Planning and Acting with Hierarchical Input/Output Automata

Sunandita Patra[1], Paolo Traverso[2], Malik Ghallab[3] and Dana Nau[1]
[1]University of Maryland, College Park, MD 20742 USA, {patras, nau}@cs.umd.edu
[2]Center for Information Technology - IRST (FBK-ICT), Trento, Italy, traverso@fbk.eu
[3]LAAS-CNRS, 7, Avenue du Colonel Roche, 31077, Toulouse, France, malik@laas.fr

Abstract

This paper introduces an original framework for planning and acting with hierarchical input/output automata for systems defined by the parallel composition of the models of their components. Typical applications are, for example, in harbor or warehouse automation. The framework extends the usual parallel composition operation of I/O automata with a hierarchical composition operation that can refine a task. It defines planning as the synthesis of a control component to drive, through I/O interactions and task refinement, the system toward desired states. A new nondeterministic algorithm performs this synthesis. We tackle these issues on a theoretical basis. We formally define the representation and prove that the two operations of parallel and hierarchical composition are *distributive*, which is essential for the correctness and completeness of the proposed planning algorithm.

Motivation

This paper introduces a knowledge representation framework and an approach for planning and acting for component-based interacting systems. We conform to the position of (Ghallab, Nau, and Traverso 2014) that planning and acting have to be tightly integrated following two principles of a *hierarchical* and *continual* online deliberation. In this view, planned actions are refined in a context-dependent way into executable commands, which drive a system toward desired objectives. We argue here that distributed domains add the requirement of a third principle: *interaction* between communicating components. The distribution considered here is not at the planning level. It is specifically focused on systems defined by the composition of the models of their components. We first explain our motivation.

A planning domain is usually modeled through a state-transition system Σ. Σ is almost never given extensively, but part of it is generated, along with the search, following the specification of what an actor can do: its actions define possible transitions in Σ. There is however another highly expressive and practical means of generating a very large state-transition system: through the *composition* of the transition models of the elementary components that constitute the entire system. Such a design method is more natural and appropriate when the system of interest is not centralized into a unique platform but distributed over numerous components. These components have their own local sensors and actuators, evolve concurrently, may be designed independently and can be assembled by model composition in modular and possibly changing manners.

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, however, 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, dealing with contingencies, local failures, modular design and easier novel deployments, would model each component (robot, shelf, refill and order preparation stations, etc.) through its possible interactions with the rest of the system. An automatically synthesized controller coordinates these interactions.

The composition approach has been in use for a long time in the area of system specification and verification, e.g., (Harel 1987). Although less popular, it has also been developed 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 a harbor or a large infrastructure (Boese and Piotrowski. 2009). The state-transition system of a component, defined with the usual action schema, evolves in its local states by *interacting* with other components, i.e., by sending and receiving messages along state transitions. Planning consists of deciding which messages to send to which components and when in order to drive the entire system toward desired states.

Such a problem can be formalized with input/output automata. Planning for a system σ means generating a

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

control automaton σ_c that receives the output of σ and sends input to σ such that the behavior of the controlled pair, σ and σ_c, drives σ toward goal states. A system composed of multiple components is defined by the parallel composition of their automata $\sigma_1 \| \ldots \| \sigma_n$, which describes all the possible evolutions of the n components. A planner for that system synthesizes a control automaton that interacts with the n σ_i's such that the system reaches certain goal states. The approach is described in (Ghallab, Nau, and Traverso 2016, Section 5.8) for the purpose of performing refinements at the acting level; it is shown to be solvable with nondeterministic planning algorithms.

In order to address planning and acting in a uniform framework, we propose to further extend this representation. We augment the parallel composition operation, used for the composition of the component models, with a hierarchical *task refinement* operation. We call the task refinement operation as hierarchical composition. We formalize planning for distributed interacting systems in a new framework of hierarchical input/output automata. The synthesis of control automaton is done with a new nondeterministic planning algorithm.

The preceding issues, being novel in the field, required to be initially tackled at a theoretical basis, which is developed in this paper (no application nor experimental results are reported). Our contributions are the following:

- We formally define the notion of refinement for hierarchical communicating input/output automata, and propose a formalization of planning and acting problems for component-based interacting systems in this original framework.
- We prove the essential properties of this class of formal machines, in particular that the operations of parallel composition and refinement are *distributive*, a critical feature needed for handling this representation, the proof of which required extensive developments.
- Distributivity allows us to show that the synthesis of a controller for a set of hierarchical communicating input/output automata can be addressed as a nondeterministic planning problem.
- We propose a new algorithm for solving that problem, and discuss its theoretical properties.

