Pushdown Module Checking with Imperfect Information ⋆

Benjamin Aminof[1], Aniello Murano[2], and Moshe Y. Vardi[3]

[1] Hebrew University, Jerusalem 91904, Israel.
[2] Università degli Studi di Napoli "Federico II", 80126 Napoli, Italy.
[3] Rice University, Houston, TX 77251-1892, U.S.A.

Abstract. The model checking problem for finite-state open systems (*module checking*) has been extensively studied in the literature, both in the context of environments with perfect and imperfect information about the system. Recently, the perfect information case has been extended to infinite-state systems (*pushdown module checking*). In this paper, we extend pushdown module checking to the imperfect information setting; i.e., the environment has only a partial view of the system's control states and pushdown store content. We study the complexity of this problem with respect to the branching-time temporal logic *CTL*, and show that pushdown module checking, which is by itself harder than pushdown model checking, becomes undecidable when the environment has imperfect information. We also show that undecidability relies on hiding information about the pushdown store. Indeed, we prove that with imperfect information about the control states, but a visible pushdown store, the problem is decidable and its complexity is the same as that of (perfect information) pushdown module checking.

1 Introduction

In system modeling we distinguish between *closed* and *open* systems [HP85]. In a closed system all the nondeterministic choices are internal, and resolved by the system. In an open system there are also external nondeterministic choices, which are resolved by the environment [Hoa85]. In order to check whether a closed system satisfies a required property, we translate the system into some formal model, specify the property with a temporal-logic formula, and check formally that the model satisfies the formula. Hence, the name *model checking* for the verification methods derived from this viewpoint ([CE81,QS81]).

In [KV96,KVW01], Kupferman, Vardi, and Wolper studied open finite-state systems. In their framework, the open finite-state system is described by a labeled state-transition graph called *module*, whose set of states is partitioned into a set of *system states* (where the system makes a transition) and a set of *environment states* (where the environment makes a transition). Given a module $\mathcal{M}$ describing the system to be verified, and a temporal logic formula φ specifying

⋆ Work supported in part by MIUR FIRB Project no. RBAU1P5SS, NSF grants CCR-9988322, CCR-0124077, CCR-0311326, CCF-0613889, and ANI-0216467, by BSF grant 9800096, and by gift from Intel.

the desired behavior of the system, the problem of model checking a module, called *module checking*, asks whether for all possible environments $\mathcal{M}$ satisfies φ. In particular, it might be that the environment does not enable all the external nondeterministic choices. Module checking thus involves not only checking that the full computation tree $\langle T_M, V_M \rangle$ obtained by unwinding $\mathcal{M}$ (which corresponds to the interaction of $\mathcal{M}$ with a maximal environment) satisfies the specification φ, but also that every tree obtained from it by pruning children of environment nodes (this corresponds to the different choices of different environments) satisfy φ. For example, consider an ATM machine that allows customers to deposit money, withdraw money, check balance, etc. The machine is an open system and an environment for it is a subset of the set of all possible infinite lines of customers, each with its own plans. Accordingly, there are many different possible environments to consider. It is shown in [KV96,KVW01] that for formulas in branching time temporal logics, module checking open finite-state systems is exponentially harder than model checking closed finite-state systems.

In [KV97] module checking has been extended to a setting where the environment has *imperfect information*[1] about the state of the system (see also [CH05,CDHR06], for related work regarding imperfect information). In this setting, every state of the module is a composition of *visible* and *invisible* variables, where the latter are hidden from the environment. While a composition of a module $\mathcal{M}$ with an environment with perfect information corresponds to arbitrary disabling of transitions in $\mathcal{M}$, the composition of $\mathcal{M}$ with an environment with imperfect information is such that whenever two computations of the system differ only in the values of internal variables along them, the disabling of transitions along them coincide. For example, in the above ATM machine, a person does not know, before he asks for money, whether or not the ATM has run out of paper for printing receipts. Thus, the possible behaviors of the environment are independent of this missing information. Given an open system $\mathcal{M}$ with a partition of $\mathcal{M}$'s variables into visible and invisible, and a temporal logic formula φ, the module-checking problem with imperfect information asks whether φ is satisfied by all trees obtained by pruning children of environment nodes from $\langle T_M, V_M \rangle$, according to environments whose nondeterministic choices are independent of the invisible variables. One of the results shown in [KV97] is that *CTL* module checking with imperfect information is Exptime-complete.

In recent years, model checking of pushdown systems has received a lot of attention (see for example [Wal96,Wal00,BEM97,EKS03]), largely due to the ability of pushdown systems to capture the flow of procedure calls and returns in programs [ABE$^+$05]. Recently, [BMP05] extended these techniques by introducing open pushdown systems (with perfect information) that interact with their environment. It is shown in [BMP05] that *CTL pushdown module checking* is 2Exptime-complete and thus much harder than pushdown model checking.

Consider again the example of the ATM machine, where the information regarding the presence of printing paper is invisible to the customers. Suppose

[1] In the literature, the term *incomplete information* is sometimes used to refer to what we call imperfect information.

also that the ATM machine shows advertisements, and that it works under the constraint that the number of advertisements the customer must view, before the card can be taken out of the machine, is equal to the number of operations the customer performed. The described machine can be modeled as an open pushdown system $\mathcal{M}$ where control states take care of the operation performed by the ATM (interacting with customers), and the pushdown store is used to keep track of the advertisements that remain to be shown. Now, suppose that we want to verify that in all possible environments, it is always possible for an inserted card to be ejected. This requirement can be modeled by the *CTL* formula $\varphi = AG(\textbf{insert-card} \rightarrow EF\textbf{eject-card})$. Since the presence of printing paper is invisible to the customers, we have imperfect information about the control states of the module. If we allow the ATM to push, after each operation the customer makes, an invisible number (possibly zero) of pending advertisements, then we also have invisible information in the pushdown store.

In this paper, we extend pushdown module checking by considering environments with imperfect information about the system's state and pushdown store content. To this aim, we first have to define how a pushdown system keeps part of its internal configuration invisible to the environment and another part visible. In [PR79], a *private pushdown store automata* is defined to be a Turing machine with two tapes: a read only public (visible) one-way input tape, and a possibly private (invisible) work tape, simulating a pushdown store. Unfortunately, their definition is not suitable for our purpose as it allows for only two levels of information hiding: either the pushdown store and control state are completely visible, or completely invisible. The definition we use instead is an extension of the idea used for finite-state systems. Like in the finite case, we assume the control states are assignments to boolean *control variables*, some of which are visible and some of which are invisible. Similarly, symbols of the pushdown store are assignments to boolean visible and invisible *pushdown store variables*.

