Coordination and Control of Hierarchically Organized Interacting Agents

Sunandita Patra$^1$, Paolo Traverso$^2$, Malik Ghallab$^3$, Dana Nau$^1$

$^1$Institute for Systems Research and Department of Computer Science, University of Maryland, College Park, USA
$^2$ Fondazione Bruno Kessler, Trento, Italy, $^3$ LAAS-CNRS, Toulouse, France

patras@umd.edu, traverso@fbk.eu, malik@laas.fr, nau@umd.edu

Abstract

The coordination and control of hierarchically organized interacting agents is an important issue in many applications, e.g., harbor or warehouse automation. A formalism of agents as hierarchical input/output automata is proposed. A system of interacting agents is modeled as the parallel composition of their automata. We extend the usual parallel composition operation of I/O automata with a hierarchical composition operation for refining abstract tasks into lower-level subtasks. We provide an algorithm to synthesize hierarchically organized controllers to coordinate the agents’ interactions in order to drive the system toward desired states. Our main contribution regards the formal definition, the representation, the theorems about its properties (i.e., the parallel and hierarchical composition are distributive operations), and the synthesis algorithm, proved to be complete and correct.

1 Motivation

Consider a collection of collaborative agents, having different capabilities and programmed to do different things under different conditions. Given a complex task or goal to accomplish, and a description of how each agent behaves, how can we organize the agents and manage their interactions in order to jointly accomplish a desired objective?

In this paper we provide a representation framework and algorithms for the above problem. In our formalism, the agents are represented as hierarchical input/output automata. Our algorithms synthesize a hierarchically organized collection of finite-state controllers for managing the interactions among the agents in order to achieve the goal.

As a motivating example, consider a warehouse automation infrastructure such as the Kiva system (D’Andrea 2012) that controls thousands of robots moving inventory shelves to human pickers preparing customers orders. According to (Wurman 2014), “planning and scheduling are at the heart of Kiva’s software architecture”. Right now, this appears to be done with extensive engineering of the environment, e.g., fixed robot tracks and highly structured inventory organization. A more flexible approach for dealing with contingencies, local failures, modular design and easier novel deployments, would be to model each agent (robot, shelf, refill, order preparation, etc.) through its possible interactions with the rest of the system, and automatically synthesize control programs to coordinate these interactions.

The idea of composing finite-state automata into a system has been used for a long time for system specification and verification, e.g., (Harel 1987). Although less popular, it has also been used in the field of automated planning for applications that naturally call for composition, e.g., planning in web services (Pistore, Traverso, and Bertoli 2005; Bertoli, Pistore, and Traverso 2010), or for the automation of or a large infrastructure (Bucchiarone et al. 2012).

In our approach, each agent is modeled as an input/output automaton $\sigma$ whose state transitions are governed by messages that are sent to and received from the other agents. If $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ is a set of such agents, planning for them does not mean generating a plan or policy as is typically done in AI planning. Instead, it means synthesizing a control automaton $\sigma_c$ to manage the interactions among the agents in $\Sigma$. The agents don’t send messages to each other directly, but instead send them to $\sigma_c$, which receives their messages and decides which messages to send to the agents to drive them toward a desired goal. Nondeterministic planning techniques can be used for synthesizing $\sigma_c$.

Known automata techniques are very useful, but but have several restrictions that limit their scope for our purpose. A large system such as a harbor (Bucchiarone et al. 2012) or a logistics network (Boese and Piotrowski 2009) is generally both distributed and hierarchical:

- These aren’t tightly-integrated monolithic systems. They are composed of agents that may even be geographically distributed. It is more convenient and scalable to rely on distributed controllers to coordinate their actions.
- Agents are composed hierarchically of components for various subtasks. One chooses which components (from among various alternatives) to incorporate into an agent.

The problem of generating a distributed hierarchy of controllers for such agents is novel in the field. It initially requires a theoretical basis, which is the purpose in this paper (no application nor experimental results are reported here).

