A Logic for Synchronous Transitions with Dynamic Conflict Detection

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. It adopts a synchronous computation model with dynamic write-conflict detection, and it handles non-termination and compositional proofs. 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.

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 [13], TLA [12], UNITY [7, 15], ST [22], and other formalisms [21]. 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 [17], and specification formalisms such as evolving algebras [10]. 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.

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 detects write-conflicts dynamically (during system execution). DSYNC also features a representation for transitions as (possibly non-terminating) imperative commands, a modular and compositional approach

*Partially supported by QaP-For/FAPERGS.
†Caixa postal 15064. 91501-970, Porto Alegre, Brazil. Fax +55(51)319-1576.

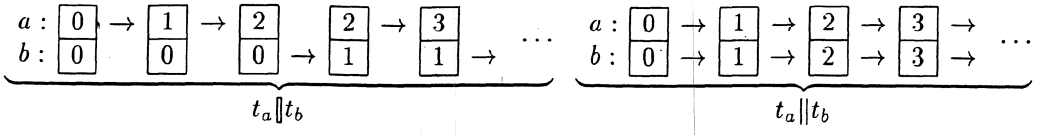

Figure 1: Asynchronous and synchronous computation models

to proof development, 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, conflict detection, and other features of DSYNC, but they are essential to the verification of VHDL and evolving algebras, for instance. Most works on formal formal verification concentrate on finite model-checkers and similar techniques (e.g., [8, 11]). Although they are quite effective for most tasks, these techniques are not appropriate to the verification of modular designs, parametric or regular systems, and data-intensive problems. DSYNC is an alternative to them, since it easily handles such conditions.

Work on DSYNC started as a wish to apply the UNITY logic on the verification of VHDL designs. Eventually, it has became a general formalism to describe and specify synchronous systems with dynamic conflict detection. It can handle hardware description languages, synchronous programming languages, and some specification formalisms (e.g., evolving algebras). In [19], we describe its application to VHDL. This paper describes DSYNC as general formalism for the verification of synchronous systems. It 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 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 $b := b + 1$. The left-half of figure 1 shows a computation of the asynchronous combination $t_a[\!]t_b$. 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 $t_a \| t_b$.

Notation $f : X \to Y$ indicates f is a partial function mapping elements of X to

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	::=	**skip**
F, G, prg	$\in$	Prg	programs			$var := exp$
α	$\in$	Value	values			cmd ; cmd
ω	$\in$	Write $\subseteq$ Var	write-sets			**if** $cond$ **then** cmd
σ	$\in$	State : Var $\to$ Value	states			**else** cmd
$\hat{\sigma}$	$\in$	Eval : Exp $\to$ Value	evaluator			**while** exp **do** cmd
Σ	$\in$	Comp : (State $\times$ Write)*	computation			

[S1] $\quad \mathbf{skip} \mid \emptyset \rhd \sigma \xrightarrow{\text{cmd}} \sigma$

[S2] $\quad x := e \mid \{x\} \rhd \sigma \xrightarrow{\text{cmd}} \sigma \oplus \{x \mapsto \hat{\sigma}(e)\}$

[S3] $$\frac{s_1 \mid \omega_1 \rhd \sigma \xrightarrow{\text{cmd}} \sigma_1 \quad s_2 \mid \omega_2 \rhd \sigma_1 \xrightarrow{\text{cmd}} \sigma_2}{s_1; s_2 \mid \omega_1 \cup \omega_2 \rhd \sigma \xrightarrow{\text{cmd}} \sigma_2}$$

[S4] $$\frac{\hat{\sigma}(b) = \mathbf{true} \quad s_1 \mid \omega_1 \rhd \sigma \xrightarrow{\text{cmd}} \sigma_1}{\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid \omega_1 \rhd \sigma \xrightarrow{\text{cmd}} \sigma_1}$$

[S5] $$\frac{\hat{\sigma}(b) = \mathbf{false} \quad s_2 \mid \omega_2 \rhd \sigma \xrightarrow{\text{cmd}} \sigma_2}{\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid \omega_2 \rhd \sigma \xrightarrow{\text{cmd}} \sigma_2}$$

[S6] $$\frac{\hat{\sigma}(b) = \mathbf{true} \quad (s;\ \mathbf{while}\ b\ \mathbf{do}\ s) \mid \omega \rhd \sigma \xrightarrow{\text{cmd}} \sigma_1}{\mathbf{while}\ b\ \mathbf{do}\ s \mid \omega \rhd \sigma \xrightarrow{\text{cmd}} \sigma_1}$$

[S7] $$\frac{\hat{\sigma}(b) = \mathbf{false}}{\mathbf{while}\ b\ \mathbf{do}\ s \mid \emptyset \rhd \sigma \xrightarrow{\text{cmd}} \sigma}$$

[S8] $$\frac{t \mid \omega \rhd \sigma \xrightarrow{\text{cmd}} \sigma'}{t \mid \omega \rhd \sigma \xrightarrow{\text{trn}} \sigma'}\ t \in \text{Cmd}$$