In [KV97], each state is partitioned into input, output, and invisible variables, where the environment supplies the input variables, and the system supplies the output and invisible variables. This idea works well for finite state-systems but not when we have to deal with imperfect information about the pushdown store. Note that a symbol pushed now, influences the computation much later, when it becomes the top of the pushdown store. Indeed, asking the environment to supply as input part of each symbol in the pushdown, is asking it to intimately participate in the internals of the computation, which is less natural. We find it more natural to think of the environment as choosing the possible transitions at certain points of the computation. For example, if the environment supplies the current reading of a physical sensor, we think of it as disabling all the transitions that are irrelevant for this reading. Thus, we model an open pushdown system with imperfect information by partitioning configurations into system and environment configurations, and also partitioning states and pushdown store symbols into visible and invisible variables, combining features from both [KV96] and [KV97].

We study the complexity of the pushdown module-checking problem with imperfect information, with respect to the branching-time logic *CTL*. We show that the problem is undecidable in the general case. We also show that undecidability relies on hiding information about the pushdown store. Indeed, we prove that *CTL* pushdown module checking with imperfect information about the internal control states, but a visible pushdown store, is decidable and 2EXPTIME-complete. Hence, it is not harder than perfect information *CTL* pushdown module checking. For the upper bound we use an automata-theoretic approach and introduce a new automata model, namely *semi-alternating pushdown Büchi tree automata* (*PD-SBT*). These are alternating pushdown Büchi tree automata [KPV02] where the universality is not allowed on the pushdown store content. That is, two copies of the automaton that read the same input, from two configurations that have the same top of pushdown store, must push the same value into the pushdown store. Our algorithm reduces the addressed problem to the emptiness problem of PD-SBT. We show that PD-SBT are equivalent to nondeterministic pushdown Büchi tree automata, for which the emptiness problem can be solved in EXPTIME [KPV02]. Note that alternating pushdown automata, in contrast to the semi-alternating ones, are *not* equivalent to nondeterministic pushdown automata. Indeed, since the emptiness problem of the intersection of two context free languages is undecidable [HU79], the emptiness problem of alternating pushdown automata is undecidable already in the case of finite words.

2 Preliminaries

Let Υ be a set. An Υ*-tree* is a prefix closed subset $T \subseteq \Upsilon^*$. The elements of T are called *nodes* and the empty word ε is the *root* of T. For $v \in T$, the set of *children* of v (in T) is $child(T, v) = \{v \cdot x \in T \mid x \in \Upsilon\}$. Given a node $v = u \cdot x$, with $u \in \Upsilon^*$ and $x \in \Upsilon$, we define $last(v)$ to be x. We also say that v *corresponds* to x. The complete Υ-tree is the tree Υ^*. For $v \in T$, a (full) path π of T from v is a *minimal* set $\pi \subseteq T$ such that $v \in \pi$ and for each $v' \in \pi$ such that $child(T, v') \neq \emptyset$, there is exactly one node in $child(T, v')$ belonging to π. Note that every infinite word $w \in \Upsilon^\omega$ can be thought of as an infinite path in the tree Υ^*, namely the path containing all the finite prefixes of w. For an alphabet Σ, a Σ-labeled Υ-tree is a pair $\langle T, V\rangle$ where T is an $\Upsilon-$tree and $V : T \to \Sigma$ maps each node of T to a symbol in Σ.

An *open system* is a system that interacts with its environment and whose behavior depends on this interaction. We consider the case where the environment has imperfect information about the system, i.e., when the system has internal variables that are not visible to its environment. We describe such a system by a *module* $\mathcal{M} = \langle AP, W_s, W_e, w_0, R, L, \cong\rangle$, where AP is a finite set of *atomic propositions*, W_s is a set of *system states*, and W_e is a set of *environment states*. We assume $W_s \cap W_e = \emptyset$, and call $W = W_s \cup W_e$ the set of $\mathcal{M}$'s *states*. $w_0 \in W$ is the *initial state*, $R \subseteq W \times W$ is a total *transition relation*, $L : W \to 2^{AP}$ is a labeling function that maps each state of $\mathcal{M}$ to the set of atomic propositions that hold in it, and $\cong$ is an equivalence relation on W.

In order to present a unified definition that is general enough to handle both finite-state and infinite-state systems, we model the fact that the environment has imperfect information about the states of the system by an equivalence relation $\cong$. States that are indistinguishable by the environment, because the difference between them is kept invisible by the system, are equivalent according to $\cong$. We write $[W]$ for the set of equivalence classes of W under $\cong$. Since states in the same equivalence class are indistinguishable by the environment, from the environment's point of view, the states of the system are actually the equivalence classes themselves. The equivalence class $[w]$ of $w \in W$, is called the *visible part* of w, since it is in a sense what the environment "sees" of w. We write $vis(w)$ instead of $[w]$, to emphasize this. Note that we can also do the converse. That is, given a function vis, whose domain is W, we can define the equivalence relation $\cong$ by letting $w \cong w'$ iff $vis(w) = vis(w')$. We can then think of the range of vis as the set of the equivalence classes $[W]$ and associate $[w]$ with the value $vis(w)$.

A module $\mathcal{M}$ is *closed* if $W_e = \emptyset$ (meaning that $\mathcal{M}$ does not interact with any environment) and *open* otherwise. Since the designation of a state as an environment state is obviously known to the environment, we require that for every $w, w' \in W$ such that $w \cong w'$, we have that $w \in W_e$ iff $w' \in W_e$. Also note that if $w \cong w'$, from the environment's point of view, the set of atomic propositions that currently hold in w may just as well be $L(w')$. We therefore define the labeling, as seen by the environment, as a function $visL : [W] \to 2^{2^{AP}}$ that maps the visible part of a state to a set of possible sets of atomic propositions: $visL([u]) = \{L(w) \mid w \in W \wedge w \cong u\}$. If it is always the case that $w \cong w' \implies L(w) = L(w')$, we say that the atomic propositions are visible.

For $\langle w, w' \rangle \in R$, we say that w' is a *successor* of w. The requirement that R be total means that every state w has at least one successor. A *computation* of $\mathcal{M}$ is a sequence $w_0 \cdot w_1 \cdots$ of states, such that for all $i \geq 0$ we have $\langle w_i, w_{i+1} \rangle \in R$. For each $w \in W$, we denote by $succ(w)$ the set (possibly empty) of w's successors. When the module $\mathcal{M}$ is in a system state w_s, then all successor states are possible next states. On the other hand, when $\mathcal{M}$ is in an environment state w_e, the environment decides, based on the visible parts of each successor of w_e, and of the history of the computation so far, to which of the successor states the computation can proceed, and to which it can not.

