A Logic to Specify and Verify Synchronous Transitions

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

Abstract

This paper introduces a formalism named SINC 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 SINC transition system adopts a synchronous computation model, includes a method to solve write-conflicts, and represents transitions as possibly non-terminating imperative commands. The SINC logic allows for formal reasoning about SINC transition systems using compositional and modular proofs. Such features are important to the verification of a large class of systems, but they are missing in other formalisms based on transition systems and temporal logics. This paper also discusses some of the pragmatics in specifying and verifying systems using SINC, and presents extensions to deal with generic parameters and regular structures. SINC 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 [17, 18], TLA [16], UNITY [8, 20], ST [29], and other formalisms [25, 28]. 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.

This paper introduces a formalism named SINC aimed at the description and verification of synchronous systems using a transition system and a first-order linear-time temporal logic. Besides the synchronous computation model, other salient features of SINC are a representation for transitions as (possibly non-terminating) imperative commands, a method to solve write-conflicts, a modular and compositional approach to proof development, a treatment for generic parameters and regular structures, and a good catalog of elementary verification techniques. These distinguishing features of SINC are essential to the verification of a class of systems that includes VHDL and evolving algebras. However, they are not appropriately addressed in the formalisms listed earlier. Therefore, we believe that SINC contributes to the research on linear-time temporal logics.

Most works on formal verification 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 is not appropriate to parametric and regular systems too, because such systems may generate large or infinite sets of states. Other works employ more general formalisms, but most of these works deal with restricted systems or result in complex formalisms which are hard to apply on the verification of actual systems [7, 27]. SINC is quite general, yet easy to use. Therefore, we may also see this formalism as a contribution to the work on the verification of synchronous systems.

*Partially supported by QaP-For/FAPERGS.

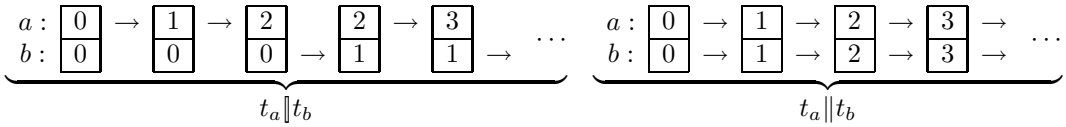

Figure 1: Asynchronous and synchronous computation models

Work on SINC started as a wish to apply the UNITY logic on 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. 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 SINC as general formalism for the verification of synchronous systems, develops some foundation work, and discusses important description and proof techniques for SINC. The resulting formalism is quite general and flexible, although it may not be fully automated as finite model-checkers.

This paper is organized as follows. Section 2 discusses the synchronous computation model and introduces the SINC transition system, and section 3 presents the SINC logic. Section 4 studies methods to apply this formalism in the specification and verification of synchronous systems. Section 5 considers some extensions to the basic formalism. Section 6 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, SINC 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 $t_a\|t_b$.

Figure 2 presents the SINC 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 assume some syntactical restriction on initial conditions to ensure they are consistent.

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). Notation $f : X \to Y$ indicates f is a partial function mapping elements of X to 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 $f(x) = \bot$. 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:

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

The rules in figure 2 present an operational semantics for the SINC transition system. The ternary relations $s \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma'$ and $t \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma'$ indicate that the execution of command s or transition t in a state σ produces a state σ'. As rules S1 to S3 indicate, we define the command semantics as usual [1]. To shorten the presentation, we omit the rules for conditional and iterative commands. Rule S4 executes elementary transitions, and rules S5 and S6 define the synchronous combination of transitions. These two rules include a 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 $t_1\|t_2$, we give a

x, var	$\in$	Var	variables	*cmd*	::=	**skip**
e, exp	$\in$	Exp	expressions			*var* := *exp*
b, cond	$\in$	Cond	conditions			*cmd* ; *cmd*
s, cmd	$\in$	Cmd	commands			**if** *cond* **then** *cmd* **else** *cmd*
t, trn	$\in$	Trn	transitions			**while** *cond* **do** *cmd*
F, G, prg	$\in$	Prg	programs	*trn*	::=	*cmd*
α	$\in$	Val	values			*trn* $\parallel$ *trn*
σ	$\in$	State : Var $\to$ Val	states	*prg*	::=	$\langle \mathit{cond} \mid \mathit{trn} \rangle$
$\widehat{\sigma}$	$\in$	Eval : Exp $\to$ Val	evaluator			*prg* $\parallel$ *prg*
Σ	$\in$	Comp : State*	computation			

[S1] $\quad \textbf{skip} \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma$

[S2] $\quad x := e \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma \oplus \{x \mapsto \widehat{\sigma}(e)\}$

[S3] $$\frac{s_1 \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma_1 \quad s_2 \rhd \sigma_1 \overset{\text{cmd}}{\rightharpoonup} \sigma_2}{s_1; s_2 \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma_2}$$

[S4] $$\frac{t \rhd \sigma \overset{\text{cmd}}{\rightharpoonup} \sigma'}{t \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma'} \; t \in \text{Cmd}$$

[S5] $$\frac{t_1 \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma_1 \quad t_2 \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma_2}{t_1 \| t_2 \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma \oplus (\sigma_1 \downarrow \text{write}(t_1)) \oplus (\sigma_2 \downarrow \text{write}(t_2))}$$

[S6] $$\frac{t_1 \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma_1 \quad t_2 \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma_2}{t_1 \| t_2 \rhd \sigma \overset{\text{trn}}{\rightharpoonup} \sigma \oplus (\sigma_2 \downarrow \text{write}(t_2)) \oplus (\sigma_1 \downarrow \text{write}(t_1))}$$

[S7] $$\frac{\widehat{\sigma}_0(b) = \textbf{true} \quad t \rhd \sigma_i \overset{\text{trn}}{\rightharpoonup} \sigma_{i+1}}{\langle b \mid t \rangle \rhd \sigma_0 \sigma_1 \sigma_2 \ldots}$$

[S8] $$\frac{\langle b_1 \wedge b_2 \mid t_1 \| t_2 \rangle \rhd \Sigma}{\langle b_1 \mid t_1 \rangle \parallel \langle b_2 \mid t_2 \rangle \rhd \Sigma}$$