Our contributions are:

- We formally define the notion of refinement for hierarchical communicating input/output automata, call them I/OAs, and propose a formalization of planning and acting problems for interacting agents in this original framework.
We propose a new algorithm for solving this problem, and we provide theorems about the main properties of this algorithm. The proof of this critical feature for the synthesis algorithm requires careful developments.

Distributivity allows us to show that the synthesis of a hierarchical control structure for a set of IOAs can be addressed as a nondeterministic planning problem.

We propose a new algorithm for solving this problem, and discuss its theoretical properties.

In the following, we develop the representation, its properties and the algorithm synthesising a hierarchical control structure with multiple distributed controllers; we then present the state of the art, future work and conclusion.

2 Representation

The proposed formalism relies on a class of automata endowed with composition and refinement operations. Furthermore, both agents and their components are modeled as hierarchical IOAs, hence in describing the formalism we sometimes will use “agent” and “component” interchangeably.

Automata. The building block of the representation is a particular input/output automaton (IOA) \( \sigma = \langle S, s_0, I, O, T, A, \gamma \rangle \), where \( S \) is a finite set of states, \( s_0 \) is the initial state, \( I, O, T \) and \( A \) are finite sets of labels called respectively input, output, tasks and actions, \( \gamma : S \times (I \cup O \cup T \cup A) \rightarrow S \) is a deterministic state transition function. Our definition of IOA is similar to that (Lynch and Tuttle 1988) apart from the fact that we also have transitions that are tasks that can be hierarchically refined. The IOA uses its inputs and outputs to interact with other IOAs and the environment. The semantics of an IOA views inputs as uncontrollable transitions, triggered by messages from the external world, while outputs, tasks, and actions are controllable transitions, freely chosen to drive the dynamics of the modeled system. An output is a message sent to another IOA; an action has some direct effects on the external world. No precondition/effect specifications are needed for actions, since a transition already spells out the applicability conditions and the effects. A task is refined into a collection of actions. We assume all transitions to be deterministic.

We define a state of an IOA as a tuple of internal state variables each of which keeps track of a particular information relevant for that IOA (a representation similar to the one described in Chapter 2 of (Ghallab, Nau, and Traverso 2016)). States are a tuple of state variables’ values, i.e., if \( \{x_1, \ldots, x_k\} \) are the state variables of \( \sigma \), and each has a finite range \( x_i \in D_i \), then the set of states is \( S \subseteq \prod_{i=1}^{k} D_i \), where \( D_i \) is a finite set of values that determine the range of the state variable \( x_i \). We assume that for any state \( s \in S \), all outgoing transitions have the same type, i.e., \( \{u \mid \gamma(s, u) \text{ is defined}\} \) consists solely of either inputs, or outputs, or tasks, or actions. For simplicity we assume \( s \) can have only one outgoing transition if that transition is an output, action or a task. Alternative actions or outputs can be modeled by a state that precedes \( s \) and receives alternative inputs, one of them leading to \( s \).

Figure 1: An IOA \( \sigma_d \) for a spring door. The bold incoming arrows are inputs of \( \sigma_d \) coming from other IOAs or the environment. The outgoing arrows are messages sent by \( \sigma_d \) to other IOAs. The red transition ‘close’ is a command.

Note that despite the assumption that our transition function \( \gamma \) is deterministic, an IOA can model nondeterminism through its inputs. It may receive multiple different inputs at any particular state. These inputs can be messages from external world modeling nondeterministic outcomes of events or commands. For example, a sensing action \( a \) in state \( s \) is a command transition, \( \langle s, a, s' \rangle \); several input transitions from \( s' \) model the possible outcomes of \( a \); these inputs to \( \sigma \) are generated by the external world. A run of an IOA is a sequence \( \langle s_0, s_1, \ldots, s_i, u_i, s_{i+1}, \ldots \rangle \) such that \( s_{i+1} = \gamma(s_i, u_i) \forall i \). It may or may not be finite.