[S9] $$\frac{t_1 \mid \omega_1 \rhd \sigma \xrightarrow{\text{trn}} \sigma_1 \quad t_2 \mid \omega_2 \rhd \sigma \xrightarrow{\text{trn}} \sigma_2}{t_1 \| t_2 \mid \omega_1 \cup \omega_2 \rhd \sigma \xrightarrow{\text{trn}} \sigma \oplus ((\sigma_1 \downarrow \omega_1) \cup (\sigma_2 \downarrow \omega_2))}\ \omega_1 \cap \omega_2 = \emptyset$$

[S10] $$\frac{\hat{\sigma}_0(b) = \mathbf{true} \quad \omega_0 = \text{var}(b) \quad t \mid \omega_{i+1} \rhd \sigma_i \xrightarrow{\text{trn}} \sigma_{i+1}}{\langle b \mid t \rangle \rhd (\sigma_0, \omega_0)(\sigma_1, \omega_1)(\sigma_2, \omega_2) \ldots}$$

[S11] $$\frac{\langle b_1 \wedge b_2 \mid t_1 \| t_2 \rangle \rhd \Sigma}{\langle b_1 \mid t_1 \rangle \parallel \langle b_2 \mid t_2 \rangle \rhd \Sigma}$$

Figure 2: DSYNC transition system

elements of Y, and $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ represents a finite mapping of x_i to y_i. When f is undefined on x, we write $f(x) = \perp$. For f and g with disjoint domains, $f \cup g$ is the union function. Operation $f \downarrow Z$ produces f with its domain restricted to a set Z of variables names, and $f \oplus g$ denotes f updated with g:

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

Figure 2 presents the DSYNC transition system. A state σ is a mapping of variable

names to values, and $\widehat{\sigma}(e)$ denotes the value in σ of an expression e (or condition, or assertion). 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.

The rules in figure 2 present an operational semantics for the DSYNC transition system. The four-argument relations $s \mid \omega \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma'$ and $t \mid \omega \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma'$ indicate that the execution of command s or transition t in a state σ produces a state σ', assigning 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 write to y always. Rules S1 to S7 define the command semantics as usual [1], except they also collect the write-sets. S2 produces a singleton write-set, and the remaining rules only combine these sets. Rule S8 executes elementary transitions, and rule S9 describes the synchronous combination of transitions with dynamic detection of conflicts.

Two transitions generate a *write-conflict* when they try to assign to the same variable at the same time. This is an error condition halting program execution. To account for write-conflicts, we define the semantics for synchronous combinations as follows. To execute $t_1 \| t_2$, 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. When there is a write-conflict, the computation does not proceed. Otherwise, it does not matter the order we apply the contributions (computations are deterministic).

Rule S9 describes the behavior above, where $(\sigma_i \downarrow \omega_i)$ is the contribution of transition t_i. This method is *dynamic* because the write-sets ω_i are built along the computation, ensuring a precise detection of conflicts. In [20], we explore a static conflict detection method, where we assume a transition t assigns to all variables on the left of the assignments occurring in t. This method is simpler than DSYNC, since we build this set of variables without looking at the actual transition computation. However, it is pessimist, because it may indicate a write-conflict when no one actually happens. Nevertheless, it is adequate to some classes of restricted systems.

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. This combinator is idempotent, commutative, associative, and **skip** is its neutral element. It is worth notice that conflicting transitions generally do not show all this properties.

The last rules in figure 2 define a binary relation $F \rhd \Sigma$ indicating that program F generates the computation $\Sigma = (\sigma_0, \omega_0)(\sigma_1, \omega_1)(\sigma_2, \omega_2)\ldots$, 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 S10, 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, S11 just unfolds the program combination.

3 Logic

We use the DSYNC logic to verify statements about a DSYNC transition system. It is derived from the Hoare logic [1] and the UNITY [7, 15] logic. The DSYNC logic employs modified Hoare triples in the form $\{p\}\ s \mid \omega\ \{q\}$ and $\{\!|p|\!\}\ t \mid \omega\ \{\!|q|\!\}$. They represent the statement "if the execution of command s or transition t begins in a state where p holds, then it does terminate in a state where q holds, and it may write to the variables in ω". Assertions p and q are formulas from standard predicate logic over state variables.

We employ distinct notations for triples over commands and transitions to emphasize that the computation of a transition does not include the intermediate states generated during the computation of commands. We need to add write-sets to triples because the rules for synchronous combinations depend on them, and they are built dynamically, along the actual computation. However, a triple may describe several computations with distinct write-sets. Therefore, the write-set of a triple can be wider then the actual write-set built along a computation.

Using the semantics presented in the previous section, we may precisely describe the modified Hoare triples. Let σ be any state where $\hat{\sigma}(p) = \mathbf{true}$. The triple $\{p\}\ s \mid \omega\ \{q\}$ holds if there exist some σ' and ω' where $(s \mid \omega' \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma')$, and $\hat{\sigma}'(q) = \mathbf{true}$, and $\omega' \subseteq \omega$. Furthermore, for all σ' and ω' where $(s \mid \omega' \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma')$, we must get $\hat{\sigma}'(q) = \mathbf{true}$, and $\omega' \subseteq \omega$. Transition triples are similarly defined.

