The Control of Synchronous Systems*,**

Luca de Alfaro*** Thomas A. Henzinger*** Freddy Y.C. Mang***

Abstract. In the synchronous composition of processes, one process may prevent another process from proceeding unless compositions without a well-defined product behavior are ruled out. They can be ruled out semantically, by insisting on the existence of certain fixed points, or syntactically, by equipping processes with types, which make the dependencies between input and output signals transparent. We classify various typing mechanisms and study their effects on the control problem.

A static type enforces fixed, acyclic dependencies between input and output ports. For example, synchronous hardware without combinational loops can be typed statically. A dynamic type may vary the dependencies from state to state, while maintaining acyclicity, as in level-sensitive latches. Then, two dynamically typed processes can be syntactically compatible, if all pairs of possible dependencies are compatible, or semantically compatible, if in each state the combined dependencies remain acyclic. For a given plant process and control objective, there may be a controller of a static type, or only a controller of a syntactically compatible dynamic type, or only a controller of a semantically compatible dynamic type. We show this to be a strict hierarchy of possibilities, and we present algorithms and determine the complexity of the corresponding control problems.

Furthermore, we consider versions of the control problem in which the type of the controller (static or dynamic) is given. We show that the solution of these fixed-type control problems requires the evaluation of partially ordered (Henkin) quantifiers on boolean formulas, and is therefore harder (nondeterministic exponential time) than more traditional control questions.

1 Introduction

The formulation of the control problem builds on the notion of parallel composition: given a transition system M (the "plant"), is there a transition system N (the "controller") such that the compound system $M \| N$ meets a given objective? Hence it is not surprising that even small variations in the definition of composition may influence the outcome of the control problem, as well as the

* A preliminary version of this paper appeared in the *Proceedings of the 11th International Conference on Concurrency Theory* (CONCUR 00), *Lecture Notes in Computer Science* **1877**, Springer-Verlag, 2000, pp. 458–473.

** This research was supported in part by the DARPA grants NAG2-1214 and F33615-C-98-3614, the SRC contract 99-TJ-683.003, the MARCO grant 98-DT-660, and the NSF CAREER award CCR-9501708.

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

hardness of its solution. (The latter distinguishes control from verification, whose complexity —PSPACE for invariant verification— is remarkably resilient against changes in the definition of parallel composition.) At the highest level, one can distinguish between asynchronous and synchronous forms of composition. Pure asynchronous (or interleaving) composition is disjunctive: one component proceeds at a time, so that an action of the compound system is an action of some component. Pure synchronous (or lock-step) composition is conjunctive: all components proceed simultaneously, so that an action of the compound system is a tuple of actions, one for each component. While many concurrency models exhibit mixed forms of composition (e.g., interleaving of internal actions and synchronization of communication actions [Mil89]), it is natural to start by considering the control problem for the two pure forms of composition. The study of these control problems corresponds to the study of winning conditions of games, where the two players (plant vs. controller) choose moves (actions) to prevent (resp. accomplish) the control objective.

In practice, the most important control objective is invariance: the controller strives to forever keep the plant within a safe set of states. The problem of invariance control can be solved by a fixed-point iteration: first, we find a strategy that keeps the plant safe for a single step; then, a strategy that keeps the plant safe for two steps; etc. We henceforth refer to invariance control as the "multi-step" control problem, and to the problem of keeping the plant safe for a single step, as the "single-step" control problem. This allows us to separate concerns: the definition of parallel composition enters the solution of the single-step problem, but independently of the type of composition, the multi-step problem can always be solved by iteratively solving single-step problems. In other words, we can independently study (1) the single-step control problem, and the definition of parallel composition plays a central role in this study, *or* (2) the multi-step control problem (for invariance or even more general, ω-regular objectives), assuming to be given a solution to the single-step problem. While (2) has been researched extensively in the literature [BL69,GH82,RW87,EJ91,McN93,TW94,Tho95], it is (1) we focus on in this paper.

We assume that the plant M is specified in a compact form, by a transition predicate on boolean variables, so that the state space of M is exponentially larger than the description of M, which is the input to the control problem. For solving the multi-step control problem, the number of single-step iterations is bound by the number of states. Therefore, if the single-step problem can be solved in exponential time, then so can the multi-step problem. Conversely, it can be shown that even if the single-step problem can be solved in constant time, the multi-step problem is still complete for EXP (deterministic exponential time). This seems to indicate that the single-step problem is of little interest, and it may explain why not much attention has been paid to the single-step problem previously. To our surprise, we found that for certain natural forms of parallel composition, the single-step control problem can *not* be solved in (deterministic) exponential time, and therefore its complexity dominates also the one of multi-step control.

An essential property of systems is to be *non-blocking*, in the sense that every state should have at least one successor state [BG88,Hal93,Kur94,Lyn96]. Non-blocking is essential for compositional techniques such as assume-guarantee reasoning [AL95,McM97,AH99]. In control, non-blocking means that the controller should never prevent the plant from moving. While the asynchronous composition of non-blocking processes is always non-blocking, synchronous composition needs to be restricted to ensure non-blocking. A second kind of restriction arises from modeling "typed" components, where the type specifies the input ports and output ports of a component, as well as permissible and impermissible dependencies between input and output signals [AH99]. In particular, hardware components are usually typed in this way, for example, in order to avoid combinational loops. In control, if we restrict our attention to typed controllers, then a controller may not exist even when an untyped controller would exist. These two kinds of common restrictions on synchronous composition, non-blocking and typing, are related, as typing can be used for syntactically enforcing the semantic concept of non-blocking for synchronously composed systems.

