The Control of Synchronous Systems, Part II[⋆,⋆⋆]

Luca de Alfaro[⋆⋆⋆] Thomas A. Henzinger[⋆⋆⋆] Freddy Y.C. Mang[⋆⋆⋆]

Abstract. A controller is an environment for a system that achieves a particular control objective by providing inputs to the system without constraining the choices of the system. For synchronous systems, where system and controller make simultaneous and interdependent choices, the notion that a controller must not constrain the choices of the system can be formalized by type systems for composability. In a previous paper, we solved the control problem for static and dynamic types: a static type is a dependency relation between inputs and outputs, and composition is well-typed if it does not introduce cyclic dependencies; a dynamic type is a set of static types, one for each state. Static and dynamic types, however, cannot capture many important digital circuits, such as gated clocks, bidirectional buses, and random-access memory. We therefore introduce more general type systems, so-called *dependent* and *bidirectional* types, for modeling these situations, and we solve the corresponding control problems.

In a system with a dependent type, the dependencies between inputs and outputs are determined gradually through a game of the system against the controller. In a system with a bidirectional type, also the distinction between inputs and outputs is resolved dynamically by such a game. The game proceeds in several rounds. In each round the system and the controller choose to update some variables dependent on variables that have already been updated. The solution of the control problem for dependent and bidirectional types is based on algorithms for solving these games.

1 Introduction

The *control problem* asks, given an open system P and a property ϕ, to construct a controller Q such that the controlled system $P||Q$ satisfies ϕ (or to answer "uncontrollable," if no such Q exists). The control problem has applications in the synthesis of reactive programs and sequential circuits [PR89], in discrete-event control [RW87], in modular verification [AdAHM99], and in early error detection [dAHM00b]. An important special case is the *single-step* control problem, for properties of the form $\phi = \bigcirc f(X)$, where $\bigcirc$ is the temporal "next" operator, and $f(X)$ is a predicate on a set X of state variables. In this case, the objective

⋆ This research was supported in part by the SRC contract 99-TJ-683.003, the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, the AFOSR MURI grant F49620-00-1-0327, and the NSF Theory grant CCR-9988172.

⋆⋆ A preliminary version of this paper will appear in the *Proceedings of the 12th International Conference on Concurrency Theory* (CONCUR 2001).

⋆⋆⋆ Electrical Engineering and Computer Sciences, University of California at Berkeley. Email: {dealfaro,tah,fmang}@eecs.berkeley.edu

of the controller Q is to ensure that the system P enters a state satisfying $f(X)$ in one transition. While "next" properties are, *per se*, of limited interest as control objectives, algorithms for all ω-regular control objectives, such as invariance properties (specifically, $\phi = \Box f(X)$), use as subroutines algorithms that solve the single-step control problem [BL69,GH82,McN93,Tho95].

We study the single-step control problem under two very general assumptions: the closed-loop assumption, and the input-enabling assumption. The *closed-loop assumption* has that the state variables which are not given values by the system P are given values by the controller Q, and vice versa. In particular, in the *unidirectional* case, the set X of state variables is partitioned into a set O of system outputs, which are also controller inputs, and a set I of system inputs, which are also controller outputs. Later, in the *bidirectional* case, what is input and output may change from one state to the next. The *input-enabling assumption* has that the state variables that are given values by the system P cannot be in any way constrained by the controller Q, and vice versa. More precisely, for all legal environments E of P, every state of the composite system $P||E$ must have a successor state, and analogously for Q. It follows, in particular, that the controlled system $P||Q$ cannot dead-lock.

The solution and hardness of the single-step control problem depends on the precise definition of the composition operator $||$. The case most commonly considered in the literature is *unidirectional turn-based composition*, where the system P and the controller Q take turns to proceed. With each turn of Q, the system outputs O remain unchanged, and with each turn of P, the controller outputs I remain unchanged. Assume that the states in which the controller outputs can change are defined by the predicate $ctr(X)$, and that the transition relation of P is defined by the predicate $\tau(X, X')$, where unprimed state variables refer to the source state of a transition, and primed state variables refer to the target state. Then, the states for which the control objective $\bigcirc f(X)$ can be met are characterized by the quantified formula

$$(ctr(X) \;\rightarrow\; (\exists X')(O' = O \;\wedge\; \tau(X, X') \;\wedge\; f(X'))) \wedge$$
$$(\neg ctr(X) \;\rightarrow\; (\forall X')(I' = I \;\wedge\; \tau(X, X') \;\rightarrow\; f(X'))).$$

Another simple case is that of *unidirectional Moore composition*, where both the system P and the controller Q can update their respective outputs in the same transition, but they cannot do so dependent on each other. In this case, the appropriate formula is

$$(\exists I')(\forall O')(\tau(X, X') \rightarrow f(X')) \quad \text{or} \quad (\forall O')(\exists I')(\tau(X, X') \wedge f(X')).$$

If $\tau(X, X')$ is the transition relation of an input-enabling Moore system, then these two formulas are equivalent.

In the Mealy case, where some system outputs may depend on simultaneous controller outputs, and some controller outputs may depend on simultaneous system outputs, there is not a single formula that characterizes the controllable states. Instead, all possible dependencies between variables must be considered

one by one. There are, however, multiple ways in which the notion of "dependency" can be formalized. For this purpose, we introduced *type systems* for composability [dAHM00a]. In essence, such a type system offers a syntactic criterion for ensuring that all well-typed composite systems are input-enabling. The simplest type system is based on static types. A *static type* is a dependency relation between input variables and output variables. Two statically typed components can be composed if the union of the component types is acyclic. Examples of statically typed components include Reactive Modules [AH99], and sequential circuits without combinational loops. As there are meaningful systems that cannot be typed statically, we generalized static types to dynamic types. A *dynamic type* is a set of static types, one for each state. In a dynamically typed component, which outputs depend on which inputs may change from state to state. An example of a dynamically typed component that cannot be typed statically is the transparent latch.