Figure 2: SINC transition system

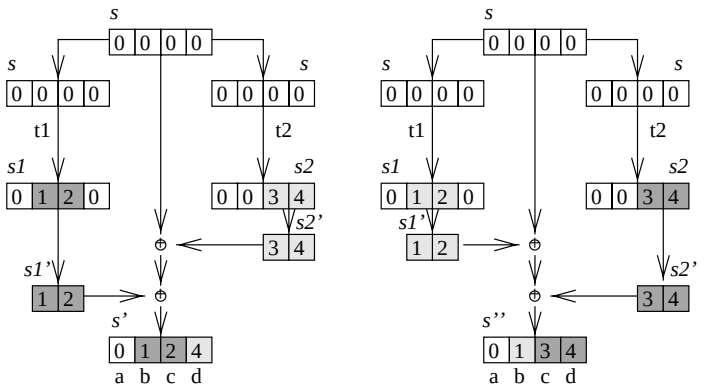

Figure 3: Execution of a synchronous combination

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 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 S5 and S6 describe the behavior above. Let write(t), the *write-set* of t, be the set of variables names that

occur on the left of some assignment in t. These are the variables that t may change when it executes. In S5 and S6, $(\sigma_i \downarrow \text{write}(t_i))$ is the contribution of transition t_i. These rules differ in the order they apply the contributions, accounting for the non-deterministic conflict resolution method. This method is static because write(t) results from a static analysis of programs. Under some conditions, it may restore conflicting variables to their initial value. Usually, this is not a problem, and this condition never happens in VHDL or evolving algebras. Nevertheless, in [23], we explore a dynamic version of SINC where this situation does not occur.

Due to write conflicts, the synchronous combination shows up some surprising behavior. It is idempotent, commutative, and **skip** is its neutral element, but it is not associative. For instance, let t_1 be $(x := 0;\ y := 0)$, and let t_2 and t_3 be $(x := 1)$ and $(y := 1)$. In this case, $t_1 \| (t_2 \| t_3) \neq (t_1 \| t_2) \| t_3$. However, if all pairs of transitions write to the same set of variables, i.e., write$(t_1) \cap$ write$(t_2) =$ write$(t_2) \cap$ write$(t_3) =$ write$(t_1) \cap$ write(t_3), then the synchronous combination is associative, i.e., $t_1 \| (t_2 \| t_3) = (t_1 \| t_2) \| t_3$. As a special case, it is associative when transitions are conflict-free.

The last rules in figure 2 define a binary relation $F \rhd \Sigma$ indicating that program F generates the computation $\Sigma = \sigma_0\sigma_1\sigma_2 \ldots$. In essence, what programs add to transitions is a description of the initial states. According to S7, a computation is a (usually infinite) sequence of states 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, S8 just unfolds the program combination.

3 Logic

We use the SINC logic to verify statements about a SINC transition system. It is derived from the Hoare logic [1] and the UNITY [8, 20] logic. The triples $\{p\}\ s\ \{q\}$ and $\{\!|p|\!\}\ t\ \{\!|q|\!\}$ represent the following statement: if the execution of command s or transition t begins in state where p holds, then it terminates, and q holds in the resulting state. 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.

Figure 4 lists the rules comprising the SINC logic. It is organized in three layers, reflecting the transition system organization. The bottom layer is the standard Hoare logic for total correctness of commands [1]. It allows the verification of statements about the computation of elementary transitions. Rules A1 to A4 are a sample of this layer. We omit other rules to shorten the presentation.

Rules B1 to B4 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 rules B3 and B4 describe the synchronous combinator. To simplify the modular development of systems, rules for combinations need to be compositional. It means $\{\!|p|\!\}\ t_1 \| t_2\ \{\!|q|\!\}$ 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, $\{\!|a{=}b|\!\}\ t_a\ \{\!|a{\neq}b|\!\}$ holds, but $\{\!|a{=}b|\!\}\ t_a \| t_b\ \{\!|a{\neq}b|\!\}$ does not. Actually, as figure 1 shows, what holds is $\{\!|a{=}b|\!\}\ t_a \| t_b\ \{\!|a{=}b|\!\}$.

To describe the synchronous combination, there are two cases to consider. First, $t_1 \| t_2$ preserves a triple $\{\!|p|\!\}\ t_1\ \{\!|q|\!\}$ over the component transition t_1 if the other component t_2 does not change any variables occurring in q. Informally, it means t_2 does not undo the effect of t_1. Let var(q) be the set of variables occurring in q. Rule B3 describes this case. The proviso in this rule ensures the condition above for both component transitions. Informally, this rule deals with statements about variables outside a write conflict. Second, $t_1 \| t_2$ also preserves a triple $\{\!|p|\!\}\ t_1\ \{\!|q|\!\}$ if both component transitions change the same variables in q, and the triple holds for both component transitions separately. Informally, it means there is a write conflict, but both transitions have the same affect on the conflicting variables. Rule B4 describes this case.

The top layer in the SINC 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 5 lists the properties and their meaning. The left column shows a property over F, and the right column shows a condition on the computations $\Sigma = \sigma_0\sigma_1\sigma_2 \ldots$ generated by F. A property holds if all computations of F satisfy the corresponding condition. In the right column,

$$\text{[A1]}\quad \frac{p \Rightarrow p' \quad \{p'\}\ s\ \{q'\} \quad q' \Rightarrow q}{\{p\}\ s\ \{q\}} \qquad \text{[A2]}\quad \{p\}\ \textbf{skip}\ \{p\}$$

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

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

$$\text{[B3]}\quad \frac{\{\!|p_1|\!\}\ t_1\ \{\!|q_1|\!\} \quad \{\!|p_2|\!\}\ t_2\ \{\!|q_2|\!\}}{\{\!|p_1 \wedge p_2|\!\}\ t_1 \| t_2\ \{\!|q_1 \wedge q_2|\!\}}\ \dagger \qquad \text{[B4]}\quad \frac{\{\!|p|\!\}\ t_1\ \{\!|q|\!\} \quad \{\!|p|\!\}\ t_2\ \{\!|q|\!\}}{\{\!|p|\!\}\ t_1 \| t_2\ \{\!|q|\!\}}\ \ddagger$$

$$\text{[C1]}\quad \frac{\textbf{init}\ p'\ \textbf{in}\ F \quad p' \Rightarrow p}{\textbf{init}\ p\ \textbf{in}\ F} \qquad \text{[C2]}\quad \textbf{init}\ b\ \textbf{in}\ \langle b \mid t \rangle$$

$$\text{[C3]}\quad \frac{p \Rightarrow p' \quad p'\ \textbf{co}\ q'\ \textbf{in}\ F \quad q' \Rightarrow q}{p\ \textbf{co}\ q\ \textbf{in}\ F} \qquad \text{[C4]}\quad \frac{\{\!|p|\!\}\ t\ \{\!|q|\!\}}{p\ \textbf{co}\ q\ \textbf{in}\ \langle b \mid t \rangle}$$

