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* σ 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* σ_c to manage the interactions among the agents in Σ. The agents don't send messages to each other directly, but instead send them to σ_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 σ_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 IOAs, and propose a formalization of planning and acting problems for interacting agents in this original framework.

- We provide theorems about the main properties of this class of automata. In particular, the operations of parallel composition and refinement are *distributive*. 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 automata (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 σ, 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.

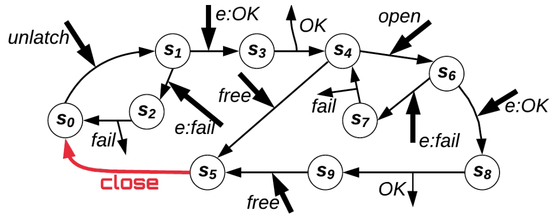

Figure 1: An IOA σ_d for a spring door. The bold incoming arrows are inputs of σ_d coming from other IOAs or the environment. The outgoing arrows are messages sent by σ_d to other IOAs. The red transition 'close' is a command.

Note that despite the assumption that our transition function γ 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 σ are generated by the external world. A *run* of an IOA is a sequence $\langle s_0, u_0, \ldots, s_i, u_i, s_{i+1}, \ldots \rangle$ 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 σ_i modeled as an IOA. These components interact by sending output and receiving input messages, while also triggering actions and tasks. The dynamics of Σ 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 σ_1 and σ_2 is $\sigma_1 \| \sigma_2 = \langle S_1 \times S_2, (s_{0_1}, s_{0_2}), I_1 \cup I_2, O_1 \cup O_2, T_1 \cup T_2, A_1 \cup A_2, \gamma \rangle$, 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, $\sigma_1 \| \sigma_2 \| \sigma_3 \| \ldots \| \sigma_n$ is the parallel composition of all of the IOAs in Σ. 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 Σ to have *no tasks* but only inputs, outputs and actions, then driving Σ towards

[1]Proofs of all of the results stated in this paper are at ⟨http://www.cs.umd.edu/%7Epatras/PatraFLAIRS21proofs.pdf⟩.

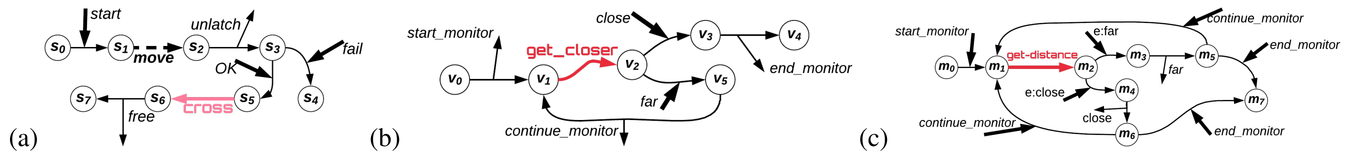

Figure 2: (a): An IOA for a robot going through a doorway. (b): The IOA σ_{move} of a method for the move task. (c): The IOA $\sigma_{monitor}$ of a monitoring method.

a set of *goal* [2] states can be addressed with a nondeterministic planning algorithm for the synthesis of a control automaton σ_c that interacts with the parallel composition $\sigma_1 \parallel \sigma_2 \parallel \sigma_3 \parallel \ldots \parallel \sigma_n$ of the automata in Σ. The control automaton's inputs are the outputs of Σ and its outputs are inputs of Σ. Several algorithms are available to synthesize such control automata, e.g., (Bertoli, Pistore, and Traverso 2010). But in this paper, we also allow the components to have hierarchy within themselves and we generate a hierarchical control structure.