The rest of the paper presents the proposed representation and its properties, the algorithm for synthesis of control automaton through planning, a discussion of the state of the art, and concluding remarks.

Representation

The proposed knowledge representation relies on a class of automata endowed with composition and refinement operations.

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*, γ is a deterministic *state transition function.* The IOA uses its *inputs* and *outputs* to communicate 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.

States are defined as tuples 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 $S \subseteq \prod_{i=1,k} D_i$. 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 or an action. Alternative actions or outputs can be modeled by a state that precedes s and receives alternative inputs, one of them leading to s.

Note that despite the assumption of deterministic transitions, an IOA σ models queries to the external world that can have nondeterminism outcomes through its inputs. For example, a sensing action a in state s is a 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 $s_{i+1} = \gamma(s_i, u_i)$ for every 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 shown in red).*

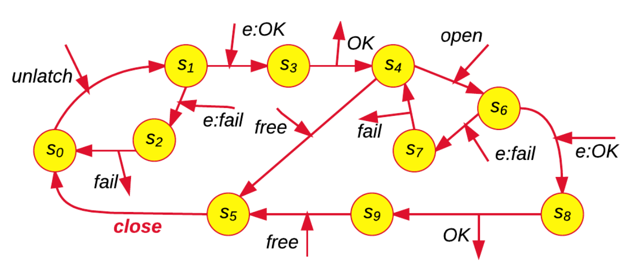

Figure 1: A simple $\sigma_{\text{spring-door}}$ model.

Parallel Composition. Consider a system consisting of n components $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, 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.

If we restrict the n components of Σ to have *no tasks* but only inputs, outputs and actions, then driving Σ towards a set of *goal* states can be addressed with a nondeterministic planning algorithm for the synthesis of a control automaton σ_c that interacts with the parallel composition $\sigma_1 \| \sigma_2 \| \sigma_3 \| \ldots \| \sigma_n$ 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).

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 definitions.

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.[2] 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

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

$$s_{0\mathfrak{R}} = \begin{cases} s_0 & \text{if } s_1 \neq s_0, \\ s_{0\mu} & \text{otherwise}, \end{cases}$$

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

Note that we do not require every run in σ_μ to actually end in $s_{f\mu}$. Some runs may be infinite, some other runs may end in a state different from $s_{f\mu}$. 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.[3]

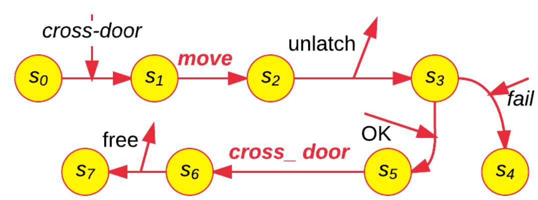

Figure 2: An IOA for a robot going through a doorway.

Example 2. *Figure 2 shows an IOA for a robot going through a doorway. It has two tasks:* move *and* cross_door*. It sends to $\sigma_{spring\text{-}door}$ the input* free *if it gets through the doorway successfully. The* move *task can be refined using the σ_{move} method in Figure 3.*

[1] The proof involves showing that every run of $\sigma_1 \| \sigma_2$ is a run of $\sigma_2 \| \sigma_1$ and vice-versa. Here is a link to the all the proofs: https://umd.box.com/s/kxhikeu9lklxdwbw1c83f6orly0oc38k

[2] As a special case, if σ contains multiple calls to t or σ_μ contains a recursive call to t, then the states of σ_μ must first be renamed, in order to avoid ambiguity. This is analogous to *standardizing* a formula in automated theorem proving.

[3] 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 $s_{f\mu}$.

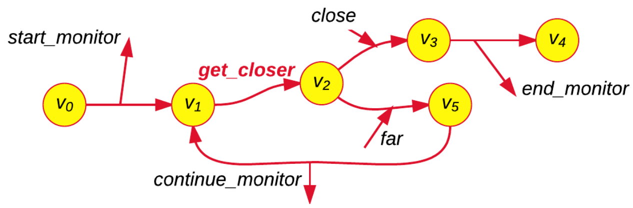

Figure 3: The IOA σ_{move} of a method for the move task.