Figure 3 lists the rules comprising the DSYNC logic. It is organized in three layers, reflecting the transition system organization. The bottom layer comprising rules A1 to A6 is the standard Hoare logic for total correctness of commands [1] extended to deal with write-sets. Rule A1 allows the enlargement of write-sets, A3 ensures the assigned variable is included in the write-set, and the remaining rules only carry these sets forward. Rules B1 to B3 constitute the middle layer, allowing the verification of statements about transitions and their combinations. These rules define triples over transitions. B1 is an adaptation (strengthening and weakening) rule, B2 describes elementary transitions, and B3 describes the synchronous combinator.

To simplify the modular development of systems, rules for combinations need to be *compositional*. It means $\{\!|p|\!\}\ t_1 \| t_2 \mid \omega\ \{\!|q|\!\}$ must be derived from triples over t_1 and t_2 alone. However, the synchronous combination $t_1 \| t_2$ does not preserve all triples over t_1 and t_2. For instance, let t_a and t_b be $a := a + 1$ and $b := b + 1$. In this case, $\{\!|a{=}b|\!\}\ t_a \mid \omega\ \{\!|a{\neq}b|\!\}$ holds, but $\{\!|a{=}b|\!\}\ t_a \| t_b \mid \omega\ \{\!|a{\neq}b|\!\}$ does not. Actually, as figure 1 shows, what actually holds

$$[\text{A1}]\ \frac{p \Rightarrow p' \quad \{p'\}\ s \mid \omega'\ \{q'\} \quad \omega' \subseteq \omega \quad q' \Rightarrow q}{\{p\}\ s \mid \omega\ \{q\}} \qquad [\text{A2}]\ \{p\}\ \mathbf{skip} \mid \omega\ \{p\}$$

$$[\text{A3}]\ \frac{x \in \omega}{\{p^x_e\}\ x{:=}e \mid \omega\ \{p\}} \qquad [\text{A4}]\ \frac{\{p\}\ s_1 \mid \omega\ \{r\} \quad \{r\}\ s_2 \mid \omega\ \{q\}}{\{p\}\ s_1; s_2 \mid \omega\ \{q\}}$$

$$[\text{A5}]\ \frac{\{p \wedge b\}\ s_1 \mid \omega\ \{q\} \quad \{p \wedge \neg b\}\ s_2 \mid \omega\ \{q\}}{\{p\}\ \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid \omega\ \{q\}} \qquad [\text{A6}]\ \frac{\{p \wedge b \wedge e{=}X\}\ s \mid \omega\ \{p \wedge e{<}X\}}{\{p\}\ \mathbf{while}\ b\ \mathbf{do}\ s \mid \omega\ \{p \wedge \neg b\}}$$

$$[\text{B1}]\ \frac{p \Rightarrow p' \quad \{\!|p'|\!\}\ t \mid \omega'\ \{\!|q'|\!\} \quad \omega' \subseteq \omega \quad q' \Rightarrow q}{\{\!|p|\!\}\ t \mid \omega\ \{\!|q|\!\}} \qquad [\text{B2}]\ \frac{\{p\}\ t \mid \omega\ \{q\}}{\{\!|p|\!\}\ t \mid \omega\ \{\!|q|\!\}}\ t \in \mathrm{Cmd}$$

$$[\text{B3}]\ \frac{\{\!|p_1|\!\}\ t_1 \mid \omega_1\ \{\!|q_1|\!\} \quad \{\!|p_2|\!\}\ t_2 \mid \omega_2\ \{\!|q_2|\!\}}{\{\!|p_1 \wedge p_2|\!\}\ t_1 \| t_2 \mid \omega_1 \cup \omega_2\ \{\!|q_1 \wedge q_2|\!\}}\ \omega_1 \cap \omega_2 = \emptyset \wedge \mathrm{var}(q_1) \cap \omega_2 = \emptyset \wedge \mathrm{var}(q_2) \cap \omega_1 = \emptyset$$

$$[\text{C1}]\ \frac{\mathbf{init}\ p'\ \mathbf{in}\ F \mid \omega' \quad p' \Rightarrow p \quad \omega' \subseteq \omega}{\mathbf{init}\ p\ \mathbf{in}\ F \mid \omega} \qquad [\text{C2}]\ \frac{\mathrm{var}(b) \subseteq \omega}{\mathbf{init}\ b\ \mathbf{in}\ \langle b \mid t \rangle \mid \omega}$$

$$[\text{C3}]\ \frac{p \Rightarrow p' \quad p'\ \mathbf{co}\ q'\ \mathbf{in}\ F \mid \omega' \quad \begin{matrix} q' \Rightarrow q \\ \omega' \subseteq \omega \end{matrix}}{p\ \mathbf{co}\ q\ \mathbf{in}\ F \mid \omega} \qquad [\text{C4}]\ \frac{\{\!|p|\!\}\ t \mid \omega\ \{\!|q|\!\}}{p\ \mathbf{co}\ q\ \mathbf{in}\ \langle b \mid t \rangle \mid \omega}$$

