Realizing Multiple Autonomous Agents through Scheduling of Shared Devices

Sebastian Sardina
School of Computer Science and Information Technology
RMIT University
Melbourne, Australia
sebastian.sardina@rmit.edu.au

Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica
Sapienza Universita' di Roma
Roma, Italy
degiacomo@dis.uniroma1.it

Abstract

Imagine a collection of available devices, such as a camera, a vacuum cleaner, or robotic arm, each of which is able to act (that is, perform actions) according to a given behavior specification, expressed as a finite transition system. Imagine next a set of virtual independent and autonomous agents, such as a surveillance agent or a cleaning agent, which are meant to operate concurrently, each within a given specification of its capabilities, again expressed as a finite transition system. The question then is: can we guarantee the realization of every agent by intelligently scheduling the available devices while fully preserving the agents' autonomy? In this paper, we define the problem formally, and propose a technique to actually generate a solution by appealing to recent results in LTL-based *synthesis* of reactive systems. We also show that the proposed technique is optimal with respect to computational complexity.

Introduction

In this paper we look at a novel problem, called below *concurrent composition*, which is foreseen in the work on complex agent systems such as robot ecology, ubiquitous robots, or intelligent spaces (Lundh, Karlsson, & Saffiotti 2008; Saffiotti & Broxvall 2005; Kim *et al.* 2004; Lee & Hashimoto 2002). Namely, we are to realize a collection of independent virtual *agents*, for example a surveillance or cleaning agent, which are meant to act autonomously and asynchronously. These agents do not have a fixed embodiment, but they must be concretely realized by using a set of shared available *devices*, for example a wall camera or a vacuum cleaner.

In such context, concrete existing devices are allowed to "join" and "leave" an agent dynamically, depending on the current action the agent is about to perform. In other words, each agent's embodiment is dynamically transformed while in execution. For example, an available camera may be used in live mode by the house surveillance agent to track and record the moves of burglar, and later on used by the cleaning agent to spot misplaced objects.

More concretely, in the concurrent composition problem, we are concerned with guaranteeing the concurrent execution of a set of agents *as if each of them were acting in isolation*, though, in reality, they are all collectively realized by the same set of shared existing devices. A solution to such problem amounts then to *intelligently scheduling* the actions requested by all the agents to the set of devices at disposal. The reader may notice that there is an apparent contradiction between the presence of a coordination module, the "*scheduler*," and the need of guaranteeing agents autonomy. This is exactly what spices up the problem. The scheduler job is to make the system, constituted by the shared devices, in which the agents execute, as "friendly" as possible to the agents, while preserving their autonomy. In doing so the scheduler has to consider the characteristics of both agents and devices:

- Agents are autonomous and decide by themselves what action to do next; the scheduler may not break such autonomy. The scheduler can however relay on a specification of each agent's *capabilities*, which form the so-called *space of deliberation* of the agent, i.e., a specification of what the agent can choose to do at each step. The scheduler can also exploit the fact that agents are asynchronous (though forms of explicit synchronization are possible, see later), and therefore unaware of when the scheduler serves their pending requests with respect to those of other agents. Obviously, though, the scheduler ought to guarantee that every pending request is served *eventually*.
- Devices can be instructed to execute an action, but the outcome of the action is not fully predictable before its execution (devices are partially controllable). The scheduler can however relay on a specification of each device's *logic* stating at each step the possible outcomes of actions.

Technically, we abstract actual agents and devices as finite state *transition systems*. More precisely, the logic of each device is represented as a nondeterministic (to model partial controllability) transition system; while each agent's space of deliberation is represented as a deterministic (to model full controllability, by the agent) transition system. Working with finite state transition systems allows us to leverage on the notable recent progress in the area of Verification (Piterman, Pnueli, & Sa'ar 2006; Alura, Henzinger, & Kupferman 2002; Clarke, Grumberg, & Peled 1999).

We tackle the concurrent composition problem, on the one hand, by borrowing ideas developed originally for service

Copyright © 2008, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

composition (Berardi *et al.* 2005; De Giacomo & Sardina 2007), and on the other hand, by making use of recent LTL-based synthesis techniques developed in the literature on Verification (Piterman, Pnueli, & Sa'ar 2006). Specifically, we develop a sound and complete technique to synthesize a scheduler for the devices, and show that such technique is indeed optimal with respect to computational complexity. Notably, the technique proposed is readily implementable by using current LTL-based state-of-the-art synthesis systems.