The set of all (maximal) computations of $\mathcal{M}$ starting from the initial state w_0 can be described by an AP-labeled W-tree $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ called a *computation tree*, which is obtained by unwinding $\mathcal{M}$ in the usual way. Each node $v = v_1 \cdots v_k$ of $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ describes the (partial) computation $w_0 \cdot v_1 \cdots v_k$ of $\mathcal{M}$, with the root ε corresponding to w_0. The children of v are exactly all nodes of the form $v_1 \cdots v_k \cdot w$, where w ranges over all the successors of v_k in $\mathcal{M}$. We extend the definition of the vis function to nodes in the natural way. Thus, the visible part of a node v is $vis(v) = vis(v_1) \cdots vis(v_k)$. The labeling $V_{\mathcal{M}}$ of a node v depends on the state it corresponds to (its last state), i.e., $V_{\mathcal{M}}(v) = L(last(v))$. Also, if v corresponds to an environment state, we say that v is an *environment node*.

The problem of deciding, for a given *CTL* formula[2] φ over the set AP of atomic propositions, whether $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ satisfies φ is the usual *model checking problem* (formally denoted $\mathcal{M} \models \varphi$) [CE81,QS81]. In model checking, we only have to consider the computation tree $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$, since the module we want to

[2] For a definition of the syntax and semantics of *CTL* see for example [KV96].

check is closed and thus its behavior is not affected by the environment. On the other hand, whenever we consider an open module, $\langle T_{\mathcal{M}}, V_{\mathcal{M}}\rangle$ corresponds to a very specific environment: a maximal environment that never restricts the set of next states. Therefore, when we examine a branching-time specification φ w.r.t. an open module $\mathcal{M}$, the formula φ should hold not only in $\langle T_{\mathcal{M}}, V_{\mathcal{M}}\rangle$, but in all the trees obtained by pruning from $\langle T_{\mathcal{M}}, V_{\mathcal{M}}\rangle$ subtrees whose roots are children (successors) of environment nodes, in accordance with all *possible* environments. It is important to note that in the case of perfect information (i.e., $\cong$ is actually the equality relation), every such pruning corresponds to some environment; however, in the case of imperfect information, only if the pruning is consistent with the partial information available to the environment, will the tree correspond to an actual environment. Formally, if two nodes v and v' are indistinguishable, i.e., if $vis(v) = vis(v')$, then a tree in which the subtree rooted at v is pruned, but the one rooted at v' is not pruned, does not correspond to any environment, and should not be considered. As noted in [KV97], the fact that given a pruning of $\langle T_{\mathcal{M}}, V_{\mathcal{M}}\rangle$, a finite automaton cannot decide if that pruning corresponds to an actual environment or not, is the main source of difficulty in dealing with module checking with imperfect information. Also note that the knowledge-based subset construction that is used to transform games of imperfect information into ones of perfect information (see for example [CDHR06]), is not applicable in this context, since in general there is no connection between the satisfiability of a branching time formula on the original structure and its satisfiability on the one obtained by the knowledge-based subset construction.

Recall that whenever $\mathcal{M}$ interacts with an environment ξ, its possible moves from environment states depends on the behavior of ξ. We can think of an environment to $\mathcal{M}$ as a strategy $\xi : [W]^* \rightarrow \{\top, \bot\}$ that maps a finite history s of a computation, as seen by the environment, to either $\top$ or $\bot$, meaning that the environment respectively allows or disallows $\mathcal{M}$ to trace s. We say that the tree $\langle [W]^*, \xi\rangle$ maintains the strategy applied by ξ, and we call it a *strategy tree*. We denote by $\mathcal{M} \lhd \xi$ the AP-labeled W-tree induced by the composition of $\langle T_{\mathcal{M}}, V_{\mathcal{M}}\rangle$ with ξ; that is, the AP-labeled W-tree obtained by pruning from $\langle T_{\mathcal{M}}, V_{\mathcal{M}}\rangle$ subtrees according to ξ. Note that by the definition above, ξ may disable all the children of a node v. Since we usually do not want the environment to completely block the system, we require that at least one child of each node is enabled. In this case, we say that the composition $\mathcal{M} \lhd \xi$ is *deadlock free*.

To see the interaction of $\mathcal{M}$ with ξ, let $v \in T_{\mathcal{M}}$ be an environment node, and $v' \in T_{\mathcal{M}}$ be one of its children. The subtree rooted in v' is pruned iff $\xi(vis(v')) = \bot$. Every two nodes v_1 and v_2 that are indistinguishable according to ξ's imperfect information have $vis(v_1) = vis(v_2)$. Also, recall that the designation of a state as an environment state is based only on the visible part of that state. Thus, if v_1 is a child of an environment node then so is v_2, and either both subtrees with roots v_1 and v_2 are pruned, or both are not. Note that once $\xi(v) = \bot$ for some $v \in [W]^*$, we can ignore $\xi(v \cdot t)$, for all $t \in [W]^*$. Indeed, once the environment disables the transition to a certain node v, it actually disables the transitions to all the nodes in the subtree with root v. We can now formally define the interaction of an open module with an environment with imperfect information. From now on, unless stated differently, we always refer to modules that are open, and environments with imperfect information. Given a module $\mathcal{M}$, and a strategy tree $\langle [W]^*, \xi \rangle$ for an environment ξ, an AP-labeled W-tree $\langle T, V \rangle$ corresponds to $\mathcal{M} \lhd \xi$ iff the following hold:

- The root of T corresponds to w_0.
- For $v \in T$ with $last(v) \in W_s$, we have $child(T, v) = \{v \cdot w_1, \ldots, v \cdot w_n\}$, where $succ(last(v)) = \{w_1, \ldots, w_n\}$.
- For $v \in T$ with $last(v) \in W_e$, there exists a nonempty subset $\{w_1, \ldots, w_k\}$ of $succ(last(v))$ such that $child(T, v) = \{v \cdot w_1, \ldots, v \cdot w_k\}$. Furthermore, for all w in $\{w_1, \ldots, w_k\}$ we have that $\xi(vis(v \cdot w)) = \top$, while for all w in $succ(last(v)) \setminus \{w_1, \ldots, w_k\}$ we have that $\xi(vis(x \cdot w)) = \bot$.
- For every node $v \in T$, we have that $V(v) = L(last(v))$.