$$[\text{C5}]\ \mathbf{inv}\ p\ \mathbf{in}\ F \mid \omega \equiv \begin{matrix}(\mathbf{init}\ p\ \mathbf{in}\ F \mid \omega) \wedge \\ (p\ \mathbf{co}\ p\ \mathbf{in}\ F \mid \omega)\end{matrix} \qquad [\text{C6}]\ \frac{\mathbf{inv}\ r\ \mathbf{in}\ F \mid \omega \quad p\ \mathbf{co}\ q\ \mathbf{in}\ F \mid \omega}{(p \wedge r)\ \mathbf{co}\ (q \wedge r)\ \mathbf{in}\ F \mid \omega}$$

$$[\text{C7}]\ \frac{\begin{matrix}\mathbf{inv}\ r\ \mathbf{in}\ F \mid \omega \\ (p \wedge r)\ \mathbf{co}\ (q \wedge r)\ \mathbf{in}\ F \mid \omega\end{matrix}}{p\ \mathbf{co}\ q\ \mathbf{in}\ F \mid \omega} \qquad [\text{C8}]\ \frac{p \Rightarrow p' \quad p'\ \mathbf{leads}\ q'\ \mathbf{in}\ F \mid \omega' \quad \begin{matrix} q' \Rightarrow q \\ \omega' \subseteq \omega \end{matrix}}{p\ \mathbf{leads}\ q\ \mathbf{in}\ F \mid \omega}$$

$$[\text{C9}]\ p\ \mathbf{leads}\ p\ \mathbf{in}\ F \mid \omega \qquad [\text{C10}]\ \frac{p\ \mathbf{co}\ q\ \mathbf{in}\ F \mid \omega}{p\ \mathbf{leads}\ q\ \mathbf{in}\ F \mid \omega}$$

$$[\text{C11}]\ \frac{\begin{matrix}p_1\ \mathbf{leads}\ q_1\ \mathbf{in}\ F \mid \omega_1 \\ p_2\ \mathbf{leads}\ q_2\ \mathbf{in}\ F \mid \omega_2\end{matrix}}{(p_1 \vee p_2)\ \mathbf{leads}\ (q_1 \vee q_2)\ \mathbf{in}\ F \mid \omega_1 \cup \omega_2} \qquad [\text{C12}]\ \langle b_1 | t_1 \rangle \,\|\, \langle b_2 | t_2 \rangle \equiv \langle b_1 \wedge b_2 \mid t_1 \| t_2 \rangle$$

Figure 3: DSYNC logic

is $\{\!|a{=}b|\!\}\ t_a \| t_b \mid \omega\ \{\!|a{=}b|\!\}$. Therefore, as a general case, the combination $t_1 \| t_2$ preserves a triple $\{\!|p|\!\}\ t_1 \mid \omega_1\ \{\!|q|\!\}$ over the component transition t_1 if the other component t_2 does not change any variables occurring in q. Rule B3 describes this case. The proviso in this rule uses the write-sets of each component transition to ensure the condition above for both components.

$$
\begin{array}{rll}
\textbf{init } p \textbf{ in } F \mid \omega & \text{iff} & \Sigma_0(p) \wedge \Sigma_0 \subseteq \omega \\
p \textbf{ co } q \textbf{ in } F \mid \omega & \text{iff} & (\forall i : \Sigma_i(p) \Rightarrow (\Sigma_{i+1}(q) \wedge \Sigma_{i+1} \subseteq \omega)) \\
\textbf{inv } p \textbf{ in } F \mid \omega & \text{iff} & (\forall i : \Sigma_i(p) \wedge \Sigma_i \subseteq \omega) \\
p \textbf{ leads } q \textbf{ in } F \mid \omega & \text{iff} & (\forall i : \Sigma_i(p) \Rightarrow (\exists j : j \geq i \wedge \Sigma_j(q) \wedge (\forall k : i \leq k \leq j \Rightarrow \Sigma_k \subseteq \omega)))
\end{array}
$$

Figure 4: Temporal properties

The top layer in the DSYNC logic is a temporal logic dealing with statements about complete program computations. Formulas in this layer are called *properties*. They are built from a temporal connective applied to assertions, they are always attached to programs, and they cannot be nested. Figure 4 lists the properties and their meaning. The left column shows a property over a program F writing to variables in ω, and the right column shows a condition on the computations $\Sigma = (\sigma_0, \omega_0)(\sigma_1, \omega_1)(\sigma_2, \omega_2)\ldots$ generated by F. A property holds if all computations of F satisfy the corresponding condition. In the right column, $\Sigma_i(p)$ means that there is a position i in Σ, and assertion p holds in the corresponding state, i.e., $\Sigma_i \neq \perp$ and $\widehat{\sigma}_i(p) = \textbf{true}$. Likewise, $\Sigma_i \subseteq \omega$ means that position i only writes to the variables in ω, i.e., $\omega_i \subseteq \omega$.