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 $\mu_t = \langle t, \sigma_\mu \rangle$, where σ_μ is an IOA that has both an initial state $s_{0\mu}$ and a *finishing* state $s_{f\mu}$. Unlike tasks in HTN planning (Nau et al. 1999), t is a single symbol rather than a term that takes arguments. Note that σ_μ may recursively contain other subtasks, which can themselves be refined. Consider an IOA $\sigma = \langle S, s_0, I, O, T, A, \gamma \rangle$ that has a transition $\langle s_1, t, s_2 \rangle$ in which t is a task. A method $\mu_t = \langle t, \sigma_\mu \rangle$ with $\sigma_\mu = \langle S_\mu, s_{0\mu}, s_{f\mu}, I_\mu, O_\mu, T_\mu, A_\mu, \gamma_\mu \rangle$ can be used to *refine* this transition by mapping s_1 to $s_{0\mu}$, s_2 to $s_{f\mu}$ and t to σ_t.[3] This produces an IOA, $\mathfrak{R}(\sigma, s_1, \mu_t) = \langle S_\mathfrak{R},\ s_{0\mathfrak{R}},\ I \cup I_\mu,\ O \cup O_\mu,\ T \cup T_\mu \setminus \{t\},\ A \cup A_\mu,\ \gamma_\mathfrak{R} \rangle$, where,

$S_\mathfrak{R} = (S \setminus \{s_1, s_2\}) \cup S_\mu$,

$s_{0\mathfrak{R}} = s_0$ if $s_1 \neq s_0$, otherwise, $s_{0\mathfrak{R}} = s_{0\mu}$,

$$\gamma_\mathfrak{R}(s,u) = \begin{cases} \gamma_\mu(s,u) & \text{if } s \in S_\mu \setminus \{s_{0\mu}, s_{f\mu}\}, \\ s_{0\mu} & \text{if } s \in S \text{ and } \gamma(s,u) = s_1, \\ s_{f\mu} & \text{if } s \in S \text{ and } \gamma(s,u) = s_2, \\ \gamma(s,u) & \text{if } s \in S \setminus \{s_1, s_2\} \text{ and} \\ & \gamma(s,u) \notin \{s_1, s_2\}, \\ \gamma(s_1,u) \cup \gamma_\mu(s,u) & \text{if } s = s_{0\mu}, \\ \gamma(s_2,u) \cup \gamma_\mu(s,u) & \text{if } s = s_{f\mu}. \end{cases}$$

Some runs in σ_μ may be infinite, some other runs may end in a state different from $s_{f\mu}$. Note that we don't require every run to actually end in $s_{f\mu}$. 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, $s_{f\mu}$ 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 σ_d (Figure 1 the input* free *if it gets through the doorway successfully. The* move *task can be refined using the σ_{move} method in Figure 2(b).*

Example 3. *Figure 2(b) shows a refinement method for the* move *task in Figure 2(a). σ_{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_closer *to approach the target. From state v_2 it receives two possible inputs:* close *or* far*. When close, it ends the monitor 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* far *or* close*. In states m_5 and m_6, 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 Σ 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 $(\sigma_1 \parallel \sigma_2 \parallel ...\sigma_n)$ 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 $P = \langle \Sigma, \mathcal{M}, S_g \rangle$, where S_g is a set of goal states. Each of the initial components in Σ 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 Σ is in one of its goal states. It is solved by finding refinements for tasks in Σ 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 Σ 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 σ contains multiple calls to t or σ_μ contains a recursive call to t, the states of σ_μ must be renamed to avoid ambiguity. This is like *standardizing* a formula in automated theorem proving.

[4] Alternatively, we may assume we have only runs that terminate, and a set of finishing states $S_{f\mu}$. We simply add a transition from every element in $S_{f\mu}$ to the *nominal* finishing state $s_{f\mu}$.

defined by introducing a *hierarchical control structure* that drives an instantiation of $(\Sigma, \mathcal{M})$ to meet the goal S_g.