For a module $\mathcal{M}$ and a temporal logic formula over the set AP, we say that $\mathcal{M}$ *reactively satisfies* φ, denoted $\mathcal{M} \models_r \varphi$, if $\mathcal{M} \lhd \xi$ satisfy φ, for every environment ξ for which $\mathcal{M} \lhd \xi$ is deadlock free. The problem of deciding whether $\mathcal{M} \models_r \varphi$ is called *module checking*, and was first introduced and studied in [KV96,KVW01] for finite-state systems with perfect information. The problem was successively extended to imperfect information in [KV97]. For *CTL* formulas it has been shown that the complexity of both problems is EXPTIME-complete[3].

3 Imperfect Information Pushdown Module Checking

In this section, we extend the notion of module checking with imperfect information to infinite-state systems induced by *Open Pushdown Systems* (*OPD*).

Definition 1. *An* OPD *is a tuple* $\mathcal{S} = \langle AP, Q, q_0, \Gamma, \flat, \delta, \mu, Env \rangle$*, where* AP *is a finite set of atomic propositions,* Q *is the set of (*control*)* states*, and* $q_0 \in Q$ *is an* initial state*. We assume that* $Q \subseteq 2^{I \cup H}$ *where* I *and* H *are disjoint finite sets of* visible *and* invisible control variables*, respectively.* Γ *is a finite pushdown store alphabet,* $\flat \notin \Gamma$ *is the* pushdown store bottom symbol*, and we use* $\Gamma_\flat$ *to denote* $\Gamma \cup \{\flat\}$*. We assume that* $\Gamma \subseteq 2^{I_\Gamma \cup H_\Gamma}$ *where* I_Γ *and* H_Γ *are disjoint finite sets of* visible *and* invisible pushdown store variables*, respectively.* $\delta \subseteq (Q \times \Gamma_\flat) \times (Q \times \Gamma_\flat^*)$ *is a finite transition relation, and* $\mu : Q \times \Gamma_\flat \to 2^{AP}$ *is a labeling function.* $Env \subseteq Q \times \Gamma_\flat$ *is used to specify the set of* environment configurations*. The size* $|\mathcal{S}|$ *of* $\mathcal{S}$ *is* $|Q| + |\Gamma| + |\delta|$*, with* $|\delta| = \sum_{((p,\gamma),(q,\beta)) \in \delta} |\beta|$.

A *configuration* of $\mathcal{S}$ is a pair (q, α) where q is a control state and $\alpha \in \Gamma^* \cdot \flat$ is a pushdown store content. We write $top(\alpha)$ for the leftmost symbol of α and call it the *top of the pushdown store* α. The *OPD* moves according to the transition relation. Thus, $((p, \gamma), (q, \beta)) \in \delta$ implies that if the *OPD* is in state p and the

[3] Although the complexity of the perfect and imperfect information cases coincide in the general case, [KVW01,KV97] show that when the formula is constant the imperfect information case is exponentially harder.

top of the pushdown store is γ, it can move to state q, pop γ and push β. We assume that if $\flat$ is popped it gets pushed right back, and that it only gets pushed in such cases. Thus, $\flat$ is always present at the bottom of the pushdown store, and nowhere else. Note that we make this assumption also about the various pushdown automata we use later. Also note that the possible moves of the system, the labeling function, and the designation of configurations as environment configurations, are all dependent only on the current control state and the top of the pushdown store.

For a control state $q \in Q$, the visible part of q is $vis(q) = q \cap I$. For a pushdown store symbol $\gamma \in \Gamma$, if $\gamma \subseteq H_\Gamma$ and $\gamma \neq \emptyset$ we set $vis(\gamma) = \varepsilon$, otherwise we set $vis(\gamma) = \gamma \cap I_\Gamma$. By setting $vis(\gamma) = \varepsilon$ whenever γ consists entirely of invisible variables, we allow the system to completely hide a push operation (obviously a corresponding pop will also be invisible). When such a push occurs, the environment does not see the symbol $\emptyset$ being pushed, rather, it sees no push at all. This is necessary since in many applications what is actually pushed is immaterial, and the information to be revealed or hidden is only the depth of the pushdown store. The visible part of a pushdown store content $s = \gamma_0 \cdots \gamma_n \cdot \flat$ is defined in the natural way: $vis(s) = vis(\gamma_0) \cdots vis(\gamma_n) \cdot \flat$. The visible part of a configuration (q, α), is thus $vis((q, \alpha)) = (vis(q), vis(\alpha))$. As for modules, the designation of a configuration of an *OPD* as an environment configuration is known to the environment. Thus, we require that for every two configurations (q, α) and (q', α') such that $vis(q, top(\alpha)) = vis(q', top(\alpha'))$, it holds that $(q, top(\alpha)) \in Env$ iff $(q', top(\alpha')) \in Env$.

Definition 2. *An* OPD $\mathcal{S} = \langle AP, Q, q_0, \Gamma, \flat, \delta, \mu, Env\rangle$ *induces an infinite-state module* $\mathcal{M}_S = \langle AP, W_s, W_e, w_0, R, L, \cong\rangle$, *where:*

- AP *is a set of atomic propositions;*
- $W_s \cup W_e = Q \times \Gamma^* \cdot \flat$ *is the set of configurations;*
- W_e *is the set of configurations* (q, α) *such that* $(q, top(\alpha)) \in Env$*;*
- $w_0 = (q_0, \flat)$ *is the initial configuration;*
- R *is a transition relation, where* $((q, \gamma \cdot \alpha), (q', \beta)) \in R$ iff *there exist* $((q, \gamma), (q', \beta')) \in \delta$ *such that* $\beta = \beta' \cdot \alpha$*;*
- $L((q, \alpha)) = \mu(q, top(\alpha))$ *for all* $(q, \alpha) \in W$*;*
- *For every* $w, w' \in W$*, we have that* $w \cong w'$ *iff* $vis(w) = vis(w')$*.*

To describe the interaction of an *OPD* $\mathcal{S}$ with its environment, we consider the interaction of the environment with the induced module $\mathcal{M}_S$. Indeed, every environment ξ of $\mathcal{S}$, can be represented by a strategy tree $\langle [W]^*, \xi\rangle$, and the composition $\mathcal{M}_S \lhd \xi$ of $\langle [W]^*, \xi\rangle$ with $\langle T_{\mathcal{M}_S}, V_{\mathcal{M}_S}\rangle$ describes all the computations of $\mathcal{S}$ allowed by the environment ξ. We can thus define the following problem.