If the plant is given by a boolean transition predicate, and parallel composition is asynchronous, then single-step control amounts to evaluating the conjunction of a $\forall$ formula ("all actions of the plant are safe") and an $\exists$ formula ("some action of the controller is safe"). Hence, the complexity class of asynchronous single-step control is DP (which contains the differences of languages in NP). For synchronous systems, the various restrictions on composition give rise to different control problems. One way of ensuring non-blocking is to consider only Moore processes. A Moore process is a non-blocking process in which the next values of the output signals do not depend on the next values of the input signals. The composition of Moore processes is again Moore, and therefore non-blocking. If both the system and the controller are Moore processes, then the single-step control formula has the quantifier prefix $\exists\forall$ ("the controller can choose the new input signals, so that regardless of the new output signals, the system is safe").

A more liberal way of ensuring non-blocking is to consider typed processes, i.e., processes that explicitly specify the dependencies between the new values of input and output signals. We distinguish between "static" types, where the input-output dependencies are fixed, and "dynamic" types, where the dependencies can change from state to state. Dynamic types may be composed either "syntactically" (by requiring that all possible combinations of dependency relations of the component processes are acyclic), or "semantically" (by requiring acyclicity at all states of the compound system). Both static and dynamic types ensure that the compound system is again typed, and therefore non-blocking. We consider two variants of the typed control problems: one in which we are free to choose both the controller and its type, and one in which we must find a controller of a specified type. If we can choose the type of the controller, the control problem can be solved by considering for the controller an exponential number of types of a simple form, namely, types that represent linearly ordered input-output dependencies. The single-step control problem resulting

from each linear order of dependencies gives rise to a boolean formula with a linear quantifier prefix, with any number of $\forall\exists$ alternations, which puts the problem into PSPACE. If the type of the desired controller is given, the single-step control problem becomes considerably harder. This is because a (static or dynamic) type may specify partially ordered input-output dependencies. These partially-ordered dependencies correspond to boolean formulas with partially ordered (Henkin) quantifiers [Hen61,Wal70,BG86], whose complexity class for satisfiability is NE (a weak form of nondeterministic exponential time) [GLV95].

The solution of control problems in presence of types gives rise to additional surprising phenomena. For example, with static or syntactically composed dynamic types, two states s and t may both be controllable even though there is not a single controller that controls both s and t (two different controllers are required). Hence, while types provide an efficient mechanism for ensuring the non-blocking of synchronously composed systems, they cause difficulties in control. On the other hand, these difficulties are often not artificial, but they correspond to real input/output constraints in the design of controllers.

2 Types for Synchronous Composition

Preliminaries. Let X be a set of variables. In this paper we consider all variables to range over the set $\mathbb{B}$ of booleans. We write $X' = \{x' \mid x \in X\}$ for the set of corresponding primed variables. A state s over X is a truth-value assignment $s\colon X \mapsto \mathbb{B}$ to the variables in X. We write s' for the truth-value assignment $s'\colon X' \mapsto \mathbb{B}$ defined by $s'(x') = s(x)$ for all $x \in X$. Given a subset $Y \subseteq X$, we write $s[Y]$ for the restriction of s to the variables in Y. Given a predicate φ over the variables in X, we write $\varphi[\![s]\!]$ for the truth value of φ when the variables in X are interpreted according to s. Given a predicate τ over the variables in $X \cup X'$, and states s, t over X, we write $\tau[\![s, t']\!]$ for the truth value of τ when the variables in X are interpreted according to s, and the variables in X' are interpreted according to t'.

Modules and composition. A *module* M consists of the following three components:

- A finite set X^o_M of *output variables*. These variables are updated by the module.
- A finite set X^i_M of *input variables*. These variables are updated by the environment. The sets X^o_M and X^i_M must be disjoint. We write $X_M = X^o_M \cup X^i_M$ for the set of all module variables. The *states* of M are the truth-value assignments to the variables in X_M.
- A predicate τ_M over the set $X_M \cup X'_M$ of unprimed and primed variables. The predicate τ_M, called *transition predicate*, relates the current (unprimed) and next (primed) values of the module variables.

Two modules M and N are *composable* if their output variables X^o_M and X^o_N are disjoint. Given two composable modules M and N, the *synchronous* (*lock-step*) *composition* $M||N$ is the module with the components $X^o_{M||N} = X^o_M \cup X^o_N$,

$X^i_{M||N} = (X^i_M \cup X^i_N) \setminus X^o_{M||N}$, and $\tau_{M||N} = (\tau_M \wedge \tau_N)$. The *asynchronous* (*interleaving*) *composition* $M|N$ is the module with the same output and input variables as $M||N$, but with the transition predicate $\tau_{M|N} = ((\tau_M \wedge (X'^o_N = X^o_N)) \vee (\tau_N \wedge (X'^o_M = X^o_M)))$.

Non-blocking modules. We are interested in the non-blocking modules, a condition necessary for compositional techniques [AH99]. A module M is *non-blocking* if every state has a successor; that is, for each state s there is a state t such that $\tau_M[\![s, t']\!]$. The asynchronous composition of two composable non-blocking modules is again non-blocking. Hence, we say that any two composable non-blocking modules are *async-composable*. However, there are composable non-blocking modules whose synchronous composition is not non-blocking.

Example 1. Let module M be such that $X^o_M = \{x\}$, $X^i_M = \{y\}$, and $\tau_M = ((y' \wedge \neg x') \vee (\neg y' \wedge x'))$. Let module N be such that $X^o_N = \{y\}$, $X^i_N = \{x\}$, and $\tau_N = ((x' \wedge y') \vee (\neg x' \wedge \neg y'))$. Then M and N are non-blocking and composable. However, the transition predicate of $M||N$ is unsatisfiable, i.e., no state of $M||N$ has a successor. ∎

It requires exponential time to check if a module M is non-blocking, which amounts to evaluating the boolean Π^p_2 formula $(\forall X_M)(\exists X'_M)\, \tau_M$. To eliminate the need for this exponential check whenever two modules are composed synchronously, we define four increasingly larger classes of modules for which the non-blocking of synchronous composition can be checked efficiently.