$$\text{[C5]}\quad \textbf{inv}\ p\ \textbf{in}\ F \ \equiv\ \begin{array}{l}(\textbf{init}\ p\ \textbf{in}\ F) \wedge \\ (p\ \textbf{co}\ p\ \textbf{in}\ F)\end{array} \qquad \text{[C6]}\quad \frac{p\ \textbf{co}\ q\ \textbf{in}\ F \quad \textbf{inv}\ r\ \textbf{in}\ F}{(p \wedge r)\ \textbf{co}\ (q \wedge r)\ \textbf{in}\ F}$$

$$\text{[C7]}\quad \frac{(p \wedge r)\ \textbf{co}\ (q \wedge r)\ \textbf{in}\ F \quad \textbf{inv}\ r\ \textbf{in}\ F}{p\ \textbf{co}\ q\ \textbf{in}\ F} \qquad \text{[C8]}\quad \frac{p \Rightarrow p' \quad p'\ \textbf{leads}\ q'\ \textbf{in}\ F \quad q' \Rightarrow q}{p\ \textbf{leads}\ q\ \textbf{in}\ F}$$

$$\text{[C9]}\quad p\ \textbf{leads}\ p\ \textbf{in}\ F \qquad \text{[C10]}\quad \frac{p\ \textbf{co}\ q\ \textbf{in}\ F}{p\ \textbf{leads}\ q\ \textbf{in}\ F}$$

$$\text{[C11]}\quad \frac{p_1\ \textbf{leads}\ q_1\ \textbf{in}\ F \quad p_2\ \textbf{leads}\ q_2\ \textbf{in}\ F}{(p_1 \vee p_2)\ \textbf{leads}\ (q_1 \vee q_2)\ \textbf{in}\ F} \qquad \text{[C12]}\quad \langle b_1 \mid t_1 \rangle \parallel \langle b_2 \mid t_2 \rangle \ \equiv\ \langle b_1 \wedge b_2 \mid t_1 \| t_2 \rangle$$

$$\begin{array}{ll}\dagger & \text{var}(q_1) \cap \text{write}(t_2) = \emptyset \ \wedge\ \text{var}(q_2) \cap \text{write}(t_1) = \emptyset \\ \ddagger & \text{var}(q) \cap \text{write}(t_1) = \text{var}(q) \cap \text{write}(t_2)\end{array}$$

Figure 4: SINC logic

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

Figure 5: Temporal properties

$\Sigma_i(p)$ means that there is a state at position i in the computation Σ, and assertion p holds in this state, i.e., $\sigma_i \neq \perp$ and $\widehat{\sigma}_i(p) = \textbf{true}$.

Informally, properties may be read as follows: **init** p **in** F means "p holds in the first state", p **co** q **in** F means "if p holds, then q holds in the next state", **inv** p **in** F means "p holds in all states", and p **leads** q **in** F means "if p holds, then q will hold in some future state". Properties stated with the **co**, **init** and **inv** connectives describe *safety* properties indicating that "nothing bad ever happens in a computation". The **leads** connective introduces *progress* properties

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

$$\text{[D3]}\quad \frac{p\ \textbf{co}\ q\ \textbf{in}\ F}{(p \wedge r)\ \textbf{co}\ (q \wedge r)\ \textbf{in}\ F}\ \dagger \qquad \text{[D4]}\quad \frac{(p \wedge X{=}e)\ \textbf{co}\ q\ \textbf{in}\ F}{p\ \textbf{co}\ q\ \textbf{in}\ F}\ \ddagger$$

$$\text{[D5]}\quad \frac{p \wedge m{=}X\ \textbf{leads}\ (p \wedge m{\prec}X) \vee q\ \textbf{in}\ F}{p\ \textbf{leads}\ q\ \textbf{in}\ F}\ \natural \qquad \text{[D6]}\quad \frac{p_1\ \textbf{co}\ q_1\ \textbf{in}\ F_1 \quad p_2\ \textbf{co}\ q_2\ \textbf{in}\ F_2}{(p_1 \wedge p_2)\ \textbf{co}\ (q_1 \wedge q_2)\ \textbf{in}\ F_1 \| F_2}\ \sharp$$

$$\text{[D7]}\quad \frac{p_1\ \textbf{leads}\ q_1\ \textbf{in}\ F \quad p_2\ \textbf{co}\ q_2\ \textbf{in}\ F}{(p_1 \wedge p_2)\ \textbf{leads}\ (q_1 \wedge q_2) \vee (\neg p_2 \wedge q_2)\ \textbf{in}\ F} \qquad \text{[D8]}\quad \frac{p\ \textbf{leads}\ q\ \textbf{in}\ F \quad r\ \textbf{co}\ r\ \textbf{in}\ F}{(p \wedge r)\ \textbf{leads}\ (q \wedge r)\ \textbf{in}\ F}$$

$\dagger$ $\text{var}(r) \cap \text{write}(F) = \emptyset$
$\ddagger$ $x \notin (\text{var}(p) \cup \text{var}(e) \cup \text{var}(q) \cup \text{var}(F))$
$\natural$ m is well-founded with respect to $\prec$
$\sharp$ $\text{var}(q_1) \cap \text{write}(t_2) = \emptyset \ \wedge\ \text{var}(q_2) \cap \text{write}(t_1) = \emptyset$

Figure 6: Additional inference rules

indicating that "something good eventually happens in a computation".

As the preceding comments suggest, properties consider only 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 $\langle a = 2 \,|\, t_b \rangle$. We cannot prove $\{|\mathit{even}(b)|\}\ t_b\ \{|\mathit{even}(b)|\}$. However, we may prove $\mathit{even}(b)$ **co** $\mathit{even}(b)$ **in** F_b because a always holds 2 in the computations of F_b. The difference results from the fact that the triple considers all pairs of initial and final states for t_b, while the property only considers the pairs of reachable states.

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 define 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 SINC 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 SINC. Furthermore, as all SINC programs are finite, the SINC logic also omits an infinitary UNITY rule for **leads** properties, and the transitivity of this connective becomes a derived rule. Additionally, the move to SINC demands for a review of all property definitions to account for non-determinism and non-termination.

As a design decision, SINC omits the **skip** steps (stuttering steps [16]) 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 [20] and existentially quantified triples [8]. However, to preserve the semantics of **leads**, we add rule C9.

To make the SINC logic useful in practice, the basic set of rules of figure 4 needs to be extended with several derived rules. Figure 6 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. Rules D3 and D4 are similarly derived from rules for command triples. Other derived inference rules are inherited (with little changes possibly) from UNITY. For instance, D5 is an induction rule for **leads** properties coming from UNITY.

Finally, there is a group of derived rules which are specific to synchronous transitions. For instance, D6 is derived

from B3, and it describes the synchronous combination of conflict-free **co** properties. A similar rule for properties with write conflicts is derived from B4. This group of derived rules is essential because they reflect basic aspects of synchronous transitions, and allow for compositional and modular verification of properties.