Example 3. *Figure 3 shows the IOA of a refinement method for the* move *task in Figure 2. 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 4 shows the IOA of 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 *transitions to m_1 to sense the distance again, otherwise the input* end_monitor *transitions to the final state m_7.*

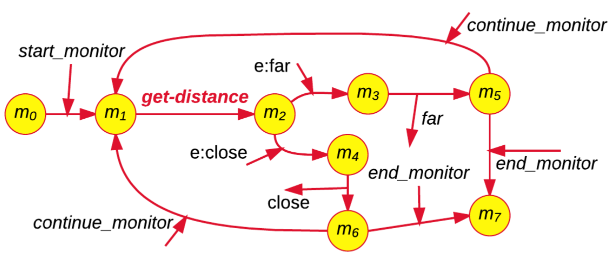

Figure 4: The IOA $\sigma_{monitor}$ of a monitoring method.

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 $(\sigma_1 \parallel \sigma_2 \parallel \ldots \sigma_n)$ 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 $P = \langle \Sigma, \mathcal{M}, S_g \rangle$, where S_g is a set of goal states. It is solved by finding refinements for tasks in Σ with methods in $\mathcal{M}$. We mentioned earlier that this is in principle akin to HTN planning. However, here we have IOAs that receive inputs from the environment or from other IOAs, thus modelling non-

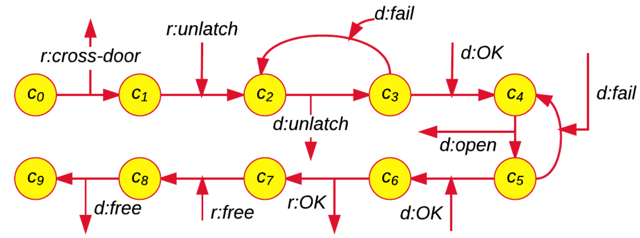

Figure 5: Control automaton for the IOAs in Figures 1 and 2. The inputs and outputs of the robot are preceded with *r:* and the inputs and outputs to the door are preceded by *d:*

determinism. 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 defined by introducing a *control automaton* that drives an instantiation of $(\Sigma, \mathcal{M})$ to meet the goal S_g. A control automaton drives an IOA σ by receiving inputs that are outputs of σ and generating outputs which act as inputs to σ. (Ghallab, Nau, and Traverso 2016, Section 5.8)

Let σ_c be a control automaton for an IOA σ_f which is an instantiation of $(\Sigma, \mathcal{M})$, i.e., the inputs of σ_c are the outputs of σ_f, and the outputs of σ_c are the inputs of σ_f. A *solution* for the planning problem $P = \langle \Sigma, \mathcal{M}, S_g \rangle$ is an IOA σ_{flat} in $(\Sigma, \mathcal{M})^*$ and a control automaton σ_c such that some runs of the parallel composition of σ_c with σ_{flat} reach a state in S_g. Note that the notion of *solution* is rather weak, since it guarantees that just some runs reach a goal state. Other runs may never end, or may reach a state that is not a goal state.

We will use the same terminology as in (Ghallab, Nau, and Traverso 2016, Section 5.2.3): a solution is *safe* if all of its finite runs terminate in goal states, and a solution is either *cyclic* or *acyclic* depending on whether it has any cycles.[4]

Example 4. *Figure 5 shows a control automaton for the IOAs in Figures 1 and 2. This control automaton is for the system when the* move *task has not been refined.*

The control automaton in Figure 6 controls the door IOA in Figure 1, the robot IOA in Figure 2, the move *IOA in Figure 3, and the monitor IOA in Figure 4.*

Solving Planning Problems

This section describes our planning algorithm, MakeControlStructure (Figure 7). It solves the planning problem $\langle \Sigma, \mathcal{M}, S_g \rangle$ where Σ is the set of IOAs, $\mathcal{M}$ is the collection of methods for refining different tasks and S_g is the set of goal states. The solution is a set of IOAs Σ_{set} and a control automaton σ_c such that

[4]In the terminology of (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.