Moore modules. A *Moore module* is a module that (a) is non-blocking, and (b) such that the next values of output variables do not depend on the next values of input variables; that is, for all states s, t, and u, if $\tau_M[\![s, t']\!]$ and $t[X^o_M] = u[X^o_M]$, then $\tau_M[\![s, u']\!]$. These two conditions can be enforced syntactically, in a way that permits checking in linear time. For example, the transition predicate of a Moore module can be specified as a set of nondeterministic guarded commands, one for each primed output variable x' in X'^o_M. The guarded command for x' assigns a value to x' such that (a) one of the guards negates the disjunction of the other guards, and (b) the guards and the right-hand sides of all assignments contain no primed variables. The synchronous composition of two composable Moore modules is again a Moore module, and therefore non-blocking. Hence, we say that any two composable Moore modules are *moore-composable*. However, since many non-blocking modules are not Moore modules, more general types of modules are of interest.

Statically typed modules (or Reactive Modules [AH99]). A *dependency relation* for a module M is an acyclic binary relation $\succ \subseteq X^o_M \times X_M$ between the output variables and the module variables (acyclicity means that the transitive closure is irreflexive). The module M *respects* the dependency relation $\succ$ at state s if, for all states t with $\tau_M[\![s, t']\!]$, for each subset $Y^i \subseteq X^i_M$ of input variables, and for each truth-value assignment u^i to the variables in Y^i, there is a state u with $\tau_M[\![s, u']\!]$ such that $u[Y^i] = u^i$, and $u[Z] = t[Z]$ for $Z = \{z \in X_M \mid (\text{not } z \succ^* y) \text{ for all } y \in Y^i\}$, where $\succ^*$ is the reflexive-transitive

closure of $\succ$. A *statically typed module* $(M, \succ_M)$ consists of a module M and a dependency relation for M, such that (a) the module M is non-blocking, and (b) the module M respects the dependency relation $\succ_M$ at all states. These two conditions, as well as the acyclicity requirement on dependency relations, can be enforced syntactically in a way that permits checking in linear time. For example, we can use guarded commands as with Moore modules, except that the guards and the right-hand sides of assignments are allowed to contain primed variables with the following proviso: if the guarded command for x' contains a primed variable y', then $x \succ_M y$. We refer to the dependency relation $\succ_M$ of a statically typed module $(M, \succ_M)$ as a *static type* for the module M. Note that if $\succ'$ is a dependency relation for M, and $\succ_M$ is a subset of $\succ'$, then $\succ'$ is also a static type for M.

Every non-blocking module has a static type (have each output variable depend on all input variables). Hence, there are composable modules with static types whose synchronous composition does not have a static type. However, static types suggest a sufficient condition for the existence of compound static types which can be checked efficiently. Two statically typed modules $(M, \succ_M)$ and $(N, \succ_N)$ are statically composable, or *static-composable*, if (1) the modules M and N are composable, and (2) the relation $\succ_{M||N} = \succ_M \cup \succ_N$ is acyclic. Then, the relation $\succ_{M||N}$ is a static type for the synchronous composition $M||N$. Since acyclicity can be checked in linear time, so can the requirement if two statically typed modules are *static*-composable. However, two statically typed modules $(M, \succ_M)$ and $(N, \succ_N)$ may not be *static*-composable even though the compound module $M||N$ is non-blocking.

Example 2. A module may have two static types, neither of which is a subset of the other. Let module M be such that $X^o_M = \{x_0, x_1\}$, $X^i_M = \{y\}$, and $\tau_M = (x'_0 \oplus x'_1 \oplus y'_0)$. Using guarded commands, we can specify M in two ways:

$$M' = \left\{ \begin{array}{lcl} [\!] \ \mathtt{T} & \rightarrow & x'_0 := \neg(x'_1 \oplus y') \\ \quad \wedge & & \\ [\!] \ \mathtt{T} & \rightarrow & x'_1 := \mathtt{T} \\ [\!] \ \mathtt{T} & \rightarrow & x'_1 := \mathtt{F} \end{array} \right\} \qquad M'' = \left\{ \begin{array}{lcl} [\!] \ \mathtt{T} & \rightarrow & x'_0 := \mathtt{T} \\ [\!] \ \mathtt{T} & \rightarrow & x'_0 := \mathtt{F} \\ \quad \wedge & & \\ [\!] \ \mathtt{T} & \rightarrow & x'_1 := \neg(x'_0 \oplus y') \end{array} \right\}$$

Note that both M' and M'' have the same transition predicate, namely τ_M, but they have different static types: the static type $\succ_{M'}$ for M' is $\{x_0 \succ x_1,\ x_0 \succ y\}$, while $\succ_{M''} = \{x_1 \succ x_0,\ x_1 \succ y\}$. Choosing different static types (i.e., implementations of the transition predicate) can have implications on composability with other modules. Let module N be such that $X^o_N = \{y\}$, $X^i_N = \{x_0, x_1\}$, and $\tau_N = (y' = x'_0)$ (or, using guarded commands, $[\!] \ \mathtt{T} \ \rightarrow \ y' := x'_0$). The static type $\succ_N$ for N is $\{y \succ x_0\}$. Then $(M', \succ_{M'})$ is not *static*-composable with $(N, \succ_N)$, but $(M'', \succ_{M''})$ is. ∎