We use the terminology of (Ghallab, 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 $\langle \Sigma_c, \textit{rDict} \rangle$ where Σ_c is a set of control automata and *rDict* is a task refinement dictionary. A single control automaton drives an IOA σ by receiving inputs that are outputs of σ and generating outputs that act as inputs to σ. $\sigma_c \triangleright \sigma$ is the controlled system, i.e., σ controlled by σ_c. The definition of controlled system is similar to the one in (Ghallab, Nau, and Traverso 2016, Section 5.8). *rDict* is a dictionary which should have as its keys all of the tasks in Σ 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 $(\Sigma, \mathcal{M})$. Finally, Σ is controlled by $\langle \Sigma_c, \textit{rDict} \rangle$, and the *hierarchical controlled system* $\boldsymbol{\phi_s} = \langle \Sigma_c, \textit{rDict} \rangle \triangleright (\Sigma, \mathcal{M})$ will have one of the following forms:

$$\sigma_c \triangleright (\boldsymbol{\phi_1} \parallel \boldsymbol{\phi_2}), \text{ where } \sigma_c \in \Sigma_c \text{ and } \boldsymbol{\phi_1}, \boldsymbol{\phi_2} \text{ are IOAs;}$$
$$\sigma_c \triangleright \mathfrak{R}(\boldsymbol{\phi_3}, s, \mu_t), \text{ where } \sigma_c \in \Sigma_c, \textit{rDict}[t] = \sigma_\mu,$$
$$\boldsymbol{\phi_3} \text{ is an IOA and } t \text{ is a task in state } s;$$
$$\sigma_c \triangleright \sigma, \text{ where } \sigma_c \in \Sigma_c \text{ and } \sigma \in \Sigma.$$

Above, $\boldsymbol{\phi_1}, \boldsymbol{\phi_2}$ and $\boldsymbol{\phi_3}$ are hierarchical controlled systems. The form it will have depends 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 $\langle \Sigma, \mathcal{M}, S_g \rangle$ where Σ is a set of IOAs, $\mathcal{M}$ 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, Σ_c and a task refinement dictionary, *rDict* such that Σ driven by Σ_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:

[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.

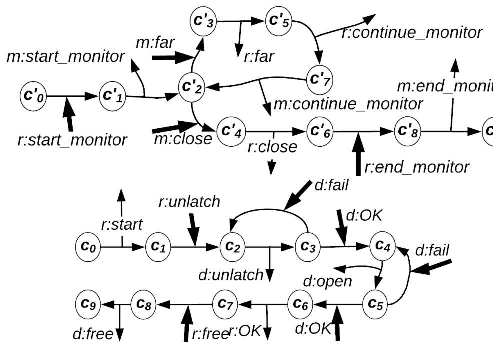

Figure 3: A hierarchical control structure for the 'door' (Figure 1), refined 'robot' for going through a doorway (Figures 2(a) and 2(b)), and the 'monitor' (Figure 2(c)). The inputs and outputs of the robot, door and monitor are preceded with *r:*, *d:* and *m:* respectively.

Theorem 1 (distributivity)**.** *Let σ_1, σ_2 be IOAs, $\langle s_1, t, s_2 \rangle$ be a transition in σ_1, and $\mu_t = \langle t, \sigma_\mu \rangle$ be a refinement method for t. Then $\mathfrak{R}(\sigma_1, s_1, \mu_t) \parallel \sigma_2 = \mathfrak{R}(\sigma_1 \parallel \sigma_2, s_1^*, \mu_t)$, where $s_1^* = \{(s_1, s) | s \in S_{\sigma_2}\}$.*

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 Σ until there are no more unrefined tasks and all pairs of interacting components have been composed.