Once a type system is adopted, we can distinguish between the fixed-type control problem and the unknown-type control problem. The *fixed-type control problem* assumes that the type of the controller is known: given a typed system P, a property ϕ, and a controller type t, construct a controller Q of type t such that $P||Q$ satisfies ϕ. The *unknown-type control problem* requires only that the controller and the resulting closed-loop system are well-typed: given a typed system P and a property ϕ, construct a typed controller Q such that $P||Q$ has a type and satisfies ϕ. In case of both static and dynamic types, the fixed-type controllable states can be characterized by formulas with partially ordered (Henkin) quantifiers [Hen61]. For example, if controller output i_1 depends on system output o_1, and controller output i_2 depends on system output o_2, and there are no other variables or dependencies, then the appropriate formula is

$$\begin{matrix}(\forall o_1')(\exists i_1')\\(\forall o_2')(\exists i_2')\end{matrix}(\tau(X, X') \wedge f(X')).$$

The unknown-type controllable states can be characterized by disjunctions of fixed-type solutions of a particularly simple kind, namely, those which contain only linearly ordered quantifiers. So, perhaps surprisingly, the unknown-type control problem is computationally simpler (in the case of boolean variables, PSPACE-complete) than the fixed-type control problem (NEXP-complete).

This concludes our review of [dAHM00a]. Following [PRSV98], we attempted to use our theory to automatically synthesize protocol converters, specifically, an interface between a PCI bus and a component using a two-phase commit protocol for communication. We found, however, that even dynamic types are too restrictive, and more general notions of synchronous composition need to be considered (cf. [BG92]). The common cases where dynamic types prove insufficient for hardware modeling fall into two classes. First, the dependencies between inputs and outputs may depend not only on the state of the system, but also on the partial successor state, as it unfolds. For example, in a digital circuit where gated clocks are used to enable or disable the latches, whether there is a dependency of the output of a latch on its input depends on whether

its clock signal is asserted or not. This in turn may depend on other parts of the system such as the current primary inputs. Second, in many digital circuits, the complete classification of ports into input and output signals is not done *a priori*. In hardware description languages, such as VHDL and Verilog, ports can be specified as bidirectional, or *IO ports*, indicating that they are sometimes used for input and at other times for output. Examples of systems that make extensive use of IO ports include bidirectional data-buses, random access memory, and the control subsystem of PCI buses, to name just a few. The use of IO ports, however, is restricted to circuit simulation; they are not included in the synthesizable fragment of HDLs.

We present a model of synchronous system composition that includes both dependent types and bidirectional types. A *dependent type* chooses the dependencies between variables based on the next-state values of other variables; a *bidirectional type* classifies variables into input and output dynamically. We then solve the single-step fixed-type and unknown-type control problems for systems with dependent and bidirectional types. Using our solution for the single-step control problem as a subroutine, controllers for more general temporal properties can be obtained in the standard way. In particular, our algorithms can be used to synthesize sequential circuits with gated clocks and IO ports.

The modeling and controller synthesis for dependent and bidirectional types is based on game-theoretic notions. In a *dependent type*, the input-output dependencies unfold in steps as the next-state variables acquire values. In a *bidirectional type*, also the commitments of variables to represent output unfold in steps as the next-state variables acquire values. These unfoldings can be viewed as a game in which the system and the controller, starting from a state, proceed to assign values to individual variables until the next state is completely specified. Our solutions of the resulting control problems are based on algorithms for solving such multi-step finite games. Although dependent and bidirectional types are significantly more general than the static and dynamic types studied previously, our results show that the computational complexity of the resulting control problems does not increase. In particular, we prove that for the most general, bidirectional types, the fixed-type control problem for boolean systems is NEXP-complete, and the unknown-type control problem is PSPACE-complete, as is already the case for static types. We also show that while the composability check for dependent types is difficult (coNP-complete), it is no more difficult than in the case of dynamic types, and simpler than checking if an untyped system is input-enabled (Π_2-complete).

2 Types for Synchronous Composition

Let V be a set of variables. In this paper all variables range over the set $\mathbb{B}$ of booleans. We denote by $\mathit{PStates}(V)$ the set of partial functions from V to $\mathbb{B}$, and by $\mathit{States}(V)$ the set of total functions. Given $v \in \mathit{PStates}(V)$, we write $\mathcal{V}ar(v) \subseteq V$ for the set of variables on which v is defined. For $X \subseteq V$, we write $v[X]$ for the restriction of v to the variables in X. For a boolean formula φ over V, we write $\varphi[\![v]\!] = \varphi[v(x_1)/x_1, \ldots, v(x_n)/x_n]$ for the formula obtained

by replacing each variable $x_i \in \mathcal{V}ar(v)$ in φ with the truth value $v(x_i)$. We write $V' = \{x' \mid x \in V\}$ for the set of corresponding primed variables, and for $v \in \mathit{PStates}(V)$, we write v' for the partial function in $\mathit{PStates}(V')$ such that $v'(x') = v(x)$ for all $x \in \mathcal{V}ar(v)$, and $v'(x')$ is undefined otherwise.

Modules. A *module* M consists of the following two components:

- A finite set V_M of *module variables*. The *states* of M are $S_M = \mathit{States}(V_M)$, and the *partial* (*next*) *states* of M are $R_M = \mathit{PStates}(V'_M)$. Unprimed variables represent current-state values; primed variables, next-state values. A pair $\langle s, t' \rangle \in S_M \times R_M$ is called an *extended state*.
- A boolean formula τ_M, called *transition predicate*, over the set $V_M \cup V'_M$ of variables; it relates the current-state and next-state values of the module variables. The state $t \in S_M$ a *macro-step successor* of the state $s \in S_M$ if $\tau_M[\![s \cup t']\!]$ is true. For a variable $x \in V_M$, the extended state $\langle s, u' \rangle \in S_M \times R_M$ is a (*micro-step*) (x, τ_M)*-successor* of the extended state $\langle s, t' \rangle$ if $x' \notin \mathcal{V}ar(t')$ and there exists $b \in \mathbb{B}$ such that $u' = t' \cup \{(x', b)\}$ and $\tau_M[\![s \cup u']\!]$ is satisfiable.