We must observe that there is no rule analogous to D6 for leads properties. It means we cannot derive a **leads** property of a composite program from leads properties over its components. The same situation happens in UNITY. To overcome this problem, we use conditional properties, and rule D7 and its particular case D8. These rules are derived from UNITY, and they describe the combination of progress and safety properties. Next section illustrates their use.

To prove a derived rule where the premises and the conclusion are formulas of the same kind, we frequently need induction on the proof length. This is the case in rules D2, D3, and D4, where both the premises and the conclusion are **co** properties. To derive a rule for transition triples from a rule for command triples, the basic rule B2 gives the base case, and the basic rules B1, B3, and B4 give the induction steps.

Likewise, to derive a rule for **co** properties from a rule for transition properties, the basic rule C4 gives the base case, and the basic rules C3, C6, and C7 give the induction steps. Before this proof, we need to expand the second premise in rules C6 and C7 by the definition of **inv** given by C5, exposing a hidden premise on **co**. We deal with **leads** properties in a similar way. It must be observed that the basic adaptation rules B1, C1, C3, and C8 in figure 4 cannot be derived from A1 using this technique; they must be basic rules.

In [23], we demonstrate that the SINC logic is sound, i.e., rules in figure 4 only generate true formulas. Each layer is considered 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 SINC 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 B4 is sound. Let σ be a state where $\widehat{\sigma}(p) =$ **true**. Assume the premises of B4 are true. From this assumption, it follows that the component transitions t_1 and t_2 terminate when their computations start in σ, and the resulting states satisfy q. Rule B4 is sound if all computations of $t_1 \| t_2$ starting in σ also terminate, and the resulting states also satisfy q. We demonstrate this statement next.

Figure 2 defines the semantics of $t_1 \| t_2$ using rules S5 and S6. We analyze rule S5 first. From the assumption above, it follows that $t_i \rhd \sigma \stackrel{\text{trn}}{\rightarrow} \sigma_i$, and $\widehat{\sigma}_i(q) =$ **true**. Therefore, according to S5, the execution of $t_1 \| t_2$ beginning in σ terminates, and produces σ' given by $\sigma \oplus (\sigma_1 \downarrow \text{write}(t_1)) \oplus (\sigma_2 \downarrow \text{write}(t_2))$.

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 holds in σ' because it holds in σ_1, and σ' and σ_1 agree on the value of all variables of q. Let x be a variable in var(q). From the proviso of S5, we get that either $x \in \text{write}(t_1)$ and $x \in \text{write}(t_2)$, or $x \notin \text{write}(t_1)$ and $x \notin \text{write}(t_2)$.

Let φ and φ' be two states, and y be a variable. The definitions of $\oplus$ and $\downarrow$ entail the propositions bellow:

[P1] if $y \in Z$, then $(\varphi \oplus (\varphi' \downarrow Z))(y) = \varphi'(y)$;
[P2] if $y \notin Z$, then $(\varphi \oplus (\varphi' \downarrow Z))(y) = \varphi(y)$;
[P3] if $y \notin \text{write}(t)$ and $t \rhd \varphi \stackrel{\text{trn}}{\rightarrow} \varphi'$, then $\varphi'(y) = \varphi(y)$.

First, assume $x \in \text{write}(t_1)$. From P1 and the definition of σ', it follows that $\sigma'(x) = \sigma_1(x)$. Alternatively, assume $x \notin \text{write}(t_1)$. In this case, we also have that $x \notin \text{write}(t_2)$. 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 var(q), and q holds in σ'. A symmetric argument shows that the resulting state of $t_1 \| t_2$ according to S6 also satisfies q. Therefore, rule B4 is sound. Similar arguments show that all rules in figure 4 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 [21]. This is a consequence of some differences between SINC and UNITY, such as the synchronous combinator, and the absence of infinitary rules and stuttering steps. We still have no completeness results for the SINC logic. However, the many examples verified in SINC suggest that it is complete. 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 strong evidences that it would be easy 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 Specification and Verification Techniques

SINC 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 SINC. 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 SINC, the rules for this construction strongly influence the organization of proofs in this formalism.

Rules B3 and B4 break 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. To satisfy these restrictions, we usually formulate sub-triples that indicate the contribution of each transition to the desired result. For instance, let t_a and t_b be $a := a+1$ and $b := b + 1$. Bellow, we list a proof for property $a = b$ **co** $a = b$ **in** $t_a \| t_b$:

(1)	$a{=}A$ **co** $a{=}A{+}1$ **in** t_a	A1, A3, B2, and C4
(2)	$b{=}B$ **co** $b{=}B{+}1$ **in** t_b	A1, A3, B2, and C4
(3)	$a{=}A \wedge b{=}B$ **co** $a{=}A{+}1 \wedge b{=}B{+}1$ **in** $t_a \| t_b$	D6 in (1, 2)
(4)	$a{=}A \wedge b{=}B \wedge A{=}B$ **co** $a{=}A{+}1 \wedge b{=}B{+}1 \wedge A{=}B$ **in** $t_a \| t_b$	D3 in (3)
(5)	$A{=}a \wedge B{=}b \wedge a{=}b$ **co** $a{=}b$ **in** $t_a \| t_b$	C3 in (4)
(6)	$B{=}b \wedge a{=}b$ **co** $a{=}b$ **in** $t_a \| t_b$	D4 in (5)
(7)	$a{=}b$ **co** $a{=}b$ **in** $t_a \| t_b$	D4 in (6)

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 values 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 in their write-sets.

Next, line 3 describes the synchronous combination of these transitions. However, this property is too general. To reach the desired conclusion, we need to restrict attention to states where $a = b$. In lines 4 and 5, D3 adds this restriction through an assertion over rigid variables, allowing C3 to produce the necessary assertions. Finally, in lines 6 and 7, D4 eliminates unnecessary rigid variables.

The vast majority of proofs in the SINC logic are organized like the proof above. They begin with a set of *elementary properties* that completely describe the contribution of each elementary transition, so later proofs steps do not need to recourse to the program text. We prefer to work with properties because the many rules for them simplify proof development. They are derived from the program text using standard techniques from Hoare logic.

In a elementary property p_i **co** q_i **in** t_i, the right-hand assertion q_i must describe the effect of t_i over all variables in the corresponding write-set, and must not refer to any other program variable. These restrictions ensure that elementary properties completely describes transitions, and help to satisfy the proviso of many rules.

The core of a proof is the derivation of the goal property from the elementary properties. For conflict free transitions, rule D6 does this task. The proviso in this rule holds due to the restriction above over the variables on the right-hand of elementary properties. However, the resulting property usually is more general than the goal property. Using D3, we introduce an assertion specializing this property to the states of interest to the goal conclusion, and employ C3 and D4 to get the this conclusion. After some training with SINC, it becomes easy to build proofs like this, where we need to find the contribution of transitions, to combine properties and then specialize them.