The technical content of paper is structured in two parts. The first part, which comprises the next three sections, describes the concurrent composition problem formally, characterizes what counts as a solution, and illustrates the different features of the problem through several examples. The second part, which comprises the remaining technical sections, presents a technique, based on a reduction to synthesis in LTL, to automatically devise a solution for the concurrent composition problem. The crux of the latter part is Theorem 2, which provides the soundness and completeness of the technique, together with Theorem 3, which shows its optimality with respect to computational complexity.

The Setting

We begin by providing formal accounts for the two basic components of the concurrent composition problem: *(i)* the set of virtual *agents* to be supported; and *(ii)* the set of available *devices* used to realize the agents. Technically, our formalization is inspired by (De Giacomo & Sardina 2007). However, since the problem we look at here is definitively more involved, for the sake of brevity, we simplify the setting by dropping non-essential aspects.[1]

Transition Systems We use transition systems (TSs) to represent "behaviors," that is, the possible executions of the components (i.e., agents and devices). Formally, a *transition system* is a tuple $\mathcal{B} = \langle \mathcal{A}, B, b_0, \delta \rangle$, where:

- $\mathcal{A}$ is a finite set of actions;
- B is the finite set of possible TS states;
- $b_0 \in B$ is the initial state of $\mathcal{B}$;
- $\delta \subseteq B \times \mathcal{A} \times B$ is the TS's transition relation: $\langle b, a, b' \rangle \in \delta$, or $b \xrightarrow{a} b'$ in $\mathcal{B}$, denotes that TS $\mathcal{B}$ may evolve to state b' when action a is executed in state b.

Transition systems are, in general, *nondeterministic*, in that they may allow many transitions to take place when the same action a is executed. Nondeterministic TSs represent *partially controllable* behaviors, since one *cannot* select the actual transition (and hence the next state of the TS) through the selection of the action to perform next.

A TS $\mathcal{B} = \langle \mathcal{A}, B, b_0, \delta \rangle$ is *deterministic* if there is no pair of transitions $b \xrightarrow{a} b'$ and $b \xrightarrow{a} b''$ in $\mathcal{B}$ such that $b' \neq b''$. Notice that, given a state and an action in a deterministic TS, we always know exactly *the* next state of the TS. Deterministic TSs represent thus *fully controllable* behaviors; one does select the (specific) transition (and hence the next state of the TS) through the selection of the action.

Available Devices An available device in our terms can be thought of as the behavior of a concrete device that can be used in the system (e.g., a video camera). Formally, an *available device* is a, possibly nondeterministic TS, which can be activated, stopped, and resumed. When activated, a device can be instructed to perform an action among the ones that are currently possible for the device—that is, for which there exists a transition from the device's current state. In this way, the TS associated with a device stands for the *logic* of the device; at every step, it provides the user of the device with the set of actions that can be performed by the device. Once a device is activated and an action instructed to be executed, the device evolves accordingly to its logic and a new set of actions is provided, and so on. Observe that, when choosing the action to execute next, one may not be certain, due to nondeterminism, of which choices will be available later on, as that depends on what transition is actually carried out. Hence, devices are only *partially controllable*.

Target Agent A target agent, or just an agent, is an entity acting *autonomously* (e.g., a surveillance agent). In contrast with the available devices, one does not instruct the agent on what to do, as this is a decision the agent takes by itself through its own deliberation mechanism. Nonetheless, we assume to have a specification of the agent's *space of deliberation*, capturing the possible executions that the agent could potentially realize. Such a specification is given again as a TS, this time, though, *deterministic*, and therefore *fully controllable* (by the agent itself). In any target agent's state, the actions available to the agent in that state are accounted for via corresponding transitions. Which action among these is actually chosen at any given step is completely determined by the (autonomous) agent. In other words, the behavior associated with a target agent can be seen as the description of the *capabilities* of the agent. They are thus understood in a analogous ways as Hierarchical Abstract Machines (HAMs) are for decision-theoretic deliberative agents (Parr & Russell 1998), and high-level programs for reasoning about (user) intentions in the situation calculus (McIlraith & Son 2002; Demolombe & Otermin Fernandez 2006).