As discussed in the previous section, $(\Sigma, \mathcal{M})^*$ 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 $(\Sigma, \mathcal{M})$ 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 $(\Sigma, \mathcal{M})$ 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 automaton $\sigma_{c_{ij}}$ for a pair of IOAs σ_i and σ_j which interact with each other. σ_i and σ_j are randomly selected from Σ. 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 (Ghallab, Nau, and Traverso 2016, Chapter 5) could also be used.

```
MakeCntrlStruct ($\Sigma_0, \mathcal{M}, S_g$)                         ▷ (a)
  $\Sigma \leftarrow \Sigma_0$; $\Sigma_c \leftarrow \emptyset$; rDict ← empty dictionary
  while (there are unrefined tasks in $\Sigma$ or $|\Sigma| > 1$):
      nondeterministically choose
          which-first $\in \{compose, refine\}$ (*)
      if (which-first = compose):
          select $\sigma_i, \sigma_j \in \Sigma$ and remove them
          $\sigma_{c_{ij}} \leftarrow$ MakeCntrlAutomaton ($\sigma_i \parallel \sigma_j, S_g$)
          if $\sigma_{c_{ij}}$ is a failure, then return failure
          else: $\Sigma_c \leftarrow \Sigma_c \cup \{\sigma_{c_{ij}}\}$
                $\Sigma \leftarrow \Sigma \cup \{\sigma_{c_{ij}} \triangleright (\sigma_i \parallel \sigma_j)\}$
      else if (which-first = refine):
          select $\sigma \in \Sigma$ which has task $t$
          (transition $\langle s_1, t, s_2 \rangle$) and remove it
          nondeterministically choose $\mu_t \in M$ to refine $t$
          $t_{new} \leftarrow$ unique new name for $t$
          rDict[$t_{new}$] $\leftarrow \sigma_\mu$; $\Sigma \leftarrow \Sigma \cup \Re(\sigma, s_1, \mu_t)$
  return $\langle \Sigma_c$, rDict$\rangle$
```

```
ControlledActingWithIOAs ($\Sigma, M, S_g$)                         ▷ (b)
  $\langle \Sigma_c$, rDict$\rangle \leftarrow$ MakeCntrlStruct($\Sigma, M, S_g$)
  for $\sigma \in \Sigma_c \cup \Sigma$ : do ExecuteAsync($\sigma$, rDict, $S_g$)

ExecuteAsync($\sigma$, rDict, $S_g$)
  $s \leftarrow$ initial state of $\sigma$
  while $s$ is not final and $s \notin S_g$ do
      $\langle s, a, s' \rangle \leftarrow$ transition coming out of $s$
      switch (type($a$)):
          case input: $a \leftarrow$ ReceiveInput( )
          case output: GenerateOutput($a$)
          case command: ExecuteCommand($a$)
          case task: $\sigma_\mu \leftarrow$ rDict[a]; ExecuteSync($\sigma_\mu$, rDict, $S_g$)
      $s \leftarrow \gamma(s, a)$
  if $s \in S_g$ then return Success else return Failure
```

Table 1: (a): Pseudocode for the controller synthesis algorithm. (b): Pseudocode for running IOAs with a synthesized hierarchical controlled structure.

If MakeCntrlAutomaton succeeds, we include $\sigma_{c_{ij}}$ in our set of solution control automata, Σ_c and add the controlled system, $\sigma_{c_{ij}} \triangleright (\sigma_i \parallel \sigma_j)$ to Σ. Otherwise, we fail and terminate this nondeterministic branch. Note that, we could allow new components to enter the system at this stage as follows. Instead of selecting σ_j randomly from Σ, we could lookup the components that interact with σ_i select σ_j from them. This simple extension allows new agents to join in at any stage of the synthesis without compromising correctness.

When MakeCntrlStruct chooses to *refine*, it chooses a refinement method μ_t selected from $\mathcal{M}$ to refine t. The task refinement dictionary *rDict* maps every instance of all tasks present in Σ to the body of the most optimal refinement method for them. So, we add σ_μ (the body of method μ_t) to the task refinement dictionary *rDict* with key t_{new}. Notice that we rename the task t to t_{new} to identify every instance of task t uniquely. Then, we add the resulting IOA, after doing the refinement, to Σ and continue the loop.

MakeCntrlStruct is sound and complete (see footnote 1 for proof). Completeness guarantees that we find the hierarchical control structure when it exists, but does not guarantee that our algorithm will terminate or return "no" when there is no control structure for the problem.

4 Related Work

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

(Ghavamzadeh, Mahadevan, and Makar 2006), (Osentoski 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 Ghallab 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 analyse 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 Wonham 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 (Bucchiarone 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, cosistency 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 $\sigma_1\| \dots \|\sigma_n$, describing the possible evolutions of its n components. A planner for such a system synthesizes a control automaton that interacts with the σ_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 N000142012257. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

References

Atkin, M.; King, G.; Westbrook, D.; Heeringa, B.; and Cohen, P. 2001. Hierarchical agent control: A framework for defining agent behavior. In *AAMAS*.