The proof technique above is the most elementary one, and the most widely applied. As we proceed to more complex systems and specifications, we need other techniques. In [23], we present an comprehensive catalog of basic specification and verification techniques for SINC which may be applied in many situations. Like the technique above,

$$
\begin{array}{l}
\mathit{Save} \;\equiv\; \langle\, \mathit{done} = \textbf{false} \;\wedge\; \mathit{buf} = \mathit{nil} \\
\quad\;\; \mid\; \textbf{if } \mathit{req} \wedge \neg \mathit{done} \textbf{ then} \\
\qquad\; (\mathit{buf} := \mathit{cons}(x, \mathit{buf}); \mathit{done} := \textbf{true}) \\
\quad\;\; \textbf{else if } \neg \mathit{req} \wedge \mathit{done} \textbf{ then} \\
\qquad\; \mathit{done} := \textbf{false} \\
\quad\;\; \textbf{else skip}\, \rangle
\end{array}
$$

Figure 7: Program saving x to *buf*

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 SINC.

As a more complex example, consider program *Save* in figure 7. Constant *nil* is the empty list, and $\mathit{cons}(x, l)$ appends element x to the beginning of list l. Whenever variable *req* indicates a request, program *Save* pushes the value of x in the list *buf*, and indicates through *done* that the operation is finished. When the request is removed, *done* is reset. This is an elementary example of *open system*, a system that interacts with an environment which is not described along the system. The proper operation of an open system usually depends on some assumptions on the environment.

To deal with environment assumptions, we use conditional properties. A conditional property is an "if-then" rule, where the premises and the conclusion are ordinary properties. A statement about an open system F is represented as a conditional property where the conclusion refers to $F \| G$ and the premises describe G. The program G is an undefined program representing the environment, and its properties are assumptions on the environment. The property in the conclusion describes the open system F through a description of the interaction between F and G.

As an example, consider the conditional property bellow:

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

The property in the premise indicates that the environment does not change the value of variables x and *req* when there is an unattended request, and the property in the conclusion indicates that x eventually will be stored in *buf*, and *done* eventually will be turned on. Therefore, this conditional property indicates that *Save* will store x in *buf* under the assumption that the environment keeps the value of x stable.

We chose to employ a **leads** property in the conclusion above to give some freedom to program *Save*. A **co** property would forbid implementations which take more than one computation step to save x, and we do not want to restrict the program *Save* that much. We now list a proof for the conditional property above:

(1)	$\mathit{req} \wedge \neg \mathit{done} \wedge x = X$ **co** $\mathit{req} \wedge x = X$ **in** G	assumption
(2)	$\mathit{req} \wedge \neg \mathit{done} \wedge x = X \wedge \mathit{buf} = B$ **co** $\mathit{buf} = \mathit{cons}(X, B) \wedge \mathit{done}$ **in** *Save*	definition of *Save*
(3)	$\mathit{req} \wedge \neg \mathit{done} \wedge x = X \wedge \mathit{buf} = B$ **co** $\mathit{buf} = \mathit{cons}(X, B) \wedge \mathit{done}$ **in** $\mathit{Save} \| G$	D6 and C3 in (1, 2)
(4)	$\mathit{req} \wedge \neg \mathit{done} \wedge x = X \wedge \mathit{buf} = B$ **leads** $\mathit{buf} = \mathit{cons}(X, B) \wedge \mathit{done}$ **in** $\mathit{Save} \| G$	C10 in (3)

In the beginning, we take the conditional property premise as an assumption. Line 2 is an elementary property derived from the program text using the techniques previously introduced. It describes the computation step when the program saves x to *buf*. Line 3 combines the program and the environment properties, and the last line announces the result as a **leads** property. We observe that the order in the last to steps is crucial. If we rephrase lines 1 and 2 as **leads** properties, we cannot combine them because these properties are not compositional.

Conditional properties are essential to state progress (**leads**) properties because these properties are not compositional. As the previous section discussed, there is no rule similar to D6 for the **leads** connective. Let P be the property p **leads** q **in** F. Without conditional properties, there is no way to use P in the proof of some property over $F \| F'$. The weakest way to state P is the following conditional property:

$$
\frac{p \textbf{ co } \textbf{true} \textbf{ in } G}{p \textbf{ leads } q \textbf{ in } F \| G}
$$

The premise of this rule only requires that the environment computation terminate in the states where F performs its computation. Termination requirements do not appear in UNITY and most other formalisms because they do not handle non-termination.

To fully specify program *Save*, we need two additional properties:

$$\begin{array}{l} \mathit{req} \wedge \mathit{done} \wedge \mathit{buf} = B \textbf{ co } \mathit{buf} = B \textbf{ in } \mathit{Save} \\ \neg \mathit{req} \wedge \mathit{buf} = B \textbf{ co } \mathit{buf} = B \wedge \neg \mathit{done} \textbf{ in } \mathit{Save} \end{array}$$

The first property describes what happens after a request is granted. In this case, the value of *buf* does not change. The second property deals with the situation where there is no request. In this case, the value of *buf* also does not change and additionally *done* remains off. These properties cannot be stated with the **leads** because these these connective would allow for unspecified states between the assertions in each property. These properties are not conditional because they are not progress properties, and they do not depend on any assumptions on the environment behavior.

The three properties listed above for program *Save* illustrate a common specification pattern in SINC, which describes a system module as an open system that should attend to operation requests. There is a set of associated set proof techniques similar to the one described earlier in this section, which aid in the verification of these properties. In [23], we analyze other common specification patterns and proof techniques. Although proofs in the SINC logic cannot be fully automated, we can use these specification and verification techniques to offer a quite extensive partially automated support for proof development. To realize this goal, we plan to embed SINC in a programmable theorem prover such as HOL.

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 [9]) impose severe limits on environment restrictions, what limits their applicability.