Available System and Target System The *available system* $\mathcal{S}_a = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle$ consists of n, possibly nondeterministic, available devices $\mathcal{D}_i$. The *target system*, on the other hand, is the multi-agent system $\mathcal{S}_t = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle$ consisting of m target agents $\mathcal{T}_i$ that are to be realized using the available system.[2] Without loss of generality, we im-

[1] Among them we drop an explicit notion of environment, cf. (De Giacomo & Sardina 2007). In fact, it can be shown that the presence of an environment can be mimicked by using special synchronization devices, see later. Details are omitted for brevity.

[2] We account for concurrent executions of agents and devices via interleaving, as often done when dealing with concurrent processes. Also, for simplicity, we assume that a single action is executed at each step. It is indeed possible to extend the results here by allowing for multiple actions occurring simultaneously as in (Sardina, Patrizi, & De Giacomo 2007).

plicitly assume that all TSs for agents and devices share the same set of actions $\mathcal{A}$, and that no state in an agent may be "blocked" with no action possible (i.e., agents never stop).[3]

Before moving on, we shall make clear two important points regarding the multi-agent target system to be "implemented." First, and as already explained, each agent is assumed to be *autonomous*, in that they themselves decide what actions they are to execute, of course, within their own space of deliberation. Hence, while the set of possible actions that could be requested by an agent is known, the actual request can neither be controlled nor restricted. Second, we assume that the agents are asynchronous, that is, they do not share time. This implies that agents are not "fussy" or "sensible" on when their requests are met with respect to those of the other agents, as long as they eventually are.[4] Put it all together then, any "*fair*" realization that preserves the agents' future autonomy can, in principle, be taken into consideration.

The Concurrent Composition Problem

In order to realize the target system with the available devices, we need to (eventually) "delegate" each action chosen by each agent to one of the devices. We call the component that does this the "*scheduler*." The problem that concerns us has to do with the synthesis of such a scheduler. More precisely, the problem we are interested in is the following: given an available system $\mathcal{S}_a = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle$ and a target system $\mathcal{S}_t = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle$, *synthesize a scheduler that fairly realizes all target agents in $\mathcal{S}_t$ by suitably operating the available devices of $\mathcal{S}_a$.* A solution to such problem is called a *concurrent composition* of $\mathcal{S}_t$ in $\mathcal{S}_a$.

Next, we formally define what a scheduler is and when one is a concurrent composition. Recall that each virtual agent in the target multi-agent system is allowed to request the execution of an action (e.g., take photo) among those within its capabilities. This means that, at any point in time, there is a set of "pending" actions, one per target agent. After an agent's request is met, the agent is allowed to request another action (again, among the ones within its capabilities), and so on. The task of the so-called scheduler amounts to continuously guaranteeing the *eventual fulfillment* of all pending requests by delegating them to the available devices. As a result, while the autonomy of each agent is to be respected, the actual scheduling of the pending actions is to be decided by the scheduler so as to facilitate the overall execution of the target system.

With these ideas spelled out, we are now ready to move on.

[3]The latter is a standard assumption in Verification. One can trivially get non-stopping agents by using "no-op" actions appropriately (Clarke, Grumberg, & Peled 1999).

[4]If instead the order of agent's requests has to be respected, then the scheduler has essentially to realize the cross-product of the agents' TSs, which can be in turn represented as a single target TS, though nondeterministic. It follows that the resulting variant of the problem is much closer to that studied in (De Giacomo & Sardina 2007), and less involved than the one studied here.

System History A *system configuration* for an available system $\mathcal{S}_a$ is a tuple of the form $\langle s^1, \ldots, s^n \rangle$ denoting that device $\mathcal{D}_i = \langle \mathcal{A}, S_i, s_{i0}, \delta_i \rangle$, with $i \in \{1, \ldots, n\}$, is in state $s^i \in S_i$. A *system history* is a, possibly infinite, alternating sequence of system configurations and pairs formed by an m-tuple of pending actions (one per agent) and the action executed by (one of) the devices. Formally, system histories are alternating sequences of the form $\langle s_1^0, \ldots, s_n^0 \rangle \cdot (\langle a_1^1, \ldots, a_m^1 \rangle, a^1) \cdot \langle s_1^1, \ldots, s_n^1 \rangle \cdots (\langle a_1^k, \ldots, a_m^k \rangle, a^k) \cdot \langle s_1^k, \ldots, s_n^k \rangle \cdots$ where:

- $s_i^0 = s_{i0}$, for $i \in \{1, \ldots, n\}$, that is, each device starts in its initial state;
- the last element of the sequence, if finite, is a system configuration;
- at each step $k \geq 1$, there exists an $i \in \{1, \ldots, n\}$ such that $s_i^{k-1} \xrightarrow{a^k} s_i^k$ in $\mathcal{D}_i$, and $s_j^k = s_j^{k-1}$, for all $j \neq i$, that is, at each step in the history, only one of the devices, namely $\mathcal{D}_i$, has made a transition (according to its transition relation δ_i), while the others have all remained still.