Beetz, M., and McDermott, D. 1994. Improving robot plans during their execution. In *AIPS*.

Bertoli; Pistore; and Traverso. 2010. Automated composition of Web services via planning in asynchronous domains. *Artif. Intel.* 174(3-4).

Boese, F., and Piotrowski, J. 2009. Autonomously controlled storage management in vehicle logistics applications of RFID and mobile computing systems. *Intl. J. RF Tech: Res. and Appl.*

Bucchiarone, A.; Marconi, A.; Pistore, M.; and Raik, H. 2012. Dynamic adaptation of fragment-based and context-aware business processes. In *ICWS*.

Bucchiarone, A.; Marconi, A.; Pistore, M.; Traverso, P.; Bertoli, P.; and Kazhamiakin, R. 2013. Domain objects for continuous context-aware adaptation of service-based systems. In *ICWS*.

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

D'Andrea, R. 2012. A revolution in the warehouse: A retrospective on Kiva Systems and the grand challenges ahead. *IEEE Trans. Automation Sci. and Engr.* 9(4):638–639.

David, A.; Larsen, K. G.; Legay, A.; Nyman, U.; and Wasowski, A. 2010. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC 2010*. ACM.

Dechter, R. 2003. *Constraint Processing*.

Erol, K.; Hendler, J.; and Nau, D. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. In *AIPS*, 249–254.

Firby, R. 1987. An investigation into reactive planning in complex domains. In *AAAI*, 202–206.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*.

Ghavamzadeh, M.; Mahadevan, S.; and Makar, R. 2006. Hierarchical multi-agent reinforcement learning. *AAMAS* 13(2).

Gundersen, T. R.; Lorber, F.; Nyman, U.; and Ovesen, C. 2018. Effortless fault localisation: Conformance testing of real-time systems in ecdar. In *GandALF 2018*, volume 277 of *EPTCS*, 147–160.

Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. of Comput. Prog.* 8(3).

Hu, J., and Feijs, L. 2003. An agent-based architecture for distributed interfaces and timed media in a storytelling application. In *AAMAS*.

Ingrand, F., and Ghallab, M. 2014. Deliberation for autonomous robots: A survey. *Artif. Intel.*

Ingrand, F.; Chatilla, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*.

Jong, N. K.; Hester, T.; and Stone, P. 2008. The utility of temporal abstraction in reinforcement learning. In *AAMAS*, 299–306.

Kessler, R.; Griss, M.; Remick, B.; and Delucchi, R. 2004. A hierarchical state machine using jade behaviours with animation visualization. In *AAMAS*.

Kuter, U.; Nau, D.; Pistore, M.; and Traverso, P. 2009. Task decomposition on abstract states, for planning under nondeterminism. *Artif. Intel.* 173.

Lynch, N., and Tuttle, M. 1988. An introduction to input output automata. *CWI Quarterly*.

Marthi, B.; Russell, S.; and Wolfe, J. 2007. Angelic semantics for high-level actions. In *ICAPS*.

Marthi, B.; Russell, S.; and Wolfe, J. 2008. Angelic hierarchical planning: Optimal and online algorithms. In *ICAPS*, 222–231.

Mohajerani, S.; Malik, R.; Ware, S.; and Fabian, M. 2011. Compositional synthesis of discrete event systems using synthesis abstraction. In *Chinese Control and Decision Conf.*, 1549–1554.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *IJCAI*.

Osentoski, S., and Mahadevan, S. 2010. Basis function construction for hierarchical reinforcement learning. In *AAMAS*, 747–754.

Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated composition of web services by planning in asynchronous domains. In *ICAPS*.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*.

Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *IROS*.

Wong, K., and Wonham, W. M. 1996. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems* 6(3):241–273.

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