The connectives in the SINC logic are not enough to specify all properties of computations. This situation is inherited from the UNITY logic [20]. To overcome this deficiency, we employ auxiliary variables (also known as history variables). An *auxiliary variable* is a program variable defined through properties. Let x be a variable in the write-set of F. The auxiliary variable *x-event* only is true in the states where x assumes a new value, and the auxiliary variable *x-last* holds the previous value of x. We define these variables through the following properties:

$$\begin{array}{ll} [\text{AUX1}] & \textbf{init } \mathit{x\text{-}event} \textbf{ in } F \\ [\text{AUX2}] & x = X \textbf{ co } (x = X \wedge \neg \mathit{x\text{-}event}) \vee (x \neq X \wedge \mathit{x\text{-}event}) \textbf{ in } F \\ [\text{AUX3}] & \textbf{init } \mathit{x\text{-}last} = x \textbf{ in } F \\ [\text{AUX4}] & x = X \wedge \mathit{x\text{-}last} = X' \textbf{ co } (x = X \wedge \mathit{x\text{-}last} = X') \vee (x \neq X \wedge \mathit{x\text{-}last} = X) \textbf{ in } F \end{array}$$

Properties and programs may read auxiliary variables as ordinary program variables. The property schemes above apply to any variable x and program F, so they are always available. For instance, assume the computation of a function f is resource-intensive. The program bellow keeps in y the value of $f(x)$, and uses an auxiliary variable to recompute f only when its input value changes:

$$F_f \;\equiv\; \langle y = f(x) \mid \textbf{if } \mathit{x\text{-}event} \textbf{ then } y := f(x) \textbf{ else skip} \rangle$$

After x changes, this program takes one computation step to update y. The property bellow uses the same auxiliary variables to indicate that the value of x and y agree, except when x has just changed:

$$\textbf{inv } \neg \mathit{x\text{-}event} \Rightarrow y = f(x) \textbf{ in } F_f$$

We present a proof for this property bellow:

(1)	**inv** $y = f(x)$ **in** F_f	C2
(2)	**inv** *x-event* $\vee\ y = f(x)$ **in** $F_f \| G$	AUX1, and (1)
(3)	$x = X$ **co** $(x = X \wedge \neg$*x-event*$) \vee (x \neq X \wedge$ *x-event*$)$ **in** G	AUX2
(4)	$x = X$ **co** $x = X \vee$ *x-event* **in** G	C3 in (3)
(5)	*x-event* $\wedge\ x = X$ **co** $y = f(X)$ **in** F_f	definition of F_f
(6)	*x-event* $\wedge\ x = X$ **co** $y = f(X) \wedge (x = X \vee$ *x-event*$)$ **in** $F_f \| G$	D6 in (4, 5)
(7)	*x-event* $\wedge\ x = X$ **co** $x = f(x) \vee$ *x-event* **in** $F_f \| G$	C3 in (6)
(8)	*x-event* **co** $x = f(x) \vee$ *x-event* **in** $F_f \| G$	D4 in (7)
(9)	$\neg$*x-event* $\wedge\ y = Y$ **co** $y = Y$ **in** F_f	definition of F_f
(10)	$\neg$*x-event* $\wedge\ y = f(X)$ **co** $y = f(X)$ **in** F_f	E3 in (9)
(11)	$\neg$*x-event* $\wedge\ y = f(X)$ **co** $y = f(X) \wedge (x = X \vee$ *x-event*$)$ **in** $F_f \| G$	D6 in (4, 10)
(12)	$\neg$*x-event* $\wedge\ y = f(X)$ **co** $x = f(x) \vee$ *x-event* **in** $F_f \| G$	C3 in (11)
(13)	$\neg$*x-event* $\wedge\ y = f(x) \wedge x = X$ **co** $x = f(x) \vee$ *x-event* **in** $F_f \| G$	C3 in (12)
(14)	$\neg$*x-event* $\wedge\ y = f(x)$ **co** $x = f(x) \vee$ *x-event* **in** $F_f \| G$	D4 in (13)
(15)	*x-event* $\vee\ (\neg$*x-event* $\wedge\ y = f(x))$ **co** $x = f(x) \vee$ *x-event* **in** $F_f \| G$	disjunct of (8, 14)
(16)	*x-event* $\vee\ y = f(x)$ **co** $x = f(x) \vee$ *x-event* **in** $F_f \| G$	C3 in (15)
(17)	**inv** *x-event* $\vee\ y = f(x)$ **in** $F_f \| G$	C5 in (2, 16)
(18)	**inv** $\neg$*x-event* $\Rightarrow y = f(x)$ **in** $F_f \| G$	from (2, 17)

Lines 2, 10, 15, and 18 employ rules not previously introduced. Lines 5 and 9 are elementary properties describing the program text. Since the environment writes to x, lines 2 and 3 associate to it the definition of *x-event*. This proof comprises two cases, reflecting the branches of the conditional command in F_f. Lines 5 to 8 are the first case, where we consider the situation when x changes and y is recomputed. Lines 9 to 14 are the second case, when x does not change and y retains its previous value. In both cases, we use the definition of *x-event* in line 4 to handle this auxiliary variable. Admittedly, this proof looks trick, but after one gets used to SINC, it becomes easy to build a proof like this.

We may introduce other auxiliary variables in programs and specification to simplify system development and verification. To ensure they also do not introduce inconsistencies, the definition of an auxiliary variable x must be a property in the form $x = X \wedge y = Y \wedge z = Z$ **co** $x = f(X, Y, Z, y)$ **in** F, where f is any function, y stands for the variables in the write-set of F, and z stands for other program variables. This definition does not introduce any inconsistencies because it may be derived from the modified program F' defined as $(y' := y;\ F;\ x := f(x, y', z, y))$. Therefore, auxiliary properties are shortcuts to program modifications.

5 Extensions

SINC 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 (e.g., size of a vector, or number of available network connections). SINC 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 $t_0 \| t_1 \| \ldots \| t_{E-2} \| t_{E-1}$, SINC adopts the notation $\langle 0 \leq I < E : t_I \rangle$. Together, 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.

Generic parameters and regular structures are not control structures, they have no representation in the semantic rules in figure 2. Conceptually, we execute a SINC program only after binding a value to each generic parameter, and after expanding the regular structures to their component transitions. To ensure that the expansion of a regular structure $\langle 0 \leq I < E : t_I \rangle$ does not depend on run-time values, I and E should not refer to program variables, only to rigid variables. Therefore, programs with generic parameters or regular structures actually are program schemes standing for a (possibly infinite) collection of programs.

