postal 15064. 91501-970, Porto Alegre, Brazil. Fax +55(51)319-1576.

This paper introduces a formalism named DSYNC aimed at the description and verification of synchronous systems using a transition system and a first-order linear-time temporal logic. The proposed synchronous computation model includes a conflict resolution method detecting conflicts dynamically, during system execution. DSYNC also features a representation for transitions as (possibly non-terminating) imperative commands, a modular and compositional approach to proof development, and a good catalog of elementary verification techniques. There are extensions to DSYNC dealing with generic parameters, regular structures, and statically detected conflicts.

We believe the development of DSYNC is a contribution to the application of first-order linear-time temporal logics on the verification of reactive systems. The main formalisms in this class listed above (UNITY, TLA, etc.) do not handle synchronous systems, dynamic resolution of write-conflicts, and other features of DSYNC. These features are essential for the verification of VHDL and evolving algebras, for instance, but they are not addressed appropriately in earlier formalisms.

We also believe DSYNC contributes to the work on the verification of synchronous systems. Most works on this field concentrate on finite model-checkers [9]. This technique is quite effective, but it does not handle some important situations. It does not allow for modular proofs, where properties of a system are derived from properties of its components only, without any knowledge on the actual definition of components. It does not solve conflicts dynamically, and it is not appropriate for parametric and regular systems either, because such systems may generate large or infinite sets of states. Other works employ more general formalisms, but most of them deal with restricted systems or result in complex formalisms which are hard to apply in the verification of actual systems [7, 28]. DSYNC is quite general, yet easy to use.

Work on DSYNC started as a wish to apply the UNITY logic in the verification of VHDL designs. While adapting UNITY to the semantics of VHDL, we came up with a formalism embodying a general model of synchronous computation with dynamic conflict resolution. The same model and verification techniques apply to other hardware description languages, to synchronous programming languages, and to some specification formalisms such as evolving algebras. We describe elsewhere the practical work aimed specifically at VHDL [24]. This paper describes DSYNC as a general formalism for the verification of synchronous systems, develops some foundation work, and discusses important description and proof techniques for DSYNC. The resulting formalism is quite general and flexible, although it cannot reach the automation level of finite model-checkers.

This paper is organized as follows. Section 2 discusses the synchronous computation model and introduces the DSYNC transition system, and section 3 presents the DSYNC logic. Section 4 studies some methods to apply this formalism in the specification and verification of synchronous systems. Section 5 comments on related works, and the last section presents some concluding remarks.

2 Transition System

The components of a transition system are a set of variables and a set of transitions. They represent the (possibly infinite) set of system states and the permitted

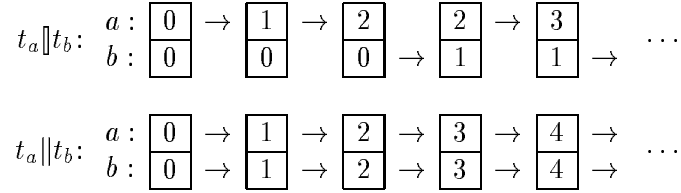


Figure 1: Asynchronous and synchronous computation models

state changes. According to the *asynchronous* computation model adopted by UNITY, TLA, and most transitions systems, each computation step non-deterministically selects and runs exactly one elementary transition. Let t_a and t_b be the transitions $a := a + 1$ and $b := b + 1$ respectively. The left-half of Figure 1 shows a computation of the asynchronous combination $t_a \parallel t_b$. Distinctly from these formalisms, DSYNC follows a *synchronous* computation model, where each computation step runs each elementary transition once. The right-half of Figure 1 shows a computation of the synchronous combination $t_a \parallel t_b$.

Figure 2 presents the DSYNC transition system. A program is a pair $\langle b|t \rangle$, where the program body t is a synchronous combination of a finite set of elementary transitions, and the initial condition b is a boolean expression describing the initial value of variables. When the initial condition is irrelevant, we annotate program $\langle b|t \rangle$ as t only. Standard imperative commands represent elementary transitions. For simplicity, we assume variables and expressions are defined as usual, expressions are total, and programs are well-typed. We also need some syntactical restriction on initial conditions to ensure they are consistent. In this paper, they are restricted to a conjunction of terms $x = k$, where k is a constant.

