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 $M \parallel N$ 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 hardness of its solution. (The latter distinguishes control from verification, whose complexity —$\text{PSPACE}$ 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 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 $s : X \to \mathbb{B}$ to the variables in $X$. We write $s'$ for the truth-value assignment $s' : X' \to \mathbb{B}$ 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 $\varphi$ over the variables in $X$, we write $\varphi[s]$ for the truth value of $\varphi$ when the variables in $X$ are interpreted according to $s$. Given a predicate $\tau$ over the variables in $X \cup X'$, and states $s,t$ over $X$, we write $\tau[s,t]$ for the truth value of $\tau$ 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 $X_M^c$ of controlled variables. These variables are updated by the module.
- A finite set $X_M^e$ of external variables. These variables are updated by the environment. The sets $X_M^c$ and $X_M^e$ must be disjoint. We write $X_M = X_M^c \cup X_M^e$ for the set of all module variables. The states of $M$ are the truth-value assignments to the variables in $X_M$.
- A predicate $\tau_M$ over the set $X_M \cup X'_M$ of unprimed and primed variables. The predicate $\tau_M$, called transition predicate, relates the current (unprimed) and next (primed) values of the module variables.

Two modules $M$ and $N$ are composable if their controlled variables $X_M^c$ and $X_N^c$ are disjoint. Given two composable modules $M$ and $N$, the synchronous (lock-step) composition $M\|N$ is the module with the components $X_M^c \| N = X_M^c \cup X_N^c$, $X_M^e \| N = (X_M^e \cup X_N^e) \setminus X_M^c \| N$, and $\tau_M \| N = (\tau_M \land \tau_N)$. The asynchronous (interleaving) composition $M[N$ is the module with the same controlled and external variables as $M[N$, but with the transition predicate $\tau_M[N = ((\tau_M \land (X_M^c = X_N^c)) \lor (\tau_N \land (X_M^e = X_N^e)))$.
Non-blocking modules. We are interested in the non-blocking modules, a condition which is 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 $X_M^c \subseteq \{x\}$, $X_M^e \subseteq \{y\}$, and $\tau_M = ((y'/ y' \wedge \neg x') \lor \neg y' \wedge \neg x')$. Let module $N$ be such that $X_N^c \subseteq \{y\}$, $X_N^e \subseteq \{x\}$, and $\tau_N = ((x' \wedge y') \lor \neg x' \wedge \neg y')$. 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 $P^2$ 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 which (a) is non-blocking, and (b) the next values of controlled variables do not depend on the next values of external variables; that is, for all states $s$, $t$, and $u$, if $\tau_M[s, t']$ and $\tau_M[s, u']$, 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 may be specified as a set of nondeterministic guarded commands, one for each primed controlled variable $x'$ in $X_M^c$. 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 $\triangleright \subseteq X_M^c \times X_M^c$ between the controlled variables and the module variables (acyclicity means that the transitive closure is irreflexive). The module $M$ respects the dependency relation $\triangleright$ at state $s$ if, for all states $t$ with $\tau_M[s, t']$, for each subset $Y^e \subseteq X_M^e$ of external variables, and for each truth-value assignment $u^c$ to the variables in $Y^e$, there is a state $u$ with $\tau_M[s, u']$ such that $u[Y^e] = u^c$, and $u[Z] = \tau[Z]$ for $Z = \{z \in X_M \mid (\text{not } z \triangleright^* y) \text{ for all } y \in Y^e\}$, where $\triangleright^*$ is the reflexive-transitive closure of $\triangleright$. A statically typed module $(M, \triangleright_M)$ 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 $\triangleright_M$ 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 may 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 \( x \triangleright_M y \). We refer to the dependency relation \( \triangleright_M \) of a statically typed module \((M, \triangleright_M)\) as a static type for the module \( M \). Note that if \( \triangleright' \) is a dependency relation for \( M \), and \( \triangleright_M \) is a subset of \( \triangleright' \), then \( \triangleright' \) is also a static type for \( M \).