Informally, properties may be read as follows: **init** p **in** $F|\omega$ means "p holds in the first state, and only the variables in ω are initialized". p **co** q **in** $F \mid \omega$ means "if p holds, then q holds in the next state, assigning to the variables in ω only", **inv** p **in** $F \mid \omega$ means "p holds in all states, and only the variables in ω are assigned", and p **leads** q **in** $F \mid \omega$ means "if p holds, then q will hold in some future state, meanwhile assigning to the variables in ω only".

It is implicit in the comments above that properties only consider the *reachable states* of a program F, i.e., the states generated through a computation of F. Depending on initial conditions, this set is smaller than the set of all possible states. This difference affects the statements we may prove. For instance, let t_b be $b := b + a$, and let F_b be $\langle a = 2 \mid t_b \rangle$. We cannot prove $\{\!|even(b)|\!\}\ t_b \mid \omega\ \{\!|even(b)|\!\}$. However, we may prove $even(b)$ **co** $even(b)$ **in** $F_b \mid \omega$ because a always holds 2 in the computations of F_b.

Rules C1 to C12 define the temporal connectives. C1, C3, and C8 are adaptation rules, C2 describes the initial states, C4 is the base case for **co** properties, and C5 defines temporal invariants. The *substitution rules* C6 and C7 allow for the introduction and elimination of temporal invariants in assertions. These two rules are the device that restricts properties to the set of reachable states. Rules C8 to C11 define the **leads** connective, and C12 describes program combinations.

The DSYNC temporal logic is derived from the UNITY logic. The link between these logics follows from the fact that a synchronous transition program corresponds to a UNITY program with a single (non-deterministic) transition. Since we consider these restricted UNITY programs only, some UNITY rules become simpler in DSYNC. When moving to DSYNC, we also review all property definitions to account for non-determinism, non-terminating transitions, and write-sets. Additionally, DSYNC omits an infinitary UNITY rule for **leads** properties, because all DSYNC programs are finite.

$$[\text{D1}]\ \frac{\{p_1\}\ s \mid \omega\ \{q_1\} \quad \{p_2\}\ s \mid \omega\ \{q_2\}}{\{p_1 \vee p_2\}\ s \mid \omega\ \{q_1 \vee q_2\}} \qquad [\text{D2}]\ \frac{p_1\ \textbf{co}\ q_1\ \textbf{in}\ F \mid \omega \quad p_2\ \textbf{co}\ q_2\ \textbf{in}\ F \mid \omega}{(p_1 \vee p_2)\ \textbf{co}\ (q_1 \vee q_2)\ \textbf{in}\ F \mid \omega}$$

$$[\text{D3}]\ \frac{p_1\ \textbf{co}\ q_1\ \textbf{in}\ F_1 \mid \omega_1 \quad p_2\ \textbf{co}\ q_2\ \textbf{in}\ F_2 \mid \omega_2}{(p_1 \wedge p_2)\ \textbf{co}\ (q_1 \wedge q_2)\ \textbf{in}\ F_1 \| F_2 \mid \omega_1 \cup \omega_2}\ \begin{array}{l}\omega_1 \cap \omega_2 = \emptyset\ \wedge \\ \text{var}(q_1) \cap \omega_2 = \emptyset\ \wedge\ \text{var}(q_2) \cap \omega_1 = \emptyset\end{array}$$

Figure 5: Additional inference rules

As a design decision, DSYNC omits the **skip** steps (stuttering steps [12]) in the definition of **co**. We believe they are not necessary in a synchronous transition system because the synchronous combination does not interleave states in a computation. As a consequence, **co** properties are enough to define progress properties (see rule C10), and we may drop the concept of transient predicates [15] and existentially quantified triples [7]. However, to preserve the semantics of **leads**, we add rule C9.

To make the DSYNC logic useful in practice, the basic set of rules of figure 3 must be extended with several derived rules. Figure 5 shows some of these rules. Derived rules include adaptation rules for command triples which are lifted to properties and transition triples. For instance, rule D1 for command triples originates rule D2 for **co** properties. Other derived inference rules are inherited (with little changes) from UNITY. Finally, some derived rules are specific to synchronous transitions. For instance, D3 describes the synchronous combination of **co** properties. This last group of derived rule is essential because they reflect basic aspects of synchronous transitions.

To prove some derived rules (for instance, D2), we need induction on the proof length. For transition triples, B2 gives the base case, and B1, B3, and B4 give the induction steps. For **co** properties, C4 gives the base case, and C3, C6, and C7 give the induction steps. In this proof, we expand the **inv** premise in rules C6 and C7 for its definition C5, exposing a hidden premise on **co**. We deal with **leads** properties in a similar way.

We claim that the DSYNC logic is *sound*, i.e., rules in figure 3 only generate true formulas. To justify this claim, we consider each layer separately, greatly simplifying the soundness proof. The soundness of the bottom layer comes from the Hoare logic [1], and the soundness of the middle and top layers follows from the semantics of the DSYNC transition system, and from the definitions of triples and properties.