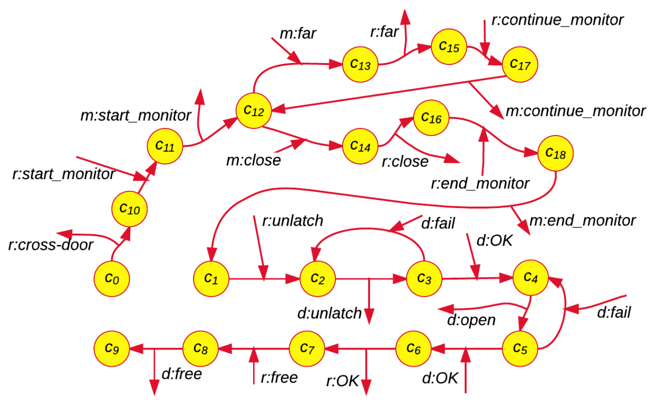

Figure 6: A control automaton for the door component(*d:*), refined robot component for going through a doorway(*r:*), and the monitor component(*m:*).

Σ_{set} driven by σ_c reaches the desired goals states, S_g. Depending on how one of its subroutines is configured, MakeControlStructure can search either for acyclic safe solutions, or for safe solutions that may contain cycles.

Before getting into the details of how MakeControlStructure works, we need to discuss a property on which it depends. Given a planning problem, MakeControlStructure constructs a solution by doing a sequence of parallel composition and refinement operations. An important property is that composition and refinement can be done in either order to produce the same result. Thus the algorithm can choose the order in which to do those operations (line (*) in Figure 7), which is useful because the order affects the size of the search space.

Theorem 1 (distributivity). *Let σ_1 and 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 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 $s_1^* = \{(s_1, s) \mid s \in S_{\sigma_2}\}$.

Algorithm. We now discuss our planning algorithm, which is shown in Figure 7. MakeControlStructure's two main steps are as follows. First, it uses the MakeFlat subroutine (described in the next paragraph) to build an IOA σ_{flat} that is an instantiation of $(\Sigma, \mathcal{M})$ and a set of IOAs Σ_{set} that fully determine σ_{flat}. Next, MakeControlStructure uses the MakeControlAutomaton subroutine to create a control automaton σ_c for Σ. We do not include pseudocode for MakeControlAutomaton, 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.

MakeControlStructure $(\Sigma, \mathcal{M}, S_g)$
  select an IOA σ from Σ and remove it
  $(\sigma_{flat}, \Sigma_{set}) \leftarrow$ MakeFlat$(\Sigma, \sigma, \mathcal{M}, \{\sigma\})$
  $\sigma_c \leftarrow$ MakeControlAutomaton (σ_{flat}, S_g)
  return (σ_c, Σ_{set})

MakeFlat $(\Sigma, \sigma_0, \mathcal{M}, \Sigma^0_{set})$
  $\sigma_f \leftarrow \sigma_0$; $\Sigma_{set} \leftarrow \Sigma^0_{set}$
  loop
    if $\sigma_f \in (\Sigma, \mathcal{M})^*$, then return (σ_f, Σ_{set})
    choose *which-first* $\in \{$*compose*, *refine*$\}$ (*)
    if (*which-first* = *compose*)
      select IOA σ from Σ that interacts with σ_f and remove it
      $\sigma_f \leftarrow \sigma_f \parallel \sigma$
      $\Sigma_{set} \leftarrow \Sigma_{set} \cup \{\sigma\}$
    else
      $(\sigma_f, \Sigma'_{set}) \leftarrow$ ComputeRefinement$(\Sigma, \sigma_f, \mathcal{M})$
      $\Sigma_{set} \leftarrow \Sigma_{set} \cup \Sigma'_{set}$

ComputeRefinement $(\Sigma, \sigma_f, \mathcal{M})$
  select a task t from T_{σ_f} such that $\langle s_1, t, s_2\rangle$ is a transition in σ_f
  select a method, $\mu_t = \langle t, \sigma_\mu\rangle$ from $\mathcal{M}$ for refining t
  $S \leftarrow \{$IOA that interact with $\sigma_\mu\}$
  t_{new}, $\sigma_{t_{new}} \leftarrow$ new unique names for t and σ_μ
  $\sigma'_\mu \leftarrow \sigma_{t_{new}} \parallel (\parallel_{\sigma' \in S} \sigma')$
  $\sigma_f \leftarrow \mathfrak{R}(\sigma_f, s_1, \langle t_{new}, \sigma'_\mu\rangle)$
  return $(\sigma_f, \{\sigma_{t_{new}}\} \cup S)$

Figure 7: Pseudocode for our planning algorithm.