Nevertheless, we may prove properties of programs with generic parameters and regular structures without expanding them. Figure 8 lists some rules dealing with these constructs. Notation A_e^v is the result of replacing the occurrences of v in A by e. Rules E1 and E2 define the unfolding of regular structures. Rule E3 instantiates a generic

[E1] $$\langle 0 \le I < 0 : t \rangle \;\equiv\; \textbf{skip}$$

[E2] $$\langle 0 \le I < n+1 : t \rangle \;\equiv\; \langle 0 \le I < n : t \rangle \parallel t_n^I$$

[E3] $$\frac{p \textbf{ co } q \textbf{ in } F}{p_E^X \textbf{ co } q_E^X \textbf{ in } F_E^X}$$

[E4] $$\frac{p_0^X \textbf{ co } q_0^X \textbf{ in } F_0^X \quad (p_n^X \textbf{ co } q_n^X \textbf{ in } F_n^X) \Rightarrow (p_{n+1}^X \textbf{ co } q_{n+1}^X \textbf{ in } F_{n+1}^X)}{p \textbf{ co } q \textbf{ in } F}$$

Figure 8: Generic and regular structures

$$\begin{array}{rcl}
\text{write}_{\text{cmd}}(\textbf{skip}) & = & \emptyset \\
\text{write}_{\text{cmd}}(x := e) & = & \{x\} \\
\text{write}_{\text{cmd}}(x[e'] := e) & = & \{x[e']\}, \text{ for a rigid expression } e' \\
\text{write}_{\text{cmd}}(x[e'] := e) & = & \{x[i] \mid i \in \mathcal{N}\}, \text{ otherwise} \\
\text{write}_{\text{cmd}}(s; s') & = & \text{write}_{\text{cmd}}(s) \cup \text{write}_{\text{cmd}}(s') \\
\text{write}_{\text{cmd}}(\textbf{if } b \textbf{ then } s \textbf{ else } s') & = & \text{write}_{\text{cmd}}(s) \cup \text{write}_{\text{cmd}}(s') \\
\text{write}_{\text{cmd}}(\textbf{while } b \textbf{ do } s) & = & \text{write}_{\text{cmd}}(s) \\
 & & \\
\text{write}(t) & = & \text{write}_{\text{cmd}}(t), \text{ for } t \in \text{Cmd} \\
\text{write}(t \| t') & = & \text{write}(t) \cup \text{write}(t') \\
\text{write}\langle 0 \le I < E : t \rangle) & = & \bigcup_{i=0}^{E-1} \text{write}(t_i^I)
\end{array}$$

Figure 9: Write-sets for indexed variables

parameter I to some value E. Like D2, rule E3 is derived from a similar rule over command triples in the Hoare logic [25]. It holds because a rigid variable stands for an arbitrary value, and there is a universal quantifier implicit in these variables. Rule E4 is the ordinary induction principle cast to properties, where n a logic parameter. It performs induction on the program text, not on run-time values, and it is combined with E1 and E2 to prove properties over regular structures.

For instance, consider the property **true co** $(\forall I : 0 \le I < N \Rightarrow a[I] = 0)$ **in** F_a indicating that the program F_a introduced earlier sets the first N elements of a to zero. The following proof uses the rules in figure 8 to prove this property. In this proof, $t(x)$ abbreviates $\langle 0 \le I < x : a[I] := 0 \rangle$.

(1)	**true co** $(\forall I : 0 \le I < 0 \Rightarrow a[I] = 0)$ **in skip**	A1, A2, B2, and C4
(2)	**true co** $(\forall I : 0 \le I < 0 \Rightarrow a[I] = 0)$ **in** $t(0)$	E1 in (1)
(3)	**true co** $(\forall I : 0 \le I < n \Rightarrow a[I] = 0)$ **in** $t(n)$	assumption
(4)	**true co** $a[n] = 0$ **in** $a[n] := 0$	A1, A3, B2, and C4
(5)	**true co** $(\forall I : 0 \le I < n+1 \Rightarrow a[I] = 0)$ **in** $t(n) \parallel a[n] := 0$	D6 in (3,4) and C3
(6)	**true co** $(\forall I : 0 \le I < n+1 \Rightarrow a[I] = 0)$ **in** $t(n+1)$	E2 in (5)
(7)	**true co** $(\forall I : 0 \le I < N \Rightarrow a[I] = 0)$ **in** F_a	E4 in (2) and (3–6)

With the introduction of regular structures, frequently we find programs where the components of a composed transition write to the same vector. As long as they write to distinct elements, we do not consider this situation as a write conflict. To account for this situation, we refine the concept of write-sets. For an indexed variable x, the write-set of a program F contains elements $x[E]$ indicating that F assigns to element E of x. The index E is a constant or a rigid expression, allowing for dependencies on generic parameters.

Figure 9 shows the rules to compute these refined write-sets. When an assignment writes to a vector element determined by constants or rigid variables only, the third rule adds to corresponding index expression to the write-set.

Otherwise, the fourth rule assumes the assignment may change any position, and it adds all vector elements to the write-set. For instance, these rules associate the write-set $\{a[i] \mid 0 \leq i < N\}$ for program F_a introduced earlier, and they associate the write-set $\{a[I]\}$ to the component transition $a[I] := 0$ of the same program.

As suggested above, a regular structure $\langle 0 \leq I < E \ : \ t_I \rangle$ does not generate write conflicts if the component transitions write to distinct vector elements. Using the refined write-sets, we state this condition as follows:

$$I \neq J \ \Rightarrow \ \text{write}(t_I) \cap \text{write}(t_J) = \emptyset$$

The new variable J is a dummy variable. Assuming it holds a value distinct from I, there must be no conflict in the component transitions.

This refinement of write-sets is consistent with the concept of static conflict detection, since it does not depend on the value that program variables assume along the computation. Generic parameters and regular structures parameters are treated as constants because they are instantiated before the program computation starts.

Since a program F with generic parameters or regular structures stands for a collection of programs, a property of F corresponds to a collection of properties. Therefore, a single proof actually proves a large or even infinite set of properties. Standard finite model-checkers and similar proof techniques do not handle generic parameters and regular structures, they only can be applied to fully expanded programs. This is another situation where SINC is more general than these approaches, although it is not automated as they are. As observed earlier, we plan to explore the complementary nature of both approaches.

6 Related Work

The features of SINC 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 [8, 20] is a typical formalism based on a transition system and a temporal logic. It is simple and flexible, and it was applied to a large set of examples. UNITY includes a synchronous combinator, but its logic does not include rules to prove properties about it. Therefore, this combinator becomes a syntactic sugar to describe complex transitions. The UNITY notation restricts transitions to conditional multiple assignments, so it syntacticly forbids non-terminating transitions. The multiple assignments must not generate write-conflicts, but UNITY does not specify when such conflicts happen, or how we check their absence.

Other formalisms based on a linear-time temporal logic present similar problems. To verify a system, the Manna and Pnueli logic [17, 18] and TLA [16] include methods to represent several programming constructions as logical formulas. However, these methods do not deal with synchronous systems. ST [29] is a special-purpose formalism that includes a method to handle synchronous transitions. However, its method only handles a restricted subset of properties and systems, and it is harder to apply because it generates an exponential number of proof subgoals.