Every non-blocking module has a static type (have each controlled variable depend on all external 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 \((M, \triangleright_M)\) and \((N, \triangleright_N)\) are statically composable, or static-composable, if (1) the modules \( M \) and \( N \) are composable, and (2) the relation \( \triangleright_M|N \) is acyclic. Then, the relation \( \triangleright_M|N \) 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 \((M, \triangleright_M)\) and \((N, \triangleright_N)\) 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 \( X^e_M = \{x_0, x_1\} \), \( X^f_M = \{y\} \), and \( \tau_M = (x_0' \oplus x_1' \oplus y'_0) \). Using guarded commands, we can specify \( M \) in two ways:

\[
M' = \begin{cases}
\|T \rightarrow x_0' := \neg(x_1' \oplus y') \\
\|T \rightarrow x_1' := T \\
\|T \rightarrow x_1' := F
\end{cases} \quad M'' = \begin{cases}
\|T \rightarrow x_0' := T \\
\|T \rightarrow x_0' := F \\
\|T \rightarrow x_1' := \neg(x_0' \oplus y')
\end{cases}
\]

Note that both \( M' \) and \( M'' \) have the same transition predicate, namely \( \tau_M \), but they have different static types: the static type \( \triangleright_M \) for \( M' \) is \( \{x_0 \triangleright x_1, x_0 \triangleright y\} \), while \( \triangleright_M'' = \{x_1 \triangleright x_0, x_1 \triangleright y\} \). Choosing different static types (i.e., implementations of the transition predicate) can have implications on compositability with other modules. Let module \( N \) be such that \( X^e_N = \{y\} \), \( X^f_N = \{x_0, x_1\} \), and \( \tau_N = (y' = x_0') \) (or, using guarded commands, \( \|T \rightarrow y' := x_0' \)). The static type \( \triangleright_N \) for \( N \) is \( y \triangleright x_0 \). Then \( (M', \triangleright_M) \) is not static-composable with \( (N, \triangleright_N) \), but \((M'', \triangleright_M'')\) is.

**Dynamically typed modules.** Example 2 suggests the following generalization of static types. A composite dependency relation for a module \( M \) is a set \( D = \{(\psi^1, \triangleright^1), \ldots, (\psi^m, \triangleright^m)\} \) of pairs, where each \( \psi^i \) is a predicate over the module variables \( X_M \), and each \( \triangleright^i \) is a dependency relation for \( M \), such that for each state \( s \) of \( M \), there is exactly one predicate \( \psi^i \), \( 1 \leq i \leq m \), with \( \psi^i[s] \). If \( \psi^i[s] \), then we write \( \triangleright^s \) for the corresponding dependency relation \( \triangleright^i \). A dynamically typed module \((M, D_M)\) consists of a module \( M \) and a composite dependency relation \( D_M = \{(\psi^i_M, \triangleright^i_M) \mid 1 \leq i \leq m\} \), such that (a) the module \( M \) is non-blocking, and (b) at every state \( s \), the module \( M \) respects the dependency
relation \( \succ_i^M \). 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 \( \psi^i_M, 1 \leq i < m \), may be required to contain the conjunct \( \bigwedge_{j \neq i} \neg \psi^j_M \), and \( \psi^m_M \) may 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 may be required to contain a conjunct of the form \( \psi^i_M \), 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 which connects the output of the \( \neg \) clocked level-sensitive latch \( M_3 \) to the input of the \( e \) 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:

\[
M_1 = \{ \{ T \to x' := \neg z' \} \} \\
M_2 = \{ \{ c \to y' := x' \} \} \\
M_3 = \{ \{ c \to z' := z \} \}
\]

The dynamic types for the modules are \( D_{M_1} = \{ (\top, x \succ z) \} \), \( D_{M_2} = \{ (c, y \succ x), (\neg c, \emptyset) \} \), 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 \( D_M = \{ (\psi^i_M, \succ^i_M) \mid 1 \leq i < m \} \) 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 **dsgnt-composable**, if (1) the modules \( M \) and \( N \) are composable, and (2) the relation \( \succ^{i,j} \) is acyclic for all \( 1 \leq i < m \) and \( 1 \leq j \leq n \). Then, \( D_{M||N} = \{ (\psi^i_M \land \theta^j_N, \succ^{i,j}) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n \} \), 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 \( 1 \leq i < m \) and \( 1 \leq j \leq n \) for which the conjunction \( \psi^i_M \land \theta^j_N \) is satisfiable. Then, \( D_{M||N} = \{ (\psi^i_M \land \theta^j_N, \succ^{i,j}) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n \text{ and } (\exists X_{M||N}) (\psi^i_M \land \theta^j_N) \} \) is a dynamic type for \( M||N \).
Note that it can be checked in quadratic time whether two dynamically typed modules are $dsgnt$-composable, while it requires exponential time (by evaluating a quadratic number of boolean $\Pi^0_1$ 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 ($\Pi^0_1$ vs. $\Pi^0_2$).