Pushdown module checking problem with imperfect information: Given an *OPD* $\mathcal{S}$ and a *CTL* formula[4] φ, decide whether $\mathcal{M}_S \models_r \varphi$, i.e., whether $\mathcal{M}_S \lhd \xi$ satisfy φ, for every environment ξ for which $\mathcal{M}_S \lhd \xi$ is deadlock free.

[4] The semantics of *CTL* is usually defined with respect to infinite paths, so we assume $\mathcal{M}_S$ has no configurations without successors. However, using a similar technique to the one used in [BMP05] our results can be adapted to the situation where terminal configurations are also allowed.

Note that starting with an *OPD* $\mathcal{S}$ having $Env = \emptyset$ (that is, the behavior of $\mathcal{S}$ is not affected by any environment) the induced module is closed. In this case, the problem we address becomes the classical *pushdown model checking problem*, and for *CTL* specifications it has been studied in [Wal96,Wal00]. Also, if the *OPD* is open ($Env \neq \emptyset$) but there is no invisible information (both H and H_Γ are empty), the addressed problem is called *pushdown module checking with perfect information*, and it has been studied in [BMP05].

In the remaining part of this section, we study the pushdown module checking problem with imperfect information and show that it is undecidable for *CTL* specifications. In the next section, we show that undecidability relies on the system's ability to hide information about the pushdown store. Namely, we prove that if we start with an *OPD* with $H_\Gamma = \emptyset$, the problem becomes decidable (even if $H \neq \emptyset$), and its complexity is the same as that of pushdown module checking with perfect information.

Undecidability of the pushdown module checking problem with imperfect information is obtained by a reduction from the universality problem of nondeterministic pushdown automata on finite words (*PDA*), which is undecidable [HU79]. That is, given a *PDA* $\mathcal{P}$, we build an *OPD* $\mathcal{S}$ and a *CTL* formula φ, such that the module induced by $\mathcal{S}$ reactively satisfies φ iff $\mathcal{P}$ is universal.

Our choice to do a reduction from the universality problem of PDA is not at all arbitrary[5]. It is well known that checking for the universality of a nondeterministic automaton can be thought of as a game between a protagonist trying to prove that the automaton is not universal, and an antagonist claiming that it is universal. The universality game is played as follows. The protagonist chooses the first symbol, the antagonist responds with the first part of the run, the protagonist chooses the next symbol, the antagonist extends the run, and so on. The protagonist wins if the resulting run is rejecting, and the antagonist wins if it is accepting. Note that if the automaton is not universal the protagonist has a winning strategy, namely, choosing the letters of a word not accepted by the automaton. However, since the automaton is nondeterministic, the converse is not true. That is, even if the automaton is universal, the antagonist may not have a winning strategy. Also note that (again due to nondeterminism) if the protagonist can see the moves of the antagonist, it may force the run to be rejecting even though the word it supplies can be accepted by the automaton. Hence, the game is sound but not complete. However, if the protagonist cannot see the moves of the antagonist the game becomes sound and complete. Deciding if the automaton is not universal can be reduced to deciding whether the antagonist has a winning strategy in the corresponding universality game with imperfect information. By casting the universality game of *PDA* to a special instance of the pushdown module checking problem with imperfect information, the latter is shown to be undecidable. The complete proof can be found in the full version.

Theorem 1. *The pushdown module-checking problem with imperfect information for* CTL *specifications is undecidable.*

[5] We thank Martin Lange for a useful discussion on the connection between the proof of Theorem 1 and the game interpretation of the universality problem.

It turns out that even if the environment has full information about the control states and (surprisingly enough) about which atomic propositions hold at each configuration the problem remains undecidable. Thus, we have.

Theorem 2. *The imperfect information pushdown module checking problem for* CTL*, with visible control states and atomic propositions, is undecidable.*

4 Module Checking with Visible Pushdown Store

In this section, we show that pushdown module checking for *CTL* with full information about the pushdown store content ($H_\Gamma = \emptyset$), but not about the control states (when $H \neq \emptyset$), is decidable and 2EXPTIME-complete, matching the complexity of pushdown module checking with complete information. For the upper bound we use an automata-theoretic approach and introduce a new automata model, namely *semi-alternating pushdown Büchi tree automata* (*PD-SBT*). Our algorithm reduces the addressed problem to the emptiness problem of PD-SBT. We show that PD-SBT are equivalent to nondeterministic pushdown Büchi tree automata, for which emptiness can be decided in EXPTIME[KPV02]. The formal definition of semi-alternating pushdown tree automata follows.

Semi-Alternating Pushdown Tree Automata. A *PD-SBT* is a tuple $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F \rangle$ where Σ is a finite input alphabet, D is a finite set of *directions*, Γ is a finite pushdown store alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\flat \notin \Gamma$ is the pushdown store bottom symbol, and $F \subseteq Q$ is a Büchi acceptance condition. Moreover, δ is a finite transition relation defined as a function $\delta : Q \times \Sigma \times \Gamma_\flat \to \mathcal{B}^+(D \times Q \times \Gamma_\flat^*)$, where $\Gamma_\flat = \Gamma \cup \{\flat\}$ as usual, and $\mathcal{B}^+(D \times Q \times \Gamma_\flat^*)$ is the set of all finite positive boolean combinations of triples (d, q, β), where d is a direction, q is a state, and β is a string of pushdown store symbols. We also allow the formulas **true** and **false**. We write $S \in \delta(p, \sigma, \gamma)$ to denote that S is a set of tuples (d, q, β) that satisfy $\delta(p, \sigma, \gamma)$.

What makes the automaton semi-alternating is the requirement that for every $d \in D$, $\sigma \in \Sigma$, $p, p' \in Q$ (possibly the same state), and $\gamma \in \Gamma$, if (d, q, β) appears in $\delta(p, \sigma, \gamma)$, and (d, q', β') appears in $\delta(p', \sigma, \gamma)$, then $\beta = \beta'$. That is, two copies of the automaton that read the same input, from two configurations that have the same top symbol of the pushdown store and proceed in the same direction, must push the same value into the pushdown store. In particular, it follows that in every run, two copies of the automaton that are reading the same node of an input tree have the same pushdown store content. Note that if we remove the semi-alternation requirement, the resulting automaton is called *alternating pushdown Büchi tree automaton* (*PD-ABT*).