Given two modules M and N, the (*synchronous*) *composition* $M||N$ is the module with $V_{M||N} = V_M \cup V_N$ and $\tau_{M||N} = \tau_M \wedge \tau_N$. A module M is *nonblocking* if every state has a macro-step successor; that is, for all states $s \in S_M$, there exists a state $t \in S_M$ such that $\tau_M[\![s \cup t']\!]$ is true. Synchronous composition is problematic because it may cause blocking even if both components are nonblocking. This is particularly undesirable in control applications, where we usually want to rule out controllers that achieve the control objective simply by blocking the plant from progressing.

Proposition 1 *It is Π_2-complete to check if the composition of two nonblocking modules is nonblocking.*

In order to simplify the check that synchronous composition is nonblocking, we augment modules with *types*. We indicate by (M, γ) the pair consisting of a module M and its type γ. We will consider several classes of module types. For each such class T, we will define a notion of T*-composability*, which specifies whether two typed nonblocking modules (M, γ) and (N, γ') with $\gamma, \gamma' \in T$ can be composed into a single nonblocking module.

Single-step control. Given a class T of types and a typed module (M, γ) with $\gamma \in T$, a T*-controller* for (M, γ) is a typed module (N, γ') with $\gamma' \in T$ which is T-composable with (M, γ). The *single-step control problem* for T asks, given a typed module (M, γ), a state $s \in S_M$, and a boolean formula φ over V_M, if there is a T-controller (N, γ') for (M, γ) such that for all states $t \in S_M$, if $\tau_{M\|N}[\![s \cup t']\!]$ is true, then $\varphi[\![t]\!]$ is true. The controller N ensures that starting from s, the predicate φ holds after one step of the *closed-loop system* $M\|N$, and it does so without blocking the progress of M. If the answer is Yes, then the state s is *single-step* T*-controllable* by (N, γ') w.r.t. the *control objective* φ. We distinguish two kinds of control problems. In the *unknown-type* control problem $((M, \gamma), s, \varphi)$, we are free to choose the type γ' of the controller; in the *fixed-type* control problem $((M, \gamma), s, \varphi, \gamma')$, the type γ' of the controller is specified as part of the problem statement.

IO-type modules. We partition the module variables into input and output, when composing modules, we disallow variables to be output from more than one module. An *IO-type module* (M, π_M), or simply *IO-module*, consists of a module M and a partition $\pi_M = (V_M^i, V_M^o)$ of the module variables V_M into a set V_M^i of input variables and a set $V_M^o = V_M \setminus V_M^i$ of output variables. We refer to π_M as an *IO-type* for M. Two IO-modules (M, π_M) and (N, π_N) are *IO-composable* if their output variables are disjoint; that is, $V_M^o \cap V_N^o = \emptyset$. The *IO-composition* is the IO-module $(M \| N, \pi_{M\|N})$, where $V_{M\|N}^o = V_M^o \cup V_N^o$ and $V_{M\|N}^i = (V_M^i \cup V_N^i) \backslash V_{M\|N}^o$. The fact that two IO-modules are IO-composable does not suffice to guarantee that their composition is nonblocking. Therefore, we either restrict the transition predicate (as in the case of Moore modules), or we augment IO-types with additional information (such as dependency relations).

Moore modules. A *Moore module* (M, π_M) is an IO-module such that (a) the module M is non-blocking, and (b) the next values of output variables V_M^o do not depend on the next values of input variables; that is, for all states $s, t, u \in S_M$, if $\tau_M[\![s \cup t']\!]$ and $t[V_M^o] = u[V_M^o]$, then $\tau_M[\![s \cup u']\!]$. The composition of two IO-composable Moore modules is nonblocking.

Statically typed modules. A *dependency relation* for an IO-module (M, π_M) is an acyclic binary relation $\succ \subseteq V_M^o \times V_M$ between the output variables and the module variables (acyclicity means that the transitive closure is irreflexive). The IO-module (M, π_M) *respects* the dependency relation $\succ$ at state $s \in S_M$ if for all states $t \in S_M$ with $\tau_M[\![s \cup t']\!]$, for each subset $Y^i \subseteq V_M^i$ of input variables, and for each truth-value assignment u^i to the variables in Y^i, there is a state u with $\tau_M[\![s \cup u']\!]$ such that $u[Y^i] = u^i$, and $u[Y] = t[Y]$ for $Y = \{z \in V_M \mid (\text{not } z \succ^* y) \text{ for all } y \in Y^i\}$, where $\succ^*$ is the reflexive-transitive closure of $\succ$. A *statically typed module* $(M, \pi_M, \succ_M)$ consists of an IO-module (M, π_M) and a dependency relation $\succ_M$ for (M, π_M) such that (a) the module M is nonblocking, and (b) the IO-module (M, π_M) respects the dependency relation $\succ_M$ at all states in S_M. We refer to the pair $(\pi_M, \succ_M)$ as a *static type* for M. Two statically typed modules $(M, \pi_M, \succ_M)$ and $(N, \pi_N, \succ_N)$ are *statically composable* if (1) they are IO-composable and (2) the relation $\succ_M \cup \succ_N$ is acyclic. The composition of two statically composable modules is nonblocking. These modules can be used to model sequential circuits without combinational loops.

Dynamically typed modules. A *composite dependency relation* for an IO-module (M, π_M) is a set $D = \{(\psi^1, \succ^1), \ldots, (\psi^m, \succ^m)\}$ of pairs, where each ψ^i is a boolean formula over the module variables V_M, and each $\succ^i$ is a dependency relation for (M, π_M), such that for every state $s \in S_M$, there is exactly one formula ψ^i, $1 \le i \le m$, such that $\psi^i[\![s]\!]$ is true. If $\psi^i[\![s]\!]$, then we write $\succ^s$ for the corresponding dependency relation $\succ^i$. A *dynamically typed module* (M, π_M, D_M) consists of an IO-module (M, π_M) and a composite dependency relation $D_M = \{(\psi_M^i, \succ_M^i) \mid 1 \le i \le m\}$ for (M, π_M) such that (a) the module M is nonblocking, and (b) at every state $s \in S_M$, the module M respects the dependency relation $\succ_M^s$. We refer to the pair (π_M, D_M) a dynamic type for the module M. Two dynamically typed modules (M, π_M, D_M) and (N, π_N, D_N) are

dynamically composable if (1) they are IO-composable and (2) the relation $\succ^{i,j}$ is acyclic for all $1 \leq i \leq m$ and $1 \leq j \leq n$ for which the conjunction $\psi_M^i \wedge \theta_N^j$ is satisfiable. The composition of two dynamically composable modules is non-blocking. These modules can be used to model circuits with transparent latches that may contain combinational loops [dAHM00a].