**Proposition 1.** The following assertions hold:

- There are two dynamically typed modules $(M, D_M)$ and $(N, D_N)$ which are $dsgnt$-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 $N$.
- There are two dynamically typed modules which are $dsem$-composable but not $dsgnt$-composable.
- There are two dynamically typed modules $(M, D_M)$ and $(N, D_N)$ which are not $dsem$-composable, even though the synchronous composition $M \parallel N$ is non-blocking.

**Example 4.** The dynamically typed modules $(M_1, D_{M_1})$ and $(M_2, D_{M_2})$ of Example 3 are $dsgnt$-composable, and the compound module $N = M_1 \parallel M_2$ has the dynamic type $D_N = \{(c, y \rightarrow x \rightarrow z), (\neg c, x \rightarrow z)\}$. The modules $(N, D_N)$ and $(M_3, D_{M_3})$ are not $dsgnt$-composable, but they are $dsem$-composable. The compound module $N \parallel M_3$ has the dynamic type $\{(c, y \rightarrow x \rightarrow z), (\neg c, x \rightarrow z \rightarrow y)\}$. 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 $\{y \rightarrow x \rightarrow z, y \rightarrow c\}$ for every state, and that for $M_3$ becomes $\{z \rightarrow y, z \rightarrow c\}$ for every state. Then $(N, D_N)$ and $(M_3, D_{M_3})$ are not $dsem$-composable, even though the synchronous composition $N \parallel M_3$ is non-blocking.

**Summary.** Let $\Lambda = \{\text{async, moore, static, dsgnt, dsem}\}$ 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 $\alpha = \text{async}$, let $\mathcal{M}_{\text{async}}$ be the set of non-blocking modules; for $\alpha = \text{moore}$, let $\mathcal{M}_{\text{moore}}$ be the set of Moore modules; for $\alpha = \text{static}$, let $\mathcal{M}_{\text{static}}$ be the set of statically typed modules; and for $\alpha = \text{dsgnt}$ and $\alpha = \text{dsem}$, let $\mathcal{M}_{\text{dsgnt}} = \mathcal{M}_{\text{dsem}}$ be the set of dynamically typed modules. Define the module class ordering $\text{async} < \text{moore} < \text{static} < \text{dsgnt} < \text{dsem}$. 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 $\parallel_\alpha$: if $\alpha = \text{async}$, then $\parallel_\alpha = \parallel$; otherwise, $\parallel_\alpha = \parallel$. 

8
3 Untyped and Typed Control Problems

**Single-step vs. multi-step verification.** Given a module $M$, a state $s$ of $M$, and a predicate $\varphi$ over the module variables $X_M$, the single-step verification problem $(M, s, \varphi)$ asks whether for all states $t$, if $\tau[s, t']$, then $\varphi[t]$. The single-step verification problem amounts to evaluating the boolean $\Pi$ formula $(\forall X'_M)(\tau_M \rightarrow \varphi'[s])$, where $\varphi'$ results from $\varphi$ 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 $\varphi$, for a predicate $\varphi$ over the set $X_M$ of module variables, if $\varphi[s_i]$ for all $0 \leq i < k$. Given a module $M$, a state $s$ of $M$, and a predicate $\varphi$ over $X_M$, the multi-step (invariant) verification problem $(M, s, \varphi)$ asks whether all $s$-rooted runs of $M$ stay in $\varphi$. 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 $\text{coNP}$. The multi-step verification problem is complete for $\text{PSPACE}$.