As an example, for $D = \{0, 1\}$, having $\delta(q, \sigma, \gamma) = ((0, q_1, \beta_1) \vee (1, q_2, \beta_2)) \wedge (1, q_1, \beta_2)$ means that when a copy of the automaton that is in a configuration where the current state is q, and the top of pushdown store is γ, reads a node in the input tree whose label is σ, it can proceed in one of two ways. In the first case, one copy proceeds in direction 0 to state q_1, by replacing γ with β_1, and one copy proceeds in direction 1 to state q_1, by replacing γ with β_2. In the second case, two copies proceed in direction 1, one to state q_1 and the other to state q_2, and in both copies γ is replaced with β_2. Hence, $\vee$ and $\wedge$ in $\delta(q, \sigma, \gamma)$ represent, respectively, choice and concurrency. As a special case of PD-ABT, we consider *nondeterministic pushdown Büchi tree automata* (*PD-NBT*) where the concurrency feature is not allowed. That is, whenever a PD-NBT visits a node x of the input tree, it sends to each successor (direction) of x at most one copy of itself. More formally, a PD-NBT is a PD-ABT in which δ is in disjunctive normal form, and in each conjunct each direction appears at most once.

A run of a PD-SBT $\mathcal{A}$ on a Σ-labeled tree $\langle T, V\rangle$, with $T = D^*$, is a $(D^* \times Q \times \Gamma^* \cdot \flat)$-labeled $\mathbb{N}$-tree $\langle T_r, r\rangle$ such that the root is labeled with $(\varepsilon, q_0, \flat)$ and the labels of each node and its successors satisfy the transition relation. Formally, a $(D^* \times Q \times \Gamma^* \cdot \flat)$-labeled tree $\langle T_r, r\rangle$ is a run of $\mathcal{A}$ on $\langle T, V\rangle$ iff

- $r(\varepsilon) = (\varepsilon, q_0, \flat)$, and
- for all $x \in T_r$ such that $r(x) = (y, p, \gamma \cdot \alpha)$, there is an $n \in \mathbb{N}$ such that the successors of x are exactly $x \cdot 1, \ldots x \cdot n$, and for all $1 \leq i \leq n$ we have $r(x \cdot i) = (y \cdot d_i, p_i, \beta_i \cdot \alpha)$ for some $\{(d_1, p_1, \beta_1), \ldots, (d_n, p_n, \beta_n)\} \in \delta(p, V(y), \gamma)$.

For a path $\pi \subseteq T_r$, let $inf_r(\pi) \subseteq Q$ be the set of states that appear in the labels of infinitely many nodes in π. For a Büchi condition $F \subseteq Q$, we have that π is *accepting* iff $inf_r(\pi) \cap F \neq \emptyset$. A run $\langle T_r, r\rangle$ is *accepting* iff all its paths are accepting. The automaton $\mathcal{A}$ accepts an input tree $\langle T, V\rangle$ iff there is an accepting run of $\mathcal{A}$ over $\langle T, V\rangle$. The language of $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of Σ-labeled trees accepted by $\mathcal{A}$. We say that an automaton $\mathcal{A}$ is nonempty iff $L(\mathcal{A}) \neq \emptyset$.

Given a PD-SBT $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F\rangle$, we define the size of $\mathcal{A}$ as $|\mathcal{A}| = |Q| + |\delta|$, where $|\delta|$ is the sum of the lengths of the satisfiable (i.e., not **false**) formulas that appear in $\delta(q, \sigma, \gamma)$ for some q, σ, and γ.

In [MH84], Miyano and Hayashi describe a translation of alternating Büchi automata on words to nondeterministic ones. We now present an extension of their translation to show the equivalence of PD-SBT and PD-NBT.

Lemma 1. *Let $\mathcal{A}$ be a PD-SBT with n states. There is a PD-NBT $\mathcal{A}'$ with $2^{O(n)}$ states, such that $L(\mathcal{A}') = L(\mathcal{A})$.*

Proof. The automaton $\mathcal{A}'$ guesses a subset construction applied to a run of $\mathcal{A}$. At a given node x of a run of $\mathcal{A}'$, it keeps in its memory the set of configurations in which the various copies of $\mathcal{A}$ visit node x in the guessed run. Since $\mathcal{A}$ is semi-alternating, all copies of $\mathcal{A}$ that visit the same node x have the same pushdown store content, and thus can all be remembered using one pushdown store and a set of states of $\mathcal{A}$. In order to make sure that every infinite path visits states in F infinitely often, $\mathcal{A}'$ keeps track of states that "owe" a visit to F. Let $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F\rangle$. Then $\mathcal{A}' = \langle \Sigma, D, \Gamma, 2^Q \times 2^Q, \langle \{q_0\}, \emptyset\rangle, \flat, \delta', 2^Q \times \{\emptyset\}\rangle$, where δ' is defined as follows. We first need the following notation. For a set $S \subseteq Q$, a letter $\sigma \in \Sigma$, and a top of pushdown store symbol $\gamma \in \Gamma$, let $sat(S, \sigma, \gamma)$ be the set of subsets of $D \times Q \times \Gamma_\flat^*$ that satisfy $\bigwedge_{q \in S} \delta(q, \sigma, \gamma)$. Also, for two sets $O \subseteq S \subseteq Q$, a letter $\sigma \in \Sigma$, and a top of pushdown store symbol $\gamma \in \Gamma$, let $pair\_sat(S, O, \sigma, \gamma)$ be such that $\langle S', O' \rangle \in pair\_sat(S, O, \sigma, \gamma)$ iff $S' \in sat(S, \sigma, \gamma)$, $O' \subseteq S'$, and $O' \in sat(O, \sigma, \gamma)$. Finally, for a direction $d \in D$, we have $S'_d = \{s \mid (d, s, \beta) \in S' \text{ for some } \beta\}$ and $O'_d = \{o \mid (d, o, \beta) \in O' \text{ for some } \beta\}$. Thus, S'_d and O'_d are, respectively, the collections of all states that appear in S' and O' along with the direction d. Since $\mathcal{A}$ is semi-alternating, for every two triplets (d, q, β) and (d, q', β') in $sat(S, \sigma, \gamma)$ having the same direction d, we have that $\beta = \beta'$. Thus, we can define $store(d, \sigma, \gamma) = \beta$.

Now, δ' is defined, for all $\langle S, O \rangle \in 2^Q \times 2^Q$, $\sigma \in \Sigma$, and $\gamma \in \Gamma$, as follows.