To illustrate the soundness proof for the middle layer, we sketch a proof that rule B3 is sound. Let σ be a state where $\hat{\sigma}(p_1) = \hat{\sigma}(p_2) = \textbf{true}$. Assume the premises and the proviso of B3 are true. From this assumption, it follows that the component transitions t_1 and t_2 terminate when their computations start in σ, the resulting states satisfy q_1 and q_2, and they only write to ω_1 and ω_2, for $\omega_1 \cap \omega_2 = \emptyset$. Rule B3 is sound if all computations of $t_1 \| t_2$ starting in σ also terminate, the resulting states also satisfy q_1 and q_2, and they only write to $\omega_1 \cup \omega_2$. We demonstrate this statement next.

Figure 2 defines the semantics of $t_1 \| t_2$ using rule S9. From this rule and the assumptions above, it follows that $t_i \mid \omega_i' \rhd \sigma \xrightarrow{\text{trn}} \sigma_i$, and $\hat{\sigma}_i(q_i) = \textbf{true}$, and $\omega_i' \subseteq \omega_i$. Therefore, according to S9, the execution of $t_1 \| t_2$ beginning in σ terminates producing σ' given by

$\sigma \oplus ((\sigma_1 \downarrow \omega_1') \cup (\sigma_2 \downarrow \omega_2'))$, and writing to $\omega_1' \cup \omega_2'$. Since $\omega_i' \subseteq \omega_i$, we already get that the synchronous combination only writes to $\omega_1 \cup \omega_2$. It remains to show that q_1 and q_2 holds in σ'. We consider q_1 only as the argument for q_2 is symmetric.

When two states agree on the value of all variables of an expression, this expression has the same value in both states. We claim that q_1 holds in σ' because it holds in σ_1, and σ' and σ_1 agree on the value of all variables of q_1. Let x be a variable in $\mathrm{var}(q_1)$. From the proviso of B3 and from $\omega_i' \subseteq \omega_i$, we get that either $x \in \omega_1'$, or $x \notin \omega_1'$ and $x \notin \omega_2'$. Let φ and φ' be states, let ν be a write-set, and let y be a variable. The definitions of $\oplus$ and $\downarrow$ entail the propositions bellow:

[P1] if $y \in \mathrm{dom}(\varphi')$, then $(\varphi \oplus \varphi')(y) = \varphi'(y)$;
[P2] if $y \notin \mathrm{dom}(\varphi')$, then $(\varphi \oplus \varphi')(y) = \varphi(y)$;
[P3] if $y \notin \nu$ and $t \mid \nu \rhd \varphi \xrightarrow{\mathrm{trn}} \varphi'$, then $\varphi'(y) = \varphi(y)$.

First, assume $x \in \omega_1'$. From P1 and the definition of σ', it follows that $\sigma'(x) = \sigma_1(x)$. Alternatively, assume $x \notin \omega_1'$. In this case, we also have that $x \notin \omega_2'$. From P2 and the definition of σ', we get that $\sigma'(x) = \sigma(x)$. But P3 and the definition of σ_1 ensure that $\sigma_1(x) = \sigma(x)$ too. So, we also get that $\sigma'(x) = \sigma_1(x)$. Since there are no other cases to consider, we get that σ' and σ_1 agree on the value of all variables in $\mathrm{var}(q_1)$, and q holds in σ'. A symmetric argument shows that the resulting state of $t_1 \| t_2$ also satisfies q_2. Therefore, rule B3 is sound. Similar arguments show that all rules in figure 3 are sound.

It is worth observe that the soundness proof of the top layer is independent and simpler than the soundness proof of UNITY, and avoids some foundation problems on the UNITY logic [16]. This is a consequence of some differences between DSYNC and UNITY, such as the synchronous combinator, and the absence of infinitary rules and stuttering steps. We still have no completeness results for DSYNC logic, but we are working on this.

The layered organization of this logic is an important design decision because it allows for a separation of concerns. Results about the bottom layer are inherited from the standard Hoare logic. In the middle layer, we analyze the synchronous combinator without concerns on the notation for elementary transitions, or on the temporal logic. The top layer deals with reachable states and complete program computations. Although this layer is based on the UNITY logic, there are evidences that we could to switch to other similar logic such as TLA or the Manna and Pnueli logic. Such change does not affect the previous layers, and the new temporal logic could equally inherit the results from other layers.

4 Pragmatics and Extensions

DSYNC inherits many specification and verification techniques from its base formalisms (UNITY and the Hoare logic), but we need to develop new techniques to cope with the specificities of DSYNC. To prove a property or triple, we usually begin at the conclusion and proceed backwards to the given premises, using the proof rules to break formulas into sub-formulas. The overall proof organization reflects the program structure, and the organization of its computations. As the synchronous combinator is the main control

structure in DSYNC, the rule for this construction strongly influence the organization of proofs in this formalism.