Example 1. The IOA in Figure 1 models a door with a spring-loaded hinge that closes automatically when the door is open and not held. To open the door requires unlatching it, which may not succeed if it is locked. Then it can be opened, unless it is blocked by some obstacle. Whenever the door is left free, the spring closes it (the “close” action in red).

Parallel Composition. Consider a system \( \Sigma = \{ \sigma_1, \ldots, \sigma_n \} \), with each \( \sigma_i \) modeled as an IOA. These components interact by sending output and receiving input messages, while also triggering actions and tasks. The dynamics of \( \Sigma \) can be modeled by the parallel composition of the components, which is a straightforward generalization of the parallel product defined in (Bertoli, Pistore, and Traverso 2010) which is same as the asynchronous product of automata. The parallel composition of two IOAs \( \sigma_1 \) and \( \sigma_2 \) is \( \sigma_1 \parallel \sigma_2 = \langle S_1 \times S_2, (s_1, s_2), I_1 \cup I_2, O_1 \cup O_2, T_1 \cup T_2, A_1 \cup A_2, \gamma \rangle \), where \( \gamma((s_1, s_2), u) = \begin{cases} \gamma_1(s_1, u) \times \{s_2\} & \text{if } u \in I_1 \cup O_1 \cup A_1 \cup T_1, \\ \{s_1\} \times \gamma_2(s_2, u) & \text{if } u \in I_2 \cup O_2 \cup A_2 \cup T_2. \end{cases} \)

By extension, \( \sigma_1 \parallel \sigma_2 \parallel \cdots \parallel \sigma_n \) is the parallel composition of all of the IOAs in \( \Sigma \). The order in which the composition operations is done is unimportant, because parallel composition is associative and commutative.1

We assume the state variables, as well as the input and output labels, are local to each IOA. This avoids potential confusion in the definition of the composed system. It also allows for a robust and flexible design, since components can be modeled independently and added incrementally to a system. However, the components are cooperative in the sense that all of them have a common goal.

If we restrict the \( n \) components of \( \Sigma \) to have no tasks but only inputs, outputs and actions, then driving \( \Sigma \) towards

1Proofs of all of the results stated in this paper are at ⟨http://www.cs.umd.edu/~Epertas/PatraFLAIRS21proofs.pdf⟩.
Hierarchical Refinement. With each task we want to associate a set of methods for hierarchically refining the task into IOAs that can perform the task. This is in principle akin to HTN planning (Erol, Hendler, and Nau 1994), but if the methods refine tasks into IOAs rather than subtasks, they produce a structure that incorporates control constructs such as branches and loops. This structure is like a hierarchical automaton (see, e.g., (Harel 1987)). However, the latter relies on a state hierarchy (a state gets expanded recursively into other automata), whereas in our case the tasks to be refined are transitions. This motivates the following definition.

A refinement method for a task $t$ is a pair $\mu_t = (t, \sigma_t)$, where $\sigma_t$ is an IOA that has both an initial state $s_0\mu_t$ and a finishing state $s_{f\mu_t}$. Unlike tasks in HTN planning (Nau et al. 1999), $t$ is a single symbol rather than a term that takes arguments. Note that $\sigma_t$ may recursively contain other subtasks, which can themselves be refined. Consider an IOA $\sigma = (S, s_0, I, O, T, A, \gamma)$ that has a transition $(s_1, t, s_2)$ in which $t$ is a task. A method $\mu_t = (t, \sigma_t)$ with $\sigma_t = (S_t, s_{0t}, s_{f\mu_t}, I_t, O_t, T_t, A_t, \gamma_t)$ can be used to refine this transition by mapping $s_1$ to $s_{0t}$, $s_2$ to $s_{f\mu_t}$, and $t$ to $\sigma_t$. This produces an IOA $R(\sigma, s_1, t, \rho_t) = (S_{\rho_t}, s_{0\rho_t}, I \cup I_t, O \cup O_t, T \cup T_{\mu_t}) \{t\} \cup A \cup A_{\rho_t} \gamma_{\rho_t})$, where,