Theorem 1 [dAHM00a] *It can be checked in linear time if two statically typed modules are statically composable. It is coNP-complete to check if two dynamically typed modules are dynamically composable.*

Theorem 2 [dAHM00a] *The single-step control problem for Moore modules is Σ_2-complete. The single-step unknown-type control problems for statically and dynamically typed modules are PSPACE-complete. The single-step fixed-type control problems for statically and dynamically typed modules are NEXP-complete.*

The goal of this paper is to extend our type systems to capture wider classes of nonblocking synchronous composition, and study the resulting control problems. In particular, we add to our list of type classes *dependent types* and *bidirectional types*. Dependent types generalize dynamic types by allowing the dependency relation to be a function not only of the current state, but also of the partial next state. Bidirectional types further generalize the dependent types by removing the requirement that module variables are partitioned into input and output variables a priori. Rather, the choice of which variables are used as inputs and outputs is performed dynamically, while the values of the variables themselves are chosen.

3 Macro-Steps as Micro-Step Graphs

The following notions will be used in the definition of both dependent and bidirectional types. The variable dependency relation establishes the possible orders in which the variables can be assigned a value in order to determine the next state, and the dependencies among the values chosen. The micro-step graph makes explicit the partial states traversed as a new macro-step successor is determined. We will solve single-step control problems by considering games on this micro-step graph.

Variable dependency relation. A *variable dependency relation* for M is a set $C = \{(\psi^1, \succcurlyeq^1), \ldots, (\psi^m, \succcurlyeq^m)\}$ of pairs, where each ψ^i is a boolean formula over the unprimed and primed module variables $V_M \cup V'_M$, and each $\succcurlyeq^i \subseteq V_M \times 2^{V_M}$ is a binary relation with the intention that if ψ^i holds in an extended state, and $x \succcurlyeq^i Y$, then x can be given a next value, and this value can depend on the next values of the variables in Y. The set C is an *IO-variable dependency relation* for an IO-module (M, π_M) if $\succcurlyeq^i \subseteq V_M^o \times 2^{V_M}$ for all $1 \leq i \leq m$; that is, dependencies are specified only for output variables. A variable dependency relation is a syntactic object; to make the variable dependencies more explicit, we define the corresponding *dependency function* $\tilde{C}$: $S_M \times R_M \times V_M \to 2^{V_M}$ as the function that, given an extended state $\langle s, t' \rangle$ and a variable x, specifies the set of variables on which x depends, as $\tilde{C}(s, t', x) = \bigcup\{Y \mid (\psi, (x, Y)) \in C$ and $\psi[\![s \cup$

$t'\rrbracket$ contains no free variables and is true$\}$. The variable x is *enabled* for (M, C) at the extended state $\langle s, t'\rangle$ if $\tilde{C}(s, t', x) \subseteq \mathcal{V}ar(t)$. The module M *respects* the variable dependency relation C if for every pair of extended states $\langle s, t'\rangle$ and $\langle s, u'\rangle$ of M, for every variable $x \in V_M$ that is enabled for (M, C) at both extended states, and for every $b \in \mathbb{B}$, if $t[\tilde{C}(s, t', x)] = u[\tilde{C}(s, u', x)]$, then the extended state $\langle s, t' \cup \{(x', b)\}\rangle$ is an (x, τ_M)-successor of $\langle s, t'\rangle$ iff the extended state $\langle s, u' \cup \{(x', b)\}\rangle$ is an (x, τ_M)-successor of $\langle s, u'\rangle$. If the transition predicate of a module is specified by a set Γ of nondeterministic guarded commands, then the variable dependency relation can be deduced from Γ as follows: for each guarded command $\llbracket g \to x' = e$ in Γ, let $(g, \succcurlyeq) \in C$, where $\succcurlyeq = \{(x, Y) \mid y \in Y$ iff y' occurs in g or in $e\}$.

Micro-step graph. Consider two modules M and N together with variable dependency relations C_M and C_N. Let $V = V_M \cup V_N$. The micro-step graph of M and N encodes the sequential process by which M and N update the variable values from a state to its macro-step successor. Formally, for a state $s \in S$, the *micro-step graph* $MG_s(M, C_M, N, C_N)$ is a directed acyclic graph whose vertices are the tuples $\langle s, t', U, W\rangle$, where $s \in States(V)$, $t' \in PStates(V')$, $U \subseteq V_M$, and $W \subseteq V_N$, together with the additional distinguished vertex $\perp$, which is used to denote an illegal configuration. The edges of $MG_s(M, C_M, N, C_N)$ are partitioned in M-edges and N-edges; they are defined as follows, for all vertices $\alpha = \langle s, t', U, W\rangle$ and all variables $x \in V$:

- If $x \notin \mathcal{V}ar(t)$ and x is enabled for (M, C_M) at the extended state $\langle s, t'\rangle$, then for each (x, τ_M)-successor $\langle s, u'\rangle$ of α, if $\langle s, u'\rangle$ is also an (x, τ_N)-successor of α, then there is an M-edge from α to the vertex $\langle s, u', U \cup \{x\}, W\rangle$; and if $\langle s, u'\rangle$ is not an (x, τ_N)-successor of α, then there is an M-edge from α to $\perp$. The N-edges are defined symmetrically.
- If $x \in (\mathcal{V}ar(t) \cap W)$ and x is enabled for (M, C_M) at the extended state $\langle s, t'\rangle$, then there is an M-edge from α to $\perp$. The N-edges are defined symmetrically.