Dynamically typed modules. Example 2 suggests the following generalization of static types. A *composite dependency relation* for a module M is a set $D = \{(\psi^1, \succ^1), \ldots, (\psi^m, \succ^m)\}$ of pairs, where each ψ^i is a predicate over the module variables X_M, and each $\succ^i$ is a dependency relation for M, such that for each

state s of M, there is exactly one predicate ψ^i, $1 \leq i \leq m$, with $\psi^i[\![s]\!]$. If $\psi^i[\![s]\!]$, then we write $\succ^s$ for the corresponding dependency relation $\succ^i$. A *dynamically typed module* (M, D_M) consists of a module M and a composite dependency relation $D_M = \{(\psi^i_M, \succ^i_M) \mid 1 \leq i \leq m\}$, such that (a) the module M is non-blocking, and (b) at every state s, the module M respects the dependency relation $\succ^s_M$. These two conditions, as well as the requirements on a composite dependency relation, can again be enforced syntactically in a way that permits checking in polynomial time. For example, each predicate ψ^i_M, $1 \leq i < m$, can be required to contain the conjunct $\bigwedge_{j \neq i} \neg\psi^j_M$, and ψ^m_M can be required to be equal to $\bigwedge_{1 \leq i < m} \neg\psi^i_M$. If we use guarded commands to specify the transition predicate, then for each guarded command, the guard can be required to contain a conjunct of the form ψ^i_M, for some $1 \leq i \leq m$, and together with the right-hand sides of assignments satisfy the proviso for the corresponding dependency relation $\succ^i_M$.

Example 3. Level-sensitive latches are commonly used in the design of high performance systems such as pipelined microprocessors. Typically different parts of a system are active depending on the phase of the clock. As an example, consider a circuit consisting of three modules M_1, M_2, and M_3. Module M_1 is an inverter that connects the output of the $\neg c$-clocked level-sensitive latch M_3 to the input of the c-clocked level-sensitive latch M_2. The output of the latch M_2 is connected to the input of the latch M_3. Using guarded commands, the three modules can be specified as follows:

$$M_1 = \{ \; [\!] \; \mathrm{T} \;\; \to x' := \neg z' \} \quad M_2 = \left\{ \begin{array}{ll} [\!] \; c & \to y' := x' \\ [\!] \; \neg c & \to y' := y \end{array} \right\} \quad M_3 = \left\{ \begin{array}{ll} [\!] \; c & \to z' := z \\ [\!] \; \neg c & \to z' := y' \end{array} \right\}$$

The dynamic types for the modules are $D_{M_1} = \{(\mathrm{T}, x \succ z)\}$, $D_{M_2} = \{(c, y \succ x), (\neg c, \emptyset)\}$, and $D_{M_3} = \{(c, \emptyset), (\neg c, z \succ y)\}$. ∎

We refer to the composite dependency relation D_M of a dynamically typed module (M, D_M) as a *dynamic type* for the module M. Like static types, dynamic types suggest sufficient conditions for the non-blocking of synchronous composition. Furthermore, the conditions for the composability of dynamic types are more liberal than *static*-composability, and thus they are applicable in more situations. Consider two dynamically typed modules (M, D_M) and (N, D_N) with $D_M = \{(\psi^i_M, \succ^i_M) \mid 1 \leq i \leq m\}$ and $D_N = \{(\theta^j_N, \succ^j_N) \mid 1 \leq j \leq n\}$. We write $\succ^{i,j}$ for the union $\succ^i_M \cup \; \succ^j_N$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. We provide two definitions of composability for dynamically typed modules, one purely syntactic, and the other in part semantic.

- The dynamically typed modules (M, D_M) and (N, D_N) are syntactically dynamically composable, or *dsynt-composable*, if (1) the modules M and N are composable, and (2) the relation $\succ^{i,j}$ is acyclic for all $1 \leq i \leq m$ and $1 \leq j \leq n$. Then, $D_{M||N} = \{(\psi^i_M \wedge \theta^j_N, \succ^{i,j}) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$, is a dynamic type for the synchronous composition $M||N$.
- The dynamically typed modules (M, D_M) and (N, D_N) are semantically dynamically composable, or *dsem-composable*, if (1) the modules M and N

are 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. Then, $D_{M||N} = \{(\psi_M^i \wedge \theta_N^j, \succ^{i,j}) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n \text{ and } (\exists X_{M||N})(\psi_M^i \wedge \theta_N^j)\}$ is a dynamic type for $M||N$.

Note that it can be checked in quadratic time whether two dynamically typed modules are *dsynt*-composable, while it requires exponential time (by evaluating a quadratic number of boolean Π_1^p formulas) to check if they are *dsem*-composable. However, checking if two dynamically typed modules are *dsem*-composable is still simpler than checking if the synchronous composition of two untyped modules is non-blocking (Π_1^p vs. Π_2^p).

Proposition 1. *(1) There are two dynamically typed modules (M, D_M) and (N, D_N) that are dsynt-composable but not static-composable, even though the union of all dependency relations in D_M is a static type for M, and the union of all dependency relations in D_N is a static type for N. (2) There are two dynamically typed modules that are dsem-composable but not dsynt-composable. (3) There are two dynamically typed modules (M, D_M) and (N, D_N) that are not dsem-composable, even though the synchronous composition $M||N$ is non-blocking.*

Example 4. The dynamically typed modules (M_1, D_{M_1}) and (M_2, D_{M_2}) of Example 3 are *dsynt*-composable, and the compound module $N = M_1 \| M_2$ has the dynamic type $D_N = \{(c, y \succ x \succ z), (\neg c, x \succ z)\}$. The modules (N, D_N) and (M_3, D_{M_3}) are not *dsynt*-composable, but they are *dsem*-composable. The compound module $N \| M_3$ has the dynamic type $\{(c, y \succ x \succ z), (\neg c, x \succ z \succ y)\}$. If the variable c in modules N and M_3 is replaced by its primed counterpart c' in both transition relations, then the dependency relation for N becomes $\{y \succ x \succ z,\ y \succ c\}$ for every state, and that for M_3 becomes $\{z \succ y,\ z \succ c\}$ for every state. Then (N, D_N) and (M_3, D_{M_3}) are not *dsem*-composable, even though the synchronous composition $N \| M_3$ is non-blocking. ∎