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. MakeFlat constructs an instantiation σ_{flat} of $(\Sigma, \mathcal{M})$ by doing a series of parallel and hierarchical compositions. It randomly selects an IOA from Σ to start with and then goes through a loop which makes the choice of whether to do a parallel composition or a refinement at each iteration. 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 the heuristics will be analogous to some of the heuristics for constraint-satisfaction problems (Dechter 2003; Russell and Norvig 2009). MakeFlat exits when the IOA σ_f is an instantiation of $(\Sigma, \mathcal{M})$ i.e., there are no more tasks to be refined and all possible interactions have been taken in account through parallel composition.

ComputeRefinement is a subroutine of MakeFlat which does task refinement.

It applies a refinement method μ_t selected from $\mathcal{M}$ to refine a task t. This involves doing both a refinement and a parallel composition.

Once the task has been refined, ComputeRefinement returns the resulting IOA σ_f from this hierarchical composition and a set of IOAs that uniquely determine σ_f.

At this step, we also rename the task t and $body(\mu_t)$ to t_{new} and $\sigma_{t_{new}}$, where t_{new} is a new unique name for t. This renaming is required to identify $body(\mu_t)$ uniquely for each refinement and avoid name conflicts when μ_t is used multiple times to refine different instances of t.

Theorem 2. *MakeControlStructure is sound and complete.*

We note that completeness guarantees that we find the control automaton when it exists, but does not guarantee that our algorithm will terminate or return "no" when there is no control automaton for the problem.

Planning and Acting

In order to synthesize a control automaton for a system, $(\Sigma, \mathcal{M})$, we need to choose among:

- alternative methods $\mu_t \in \mathcal{M}$ for refining a task t; and
- alternative inputs in a state s that is followed by distinct actions or outputs.

These decision steps are determined by the planning algorithm through the synthesis of a pair $(\sigma_{flat}, \sigma_c)$.

Acting with such a system once a solution has been planned for means running its components $\{\sigma_1, \ldots, \sigma_n\}$ asynchronously, while *(i)* triggering the only action or output associated to a state whose outgoing transition is an action or an output, and *(ii)* following the received input for a state whose outgoing transitions are inputs. In some states, these received inputs are nondeterministic outputs from the external world. Hence, the pair $(\sigma_{flat}, \sigma_c)$ can be viewed as a classical reactive system, which interacts deterministically with a nondeterministic external world. This is, in a way, equivalent to a deterministic policy, and in our case, it is distributed over n components.

Acting according to a deterministic automaton may appear to be straightforward;but in the proposed framework, it raises several important issues that still lie ahead in our research agenda. Here is a brief overview of the issues:

- *Monitoring*: planning for any solution is generally easier than planning for a *safe* solution, which may not exist. For unsafe solutions, it is important to monitor the system to check whether the interaction with the nondeterministic external world is driving the system away from the intended goals and whether replanning is needed. Furthermore, even if a safe solution has been planned for, a system does not always behave as modeled; acting needs to be coupled with monitoring.
- *Interleaving* planning and acting, which is particularly desirable given the hierarchical nature of our framework and the interaction with a nondeterministic external world.

To handle the first issue of monitoring, the idea is to synthesize, in the planning stage (our module MakeControlStructure), conditions which guarantee that a given pair $(\sigma_{flat}, \sigma_c)$ remains a solution, i.e., some runs will reach a goal state. These conditions may be expressed as a *monitoring* IOA which interact with the components and σ_c, raising failure detection outputs when needed. Such a monitoring IOA is shown in Figure 4. The second issue of interleaving planning and acting corresponds to one of the motivation of our proposed framework. The idea here is to ignore some of the tasks in T_{σ_f} at the planning stage in ComputeRefinement and refine these tasks at the acting stage only. Tasks that have a single refinement method and whose recursive refinement is an IOA without any decision step (as summarized above) can safely be ignored in T_{σ_f} while planning. Even for these simple cases, postponing the refinement may lead to retrials in the physical world (instead of backtracking at the planning stage). This is acceptable for some noncritical tasks and domains. A postponed refinement may be addressed at the acting stage with domain-dependent heuristics for the choice of methods and inputs. It may also be addressed with a look-ahead mechanism, basically an adaptation of planning approach to the ongoing solution.