A vertex of $MG_s(M, C_M, N, C_N)$ is *terminal* if it does not have any outgoing edges. Note that the micro-step graph has the following properties: there are at most $2^{O(|V|)}$ vertices, the size of each vertex is at most $4 \cdot |V|$, and the depth of the graph is at most $|V| + 1$. If a vertex α of $MG_s(M, C_M, N, C_N)$ has both outgoing M- and N-edges, then α is *mixed.* The *M-reduced micro-step graph* $RMG_s^M(M, C_M, N, C_N)$ is the micro-step graph obtained from $MG_s(M, C_M, N, C_N)$ by pruning, for all mixed vertices α, all N-edges outgoing from α. Intuitively, the reduced graph represents the situation in which the module M has precedence over N in updating variable values.

4 Dependent-type Modules

Consider an IO-module (M, π_M) together with an IO-variable dependency relation C_M for (M, π_M). Let (E, π_E, C_E) consist of the module E with the input variables $V_E^i = V_M^o$, the output variables $V_E^o = V_M^i$, the transition predicate $\tau_E = \mathrm{T}$, and the IO-variable dependency relation $\{(\mathrm{T}, \{(x, \emptyset)\}) \mid x \in V_E^o\}$. The triple (E, π_E, C_E) is the *most general dependent-type environment* for (M, π_M);

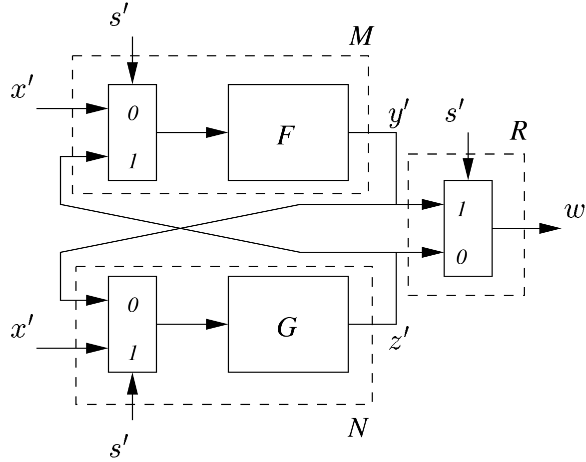


Fig. 1. A cyclic circuit composed of three modules M, N, and R. It performs the following function: if s' then $w' = F(G(x'))$ else $w' = G(F(x'))$, where F and G are two combinational blocks, such as a shifter and adder.

it assigns nondeterministically values to the input variables of (M, π_M). The triple (M, π_M, C_M) is a *dependent-type module* if (a) the module M respects the variable dependency relation C_M and (b) for every state $s \in S_M$, if there is a path in the micro-step graph $MG_s^M(M, C_M, E, C_E)$ from the initial vertex $\langle s, \emptyset, \emptyset, \emptyset \rangle$ to a terminal vertex α, then $\alpha \neq \perp$, and $\alpha = \langle s, t', V_M^o, V_M^i \rangle$ for some state $t \in \mathit{States}(V_M)$. Condition (b) states that for all environment inputs, the module M does not block. We refer to the pair (π_M, C_M) as a *dependent type* for M.

Example 1 [Mal94] Cyclic circuits are often used in hardware systems for minimizing the circuit size. As an example, consider the circuit in Figure 1. The output w is a function of the inputs s and x. The circuit consists of three dependent-type modules M, N and R. In guarded commands, they are

$$M = \begin{array}{l} [\!] \; \neg s' \;\; \to y' = F(x') \\ [\!] \; s' \;\;\;\; \to y' = F(z') \end{array} \quad N = \begin{array}{l} [\!] \; \neg s' \;\; \to z' = G(y') \\ [\!] \; s' \;\;\;\; \to z' = G(x') \end{array} \quad R = \begin{array}{l} [\!] \; \neg s' \;\; \to w' = z' \\ [\!] \; s' \;\;\;\; \to w' = y' \end{array}$$

Module M has the variables $V_M^o = \{y\}$ and $V_M^i = \{s, x, z\}$, and the IO-variable dependency relation $\{(\neg s', \{(y, \{x, s\})\}), (s', \{(y, \{z, s\})\})\}$. Module N has the variables $V_N^o = \{z\}$ and $V_N^i = \{s, x, y\}$, and the IO-variable dependency relation $\{(\neg s', \{(z, \{y, s\})\}), (s', \{(z, \{x, s\})\})\}$. Module R has the variables $V_R^o = \{w\}$ and $V_R^i = \{y, z\}$, and the IO-variable dependency relation $\{(\neg s', \{(w, \{z, s\})\}), (s', \{(w, \{y, s\})\})\}$. ∎

Proposition 2 *Every IO-module with a dependent type is nonblocking. Every nonblocking IO-module has a dependent type.*

Composition. Two dependent-type modules (M, π_M, C_M) and (N, π_N, C_N) are *dependent-type composable* if (1) they are IO-composable and (2) the IO-composition $(M\|N, \pi_{M\|N})$ respects the IO-variable dependency relation $C_{M\|N} = C_M \cup C_N$. Then the pair $(\pi_{M\|N}, C_{M\|N})$ is a dependent type for the composite module $M\|N$. The following theorem shows that checking composability for dependent-type modules has the same worst-case complexity as for dynamically typed modules.

Theorem 3 *It is coNP-complete to check if two dependent-type modules are dependent-type composable.*

Proof. (sketch) Given two dependent-type modules (M, π_M, C_M) and (N, π_N, C_N), to show that they are not composable one can guess a path $\langle s, \emptyset, \emptyset, \emptyset\rangle, \langle s, t'_1, U_1, W_1\rangle, \langle s, t'_2, U_2, W_2\rangle, \ldots, \langle s, t'_n, U_n, W_n\rangle = \alpha$ in the micro-step graph $MG_s(M, C_M, N, C_N)$, and check that $\mathcal{V}ar(t_n) \subsetneq V_{M\|N}$, and that there is no outgoing edge from α. This last condition can be checked by checking that $\mathcal{V}ar(t_n)$ contains all input variables $V^i_{M\|N}$, and that no output variable undefined in t'_n is enabled at the extended state $\langle s, t'_n\rangle$. Note that this check requires only polynomial time, because a variable can be enabled only when all variables that occur in its enabling condition are assigned a value by $s \cup t'_n$. ■

Dependent types capture a larger class of nonblocking synchronous composition than dynamically typed modules, as shown by the following proposition. For a dependent-type module (M, π_M, C_M), the variable $x \in V_M$ depends on $y \in V_M$ at a state $s \in S_M$, written $x \succ^s y$, if there exists a partial state $t' \in S_M$ and a pair $(\psi, \succcurlyeq) \in C_M$ such that $\psi[\![s \cup t']\!]$ is true and $x \succcurlyeq Y$ with $y \in Y$. Then $D_M = \{(s, \succ^s) \mid s \in S_M\}$ is a dynamic type for M.