Summary. Let $\Lambda = \{\mathit{async}, \mathit{moore}, \mathit{static}, \mathit{dsynt}, \mathit{dsem}\}$ be the set of *module classes*. We summarize this section by defining, for each module class $\alpha \in \Lambda$, a set $\mathcal{M}_\alpha$ of modules: for $\alpha = \mathit{async}$, let $\mathcal{M}_{\mathit{async}}$ be the set of non-blocking modules; for $\alpha = \mathit{moore}$, let $\mathcal{M}_{\mathit{moore}}$ be the set of Moore modules; for $\alpha = \mathit{static}$, let $\mathcal{M}_{\mathit{static}}$ be the set of statically typed modules; and for $\alpha = \mathit{dsynt}$ and $\alpha = \mathit{dsem}$, let $\mathcal{M}_{\mathit{dsynt}} = \mathcal{M}_{\mathit{dsem}}$ be the set of dynamically typed modules. Define the *module class ordering* $\mathit{async} < \mathit{moore} < \mathit{static} < \mathit{dsynt} < \mathit{dsem}$. Then, for $\alpha, \beta \in \Lambda$ with $\alpha < \beta$, every module $M \in \mathcal{M}_\alpha$ can be considered to be a module in $\mathcal{M}_\beta$ by adjusting its type or the semantics of composition, if necessary. Precisely: an *async*-module can be considered as a *moore*-module by changing the semantics of composition; a *moore*-module can be considered a *static*-module with the empty dependency relation; and a *static*-module can be considered a dynamically typed module with a single dependency relation. We also define for each module class $\alpha \in \Lambda$ a corresponding composition operator $\|_\alpha$: if $\alpha = \mathit{async}$, then $\|_\alpha = |$; otherwise, $\|_\alpha = \|$.

3 Untyped and Typed Control Problems

Single-step vs. multi-step verification. Given a module M, a state s of M, and a predicate φ over the module variables X_M, the *single-step verification problem* (M, s, φ) asks whether for all states t, if $\tau[\![s, t']\!]$, then $\varphi[\![t]\!]$. The single-step verification problem amounts to evaluating the boolean Π_1^p formula $(\forall X'_M)(\tau_M \rightarrow \varphi')[\![s]\!]$, where φ' results from φ by replacing all variables with their primed counterparts. A *run* r of a module M is a finite sequence $s_0 s_1 \ldots s_k$ of states of M such that $\tau_M[\![s_i, s'_{i+1}]\!]$ for all $0 \leq i < k$. The run r is *s-rooted*, for a state s of M, if $s_0 = s$. The run r *stays in* φ, for a predicate φ over the set X_M of module variables, if $\varphi[\![s_i]\!]$ for all $0 \leq i \leq k$. Given a module M, a state s of M, and a predicate φ over X_M, the *multi-step* (*invariant*) *verification problem* (M, s, φ) asks whether all s-rooted runs of M stay in φ. The multi-step verification problem can be solved by iterating the solution for the single-step verification problem. The number of states, which is exponential, gives a tight bound on the number of iterations.

Theorem 1. (cf. [AH98]) *The single-step verification problem is complete for coNP. The multi-step verification problem is complete for PSPACE.*

In control, it is natural to require that the controller falls into the same module class as the plant. Consider a module class $\alpha \in \Lambda$ and a module $M \in \mathcal{M}_\alpha$. The module $N \in \mathcal{M}_\alpha$ is an *α-controller* for M if (1) M and N are α-composable, and (2) $X_N^o = X_M^i$ and $X_N^i = X_M^o$. According to this definition, a controller for M is an environment of M that has no state on its own. For the control problems we consider in this paper, the results would remain unchanged if we were to consider controllers with state. As in verification, we distinguish between single-step and multi-step control. The single-step (resp. multi-step) control problem asks if there is a controller for a module that ensures that, starting from a given state, a given predicate holds after one step (resp. any number of steps). Precisely, for a module class α, a module $M \in \mathcal{M}_\alpha$, a state s of M, and a predicate φ over the set X_M of module variables, the *single-step* (resp. *multi-step*) *α-control problem* (M, s, φ) asks whether there is an α-controller N for M such that the answer to the single-step (resp. multi-step) verification problem $(M \parallel_\alpha N, s, \varphi)$ is Yes. If the answer is Yes, then the state s is single-step (resp. multi-step) *controllable* by N with respect to the *control objective* φ.

Fixed-type control. For $\alpha \in \{static, dsynt, dsem\}$, we also consider a variant of the control problems in which the type of the controller module is known (but its transiton relation is not). An instance (M, γ, s, φ) of the *fixed-type* single-step (resp. multi-step) α-control problem consists of an instance (M, s, φ) of the single-step (resp. multi-step) α-control problem together with a type γ for the controller. For $\alpha = static$, the type γ is a dependency relation for an α-controller for M; for $\alpha \in \{dsynt, dsem\}$, the type γ is a composite dependency relation for an α-controller for M. The instance (M, γ, s, φ) asks whether there is an α-controller N of type γ for M such that the answer to the single-step (resp. multi-step) verification problem $(M \parallel_\alpha N, s, \varphi)$ is Yes.

Class	Composability Check	Single-Step		Multi-Step	
		Arbitrary	Fixed	Arbitrary	Fixed
async	$\mathcal{O}(n)$	DP	—	EXP	—
moore	$\mathcal{O}(n)$	Σ_2^p	—	EXP	—
static	$\mathcal{O}(n)$	PSPACE	NE	EXP	NE
dsynt	$\mathcal{O}(n^2)$	PSPACE	NE	EXP	NE
dsem	coNP	PSPACE	NE	EXP	NE

(a) Complexity Results.

Class	MSG	MG
async	yes	yes
moore	yes	yes
static	no	no
dsynt	no	no
dsem	yes	no

(b) Existence of controllers.