Rule B3 breaks a triple over a synchronous combination into sub-triples over its component transitions, but the proviso in these rules impose restrictions on the sub-triples and their write-sets. To satisfy these restrictions, we usually formulate sub-triples that reflect the contribution of each transition to the desired result. For instance, let t_1 and t_2 be (**if** $x \geq 0$ **then** $y := x$ **else skip**) and (**if** $x < 0$ **then** $y := -x$ **else skip**). Bellow, we list a proof for property $x = X$ **co** $y = abs(X)$ **in** $t_1 \| t_2 \mid \{x\}$:

(1)	$x = X \wedge x \geq 0$ **co** $y = X \wedge X \geq 0$ **in** $t_1 \mid \{x\}$	A1–6, B2, and C4
(2)	$x \geq 0$ **co true in** $t_2 \mid \emptyset$	A1–6, B2, and C4
(3)	$x = X \wedge x \geq 0$ **co** $y = X \wedge X \geq 0$ **in** $t_1 \| t_2 \mid \{x\}$	D6 in (1, 2), and C3
(4)	$x = X \wedge x < 0$ **co** $y = -X \wedge X < 0$ **in** $t_1 \| t_2 \mid \{x\}$	similar to (1–3)
(5)	$(x = X \wedge x \geq 0) \vee (x = X \wedge x < 0)$ **co** $(y = X \wedge X \geq 0) \vee (y = -X \wedge X < 0)$ **in** $t_1 \| t_2 \mid \{x\}$	D2 in (3, 4)
(6)	$x = X$ **co** $y = abs(X)$ **in** $t_1 \| t_2 \mid \{x\}$	C3 in (5)

Upper-case variables stand for *rigid variables*. These variables are not modified through assignment, preserving their value between states. We use these variables in properties to state relations between the initial and final value of program variables. Using these variables, the properties in lines 1 and 2 describe the effect (contribution) of the elementary transitions on the final value of variables when $x \geq 0$. Next, line 3 describes the synchronous combination of these transitions, where the dynamically built write-sets ensure there is no conflict between t_1 and t_2. As a complement, line 4 deals with $x < 0$. The desired property follows from the disjunction of both cases.

This example illustrates a general proof tactic. Working backwards, we split a property into sub-properties corresponding to computation paths sharing the same write-sets, and then slip the sub-properties and their write-sets into the component transitions. However, as we proceed to more complex systems and specifications, we need more sophisticate proof techniques. Elsewhere [18], we present a catalog of basic specification and verification techniques for DSYNC which may be applied in many situations. Like the tactic above, they depend on the program structure, on the organization of its computations, and on the kind of property that must be proved. Broadly, these techniques reflect the programmer knowledge on the system it is designing, and they are easy to select and apply. Therefore (except for some hard cases), usually it is quite easy to find a proof in DSYNC.

DSYNC deals with *open systems*, which are systems interacting with an environment that is not described along the system. The proper operation of an open system usually depends on some assumptions on the environment. To specify an open system F, we employ a *conditional property* ("if-then" rule) where a generic (unspecified) program G represents the environment. The premises are ordinary properties over G stating environment assumptions, and the conclusion is an ordinary property over $F \| G$ stating the system interaction. For example, consider a program *Save* behaving as follows. When *req* signals a request, it pushes the value of x into *buf*, and sets *done*. When the request is removed, *done* is reset. The conditional property bellow is part of the description of *Save*.

$$\frac{req \wedge x = X \wedge \neg done \ \textbf{co}\ req \wedge x = X \ \textbf{in}\ G \mid req, x, \omega}{req \wedge x = X \wedge buf = B \ \textbf{leads}\ buf = cons(X, B) \wedge done \ \textbf{in}\ Save \| G \mid done, buf, req, x, \omega}$$

The premise indicates that the environment does not change the value of x and *req* when there is an unattended request, and the conclusion indicates that eventually x will be stored in *buf*, *done* will be set. It means *Save* works well under the assumption the environment keeps a request until it is attended. This is a common specification pattern in DSYNC. Elsewhere [18], we analyze similar specification and verification patterns.

Open systems are fundamental to modular system development. Each component in a library may be regarded as an open system to be plugged to a complete system latter. Conditional properties allow for the verification of properties of a component based on properties of other components. It also allows for the derivation of system properties from component properties. These tasks do not depend on any knowledge on the actual components. It means a component may be replaced by other, as long as both present the same properties. Since any property may be a premise, we may place arbitrary restrictions on the environment. Other approaches (e.g., model-checkers [8]) impose severe limits on environment restrictions, what limits their applicability.

DSYNC may be extended in several directions. Simple but very useful additions are generic parameters and regular structures. Generic parameters are unspecified constants. They usually stand for the amount of some system resource (size of a vector, number of available network connections, and so on). DSYNC represents a generic parameter as a rigid variable. A regular structure is a set of transitions following a common syntactical pattern parametrized on an argument I. To represent the regular structure $t_0 \| t_1 \| \ldots \| t_{E-2} \| t_{E-1}$, DSYNC adopts the notation $\langle 0 \leq I < E : t_I \rangle$. Generic parameters and regular structures allow for very compact system descriptions. For instance, assume N is a generic parameter. Let a be a vector, and let F_a be the program $\langle 0 \leq I < N : a[I] := 0 \rangle$. This program sets the first N elements of a to zero. In [20], we study a treatment for generic parameters and regular structures in a static version of DSYNC. A similar approach may be adopted for the present version of DSYNC.