A state σ is a mapping of variable names to values, and $\hat{\sigma}(e)$ denotes the value in σ of an expression e (or condition, or assertion). Notation $f : X \rightarrow Y$ indicates f is a partial function mapping elements of X to elements of Y , and $\{x_1 \mapsto y_1, \dots, x_n \mapsto y_n\}$ represents a finite mapping of x_i to y_i . When f is undefined on x , we write $f(x) = \perp$. Notation $f \downarrow Z$ denotes f with its domain restricted to a set Z of variables names, and $f \oplus g$ denotes f updated with function g :

$$(f \downarrow Z)(x) = \begin{cases} f(x) & \text{when } x \in Z \\ \perp & \text{otherwise} \end{cases} \quad (f \oplus g)(x) = \begin{cases} g(x) & \text{when } g(x) \neq \perp \\ f(x) & \text{otherwise} \end{cases}$$

The rules in Figure 2 present an operational semantics for the DSYNC transition system. The four-argument relations $s \mid \omega \triangleright \sigma \xrightarrow{\text{cmd}} \sigma'$ and $t \mid \omega \triangleright \sigma \xrightarrow{\text{trn}} \sigma'$ indicate that the execution of command s or transition t in a state σ produces a state σ' , assigning new values to the variables named in the write-set ω . *Write-sets* are sets of variable names recording the assignments that the program performs. We need to build them along the computation because they depend on the initial state. For instance, **if** $x > 0$ **then** $y := x$ **else skip** does not always write to y .

Rules S1 to S7 define the command semantics as usual [1], except that they also collect the write-sets. S2 adds a variable name to the write-set, and the remaining rules only combine these sets. Rules S8 to S10 define a semantics for transitions. Rule S8 executes elementary transitions, beginning the command computation with an

x, var	\in Var	variables	$prg ::= \langle cond \mid trn \rangle$
e, exp	\in Exp	expressions	$prg \parallel prg$
$b, cond$	\in Cond	conditions	$trn ::= cmd$
s, cmd	\in Cmd	commands	$trn \parallel trn$
t, trn	\in Trn	transitions	$cmd ::= \mathbf{skip}$
F, G, prg	\in Prg	programs	$var := exp$
α	\in Value	values	$cmd ; cmd$
ω, ν	\in Write \subseteq Var	write-sets	$\mathbf{if\ cond\ then\ cmd}$
σ	\in State : Var \rightarrow Value	states	$\mathbf{else\ cmd}$
$\hat{\sigma}$	\in Eval : Exp \rightarrow Value	evaluator	$\mathbf{while\ exp\ do\ cmd}$
Σ	\in Comp : (State \times Write)*	computation	

[S1]	$\mathbf{skip} \mid \emptyset \triangleright \sigma \xrightarrow{cmd} \sigma$	[S2]	$x := e \mid \{x\} \triangleright \sigma$ $\xrightarrow{cmd} \sigma \oplus \{x \mapsto \hat{\sigma}(e)\}$
[S3]	$\frac{s_1 \mid \omega_1 \triangleright \sigma \xrightarrow{cmd} \sigma_1 \quad s_2 \mid \omega_2 \triangleright \sigma_1 \xrightarrow{cmd} \sigma_2}{s_1 ; s_2 \mid \omega_1 \cup \omega_2 \triangleright \sigma \xrightarrow{cmd} \sigma_2}$	[S4]	$\frac{\hat{\sigma}(b) = \mathbf{true} \quad s_1 \mid \omega_1 \triangleright \sigma \xrightarrow{cmd} \sigma_1}{\mathbf{if\ } b \mathbf{\ then\ } s_1 \mathbf{\ else\ } s_2 \mid \omega_1 \triangleright \sigma \xrightarrow{cmd} \sigma_1}$
[S5]	$\frac{\hat{\sigma}(b) = \mathbf{false} \quad s_2 \mid \omega_2 \triangleright \sigma \xrightarrow{cmd} \sigma_2}{\mathbf{if\ } b \mathbf{\ then\ } s_1 \mathbf{\ else\ } s_2 \mid \omega_2 \triangleright \sigma \xrightarrow{cmd} \sigma_2}$	[S6]	$\frac{\hat{\sigma}(b) = \mathbf{true} \quad (s ; \mathbf{while\ } b \mathbf{\ do\ } s) \mid \omega \triangleright \sigma \xrightarrow{cmd} \sigma_1}{\mathbf{while\ } b \mathbf{\ do\ } s \mid \omega \triangleright \sigma \xrightarrow{cmd} \sigma_1}$
[S7]	$\frac{\hat{\sigma}(b) = \mathbf{false}}{\mathbf{while\ } b \mathbf{\ do\ } s \mid \emptyset \triangleright \sigma \xrightarrow{cmd} \sigma}$	[S8]	$\frac{t \mid \omega \triangleright \sigma \xrightarrow{cmd} \sigma'}{t \mid \omega \triangleright \sigma \xrightarrow{trn} \sigma'} \quad t \in \text{Cmd}$
[S9]	$\frac{t_1 \mid \omega_1 \triangleright \sigma \xrightarrow{trn} \sigma_1 \quad t_2 \mid \omega_2 \triangleright \sigma \xrightarrow{trn} \sigma_2}{t_1 \parallel t_2 \mid \omega_1 \cup \omega_2 \triangleright \sigma \xrightarrow{trn} \sigma \oplus (\sigma_1 \downarrow \omega_1) \oplus (\sigma_2 \downarrow \omega_2)}$	[S10]	$\frac{t_1 \mid \omega_1 \triangleright \sigma \xrightarrow{trn} \sigma_1 \quad t_2 \mid \omega_2 \triangleright \sigma \xrightarrow{trn} \sigma_2}{t_1 \parallel t_2 \mid \omega_1 \cup \omega_2 \triangleright \sigma \xrightarrow{trn} \sigma \oplus (\sigma_2 \downarrow \omega_2) \oplus (\sigma_1 \downarrow \omega_1)}$
[S11]	$\frac{\hat{\sigma}_0(b) = \mathbf{true} \quad \omega_0 = \text{var}(b) \quad t \mid \omega_{i+1} \triangleright \sigma_i \xrightarrow{trn} \sigma_{i+1}}{\langle b \mid t \rangle \triangleright (\sigma_0, \omega_0)(\sigma_1, \omega_1)(\sigma_2, \omega_2) \dots}$	[S12]	$\frac{\langle b_1 \wedge b_2 \mid t_1 \parallel t_2 \rangle \triangleright \Sigma}{\langle b_1 \mid t_1 \rangle \parallel \langle b_2 \mid t_2 \rangle \triangleright \Sigma}$