Related Work

To the best of our knowledge, no previous approach to planning has proposed a formal framework based on hierarchical input/output automata. The idea of hierarchical planning was proposed long time ago, see, e.g., (Sacerdoti 1974; Tate 1977; Yang 1990). In these works, high level plans are sequences of abstract actions that are refined into sequences of actions at a lower level of abstraction. In our work, plans are represented with automata. This allows us to express plans with rich control constructs, not only sequences of actions. We can thus represent conditional and iterative plans that are most often required when we refine plans at a lower level of abstraction. Moreover, none of these systems can represent components that interact among each other, and that act by interacting with the external environment, like in our representation based on input/output automata. These are also the main differences with more recent work, such as the work on angelic hierarchical planning (Marthi, Russell, and Wolfe 2007) and its extension with optimal and online algorithms (Marthi, Russell, and Wolfe 2008).

Some approaches manage plans with rich control constructs, see, e.g., the PRS (Georgeff, Lansky, and Bessière 1985) and Golog (McIlraith and Fadel 2002) systems. However, PRS does not provide the ability to reason about alternative refinements. Some limited planning capabilities were added to PRS by (Despouys and Ingrand 1999) to anticipate execution paths leading to failure by simulating the execution of procedures and exploring different branches. In Golog, it is possible to specify plans with complex actions and to reason about them in a logical framework. However, the formal framework is not hierarchical and the notion of refinement is not formalized. Moreover, neither PRS nor Golog allows for an explicit representation of interactions among different systems or components. Hierarchical and procedure based frameworks (similar to PRS) have been used in robotic systems, see, e.g., RAP (Firby 1987), TCA (Simmons 1992; Simmons and Apfelbaum 1998), XFRM (Beetz and McDermott 1994), and the survey (Ingrand and Ghallab 2014). All these works have addressed the practical problem of providing reactive systems for robotics, but none of them is based on a formal account as the one provided in this paper.

Our approach shares some similarities with the hierarchical state machines (Harel 1987), which has been used to address problems different from planning and acting, like the problem of the specification and verification of reactive systems. Our work is based on the theory of input/output automata (see, e.g., (Lynch and Tuttle 1988)), which has been used to specify distributed discrete event systems, and to formalize and analyse communication and concurrent algorithms.

I/O automata have been used to formalize interactions among web services and to plan for their composition (Pistore, Traverso, and Bertoli 2005; Bertoli, Pistore, and Traverso 2010). That work was the basis for the approach described in (Ghallab, Nau, and Traverso 2016, Section 5.8). In our paper, beyond providing a framework that is independent of the web service domain, we extend the representation significantly with hierarchical input/output automata, that contain compound tasks. Moreover, we provide a theory that formalizes the refinement of tasks. Finally, we describe a novel planning algorithm for synthesizing control automata, that can deal with hierarchical refinements. Our work is also significantly different from the work in (Bucchiarone et al. 2012; Bucchiarone et al. 2013), where abstract actions are represented with goals, and where (online) planning can be used to generate interacting processes that satisfy such goals.

Our framework has some similarities with HTN planning (Nau et al. 1999), since tasks can be refined with different methods. However, our methods are significantly different from HTN ones in at least two fundamental aspects: (i) our methods are automata that can encode rich control constructs rather than simple sequences of primitive tasks; (ii) our methods can interact among themselves. This allows us to represent interacting and distributed systems and components. This is also a main difference with the work proposed in extensions of HTN planning to deal with nondeterministic domains (Kuter et al. 2009).

There has been some work on synthesizing controllers for MDPs/POMDPs (Hansen and Zhou 2003). But for modeling any component as a MDP/POMDP, we need the information of the state transition probabilities, which might not be always available. Further, the controllers synthesized in that work are policies rather than automata. A control automaton is a richer structure, because its transitions depend not only on the state of the world but also the state of the automaton (which depends on the history of previous transitions).

There is also a vast amount of literature on controllers for discrete-event systems, e.g., (Wong and Wonham 1996; Mohajerani et al. 2011). But they do verification and do not synthesize controllers to the best of our knowledge. Also, they don't consider hierarchical refinement and alternate choices, and the controller synthesized is limited in terms of the execution structure it leads to. The work by Mikaelian (Mikaelian, Williams, and Sachenbacher 2005) models hardware and software systems as automata and is limited to monitoring and fault diagnosis in the systems, and does not design a controller for it.

Conclusions and Future Work

We have developed a formalism for synthesizing systems that are composed of communicating components. This synthesis is done by combining parallel composition of input/output automata with hierarchical refinement of tasks into input/output automata. This approach can be used to synthesize plans that are not just sequences of actions, but include rich control constructs such as conditional and iterative plans. For synthesis of such plans, we describe a novel planning algorithm for synthesizing control automaton, that can deal with hierarchical refinements.