5 Related Work

The features of DSYNC discussed along this paper are essential to the verification of synchronous systems. However, they are not present in other formalisms based on a first-order liner-time temporal logic. UNITY [7, 15] is a typical formalism based on a transition system and a temporal logic. It includes a synchronous combinator as a syntactic sugar for complex transitions, as its logic does not include rules for this combinator. Therefore, this combinator becomes a syntactic sugar to describe complex transitions. UNITY syntacticly forbids non-terminating transitions. It also forbids write-conflicts, but does not specify any test to prevent them. Other formalisms based on a first-order linear-time temporal logic (such as the Manna and Pnueli logic [13], TLA [12], and ST [22]) present similar problems.

Besides the transition systems and temporal logics, DSYNC is also related to research on the semantics and verification of VHDL [5, 9]. Some works [3] on this field amount to

complex descriptions which are not appropriate to the verification of actual designs, while other works [4, 6] cover only restricted language subsets. The static version of DSYNC also addresses a restricted subset of VHDL [19]. However, it handles some features that most semantics ignore (e.g., generic parameters and regular structures), and it can be easily extended to a more general (dynamic) setting. It is worth observe that DSYNC includes a general and clear analysis of the synchronous combinator, while most works mix such analysis with the study of other aspects of VHDL. This approach may lead to very subtle errors.

We may employ DSYNC in other verification tasks beyond VHDL. Evolving algebras [10] are a specification formalism which adopts a synchronous computation model. Therefore, we may verify properties of an evolving algebra using the DSYNC logic. Nowadays, there is a lot of work on formal verification which employs finite model-checkers, automata, and similar techniques [14]. Such approaches are fully automated and quite effective, but they are not appropriate to all situations. For instance, they are not do not deal with first-order (non-propositional) specifications, and they are not good at the verification of modular systems. A linear-time temporal logic such as DSYNC is a nice complement to these techniques, since it easily handles those hard situations. Some works explore this complementarity nature [4], and we plan to explore this path in some future developments of DSYNC.

6 Last Remarks

DSYNC is composed of a transition system and an associated linear-time temporal logic. The DSYNC transition system adopts the synchronous computation model with dynamic conflict detection and non-termination. and the DSYNC allows the verification of properties of such transition systems. Proofs in this logic are compositional, meaning properties of a system may be derived from properties of its components without recourse to the actual definition of components. The DSYNC logic is organized in layers, and there is a catalog of specification and verification techniques for this logic. These features are essential to the verification of synchronous system, but they are not present in similar formalisms. DSYNC allows for the application of a first-order linear-time temporal logic to the verification of synchronous systems. It may be employed in several application fields, it is quite general and easy to use, and it is complementary to other more restricted and automated verification methods. Therefore, we believe DSYNC is a good formalism, and we plan to continue its development.

References

[1] K. R. Apt et al. *Verification of Sequential and Concurrent Programs.* Springer, 1991.

[2] G. Berry. The Esterel v5 language primer. Ecole des Mines and INRIA, 1997.

[3] E. Börger et al. The semantics of behavioral VHDL'93 descriptions. In *EURO-VHDL'94*, pages 500–5, 1994.

[4] D. Borrione et al. Formal verification of VHDL descriptions in the Prevail environment. *IEEE Design & Test of Computers*, 9(2):42–55, 1992.
[5] D. Borrione, editor. Special issue on VHDL semantics. *Formal Methods in System Design*, 7(1/2), 1995.
[6] P. T. Breuer et al. A refinement calculus for the synthesis of verified hardware descriptions in VHDL. *ACM TOPLAS*, 19(4):586–616, 1997.
[7] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.
[8] D. Déharbe et al. The CV model-checker. In *FMCAD'98*. Springer, 1998.
[9] C. Delgado Kloos et al., editors. *Formal Semantics for VHDL*. Kluwer, 1995.
[10] Y. Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*. Oxford University, 1995.
[11] L. Jagadeesan et al. Safety property verification of ESTEREL programs and applications to telecommunications software. In *CAV-95*, 1995.
[12] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, 1994.
[13] Z. Manna et al. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
[14] K. C. McMillan. *Symbolic Model Checking*. Kluwer, Boston, 1993.
[15] J. Misra. A logic for concurrent programming. University of Texas at Austin, 1994.
[16] P. Päppinghaus. On the logic of UNITY. *Theoretical Computer Science*, 139:27–67, 1995.
[17] D. L. Perry. *VHDL*. McGraw-Hill, 1991.
[18] V. M. Rodrigues. *Transições Síncronas, Lógica Temporal e VHDL*. Tese de doutorado, CPGCC, UFRGS, 1998.
[19] V. M. Rodrigues and F. R. Wagner. A temporal logic for data-flow VHDL. In *SBCCI'98*, pages 91–94. IEEE Computer Society, 1998.
[20] V. M. Rodrigues and F. R. Wagner. A logic for synchronous transition. In *SBLP'99*. Sociedade Brasileira de Computação, 1999.
[21] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.
[22] J. Staunstrup. *A Formal Approach to Hardware Design*. Kluwer, 1994.