\[ S_{\rho_t} = (S_0 \backslash \{s_1, s_2\}) \cup S_{\mu_t}, \]

\[ s_{0\rho_t} = s_0 \text{ if } s_1 \neq s_0, \text{ otherwise, } s_{0\rho_t} = s_{0t}, \]

\[ \gamma_{\rho_t}(s, u) = \begin{cases} \gamma(s, u) & \text{if } s \in S_{\mu_t} \setminus \{s_{0t}, s_{f\mu_t}\}, \\
\gamma_t(s, u) & \text{if } s \in S_t \text{ and } \gamma_t(s, u) = s_1, \\
\gamma_t(s, u) & \text{if } s \in S_t \text{ and } \gamma_t(s, u) = s_2, \\
\gamma(s, u) & \text{if } s \in S_\rho \setminus \{s_1, s_2\} \text{ and } \\
\gamma(s, u) & \notin \{s_1, s_2\}, \\
\gamma_t(s_1, u) \cup \gamma_{t}(s_2, u) & \text{if } s = s_{0\rho_t}, \\
\gamma_t(s_1, u) \cup \gamma_{t}(s_2, u) & \text{if } s = s_{f\mu_t}. \end{cases} \]

Some runs in $\sigma_t$ may be infinite, some other runs may end in a state different from $s_{f\mu_t}$. Note that we don’t require every run to actually end in $s_{f\mu_t}$. Such a requirement would be unrealistic, since the IOA of a method may receive different inputs from other IOA, which cannot be controlled by the method. Intuitively, $s_{f\mu_t}$ represents the “nominal” state in which a run should end, i.e., the nominal path of execution.\(^4\)

Example 2. Figure 2(a) shows an IOA for a robot going through a doorway. It has one task, move and one action, cross. It sends to $\sigma_4$ (Figure 1 the input tree if it gets through the doorway successfully. The move task can be refined using the $\sigma_{\text{move}}$ method in Figure 2(b).

Example 3. Figure 2(b) shows a refinement method for the move task in Figure 2(a). $\sigma_{\text{move}}$ starts with a start_monitor output to activate a monitor IOA that senses the distance to a target. It then triggers the task get_distance to approach the target. From state $v_2$ it receives two possible inputs: close or far. When close, it ends the monitoring activity and terminates in $v_4$, otherwise it gets closer again.

Figure 2(c) shows a method for the monitor task. It waits in state $m_0$ for the input start_monitor, then triggers the sensing action get_distance. In response, the execution platform may return either close or far. In states $v_2$ and $v_3$, the input continue_monitor goes to $m_1$ to sense the distance again, otherwise the input end_monitor goes to the final state $m_7$.

Planning Problem. We are now ready to define the planning problem for this representation. Consider a system modeled by $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ and a finite collection of methods $\mathcal{M}$, such that for every task $t$ in $\Sigma$ or in the methods of $\mathcal{M}$ there is at least one method $\mu_t \in \mathcal{M}$ for task $t$. An instantiation of $(\Sigma, \mathcal{M})$ is obtained by recursively refining every task in the composition $(\sigma_1 \parallel \sigma_2 \parallel \ldots \sigma_n)$ with a method in $\mathcal{M}$, down to primitive actions. Let $(\Sigma, \mathcal{M})^*$ be the set of all possible instantiations of that system, which is enumerable but not necessarily finite. Infinite instances are possible when the body of a method contains the same or another task which can further be refined leading to an infinite chain of refinements. A planning problem is defined as a tuple $P = (\Sigma, \mathcal{M}, S_g)$, where $S_g$ is a set of goal states. Each of the initial components in $\Sigma$ has a set of goal states, and $S_g$ is the Cartesian product of those sets. In other words, the job of the synthesized controller is to make the overall system reach a state such that each component in $\Sigma$ is in one of its goal states. It is solved by finding refinements for tasks in $\Sigma$ with methods in $\mathcal{M}$. In principle this is akin to HTN planning, but we have IOAs that receive inputs from the environment or from other IOAs, thus modelling nondeterminism. We need to control the set of IOAs $\Sigma$ in order to reach (or to try to reach) a goal in $S_g$. For this reason a solution is

\(^2\)goal is represented through a set of states of IOA

\(^3\)If $\sigma$ contains multiple calls to $t$ or $\sigma_t$, contains a recursive call to $t$, the states of $\sigma_t$ must be renamed to avoid ambiguity. This is like standardizing a formula in automated theorem proving.
defined by introducing a hierarchical control structure that drives an instantiation of \((\Sigma, M)\) to meet the goal \(S_g\).