Proposition 3 *There are two dependent-type modules that are dependent-type composable but not composable if viewed as dynamically typed.*

Example 2 The dependent-type modules M, N, and R from Example 1 are dependent-type composable. If these modules are viewed as dynamically typed modules, the output of each module will depend on the respective inputs. Hence there will be a cyclic dependency in the union of their dependency relations, namely, $y \succ z$ and $z \succ y$ at all states. Since dynamically typed modules do not permit cyclic dependencies, these modules are not dynamically composable. ■

Unknown-type control. By relaxing the composability requirement of modules from dynamic to dependent types, we can control a larger class of modules.

Proposition 4 *There is a dependent-type module (M, π_M, C_M), a control objective φ over V_M, and a state $s \in S_M$ such that s is single-step controllable w.r.t. φ by a dependent-type controller but not by a dynamically typed controller.*

Example 3 Let M be the module with $V^i_M = \{u, v\}$ and $V^o_M = \{x\}$, and the following guarded commands:

$$M = \begin{array}{l} [\!]\ u' \rightarrow\ x' = \neg v' \\ [\!]\ \neg u' \rightarrow\ x' = \mathrm{T} \\ [\!]\ \neg u' \rightarrow\ x' = \mathrm{F} \end{array}$$

The control objective is $x = v$. There is no dynamically typed controller (at any state), because such a controller would have x depend on v, and M can set x to be $\neg v$. But a dependent-type controller can change the dependencies, namely, have x not depend on any variable and have v depend on x. The following is a dependent-type controller:

$$N = \begin{array}{l} [\![\text{T} \rightarrow u' = \text{F} \\ [\![\text{T} \rightarrow v' = x' \end{array} \quad \blacksquare$$

Consider the single-step unknown-type control problem $((M, \pi_M, C_M), s, \varphi)$. It is convenient to view this control problem as a game between the dependent-type module (M, π_M, C_M) and its controller. The game is played on the M-reduced micro-step graph $RMG_s^M(M, C_M, E, C_E)$, where (E, C_E) is the most general dependent-type environment for (M, π_M). Note that every nonterminal vertex of the reduced micro-step graph either has only outgoing M-edges or has only outgoing E-edges. We call the vertices with only outgoing M-edges the *module vertices*, and the vertices with only outgoing E-edges the *environment vertices*. Then we solve the game by the following marking algorithm. A terminal vertex $\alpha = \langle s, t', U, W\rangle$ is marked if $U \cup W = V_M$ and $\varphi[\![t]\!]$ is true. A module vertex α is marked if all successors β of α are marked. An environment vertex α is marked if some successor β of α is marked. The answer to the given single-step unknown-type control problem is Yes iff the vertex $\langle s, \emptyset, \emptyset, \emptyset\rangle$ is marked.

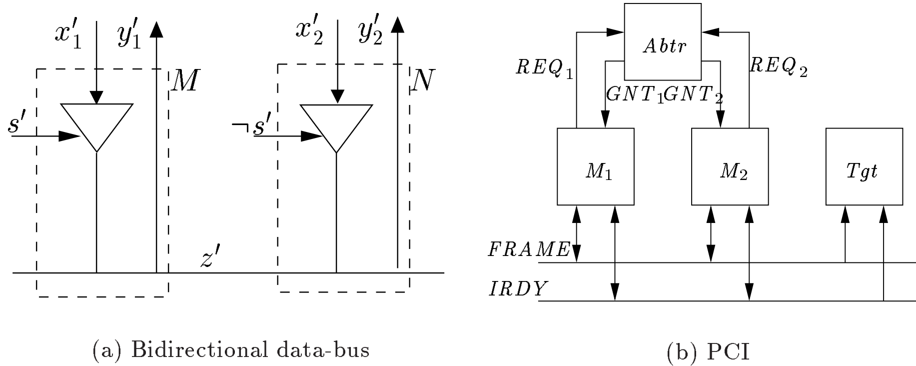
If the answer to the control problem is Yes, then the method also suggests a way of synthesizing a dependent-type controller (N, π_N, C_N) as a set of guarded commands Γ. Given a state $s \in S_M$, denote by χ_s the *characteristic formula* of s, defined by $\chi_s = \bigwedge\{x \mid (x, \text{T}) \in s\} \wedge \bigwedge\{\neg x \mid (x, \text{F}) \in s\}$. The controller has the output variables $V_N^o = V_M^i$ and input variables $V_N^i = V_M^o$. For every marked environment vertex $\alpha = \langle s, t', U, W\rangle$ in the reduced micro-step graph, choose one marked successor $\langle s, t' \cup \{(x', b)\}, U, W \cup \{x\}\rangle$ of α, and add to Γ the guarded command $[\![\chi_s \wedge \chi_{t'} \rightarrow x' = b$. Like composability checking, the single-step unknown-type control problem for dependent-type modules is no harder than its counterpart for dynamically typed modules.

Theorem 4 *The single-step unknown-type control problem for dependent-type modules is PSPACE-complete. Moreover, if the answer is Yes, then a dependent-type controller can be synthesized.*

Fixed-type control. The fixed-type control problem is computationally harder than the unknown-type control problem, although it is no harder than its counterpart for dynamically typed modules. The additional complexity is due to the fact that we need to construct explicitly the micro-step graph.