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 $\alpha$-controller for $M$ if (1) $M$ and $N$ are $\alpha$-composable, and (2) $X'_N = X'_M$ and $X''_N = X''_M$. According to this definition, a controller for $M$ is an environment of $M$ which 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 $\alpha$, a module $M \in \mathcal{M}_\alpha$, a state $s$ of $M$, and a predicate $\varphi$ over the set $X_M$ of module variables, the single-step (resp. multi-step) $\alpha$-control problem $(M, s, \varphi)$ asks whether there is an $\alpha$-controller $N$ for $M$ such that the answer to the single-step (resp. multi-step) verification problem $(M ||_\alpha N, s, \varphi)$ 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 $\varphi$.

**Fixed-type control.** For $\alpha \in \{\text{static, dysnt, dsem}\}$, we also consider a variant of the control problems in which the type of the controller module is known (but its transition relation is not). An instance $(M, \gamma, s, \varphi)$ of the fixed-type single-step (resp. multi-step) $\alpha$-control problem consists of an instance $(M, s, \varphi)$ of the single-step (resp. multi-step) $\alpha$-control problem together with a type $\gamma$ for the controller. For $\alpha = \text{static}$, the type $\gamma$ is a dependency relation for an $\alpha$-controller for $M$; for $\alpha \in \{\text{dysnt, dsem}\}$, the type $\gamma$ is a composition dependency relation for an $\alpha$-controller for $M$. The instance $(M, \gamma, s, \varphi)$ asks whether there is an $\alpha$-controller $N$ of type $\gamma$ for $M$ such that the answer to the single-step (resp. multi-step) verification problem $(M ||_\alpha N, s, \varphi)$ is Yes.
\begin{table}[ht]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Class & Composability Check & Single-Step Arbitrary & Multi-Step Arbitrary & Fixed & Fixed \\
\hline
async & $O(n)$ & DP & — & EXP & — \\
moore & $O(n)$ & $Σ_2^p$ & — & EXP & — \\
static & $O(n)$ & $\text{PSPACE}$ & NE & EXP & NE \\
dsyst & $O(n^2)$ & $\text{PSPACE}$ & NE & EXP & NE \\
dsem & coNP & $\text{PSPACE}$ & NE & EXP & NE \\
\hline
\end{tabular}
\caption{(a) Complexity Results.}
\end{table}

\begin{table}[ht]
\centering
\begin{tabular}{|c|c|c|}
\hline
Class & MSG & MG \\
\hline
async & yes & yes \\
moore & yes & yes \\
static & no & no \\
dsys & no & no \\
dsem & yes & no \\
\hline
\end{tabular}
\caption{(b) Existence of controllers.}
\end{table}

**Generality of controllers.** For a module class $\alpha$, consider a module $M \in \mathcal{M}_\alpha$, a single-step (resp. multi-step) control objective $\varphi$, and two $\alpha$-controllers $N$ and $N'$ for $M$. The controller $N$ is \textit{as state-general as} $N'$ if all states $s$ of $M$ which are single-step (resp. multi-step) controllable by $N'$ with respect to $\varphi$ are also single-step (resp. multi-step) controllable by $N$ with respect to $\varphi$. Moreover, if $N$ and $N'$ are equally state-general (i.e., $N$ is as state-general as $N'$, and vice versa), then $N$ is \textit{as choice-general as} $N'$ if the transition predicate $\tau_N$ implies $\tau_{N'}$ (i.e., $N$ permits as much nondeterminism as $N'$). An $\alpha$-controller is \textit{most state-general} if it is as state-general as any other $\alpha$-controller. A $\alpha$-controller is \textit{most general} if (1) it is most state-general, and (2) it is as choice-general as any other most state-general $\alpha$-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 which are intersections of an NP language and a coNP language. If $n$ is the input size, the complexity class NE is $\bigcup_{k>0} \text{NTIME}(2^{kn})$, and the complexity class EXP is $\bigcup_{k>0} \text{DTIME}(2^{n^k})$. By the padding argument, any problem complete for NE is also complete for NEXP = $\bigcup_{k>0} \text{NTIME}(2^{n^k})$ [Pap94]. Hence, assuming P $\neq$ NP, for the module classes \textit{static}, \textit{dsynt}, and \textit{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 $\alpha$-control problems for all five module classes $\alpha \in \mathcal{A}$. 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 $\varphi$ over the module variables $X_M$, the single-step *async*-control problem amounts to evaluating the boolean formula

$$\left( (\forall X_M')(\tau_M \land (X_M' M = X_M) \rightarrow \varphi') \land (\exists X_M' \land (X_M' M = X_M') \land \varphi') \right)[s].$$