We believe this work will be important as a basis for algorithms to synthesize real-time systems for web services, automation of large physical facilities such as warehouses or harbors, and other applications. 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. 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. Some of these heuristics are likely to be similar to the heuristics used to guide constraint-satisfaction algorithms (Dechter 2003; Russell and Norvig 2009).

Finally, there are several ways in which it would be useful to generalize our formalism. One is to allow tasks and methods to have parameters, so that a method can be used to refine a variety of related tasks. Another is to extend the formalism to support cases where two or more methods can collaborate to perform a single task.

References

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

[Bertoli, Pistore, and Traverso 2010] Bertoli, P.; Pistore, M.; and Traverso, P. 2010. Automated composition of Web services via planning in asynchronous domains. *Artificial Intell.* 174(3-4):316–361.

[Boese and Piotrowski. 2009] Boese, F., and Piotrowski., J. 2009. Autonomously controlled storage management in vehicle logistics applications of RFID and mobile computing systems. *Intl. J. RF Technologies: Research and Applications* 1(1):57–76.

[Bucchiarone et al. 2012] Bucchiarone, A.; Marconi, A.; Pistore, M.; and Raik, H. 2012. Dynamic adaptation of fragment-based and context-aware business processes. In *Intl. Conf. Web Services*, 33–41.

[Bucchiarone et al. 2013] 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*, 571–578.

[Cimatti et al. 2003] Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intell.* 147(1-2):35–84.

[D'Andrea 2012] 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.

[Dechter 2003] Dechter, R. 2003. *Constraint Processing.* Morgan Kaufmann.

[Despouys and Ingrand 1999] Despouys, O., and Ingrand, F. 1999. Propice-Plan: Toward a unified framework for planning and execution. In *ECP*.

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

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

[Georgeff, Lansky, and Bessière 1985] Georgeff, M. P.; Lansky, A. L.; and Bessière, P. 1985. A procedural logic. In *IJCAI*, 516–523.

[Ghallab, Nau, and Traverso 2014] Ghallab, M.; Nau, D.; and Traverso, P. 2014. The actor's view of automated planning and acting: A position paper. *Artificial Intell.* 208:1–17.

[Ghallab, Nau, and Traverso 2016] Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting.* Cambridge University Press.

[Hansen and Zhou 2003] Hansen, E. A., and Zhou, R. 2003. Synthesis of hierarchical finite-state controllers for pomdps. In *ICAPS*, 113–122.

[Harel 1987] Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3):231–274.

[Ingrand and Ghallab 2014] Ingrand, F., and Ghallab, M. 2014. Deliberation for autonomous robots: A survey. *Artificial Intell.*

[Kuter et al. 2009] Kuter, U.; Nau, D.; Pistore, M.; and Traverso, P. 2009. Task decomposition on abstract states, for planning under nondeterminism. *Artificial Intell.* 173:669–695.

[Lynch and Tuttle 1988] Lynch, N. A., and Tuttle, M. R. 1988. An introduction to input output automata. *CWI Quarterly* 2(3):219–246.

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

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

[McIlraith and Fadel 2002] McIlraith, S. A., and Fadel, R. 2002. Planning with complex actions. In *Proc. NMR 2002*, 356–364.

[Mikaelian, Williams, and Sachenbacher 2005] Mikaelian, T.; Williams, B. C.; and Sachenbacher, M. 2005. Model-based monitoring and diagnosis of systems with software-extended behavior. In *AAAI*, 327–333.

[Mohajerani et al. 2011] 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 et al. 1999] Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *IJCAI*, 968–973.

[Pistore, Traverso, and Bertoli 2005] Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated composition of web services by planning in asynchronous domains. In *ICAPS*, 2–11.

[Russell and Norvig 2009] Russell, S. J., and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach.*

[Sacerdoti 1974] Sacerdoti, E. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intell.* 5:115–135.

[Simmons and Apfelbaum 1998] Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *IROS*, 1931–1937.

[Simmons 1992] Simmons, R. 1992. Concurrent planning and execution for autonomous robots. *IEEE Control Systems* 12(1):46–50.

[Tate 1977] Tate, A. 1977. Generating project networks. In *IJCAI*, 888–893.

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

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

[Yang 1990] Yang, Q. 1990. Formalizing planning knowledge for hierarchical planning. *Comput. Intell.* 6(1):12–24.