- if $O \neq \emptyset$, then
$$\delta'(\langle S, O \rangle, \sigma, \gamma) = \bigvee_{\substack{\langle S', O' \rangle \in \\ pair\_sat(S, O, \sigma, \gamma)}} \bigwedge_{d \in D} (d, \langle S'_d, O'_d \setminus F \rangle, store(d, \sigma, \gamma))$$
Thus, when reading σ, from a configuration with a top of pushdown store symbol γ, the automaton $\mathcal{A}'$ sends to a direction $d \in D$ the set S'_d of states that the different copies of $\mathcal{A}$ send to direction d in the guessed run. Each such S'_d is paired with a subset O'_d of S'_d of the states that still "owe" a visit to F. The key observation is that since $\mathcal{A}$ is semi-alternating, all the copies that $\mathcal{A}$ sends to direction d replace γ with exactly the same pushdown store string, namely, with $store(d, \sigma, \gamma)$. Hence, the pushdown stores of all the copies that $\mathcal{A}$ sends to direction d are identical, and $\mathcal{A}'$ can keep track of them all using the single stack of the copy it send to direction d.
- if $O = \emptyset$, then
$$\delta'(\langle S, O \rangle, \sigma, \gamma) = \bigvee_{\substack{\langle S', O' \rangle \in \\ pair\_sat(S, O, \sigma, \gamma)}} \bigwedge_{d \in D} (d, \langle S'_d, S'_d \setminus F \rangle, store(d, \sigma, \gamma))$$
Thus, when no state "owes" a visit to F we know that every path in the guessed run of $\mathcal{A}$ visited F one more time, and the requirement to visit F is reinforced. $\square$

We can now show decidability for pushdown module checking for *CTL* with visible pushdown store. The decidability follows from Lemma 1, the fact that emptiness of PD-NBT is decidable, and the following theorem.

Theorem 3. *For an OPD $\mathcal{S}$ with $H_\Gamma = \emptyset$ and a CTL formula φ over $\mathcal{S}$'s atomic propositions, there is a PD-SBT $\mathcal{A}_{\mathcal{S},\varphi}$ of size $O(|\mathcal{S}| * |\varphi|)$, such that $L(A_{\mathcal{S},\varphi})$ is the set of strategies ξ such that $\mathcal{M}_S \lhd \xi$ is deadlock free and satisfies φ.*

Proof (Sketch). Essentially, the automaton $\mathcal{A}_{\mathcal{S},\varphi}$ we build is an extension of the product automaton obtained in the alternating-automata theoretic approach for *CTL* module checking with imperfect information [KV97]. The extension we consider here concerns the simulation of the pushdown store of the *OPD*.

Let $\mathcal{S} = \langle AP, Q, q_0, \Gamma, \flat, \delta, \mu, Env \rangle$ be an OPD, let φ be a *CTL* formula in positive normal form, and let $\mathcal{M}_S = \langle AP, W_s, W_e, w_0, R, L, \cong \rangle$ be the module induced by $\mathcal{S}$. We build an automaton $\mathcal{A}_{\mathcal{S},\varphi}$ that accepts $\{\top, \bot\}$-labeled trees corresponding to strategies ξ, whose composition with $\mathcal{M}_S$ is deadlock free and satisfy φ. Intuitively, a run of $\mathcal{A}_{\mathcal{S},\varphi}$ on an input strategy tree ξ, proceeds by simulating an unwinding of the module $\mathcal{M}_S$, pruned at each step according to the strategy ξ. Copies of the automaton simulating nodes in the computation tree of $\mathcal{M}_S$ that are indistinguishable by the environment are sent to the same direction in the input tree. The resulting run tree of $\mathcal{A}_{\mathcal{S},\varphi}$ on ξ is basically a replica of the composition $\mathcal{M}_S \lhd \xi$, and the fact that it satisfies the formula φ is checked on the fly, by employing in $\mathcal{A}_{\mathcal{S},\varphi}$ the usual alternating-automata approach for CTL model checking. In the full computation tree of $\mathcal{M}_S$, the set of directions is $G = \{(q, \beta) \mid ((p, \alpha), (q, \beta)) \in R$ for some p, α and $\beta\}$. Since in $\mathcal{S}$ the pushdown store is completely visible to the environment, the set of directions of the input strategy trees is $D = \{(vis(q), \beta) \mid ((p, \alpha), (q, \beta)) \in R$ for some p, α and $\beta\}$. Finally, due to the fact that all copies of the automaton sent to direction $(vis(q), \beta)$ push β into the pushdown store, the resulting automaton $\mathcal{A}_{\mathcal{S},\varphi}$ is semi-alternating.

We formally define $\mathcal{A}_{\mathcal{S},\varphi} = \langle \{\top, \bot\}, D, \Gamma, Q', q'_0, \flat, \delta', F \rangle$, where

- $Q' = (Q \times (cl(\varphi) \cup \{p_\top\}) \times \{\forall, \exists\} \times \{p_e, p_s\}) \cup \{q'_0\}$. States with the component $p_\top$ are used to check that the composition of $\mathcal{M}_S$ with the strategy is deadlock free, while states with a component in $cl(\varphi)$ check that this composition satisfies φ. The components p_e and p_s are used to flag that a currently simulated node, of the computation tree of $\mathcal{M}_S$, is a child of an environment or a system node, respectively. Clearly, the simulation should respect the strategy pruning specifications only if they correspond to children of environment nodes; that is, only if the current state q contains p_e. Every state is either in an existential or a universal mode, as specified by the $\forall$ and $\exists$ components. When the automaton is in a universal state $(q, \varphi, \forall, p_e)$ with a pushdown store content α, it accepts all strategies for which (q, α) in $\mathcal{M}_S$ is either pruned or satisfies φ (where $p_\top$ is satisfied iff the root of the strategy is labeled $\top$). When the automaton is in an existential state $(q, \varphi, \exists, p_e)$ with a pushdown store content α, it accepts all strategies for which (q, α) in $\mathcal{M}_S$ is not pruned and satisfies φ.
- The formal definition of $\delta' : Q' \times \Sigma \times \Gamma_\flat \rightarrow \mathcal{B}^+(D \times Q' \times \Gamma_\flat^*)$ is reported in the full version. Here, we just give an example of a transition rule. Consider, a transition from the configuration $(\langle p, \forall X\psi, \exists, p_e \rangle, \gamma \cdot \alpha)$, where $(p, \gamma) \in Env$. First, if the transition to $(p, \gamma \cdot \alpha)$ is disabled (that is, the automaton reads $\bot$), then, as the current mode is existential, the run is rejecting. If the transition to $(p, \gamma \cdot \alpha)$ is enabled, then the successors of $(p, \gamma \cdot \alpha)$ that are enabled should satisfy ψ. Note that all the successors of $(p, \gamma \cdot \alpha)$ that are indistinguishable by the environment are sent by the automaton to the same direction v. This guarantees that either all these successors are enabled by the strategy (in case the letter to be read in direction v is $\top$) or all are disabled (in case the letter in direction v is $\bot$). In addition, since the requirement to satisfy ψ concerns only successors of $(p, \gamma \cdot \alpha)$ that are enabled, the mode of the new states is universal. The copies of $\mathcal{A}_{\mathcal{S},\varphi}$ that check the composition with the strategy to be deadlock free guarantee that at least one successor of $(p, \gamma \cdot \alpha)$