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 $\varphi$ over $X_M$, the single-step *moore*-control problem amounts to evaluating the boolean $\Sigma_2^p$ formula $(\exists X_M')(\forall X_M')(\tau_M \rightarrow \varphi')[s]$. The multi-step hardness proof is similar to the asynchronous case.

**Theorem 3.** The single-step *moore*-control problem is complete for $\Sigma_2^p$. 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 controlled 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 $\varphi$ over $X_M$, for each linear order $\ell = x_{i_1}, x_{i_2}, \ldots, x_{i_n}$, we define the boolean formula

$$C(\ell, \varphi) = (\lambda_{i_1} x_{i_1}' \lambda_{i_2} x_{i_2}' \ldots (\lambda_{i_n} x_{i_n}')(\tau_M \rightarrow \varphi'),$$

where for $1 \leq k \leq n$, we have $\lambda_{i_k} = \forall$ if $x_{i_k} \in X_M$, and $\lambda_{i_k} = \exists$ if $x_{i_k} \in X_M$. 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 $\varphi$ over $X_M$, and a state $s$ of $M$, the state $s$ is single-step static-controllable with respect to $\varphi$ iff there is a linear order $\ell$ of $X_M$ compatible with $\succ_M$ such that $C(\ell, \varphi)[s]$. 

11
The lemma is proved by showing that (1) if a state of the statically typed module
\((M, \triangleright_M)\) is controlled by a statically typed module \((N, \triangleright_N)\), then there is a
linear order \(\ell\) that strengthens the transitive closure of \(\triangleright_M \cup \triangleright_N\) such that
\(C(\ell, \varphi)\), and (2) if \(C(\ell, \varphi)\) for some linear order \(\ell\), then we can extract from \(\ell\) a
dependency relation \(\triangleright_N\) for the controller which ensures controllability.

**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 \((\forall x_0)(\exists y_0) \cdot \ldots \cdot (\forall x_n)(\exists y_n)\varphi\), we can encode the problem of
deciding its truth value as the static-control problem with the control objective
\(\varphi\) for a module \(M\) with the variables \(X_M = \{x_0, \ldots, x_n\}\) and \(X_M = \{y_0, \ldots, y_n\}\),
the valid transition relation \(\tau_M\), and the dependency relation \(\triangleright_M = \{(x_i, y_j) \mid 1 \leq j < i \leq n\}\).

Note that in the common special case that the dependency relation \(\triangleright_M\) is empty, the single-step static-control problem amounts to evaluating the Boolean
formula \((\forall X_M^\varnothing)(\exists X_M^\varnothing)(\tau_M \land \varphi)'[s]\). This case is dual to the Moore case,
because here the controller can choose the next values of the external variables
dependent on the next values of all controlled variables. The corresponding multi-
step problem is again complete for EXP.

We consider now the case in which the type \(\triangleright_N \subseteq X_M^\varnothing \times X_M\) of the controller
is fixed. We assume that \(\triangleright_M \cup \triangleright_N\) is acyclic; otherwise, \(M\) and \(N\) are not static-
composable, and the answer to the fixed-type control problems is No. Let \(X_M^\varnothing = \{x_1, \ldots, x_m\}\) and \(X_M^\varnothing = \{y_1, \ldots, y_k\}\). 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 value of the controlled variables on which \(y_i\) depends. Hence, a controller with fixed static type \(\triangleright_N\) 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 \(X_M \cup \{x' \in X_M^\varnothing \mid y_i \triangleright_N x\}\). This set of Skolem functions corresponds to the following Boolean formula with Henkin quantifiers [Hen61, Wal70]:

\[
H(\triangleright_N, \varphi) = \left( (\forall \{x' \in X_M^\varnothing \mid y_1 \triangleright_N x\}) (\exists y_1') \ldots \right) (\forall \{x' \in X_M^\varnothing \mid y_k \triangleright_N x\}) (\exists y_k') (\tau_M \rightarrow \varphi').
\]

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

**Lemma 2.** Given a statically typed module \((M, \triangleright_M)\), a static controller type
\(\triangleright_N \subseteq X_M^\varnothing \times X_M\) which is static-composable with \(\triangleright_M\), a control objective \(\varphi\)
over \(X_M\), and a state \(s\) of \(M\), the state \(s\) is single-step static-controllable with
respect to \(\varphi\) by a controller with static type \(\triangleright_N\) iff \(H(\triangleright_N, \varphi)[s]\).