Table 1. (a) Complexity of composability checking, as well as single-step and multi-step control for the various module classes. For statically and dynamically typed modules, we consider both arbitrary and fixed controller types. The quantity n is the size of the module description. Each problem is complete for the corresponding complexity class. (b) Existence of most state-general (MSG) and most general (MG) controllers.

Generality of controllers. For a module class α, consider a module $M \in \mathcal{M}_\alpha$, a single-step (resp. multi-step) control objective φ, and two α-controllers N and N' for M. The controller N is *as state-general as* N' if all states s of M that are single-step (resp. multi-step) controllable by N' with respect to φ are also single-step (resp. multi-step) controllable by N with respect to φ. Moreover, if N and N' are equally state-general (i.e., N is as state-general as N', and vice versa), then N is *as choice-general as* N' if the transition predicate $\tau_{N'}$ implies τ_N (i.e., N permits as much nondeterminism as N'). An α-controller is *most state-general* if it is as state-general as any other α-controller. A α-controller is *most general* if (1) it is most state-general, and (2) it is as choice-general as any other most state-general α-controller.

Summary of results. In the following section, we present algorithms for solving the various types of control problems. The complexity results are summarized in Table 1(a). We recall that the complexity class DP consists of the languages that are intersections of an NP language and a coNP language. If n is the input size, the complexity class NE is $\bigcup_{k>0} \mathrm{NTIME}(2^{kn})$, and the complexity class EXP is $\bigcup_{k>0} \mathrm{DTIME}(2^{n^k})$. By the padding argument, any problem complete for NE is also complete for NEXP $= \bigcup_{k>0} \mathrm{NTIME}(2^{n^k})$ [Pap94]. Hence, assuming P $\neq$ NP, for the module classes *static*, *dsynt*, and *dsem*, the fixed-type multi-step control problems are harder than the multi-step control problems with arbitrary controller type. In addition, we summarize in Table 1(b) all results on the existence of most state-general and most general controllers.

4 Algorithms and Complexity of Control

We determine the complexity for solving the single-step and multi-step α-control problems for all five module classes $\alpha \in \Lambda$. In each case, the multi-step control problem can be solved by iterating an exponential number of times the solution for the corresponding single-step control problem.

Asynchronous control. Given a non-blocking module M, a state s of M, and a predicate φ over the module variables X_M, the single-step *async*-control problem amounts to evaluating the boolean formula

$$\big((\forall X_M'^o)(\tau_M \wedge (X_M'^i = X_M^i) \rightarrow \varphi') \wedge (\exists X_M'^i)(\tau_M \wedge (X_M'^o = X_M^o) \wedge \varphi')\big)[\![s]\!].$$

Hence, in the asynchronous case, the single-step control problem is complete for DP. It follows from [CKS81] that the multi-step version is complete for exponential time (cf. [HK97]).

Theorem 2. *The single-step async-control problem is complete for DP. The multi-step async-control problem is complete for EXP.*

Proposition 2. *For every non-blocking module and every control objective, there is a most general single-step async-controller, and there is a most general multi-step async-controller.*

Moore control. Given a Moore module M, a state s of M, and a predicate φ over X_M, the single-step *moore*-control problem amounts to evaluating the boolean Σ_2^p formula $(\exists X_M'^i)(\forall X_M'^o)(\tau_M \rightarrow \varphi')[\![s]\!]$. The multi-step hardness proof is similar to the asynchronous case.

Theorem 3. *The single-step moore-control problem is complete for Σ_2^p. The multi-step moore-control problem is complete for EXP.*

Proposition 3. *For every Moore module and every control objective, there is a most general single-step moore-controller, and there is a most general multi-step moore-controller.*

Statically typed control. Consider a statically typed module $(M, \succ_M)$, and let $X_M = \{x_1, \ldots, x_n\}$. A linear order $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ of the variables in X_M is *compatible* with the dependency relation $\succ_M$ if each output variable follows in the ordering the variables on which it depends. Precisely, $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ is compatible with $\succ_M$ if for all $1 \leq j, k \leq n$, if $x_{i_j} \succ x_{i_k}$, then $k < j$. Given a predicate φ over X_M, for each linear order $\ell = x_{i_1}, x_{i_2}, \ldots, x_{i_n}$, we define the boolean formula $C(\ell, \varphi) = (\lambda_{i_1} x'_{i_1})(\lambda_{i_2} x'_{i_2}) \cdots (\lambda_{i_n} x'_{i_n})(\tau_M \rightarrow \varphi')$, where for $1 \leq k \leq n$, we have $\lambda_{i_k} = \forall$ if $x_{i_k} \in X_M^o$, and $\lambda_{i_k} = \exists$ if $x_{i_k} \in X_M^i$. The following lemma states that, in order to decide whether a state is single-step *static*-controllable, it suffices to consider all linear orders of variable dependencies.

Lemma 1. *Given a statically typed module $(M, \succ_M)$, a control objective φ over X_M, and a state s of M, the state s is single-step static-controllable with respect to φ iff there is a linear order ℓ of X_M compatible with $\succ_M$ such that $C(\ell, \varphi)[\![s]\!]$.*

Theorem 4. *The single-step static-control problem is complete for PSPACE. The multi-step static-control problem is complete for EXP.*

The single-step *static*-control problem is in PSPACE, because we can check each linear order in PSPACE. Hardness for PSPACE follows from the fact that, given a boolean formula $(\forall x_0)(\exists y_0) \cdots (\forall x_n)(\exists y_n)\varphi$, we can encode the problem of

deciding its truth value as the *static*-control problem with the control objective φ for a module M with the variables $X^o_M = \{x_0, \ldots, x_n\}$ and $X^i_M = \{y_0, \ldots, y_n\}$, the valid transition relation τ_M, and the dependency relation $\succ_M = \{(x_i, y_j) \mid 1 \leq j < i \leq n\}$. The corresponding multi-step problem is again complete for EXP. Note that controllability by a Moore module corresponds to the special case in which every output variable depends on all input variables; that is, $\succ_M = X^o_M \times X^i_M$. The dual case is that on an empty dependency relation $\succ_M = \emptyset$. Here, the controller can choose the next values of the input variables dependent on the next values of all output variables, and the single-step control problem amounts to evaluating the boolean Π^p_2 formula $(\forall X'^o_M)(\exists X'^i_M)(\tau_M \wedge \varphi')[\![s]\!]$.