Theorem 5 *The single-step fixed-type control problem for dependent-type modules is NEXP-complete. Moreover, if the answer is Yes, then a dependent-type controller can be synthesized.*

Proof. (sketch) Given a dependent-type module (M, π_M, C_M), a state $s \in S_M$, a control objective φ over V_M, and a dependent type (π_N, C_N) for



(a) Bidirectional data-bus

(b) PCI

Fig. 2. Examples of bidirectional modules. Figure 2(a): A system with two processors communicating through a bidirectional bus. The variable z is output of the module M if $s' = \text{T}$, and it is output of N if $s' = \text{F}$. Figure 2(b): A PCI system with two master devices and one target device. The figure also shows the arbiter.

the controller, one can guess a subgraph G of the reduced micro-step graph $RMG_s^M(M, C_M, E, C_E)$, for the most general dependent-type environment (E, π_E, C_E), and check that (1) the vertex $\langle s, \emptyset, \emptyset, \emptyset \rangle$ is in G; (2) for all module vertices α in G, all successors of α are also in G; (3) for all environment vertices α in G, there is exactly one successor β of α in G such that (α, β) is an edge in G; (4) for all terminal vertices $\langle s, t', \cdot, \cdot \rangle$, the formula $\varphi[\![t]\!]$ is true; (5) the controller respects the IO-variable dependency relation C_N at all environment vertices in G. All of these conditions can be checked in time polynomial in the size of G. NEXP-hardness comes from the fact that dependent-type modules can encode dynamically typed modules. ∎

5 Bidirectional Modules

A *bidirectional module* (M, C_M) consists of a module M and a variable dependency relation C_M for M such that M respects C_M. We refer to C_M as a *bidirectional type* for M. Note that a bidirectional type does not contain an IO-type. The information about which variables can be outputs of a bidirectional module is encoded instead by its variable dependency relation: if a variable $x \notin \mathcal{V}ar(t')$ is enabled at the extended state $\langle s, t' \rangle$, then x can be assigned a value at $\langle s, t' \rangle$, and thus used as an output. Unlike the various typed modules we defined previously, bidirectional modules do not guarantee nonblocking. This is because a bidirectional module may not be able to produce suitable "outputs" (i.e., values for some module variables) for all possible environment "inputs" (i.e., values for the other module variables). For instance, the environment may try to assign a value to a variable that already has a value assigned by the module. Hence,

the environment has to be specified explicitly, and two bidirectional modules are composable only if the result is nonblocking.

Composition. Informally speaking, two bidirectional modules are composable if in all macro-steps, every module variable is assigned a value exactly once (by either of the modules). Then the composition is nonblocking, and each variable is output of exactly one component. Given two bidirectional modules (M, C_M) and (N, C_N), let $V = V_M \cup V_N$. The modules (M, C_M) and (N, C_N) are *bidirectionally composable* if for all $s \in States(V)$, if there is a path in the micro-step graph $MG_s(M, C_M, N, C_N)$, from the initial vertex $\langle s, \emptyset, \emptyset, \emptyset \rangle$ to a terminal vertex α, then $\alpha \neq \perp$, and $\alpha = \langle s, t', U, W \rangle$ with $U \cup W = V$. Unfortunately, checking bidirectional composability is computationally harder than checking composability for the other previous types.

Theorem 6 *It is Π_2-complete to check if two bidirectional modules are bidirectionally composable.*

The additional hardness comes from the fact that when composing two bidirectional modules, one of the modules may choose a value for the common variables such that the other module blocks. This cannot not happen with statically typed, dynamically typed, or dependent-type modules, because they do not block on any inputs they receive. For a bidirectional module (M, C_M), let $V_M^o = \{x \in V_M \mid$ there exists a pair $(\psi, \succcurlyeq) \in C_M$ such that $x \succcurlyeq Y$ for some $Y \subseteq V_M\}$. Then the pair (π_M, C_M), where $\pi_M = (V_M^o, V_M \setminus V_M^o)$, is a dependent type for M.

Proposition 5 *There are two bidirectional modules that are bidirectionally composable but not composable if viewed as dependent-type modules.*

Example 4 In a multi-processor system, a bidirectional data-bus can be modeled as a bidirectional module. Typically, at most one processor has the right to write to the data-bus, while the others can only read from the data-bus. Figure 2(a) shows a simplified system in which there are two processors M and N communicating via the data-bus z. The variable z may be an output of either M or N, depending on the value of s. In guarded commands, M and N are:

$$M = \begin{array}{l} [\!] \; s' \to z' = x_1' \\ [\!] \; \mathrm{T} \to y_1' = x_1' \end{array} \qquad N = \begin{array}{l} [\!] \; \neg s' \to z' = x_2' \\ [\!] \; \mathrm{T} \to y_2' = x_2' \end{array}$$

We assume that M and N are composed with an environment that nondeterministically chooses values for the variables s, x_1, and x_2. These three modules are bidirectionally composable, but they are not composable if viewed as dependent-type modules because z is output of both M and N. ∎

Example 5 (PCI bus arbitration) Bidirectional modules can be used to model the PCI bus protocol[SIG]. The PCI bus is an industry standard commonly used for interfacing between the core computer system (e.g., CPU, memory, etc.) and the peripheral devices (e.g., audio, video etc.). Typically, the master devices attached to a PCI bus may request to own the bus in order to communicate with the respective target devices. Once a request is granted by

the arbiter, the bus will be owned by the selected master device and only this master device or its target device can write onto the bus.

Consider an instance of the bus protocol depicted in Figure 2(b). There are two master devices and one target device. During the arbitration phase, the master devices may request to own the bus. When ownership is granted by the arbiter, the selected master device checks if the bus is idle (by observing that the values of both signals *FRAME* and *IRDY* are high), and then drives these two signals, so that they become outputs of this master device. Note that the two signals can be output of either master device, depending on the decision of the arbiter. The following guarded commands model the arbitration phase of the system. The two master devices M_i, $i \in \{1, 2\}$ can be described as:

$$M_i = \begin{array}{l} [\!] \ \textsc{t} \rightarrow \ REQ_i = \textsc{t} \\ [\!] \ \textsc{t} \rightarrow \ REQ_i = \textsc{f} \\ [\!] \ FRAME \wedge IRDY \wedge GNT_i \rightarrow \ FRAME' = \textsc{f}; IRDY' = \textsc{t} \end{array}$$

The arbiter *Abtr* can be described as:

$$Abtr = \begin{array}{l} [\!] \ \textsc{t} \rightarrow \ GNT'_1 = \textsc{f}; GNT'_1 = \textsc{f} \\ [\!] \ REQ_1 \rightarrow \ GNT'_1 = \textsc{t}; GNT'_1 = \textsc{f} \\ [\!] \ REQ_2 \rightarrow \ GNT'_1 = \textsc{f}; GNT'_2 = \textsc{t} \end{array} \blacksquare$$

Unknown-type control. Relaxing the composability requirement of modules from dependent to bidirectional types, we can control a larger class of modules.