We use the terminology of (Ghalab, Nau, and Traverso 2016, Section 5.2.3). A solution means that some of the runs will reach a goal state. Other runs may never end, or may reach a state that is not a goal state. A solution is safe if all of its finite runs terminate in goal states, and a solution is cyclic or acyclic depending on whether it has cycles.\(^5\)

The hierarchical control structure is a pair \((\Sigma_c, rDict)\) where \(\Sigma_c\) is a set of control automata and \(rDict\) is a task refinement dictionary. A single control automation drives an IOA \(\sigma\) by receiving inputs that are outputs of \(\sigma\) and generating outputs that act as inputs to \(\sigma\). \(\sigma,\sigma\) is the controlled system, i.e., \(\sigma\) controlled by \(\sigma_c\). The definition of controlled system is similar to the one in (Ghalab, Nau, and Traverso 2016, Section 5.8). \(rDict\) is a dictionary which should have as its keys all of the tasks in \(\Sigma\) and its refinement. \(rDict[t]\) is a method which should be used to refine task \(t\) in order to achieve \(S_g\). So, \(rDict\) uniquely defines an instantiation of \((\Sigma, M)\). Finally, \(\Sigma\) is controlled by \((\Sigma_c, rDict)\), and the hierarchical control system \(\phi_\Sigma = (\Sigma_c, rDict)\phi(\Sigma, M)\) will have one of the following forms:

\[
\begin{align*}
\sigma_c &\triangleright (\phi_1 \parallel \phi_2), \text{ where } \sigma_c \in \Sigma_c \text{ and } \phi_1, \phi_2 \text{ are IOAs;} \\
\sigma_c &\triangleright R((\phi_3, s, \mu_1), \text{ where } \sigma_c \in \Sigma_c, rDict[t] = \sigma_\mu, \\
\phi_3 &\text{ is an IOA and } t \text{ is a task in state } s; \\
\sigma_c &\triangleright \sigma, \text{ where } \sigma_c \in \Sigma_c \text{ and } \sigma \in \Sigma.
\end{align*}
\]

Above, \(\phi_1, \phi_2\) and \(\phi_3\) are hierarchical controlled systems. The form it will depend on the ordering of parallel and hierarchical composition chosen by MakeCntrlStruct to synthesize the controller (see Section 3).

**Example 4.** The IOA on the bottom in Figure 3 is a control automaton for the IOAs in Figures 1 and 2(a). This control automaton is for the system when the move task has not been refined. The IOA on the top controls the refined robot IOA in Figure 2(b) and the monitor IOA in Figure 2(c).

### 3 Solving Planning Problems

The algorithm MakeCntrlStruct (Table 1(a)) solves a planning problem \((\Sigma, M, S_g)\) where \(\Sigma\) is a set of IOAs, \(M\) is a set of methods for refining tasks and \(S_g\) is a set of goal states. The solution is a set of control automata, \(\Sigma_c\) and a task refinement dictionary, \(rDict\) such that \(\Sigma\) driven by \(\Sigma_c\) and refined following \(rDict\) reaches the desired goals states, \(S_g\). Depending on how one of its subroutines is configured, MakeCntrlStruct can search either for acyclic safe solutions, or for safe solutions that may contain cycles.