We consider now the case in which the type $\succ_N \subseteq X^i_M \times X_M$ of the controller is fixed. We assume that $\succ_M \cup \succ_N$ is acyclic; otherwise, M and N are not *static*-composable, and the answer to the fixed-type control problems is No. Let $X^o_M = \{x_1, \ldots, x_m\}$ and $X^i_M = \{y_1, \ldots, y_k\}$. Intuitively, for $1 \leq i \leq k$, the next value for y_i can be chosen in terms of the current values of the module variables, as well as in terms of the next values of the output variables on which y_i depends. Hence, a controller with fixed static type $\succ_N$ can be thought of as a set $\{f_1, \ldots, f_k\}$ of Skolem functions: for $1 \leq i \leq k$, the Skolem function f_i provides a next value for y_i, and has as arguments the variables in $X_M \cup \{x' \in X'^o_M \mid y_i \succ_N x\}$. This set of Skolem functions corresponds to the following boolean formula with Henkin quantifiers [Hen61,Wal70]:

$$H(\succ_N, \varphi) \;=\; \begin{pmatrix} (\forall \{x' \in X'^o_M \mid y_1 \succ_N x\})(\exists y'_1) \\ \cdots \\ (\forall \{x' \in X'^o_M \mid y_k \succ_N x\})(\exists y'_n) \end{pmatrix} (\tau_M \rightarrow \varphi').$$

The fixed-type single-step *static*-control problem can be solved as follows.

Lemma 2. *Given a statically typed module $(M, \succ_M)$, a static controller type $\succ_N \subseteq X^i_M \times X_M$ that is static-composable with $\succ_M$, a control objective φ over X_M, and a state s of M, the state s is single-step static-controllable with respect to φ by a controller with static type $\succ_N$ iff $H(\succ_N, \varphi)[\![s]\!]$.*

Deciding the truth value of a boolean formula with Henkin quantifiers is complete for NE, even if the formula has the restricted form shown above [GLV95].

Theorem 5. *The fixed-type single-step and multi-step static-control problems are complete for NE.*

Unlike Moore modules, a statically typed module may not have a most state-general controller.

Proposition 4. *There is a statically typed module and a control objective such that there is no most state-general single-step static-controller, nor a most state-general multi-step static-controller.*

Example 5. Let module M have the output variables $X^o_M = \{x_0, x_1, z\}$, the input variables $X^i_M = \{y_0, y_1\}$, the transition predicate $\tau_M = (z' = z)$, and the static type $\succ_M = \{x_0 \succ y_0,\ x_1 \succ y_1\}$. The control objective is $\varphi = (z \wedge (y_1 =$

$x_0)) \vee (\neg z \wedge (y_0 = x_1))$. For every state s of M there is a controller N such that s is *static*-controllable by N with respect to φ: if $z[\![s]\!]$, then N has the transition predicate $\tau_N = (y_1' = x_0')$ and the static type $\succ_N = \{y_1 \succ x_0\}$; if $\neg z[\![s]\!]$, then $\tau_N = (y_0' = x_1')$ and $y_0 \succ_N x_1$. However, because of the acyclicity requirement for dependency relations, there is no single *static*-controller that controls all states of M. For the same reason, M also does not have a most state-general multi-step *static*-controller for the control objective φ. ∎

Dynamically typed control. The solution of control problems for dynamically typed modules closely parallels the solution for statically typed modules.

Lemma 3. *Given a dynamically typed module $(M, \{(\psi_M^i, \succ_M^i) \mid 1 \leq i \leq m\})$, a control objective φ over X_M, and a state s of M, the following assertions hold:*

- *The state s is single-step dsynt-controllable with respect to φ iff there is a linear order ℓ of X_M that is compatible with all $\succ_M^i$, for $1 \leq i \leq m$, such that $C(\ell, \varphi)[\![s]\!]$.*
- *The state s is single-step dsem-controllable with respect to φ iff there is a linear order ℓ of X_M that is compatible with $\succ_M^s$, such that $C(\ell, \varphi)[\![s]\!]$.*

Theorem 6. *For $\alpha \in \{dsynt, dsem\}$, the single-step α-control problem is complete for PSPACE, and the multi-step α-control problem is complete for EXP.*

Hence, the control problems for statically and dynamically typed modules have the same complexity. This applies also to the fixed-type control problems.

Lemma 4. *Given a dynamically typed module (M, D_M), a module class $\alpha \in \{dsynt, dsem\}$, a dynamic controller type $D_N = \{(\psi_N^i, \succ_N^i) \mid 1 \leq i \leq m\}$ that is α-composable with D_M, a control objective φ over X_M, and a state s of M, the state s is single-step α-controllable with respect to φ by a controller with dynamic type D_N iff $H(\succ_N^s, \varphi)[\![s]\!]$.*

Theorem 7. *For $\alpha \in \{dsynt, dsem\}$, the fixed-type single-step and multi-step α-control problems are complete for NE.*

Dynamically typed modules with syntactic composition do not necessarily have a most state-general controller. In contrast, dynamically typed modules with semantic composition always have a most state-general controller, but they may not have a most general one.