Figure 2: DSYNC transition system

empty write-set. Rules S9 and S10 describe the synchronous combination of transitions, including the dynamic method to solve write-conflicts.

Two transitions generate a *write-conflict* when they try to assign to the same variable at the same time. To account for this situation, we define the semantics for synchronous combinations as follows. To execute $t_1 \parallel t_2$, we give a distinct copy of the initial state to each transition, execute them independently, and then update the initial state with the contribution of each transition. The contribution of a transition is the set of variables the transition writes while executing. The order we apply the contributions to the final state is not determined. Therefore, the last transition to apply its contribution will define the final value of conflicting variables, and the computation of a synchronous

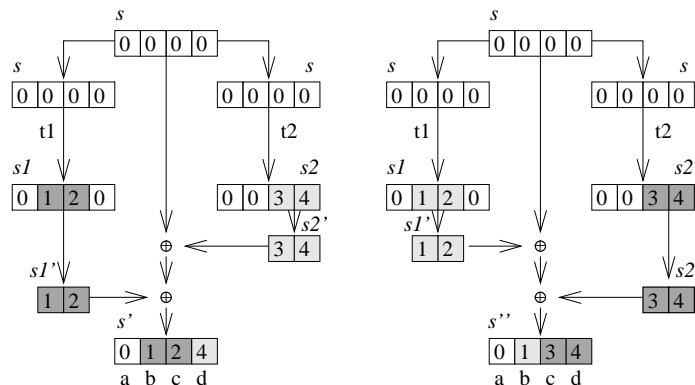


Figure 3: Execution of a synchronous combination

combination generating write-conflicts is non-deterministic. For instance, let t_1 and t_2 be $(b := 1; c := 2)$ and $(c := 3; d := 4)$. Figure 3 shows the two possible computations of $t_1 || t_2$, where s' and s'' are the final states.

Rules S9 and S10 describe the behavior above, where $(\sigma_i \downarrow \omega_i)$ is the contribution of transition t_i . These rules only differ in the order they apply these contributions, accounting for the non-deterministic conflict resolution method. This method is *dynamic* because the write-sets ω_i are built along the computation, ensuring a precise detection of conflicts. In [25], we explore a static conflict resolution method, where we assume a transition t assigns to all variables on the left of the assignment occurring in t . This method is simpler than DSYNC, since we build this set of variables without looking at the actual transition computation. It is adequate to some classes of restricted systems, but it is pessimistic, because it may indicate a write-conflict when no one actually happens. The present method offers a more precise treatment for write-conflicts.

We chose the synchronous combinator semantics above for many reasons. It seems to generalize the semantics of several synchronous formalisms, such as VHDL and evolving algebras, it restricts the interaction between component transitions, and it represents the conflict resolution method explicitly as a single and well-defined operation (the state update operation). As a consequence, it is easier to reason about component transitions independently, and the synchronous combinator shows up several properties. For instance, these combinator is idempotent, commutative, and **skip** is its neutral element.

The last rules in Figure 2 define a binary relation $F \triangleright \Sigma$ indicating that program F generates the computation $\Sigma = (\sigma_0, \omega_0)(\sigma_1, \omega_1)(\sigma_2, \omega_2) \dots$, where each ω_i names the variables written during the computation of σ_i . In essence, what programs add to transitions is a description of the initial states. According to S11, a computation is a (usually infinite) sequence of states and write-sets generated through the repeated execution of the program body that starts in a state satisfying the initial program condition. To define the synchronous combination of programs, S12 just unfolds the program combination.