Besides the transition systems and temporal logics, SINC is also related to research on the semantics and verification of VHDL [5, 10, 11]. The works on this field differ on the subset of VHDL they handle, and the formalism they use. Some works [3, 31] cover a large subset of this language, but the resulting semantics are too complex, and are not appropriate to the formal verification of actual designs. Other works are oriented towards the formal verification of VHDL [4, 7, 30], but these works only cover restricted subsets of the language. SINC addresses a similarly restricted subset of VHDL [24], but it deals with some features that other semantics usually ignore (e.g., generic parameters and regular structures). SINC also includes a 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 [6].

We may employ SINC in other verification tasks beyond VHDL. Evolving algebras [12, 13, 14] are a specification formalism which is applied to the description of programming languages and computational systems. Since it adopts a synchronous computation model, we may verify properties of an evolving algebra using the SINC logic. We are developing a version of SINC tailored to evolving algebras, what amounts to adapting the SINC logic to the conflict detection and resolution method of evolving algebras. We know about only one other logic for evolving algebras [27], and we believe the SINC logic is more expressive and easier to use than this logic.

SINC also may be applied to the verification of synchronous programming languages such as Esterel [2]. These languages adopt the same computation model of SINC, although usually they also include a notion of real time

measured in some actual time unit. Through the introduction of some restrictions in these languages, programs are represented as a finite automata, which allows for efficient verification techniques [15]. SINC is a more general formalism, which does not impose restrictions on programs. We believe we may use the SINC logic as a complement to standard verification methods for synchronous programming languages, which we apply in systems or descriptions violating the restrictions above. To fulfill this goal, we began the exploration of a model of real time for SINC in [23].

Nowadays, there is a lot of work on formal verification which employs finite model-checkers, automata, and similar techniques [19]. Such approaches have been successfully applied on many fields, including hardware description languages and synchronous programming languages [9, 15, 26]. They are fully automated and quite effective. However, despite advances in the field, these techniques are not appropriate to all situations. They do not deal with first-order (non-propositional) specifications, and they are not good at the verification of modular systems, regularly structured systems, parametric descriptions, or data-intensive designs. It seems that a linear-time temporal logic such as SINC is a nice complement to these techniques, since it easily handles those hard situations. Some works explore this complementarity nature among distinct formalisms [4], and we plan to explore this path in some future developments of SINC.

7 Last Remarks

SINC is composed of a transition system and an associated linear-time temporal logic. The SINC transition system adopts the synchronous computation model, represents transitions as possibly non-terminating imperative commands, includes a method to solve write-conflicts, and handles generic parameters and regular structures. The SINC logic is designed to verify 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. Therefore, SINC allows for modular system development and verification. The SINC logic is organized in layers, allowing for a separation of concerns, and simplifying the formalism development and its soundness proof. These features are essential to the verification of synchronous system. However, as the previous section discussed, they are not present in other formalisms based on a first-order liner-time temporal logic.

This work was motivated by a wish to apply a first-order linear-time temporal logic to the verification of VHDL designs. Such logics have been successfully applied to several application fields, but they deal with asynchronous systems mostly. SINC resulted from adapting UNITY, a well-known formalism based on a linear-time temporal logic, to the verification of synchronous systems. Besides VHDL, the resulting formalism may be applied to the verification of other hardware description language, to synchronous programming languages, and to specification formalisms.

Besides developing the formalism, we also study the pragmatics in its usage. As discussed in section 4, there is a catalog of elementary specification and verification techniques in SINC which covers several sorts of systems and properties. Some of these techniques were inherited (possibly with changes) from other formalisms, while some are specific to SINC. These catalog greatly simplifies the verification of actual systems. We plan to embed SINC into a theorem proving tool such as HOL. These verification techniques should be added as programmed proof tactics in this tool.

There are many aspects of SINC which need to be further developed. We would like to extend the SINC programming notation with named cells with formal parameters, and we need to show that the SINC logic is complete. We are adapting the SINC logic to evolving algebras, and we plan to develop some automated support to SINC. We also plan to further study the complementary link between SINC and some automated verification techniques such as model-checkers.

To summarize, SINC 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 SINC is a good formalism and we plan to continue its development.

References

[1] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

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

[3] E. Börger, U. Glässer, and W. Müller. 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 simple denotational semantics, proof theory and validation condition generator for unit-delay VHDL. *Formal Methods in System Design*, 7(1/2):27–52, 1995.

[7] P. T. Breuer et al. A refinement calculus for the synthesis of verified hardware descriptions in VHDL. *ACM TOPLAS*, 19(4):586–616, 1997.

[8] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[9] D. Déharbe et al. The CV model-checker. In *FMCAD'98*. Springer-Verlag, 1998.

[10] C. Delgado Kloos and P. T. Breuer, editors. *Formal Semantics for VHDL*. Kluwer, 1995.

[11] C. Delgado Kloos and W. Damm, editors. *Practical Formal Methods for Hardware Design*. Springer-Verlag, 1997.

[12] Y. Gurevich. Evolving algebras: An attempt to discover semantics. Universty of Michigan, 1994.

[13] Y. Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*. Oxford University, 1995.

[14] Y. Gurevich. May 1997 draft of the ASM guide. Universty of Michigan, 1997.

[15] L. Jagadeesan et al. Safety property verification of ESTEREL programs and applications to telecommunications software. In *CAV-95*, 1995.

[16] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, 1994.

[17] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[18] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[19] K. C. McMillan. *Symbolic Model Checking*. Kluwer, Boston, 1993.

[20] J. Misra. A logic for concurrent programming. University of Texas at Austin, 1994.

[21] P. Päppinghaus. On the logic of UNITY. *Theoretical Computer Science*, 139:27–67, 1995.

[22] D. L. Perry. *VHDL*. McGraw-Hill, 1991.

[23] V. M. Rodrigues. *Transições Síncronas, Lógica Temporal e VHDL*. Tese de doutorado, CPGCC, UFRGS, 1998.

[24] V. M. Rodrigues and F. R. Wagner. A temporal logic for data-flow VHDL. In *SBCCI'98*, pages 91–94. IEEE Computer Society, 1998.

[25] F. B. Schneider. *On Concurrent Programming*. Springer-Verlag, 1997.

[26] A. Scholz et al. The FORMAT model checker. In *Practical Formal Methods for Hardware Design*, pages 175–183. Springer-Verlag, 1997.

[27] A. Schönegge. Extending dynamic logic for reasoning about evolving algebras. Technical report, Universität Karlsruhe, 1995.

[28] A. U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, 1993.

[29] J. Staunstrup. *A Formal Approach to Hardware Design*. Kluwer, 1994.

[30] J. P. van Tassel. *Femto-VHDL: The Semantics of a Subset of VHDL and its Embedding in the HOL Proof Assistant*. PhD thesis, University of Cambridge, 1992.

[31] P. A. Wilsey et al. A model of VHDL for the analysis, transformation, and optimization of digital system designs. In *CHDL-95*, pages 611–6, 1995.