Proposition 6 *There is a bidirectional module (M, C_M), a control objective φ over V_M, and a state of $s \in S_M$ such that s is single-step controllable w.r.t. φ by a bidirectional controller but not by a dependent-type controller.*

Example 6 Let M be the module with the variables $V = \{x, u\}$ and the following guarded commands:

$$M = \begin{array}{l} [\!] \ u' \rightarrow \ x' = \textsc{f} \\ [\!] \ \neg u' \rightarrow \end{array}$$

The control objective is $x = \textsc{t}$. There is no dependent-type controller (at any state), because if the controller sets u' to $\textsc{t}$, then M will set x' to $\textsc{f}$. On the other hand, if the controller sets u' to $\textsc{f}$, then M does not have an enabled guarded command to assign a value to x'. But a bidirectional controller can bind the variable x to its output and set its value to $\textsc{t}$. The following is a possible bidirectional controller:

$$N = \begin{array}{l} [\!] \ \textsc{t} \ \rightarrow \ u' = \textsc{f} \\ [\!] \ \textsc{t} \ \rightarrow \ x' = \textsc{t} \end{array} \blacksquare$$

Consider the single-step unknown-type control problem $((M, C_M), s, \varphi)$. As for dependent-type modules, this control problem can be viewed as a game played between the module and the controller on the M-reduced micro-step graph $RMG_s^M(M, C_M, E, C_E)$, where (E, C_E) is the most general bidirectional environment for M, defined by $V_E = V_M$, $\tau_E = \textsc{t}$, and variable dependency relation $C_E = \{(\textsc{t}, \{(x, \emptyset)\}) \mid x \in V_E\}$. A vertex $\langle s, t', U, W \rangle$ is a *stop node* if

$U \cup W = V_M$. To solve the game, we use the following marking algorithm. A stop node $\langle s, t', U, W \rangle$ is marked if $\varphi[\![t]\!]$ is true, or if it does not have any outgoing M-edges. For all other vertices α, if it is a module vertex, then it is marked if all successors of α are marked; and if it is an environment vertex, then it is marked if some successor of α is marked. The answer to the given single-step unknown-type control problem is Yes iff the vertex $\langle s, \emptyset, \emptyset, \emptyset \rangle$ is marked.

If the answer to the control probem is Yes, then we can synthesize a bidirectional controller as follows. We construct a subgraph of the reduced micro-step graph by keeping all successors of each marked environment vertex, but only one marked successor of each marked module vertex. This subgraph, called the *control graph,* may not be unique. The following observation is crucial: in every control graph there are no distinct vertices $\langle s, t'_1, U_1, W_1 \rangle$ and $\langle s, t'_2, U_2, W_2 \rangle$ with $t_1 = t_2$ but $U_1 \neq U_2$. We can therefore synthesize a bidirectional controller (N, C_N) as a set of guarded commands Γ from *any* control graph. The controller has module variables $V_N = V_E$. For every environment vertex $\alpha = \langle s, t', U, W \rangle$ in the control graph, if $\beta = \langle s, u', U, W \cup x \rangle$ is the unique successor of α, then we add to Γ the guarded command $[\![\chi_s \wedge \chi_{u'} \rightarrow x' = b$. Note that by the observation on control graphs, no two guarded commands have identical guards. In summary, the single-step unknown-type control problem for bidirectional modules is no harder than its counterpart for dependent-type modules.

Theorem 7 *The single-step unknown-type control problem for bidirectional modules is PSPACE-complete. Moreover, if the answer is Yes, then a bidirectional controller can be synthesized.*

Fixed-type control. Also the single-step fixed-type control problem for bidirectional modules is no harder than its counterpart for dependent-type modules.

Theorem 8 *The single-step fixed-type control problem for bidirectional modules is NEXP-complete. Moreover, if the answer is Yes, then a bidirectional controller can be synthesized.*

Proof. (sketch) The proof is very similar to that for dependent-type modules. One adds to the list of things to check that at each node of the guessed subgraph there is no conflict in the value assignments to the variables. ■

References

[AdAHM99] R. Alur, L. de Alfaro, T.A. Henzinger, F.Y.C. Mang. Automating modular verification. In *Concurrency Theory*, LNCS 1664, pp. 82–97. Springer, 1999.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[dAHM00a] L. de Alfaro, T.A. Henzinger, F.Y.C. Mang. The control of synchronous systems. In *Concurrency Theory*, LNCS 1877, pp. 458–473. Springer, 2000.

[dAHM00b] L. de Alfaro, T.A. Henzinger, F.Y.C. Mang. Detecting errors before reaching them. In *Computer-Aided Verification*, LNCS 1855, pp. 186–201. Springer, 2000.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: design, semantics, implementation. *Science Computer Programming*, 19:87–152, 1992.

[GH82] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proc. Symp. Theory of Computing*, pp. 60–65. ACM Press, 1982.

[Hen61] L. Henkin. Some remarks on infinitely long formulas. In *Infinitistic Methods*, pp. 167–183. Polish Scientific Publishers, 1961.

[Mal94] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13:950–956, 1994.

[McN93] R. McNaughton. Infinite games played on finite graphs. *Ann. Pure and Applied Logic*, 65:149–184, 1993.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Symp. Principles of Programming Languages*, pp. 179–190. ACM Press, 1989.

[PRSV98] R. Passerone, J.A. Rowson, A.L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proc. Design Automation Conference*, pp. 8–13. ACM Press, 1998.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control and Optimization*, 25:206–230, 1987.

[SIG] PCI SIG. PCI local bus specification, rev. 2.2.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Theoretical Aspects of Computer Science*, LNCS 900, pp. 1–13. Springer, 1995.