is enabled. As noted earlier, the enable/disable instructions of the strategy are ignored in every configuration $(p, \gamma \cdot \alpha)$ that is a successor of a system configuration. Also note that since we assume that no configuration in $\mathcal{M}_S$ has no successors, the conjunctions and disjunctions in δ' cannot be empty.
- $F = Q \times \widetilde{U}(\varphi) \times \{\exists, \forall\} \times \{p_e, p_s\}$, where $\widetilde{U}(\varphi)$ is the set of all formulas of the form $\forall\psi_1\widetilde{U}\psi_2$ or $\exists\psi_1\widetilde{U}\psi_2$ in $cl(\varphi)$.

In the full version we prove that $\mathcal{A}_{\mathcal{S},\varphi}$ is semi-alternating and that the size of δ' is $O(|\delta| * |\varphi|)$. Since $|Q'| = O(|Q| * |\varphi|)$, the size of $\mathcal{A}_{\mathcal{S},\varphi}$ is $O(|\mathcal{S}| * |\varphi|)$. $\square$

We now consider the complexity bounds that follow from our algorithm.

Theorem 4. CTL *pushdown module checking with imperfect information about the control states but a visible pushdown store is* 2EXPTIME-*complete.*

Proof. The lower bound follows from the known bound for *CTL* pushdown module checking with perfect information [BMP05]. For the upper bound, Theorem 3 implies that $\mathcal{M}_S \models_r \varphi$ iff the language of the automaton $\mathcal{A}_{\mathcal{S},\neg\varphi}$ is empty. We recall that $\mathcal{A}_{\mathcal{S},\neg\varphi}$ is a PD-SBT of size $O(|\mathcal{S}| * |\varphi|)$. By Lemma 1, we can obtain a PD-NBT $\mathcal{A}$ equivalent to $\mathcal{A}_{\mathcal{S},\varphi}$, with an exponential blow-up. By [KPV02], the emptiness of $\mathcal{A}$ can be checked in exponential time. Thus, checking the emptiness of $\mathcal{A}$ is double-exponential in the sizes of $|\mathcal{S}|$ and $|\varphi|$. $\square$

5 Discussion

We have shown that the pushdown module checking problem with imperfect information is undecidable for specifications given in *CTL*. Moreover, since the formula used in the proof of Theorems 1 and 2 is an existential formula, the problem is already undecidable for the existential fragment *ECTL* of *CTL*. This obviously implies the undecidability of the problem with respect to more expressive logics such as *CTL** and μ-calculus. Recall that in our setting, whenever we push a symbol consisting entirely of invisible variables, the environment does not see the push at all. One can think of a variant of the problem where the environment does see that a push occurred, but not what was pushed. Thus, the depth of the stack is always known to the environment. It is an open question whether this variant of the problem is decidable or not. As good news, we also showed that if the pushdown store is visible, the problem is decidable, and not harder than perfect information pushdown module checking. An interesting question is whether this variant of the problem remains decidable also for more expressive logics like *CTL**. By using an approach similar to the one used for *CTL*, we can reduce the problem for *CTL** to the emptiness problem of a semi-alternating pushdown tree automaton, but with a stronger acceptance condition, such as the parity condition. We do not know, however, if the emptiness problem for such automata is decidable or not. The main source of difficulty is that all known methods to remove alternation from parity finite tree automata involve a co-determinization step, and thus can not be easily adapted to pushdown automata. Even in [KV05] where the emptiness problem of alternating parity tree automata is reduced to that of nondeterministic automata, without a co-determinization step, the correctness *proof* of the construction does contain such a step. Nevertheless, it is our conjecture that despite these difficulties, CTL^* pushdown module checking with visible pushdown store is decidable.

References

[ABE+05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *CONCUR'97*, LNCS 1243, pages 135–150. Springer-Verlag, 1997.

[BMP05] L. Bozzelli, A. Murano, and A. Peron. Pushdown module checking. In *LPAR'05*, LNCS 3835, pages 504–518. Springer-Verlag, 2005.

[CDHR06] K. Chatterjee, L. Doyen, T. A. Henzinger, and J. Raskin. Algorithms for omega-regular games with imperfect information. In *CSL'06*, volume 4207 of *LNCS*, pages 287–302. Springer-Verlag, 2006.

[CE81] E.M. Clarke and E.A. Emerson. Design and verification of synchronization skeletons using branching time temporal logic. In *Logics of Programs Workshop*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CH05] K. Chatterjee and T. A. Henzinger. Semiperfect-information games. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 1–18. Springer-Verlag, 2005.

[EKS03] J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer-Verlag, 1985.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[KPV02] O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown specifications. In *LPAR'02*, LNCS 2514, pages 262–277. Springer-Verlag, 2002.

[KV96] O. Kupferman and M.Y. Vardi. Module checking. In *CAV'96*, LNCS 1102, pages 75–86. Springer-Verlag, 1996.

[KV97] O. Kupferman and M. Y. Vardi. Module checking revisited. In *CAV'96*, LNCS 1254, pages 36–47. Springer-Verlag, 1997.

[KV05] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *IEEE FOCS'05*, pages 531–540, Pittsburgh, October 2005.

[KVW01] O. Kupferman, M.Y. Vardi, and P. Wolper. Module Checking. *Information and Computation*, 164(2):322–344, 2001.

[PR79] G. L. Peterson and J. H. Reif. Multiple-person alternation. In *FOCS'79*, pages 348 – 363. IEEE Computer Society, 1979.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in Cesar. In *Symp. on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1981.

[Wal96] I. Walukiewicz. Pushdown processes: Games and Model Checking. In *CAV'96*, LNCS 1102, pages 62–74. Springer-Verlag, 1996.

[Wal00] I. Walukiewicz. Model checking CTL properties of pushdown systems. In *FSTTCS'00*, LNCS 1974, pages 127–138. Springer-Verlag, 2000.