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 controlled variables $X_{M}^{c} = \{x_{0}, x_{1}, z\}$, the external variables $X_{M}^{e} = \{y_{0}, y_{1}\}$, the transition predicate $\tau_{M} = (z' = z)$, and the static type $\vdash_{M} = \{x_{0} \vdash y_{0}, x_{1} \vdash y_{1}\}$. 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 $\varphi$; if $z[s]$, then $N$ has the transition predicate $\tau_{N} = (y_{1}' = x_{0}')$ and the static type $\vdash_{N} = \{y_{1} \vdash x_{0}\}$; if $\neg z[s]$, then $\tau_{N} = (y_{0}' = x_{1}')$ and $y_{0} \vdash_{N} x_{1}$. 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 $\varphi$.

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 $(M, \{(\psi_{M}^{i}, \vdash_{M}^{i}) | 1 \leq i \leq m\})$, a control objective $\varphi$ over $X_{M}$, and a state $s$ of $M$, the following assertions hold:

- the state $s$ is single-step dsym-controllable with respect to $\varphi$ iff there is a linear order $\ell$ of $X_{M}$ which is compatible with $\vdash_{M}$, for $1 \leq i \leq m$, such that $C(\ell, \varphi)[s]$.
- The state $s$ is single-step dsem-controllable with respect to $\varphi$ iff there is a linear order $\ell$ of $X_{M}$ which is compatible with $\vdash_{M}$, such that $C(\ell, \varphi)[s]$.

Theorem 6. For $\alpha \in \{dsym, dsem\}$, the single-step $\alpha$-control problem is complete for $PSPACE$, and the multi-step $\alpha$-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 $\alpha \in \{dsym, dsem\}$, a dynamic controller type $D_{N} = \{(\psi_{N}^{i}, \vdash_{N}^{i}) | 1 \leq i \leq m\}$ which is $\alpha$-composable with $D_{M}$, a control objective $\varphi$ over $X_{M}$, and a state $s$ of $M$, the state $s$ is single-step $\alpha$-controllable with respect to $\varphi$ by a controller with dynamic type $D_{N}$ iff $H(\vdash_{N}, \varphi)[s]$.

Theorem 7. For $\alpha \in \{dsym, dsem\}$, the fixed-type single-step and multi-step $\alpha$-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. The following assertions hold:

- There is a dynamically typed module and a control objective such that there is no most state-general single-step dsyn-t-controller, nor a most state-general multi-step dsyn-controller.
- 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. However, 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 dsyn-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 $\varphi = ((y_1 = x_0) \lor (y_0 = x_1))$. There exist at least two single-step dsem-controllers which 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 $\varphi$ in a most general way, a controller $N$ with the transition predicate $\tau_N = ((y'_0 = x'_1) \lor (y'_1 = x'_0))$ would be required. Such a controller can be typed only if dynamic types are generalized to admit disjunctions of composite dependency relations.

Unrestricted control. 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 $\varphi$ 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, however, makes no distinction between controlled and external variables, and thus permits the controller $N$ to arbitrarily constrain the controlled 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 $\Sigma^2_1$ formula $(\exists X'_M)(\tau_M \land \varphi)[s]$, 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 $M$.

The relative power of controllers. Recall the module class ordering $async < moore < static < dsyn < dsem$. 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 $\varphi$ over $X_M$, and a state $s$ of $M$, such that $s$ is not single-step (resp. multi-step) $\alpha$-controllable with respect to $\varphi$, but $s$ is single-step (resp. multi-step) $\beta$-controllable with respect to $\varphi$.\[14\]
References