Proposition 5. *(1) There is a dynamically typed module and a control objective such that there is no most state-general single-step dsynt-controller, nor a most state-general multi-step dsynt-controller. (2) For every dynamically typed module and every control objective, there is a most state-general single-step dsem-controller, and there is a most state-general multi-step dsem-controller. (3) There is a dynamically typed module and a control objective such that there is no most general single-step dsem-controller, nor a most general multi-step dsynt-controller.*

Example 6. The module M of Example 5 can be viewed as a dynamically typed module whose dependency relation is the same for every state. The control objective is $\varphi = ((y_1 = x_0) \vee (y_0 = x_1))$. There exist at least two single-step *dsem*-controllers that control every state of M: the first controller N_1 has the transition predicate $\tau_{N_1} = (y'_1 = x'_0)$; the second controller N_2 has the transition predicate $\tau_{N_2} = (y'_0 = x'_1)$. However, there is no most general single-step *dsem*-controller. To control M with respect to φ in a most general way, a controller N with the transition predicate $\tau_N = ((y'_0 = x'_1) \vee (y'_1 = x'_0))$ would be required. Such a controller can be typed only if dynamic types are generalized to admit disjunctions of composite dependency relations. ∎

The relative power of controllers. Recall the module class ordering $\mathit{async} < \mathit{moore} < \mathit{static} < \mathit{dsynt} < \mathit{dsem}$. The following proposition establishes that this ordering strictly orders the power of controllers.

Proposition 6. *For all module classes $\alpha, \beta \in \Lambda$ with $\alpha < \beta$, there is a module $M \in \mathcal{M}_\alpha$, a control objective φ over X_M, and a state s of M, such that s is not single-step (resp. multi-step) α-controllable with respect to φ, but s is single-step (resp. multi-step) β-controllable with respect to φ.*

Discussion. One may be inclined to define the following "unrestricted synchronous control problem": given a non-blocking module M, a state s of M, and a predicate φ over X_M, is there a module N composable with M such that (1) the synchronous composition $M \| N$ is non-blocking, and (2) the answer to the single-step (resp. multi-step) verification problem $(M \| N, s, \varphi)$ is Yes? This formulation does not distinguish between output and input variables, and thus permits the controller N to arbitrarily constrain the output variables of M, as long as the compound system $M \| N$ is non-blocking. Thus, the "unrestricted synchronous control problem" is not a control problem at all in the traditional sense, because it simply asks for the existence of a transition (in the single-step case) or run (in the multi-step case). The single-step solution amounts to evaluating the boolean Σ^p_1 formula $(\exists X'_M)(\tau_M \wedge \varphi')[\![s]\!]$, and like invariant verification, the multi-step problem is complete for PSPACE. Note that if the non-blocking requirement (1) is also dropped, then the appropriate single-step formula is $(\exists X'_M)(\tau_M \rightarrow \varphi')[\![s]\!]$, which permits the controller to block the progress of M.

To introduce a semantic (i.e., non-type-based) distinction between output and input variables, one may define the condition of input universality, that states that a module should not constrain its inputs. Formally, a module M is *input universal* if the Π^p_2 formula $(\forall X_M)(\forall X'^i_M)(\exists X'^o_M)\tau_M$ is true. Input universality is a reasonable requirement, which, in particular, is satisfied by all typed modules. Input universality by itself, however, is not compositional: the composition of two input-universal modules is not necessarily input universal, and may be blocking (cf. Example 1). This is because input universality does not treat the module and its environment symmetrically: it states that a module, when composed with a Moore environment, will not block —but the module itself is not required to be Moore. This mismatch is reflected in the "input-universal control problem," that asks, given an input-universal module M, a state s of M, and a

predicate φ over X_M, if there is an input-universal module N composable with M such that (1) the synchronous composition $M \| N$ is non-blocking, and (2) the answer to the single-step (resp. multi-step) verification problem $(M \| N, s, \varphi)$ is Yes. The single-step solution amounts to evaluating the boolean Π_2^p formula $((\forall X_M'^o)(\exists X_M'^i)(\tau_M \rightarrow \varphi') \wedge (\exists X_M'^o)(\exists X_M'^i)(\tau_M \wedge \varphi'))[\![s]\!]$. The two conjuncts of this formula imply different powers for the controller. The first conjunct states that the controller can look at the next values of the output variables, and propose new values for the input variables that either cause blocking, or achieve control (hence the controller is input universal). The second conjuct gives the controller the additional power of "guessing" the next values of the output variables in order to ensure non-blocking of the compound system. Types solve this mismatch by providing a stronger, compositional condition than input universality, which treats the module and the controller symmetrically.

References

[AH98] R. Alur and T.A. Henzinger. *Computer-aided Verification: An Introduction to Model Building and Model Checking for Concurrent Systems.* Draft, 1998.

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

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Programming Languages and Systems*, 17:507–534, 1995.

[BG86] A. Blass and Y. Gurevich. Henkin quantifiers and complete problems. *Ann. Pure and Applied Logic*, 32:1–16, 1986.

[BG88] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. Technical Report 842, INRIA, 1988.

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

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *J. ACM*, 28:114–133, 1981.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus, and determinacy. In *Proc. Symp. on Foundations of Computer Science*, pp. 368–377. IEEE Press, 1991.

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

[GLV95] G. Gottlob, N. Leone, and H. Veith. Second-order logic and the weak exponential hierarchies. In *Mathematical Foundations of Compuer Science*, LNCS 969, pp. 66–81. Springer-Verlag, 1995.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

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

[HK97] T.A. Henzinger and P.W. Kopke. Discrete-time control for rectangular hybrid automata. In *Automata, Languages, and Programming*, LNCS 1256, pp. 582–593. Springer-Verlag, 1997.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In *Computer-aided Verification*, LNCS 1254, pp. 24–35. Springer-Verlag, 1997.

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

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[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.

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

[TW94] J.G. Thistle and W.M. Wonham. Control of infinite behavior of finite automata. *SIAM J. Control and Optimization*, 32:1075–1097, 1994.

[Wal70] W. Walkoe. Finite partially-ordered quantification. *J. Symbolic Logic*, 35:535–555, 1970.