Before getting into the details of how MakeCntrlStruct works, we need to discuss a property on which it depends. Given a planning problem, MakeCntrlStruct constructs a solution by doing a sequence of parallel composition and refinement operations. The following theorem shows that composition and refinement can be done in either order to produce the same result:

\[\text{Theorem 1 (distributivity). Let } \sigma_1, \sigma_2 \text{ be IOAs, } \langle s_1, t, s_2 \rangle \text{ be a transition in } \sigma_1, \text{ and } \mu_t = (t, \sigma_\mu) \text{ be a refinement method for } t. \text{ Then } R(\sigma_1, s_1, \mu_t) \parallel \sigma_2 = R(\sigma_1 \parallel \sigma_2, s_1^*, \mu_t), \text{ where } s_1^* = \{s_1, s\}|s \in S_g.\]

Thus the algorithm can choose the order in which to do those operations (line (*) in Table 1(a)), which is useful because the order affects the size of the search space.

**Algorithm.** Table 1(a) shows our algorithm for synthesizing hierarchical control structures using planning. It does a sequence of parallel and hierarchical compositions of the IOAs in \(\Sigma\) until there are no more unrefined tasks and all pairs of interacting components have been composed.

As discussed in the previous section, \((\Sigma, M)^*\) is the set of all possible instantiations of our system, which is enumerable but not necessarily finite. Among this set, some instantiations are desirable with respect to our goal. The while loop in MakeCntrlStruct implicitly constructs an instantiation of \((\Sigma, M)\) by doing a series of parallel and hierarchical compositions. In each iteration of the loop the algorithm makes the choice of whether to do a parallel composition or a refinement. The size of the search space depends on the order in which the choices are made. In an implementation, the choice would be made heuristically. We believe some of the heuristics will be analogous to constraint-satisfaction heuristics (Dechter 2003). The while loop exits when the implicit instantiation of \((\Sigma, M)\) is complete, i.e., there are no more tasks to refine, and all interactions between pairs of IOAs have been taken into account through parallel composition.

When MakeCntrlStruct chooses to compose, it uses the MakeCntrlAutomaton subroutine to create a control automation \(\sigma_{cij}\) for a pair of IOAs \(\sigma_i\) and \(\sigma_j\) which interact with each other. \(\sigma_i\) and \(\sigma_j\) are randomly selected from \(\Sigma\). We do not include pseudocode for MakeCntrlAutomaton, because it may be any of several planning algorithms published elsewhere. For example, the algorithm in (Bertoli, Pistore, and Traverso 2010) will generate an acyclic safe solution if one exists, and (Bertoli, Pistore, and Traverso 2010) discusses how to modify that algorithm so that it will find safe solutions that aren’t restricted to be acyclic. Several of the algorithms in (Ghalab, Nau, and Traverso 2016, Chapter 5) could also be used.

---

\(^5\)In (Cimatti et al. 2003), a weak solution is what we call a solution, a strong cyclic solution is what we call a safe solution, and a strong solution is what we call an acyclic safe solution.
4 Related Work

To our knowledge, there is no previous formalism for the synthesis of hierarchical distributed controllers for coordinating multiple agents.

(Chavazadeh, Mahadevan, and Makar 2006), (Osenowski and Mahadevan 2010) and (Jong, Hester, and Stone 2008) use the notion of hierarchy for multi-agent reinforcement learning. These works allow for a hierarchical representation of the target plan, to be executed in a collaborative manner. In our framework, the hierarchical representation is in the agent itself; the synthesized controllers coordinate interactions among hierarchical agents.

(Atkin et al. 2001) proposes a Hierarchical Agent Control Architecture (HAC) with a hierarchical representation of actions, sensors, and goals. HAC includes a least-commitment partial hierarchical planner, relying on plan skeletons. Given a set of goals, plans are retrieved, simulated, and executed. HAC combines hierarchical planning with reasoning by procedural knowledge. Our approach is different since we allow for reasoning about alternative refinements of tasks through the automated synthesis of controllers.

Hierarchical and procedure-based frameworks have been used in robotic systems, e.g., PRS (Ingrand et al. 1996), RAP (Firby 1987), TCA (Simmons and Apfelbaum 1998), XFRM (Beetz and McDermott 1994), and the survey of (Ingrand and Ghalib 2014). These approaches propose reactive systems, but none of them is based on a formal account with the synthesis techniques provided in this paper.

(Hu and Feijs 2003) describes an agent-based architecture for networked devices, where each agent has a controller. However, the controller does not control inter-agent communication, and no synthesis of interactions is provided.

Hierarchical planning formalisms (including angelic hierarchical planning (Marthi, Russell, and Wolfe 2007) and its extension (Marthi, Russell, and Wolfe 2008), (Kuter et al. 2009)) do not represent agents that interact together and with the external environment. The hierarchical framework proposed in (Shivashankar et al. 2012) refines goals instead of tasks; no synthesis of controllers is provided.

Our approach shares some similarities with the hierarchical state machines of (Harel 1987), which have been used for the specification and verification of reactive systems. We rely on the theory of input/output automata (Lynch and Tuttle 1988), which has been used to specify distributed discrete event systems, and to formalize and analyze communication and concurrent algorithms. The work in (Kessler et al. 2004) is based on hierarchical state machines, however no automated synthesis is provided. There is also a vast amount of literature on controllers for discrete-event systems, e.g., (Wong and Wonnham 1996; Mohajerani et al. 2011). All these works focus on the verification rather than on the synthesis of hierarchical agents through input/output automata.

I/O automata have also been used to formalize non-hierarchical interactions of web services and to plan for their composition (Pistore, Traverso, and Bertoli 2005; Bertoli, Pistore, and Traverso 2010). Our work is also different from the work in (Bucchiaroni et al. 2012; 2013), where abstract actions are represented with goals, and online planning is used to generate interacting processes that satisfy such goals.
(David et al. 2010) proposes a theoretical framework for the specification of real-time systems using timed I/O automata. It provides constructs for refinement, consistence checking, logical and structural composition, and quotient of specifications. We do not represent and reason explicitly about time. However, the main difference with our work is that (David et al. 2010) is a framework for the specification, design, and verification of timed I/O automata, while we address a synthesis problem by generating automatically a control automaton. The same difference holds with (Gundersen et al. 2018), which integrates conformance testing into a framework for model checking.

Our contribution builds on (Ghallab, Nau, and Traverso 2016, Section 5.8), where a system is defined as the parallel composition of automata $\sigma_1 \parallel \ldots \parallel \sigma_n$, describing the possible evolutions of its $n$ components. A planner for such a system synthesizes a control automaton that interacts with the $\sigma_i$’s to drive them to specified goals. The approach is shown to be solvable with nondeterministic planning algorithms. It is however limited to flat nonhierarchical automata.

5 Future Work and Conclusion

We described a formalism for synthesizing hierarchical control structure for systems that are composed of communicating components. Components are represented as I/O automata that support parallel composition and task refinement. The synthesized plans have rich control constructs such as conditional and iterative plans. We describe a novel planning algorithm for synthesizing such controllers.

We believe this work provides the basis for the online synthesis of real-time systems, e.g., for web services, automation of large physical facilities such as warehouses or harbors, etc. In our future work, we intend to implement our algorithm and test it on representative problems from such problem domains. For that purpose, an important topic of future work will be to extend our algorithm for use in continual online planning. This should be straightforward, since our acting algorithm already synthesizes the control structure online. As another topic for future work, recall that Theorem 1 (Distributivity) shows that parallel and hierarchical composition operations can be done in either order and produce the same result. The size of the planner’s search space depends on the order in which these operations are done, and we want to develop heuristics for choosing the best order.

Acknowledgements. This work has been supported in part by NRL grants N0017320P0399 and N00173191G001, and ONR grant N00142012257. The information in this paper does not necessarily reflect the position or policy of the fun-
ders, and no official endorsement should be inferred.

References


Cimatti; Pistore; Roveri; and Traverso. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. Artif. Intell. 147(1-2).


Wurman, P. 2014. How to coordinate a thousand robots (invited talk). In ICAPS.