The set of all system histories is denoted by $\mathcal{H}$.

A history h' is a *subhistory* of a history h, denoted by $h' \sqsubseteq h$, if h' is a prefix of h. Moreover, when $h \in \mathcal{H}$ is a history, $[h]$ denotes the sequence of actions executed in h, i.e., $[h] = a^1 \cdot a^2 \cdots$. (if $h = \langle s_1^0, \ldots, s_n^0 \rangle$, then $[h] = \epsilon$ and $|[h]| = 0$.) [5] Lastly, $h|_k$ denotes the subhistory of h of action length $k \geq 0$, i.e., $h|_k \sqsubseteq h$ and $|[h|_k]| = k$ (if $k \geq |[h]|$, then $h|_k = h = h|_{|[h]|}$).

Scheduler A *scheduler* $P = \langle P_a, P_t \rangle$ for an available system $\mathcal{S}_a$ and a target system $\mathcal{S}_t$ is a pair of functions:

1. $P_a : \mathcal{H} \times \mathcal{A}^m \mapsto \mathcal{A} \times \{1, \ldots, n\}$ that, given a history $h \in \mathcal{H}$, returns the action to execute next and the (index of the) available device that shall perform such action.
2. $P_t : \mathcal{H} \mapsto 2^{\{1,\ldots,m\}}$ that, given a history $h \in \mathcal{H}$, returns which target agents, if any, may advance one step.

Intuitively, at any point, a scheduler is able to do two things. First, it is capable of instructing the execution of an action in an available device (via function P_a). Second, it can instruct some target agents to progress and request new actions (via function P_t).

Let us now focus on when a scheduler amounts to a solution for the concurrent composition problem.

Agent Traces and Target System Trace Given a target agent $\mathcal{T} = (\mathcal{A}, T, t_0, \delta)$, we define the possible *agent traces* of $\mathcal{T}$ as the, possibly infinite, sequences of actions of the form $\lambda = a^1 \cdot a^2 \cdots$ such that $t^0 = t_0 \xrightarrow{a^1} t^1 \xrightarrow{a^2} t^2 \cdots$, for some t^i's, with $i \geq 0$—that is, $(t^i, a^i, t^{i+1}) \in \delta$. As with histories, when λ is an agent trace, $\lambda|_k$ stands for the k-long prefix of λ ($\lambda|_0 = \epsilon$, where ϵ is the empty trace). Notice that since target agents are in fact deterministic behaviors, any

[5]As standard, $|\cdot|$ denotes the "size" of an element (e.g., the length of a sequence, the cardinality of a set, the size of a TS).

initial fragment of a trace leads to a single state in the target agent. Thus, a target agent can be seen as a specification of an (infinite) set of traces. (Remember that we assume agents to be non-stopping.)

Given a target system $\mathcal{S}_t = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle$, a *target system trace* for $\mathcal{S}_t$ is a tuple $\Lambda = \langle \lambda_1, \ldots, \lambda_m \rangle$ where λ_i is an infinite agent trace for target agent $\mathcal{T}_i$, for $i \in \{1, \ldots, m\}$.

Runs Prescribed by P with respect to Λ We now define the set $\mathcal{H}^\ell_{\Lambda,P}$ of all system histories prescribed by scheduler P relative to target system trace $\Lambda = \langle \lambda_1, \ldots, \lambda_m \rangle$, in which ℓ actions have been executed. Below, we shall use the following notation: given a system history $h \in \mathcal{H}$ and a target index $i \in \{1, \ldots, m\}$, $adv_P(h, i)$ stands for the number of times that P_t *has advanced* agent $\mathcal{T}_i$ during h; formally, $adv_P(h, i) = |\{h' \mid h' \sqsubseteq h \text{ and } i \in P_t(h')\}|$.

The set of $\mathcal{H}^\ell_{\Lambda,P}$ is inductively defined as follows: $\mathcal{H}^0_{\Lambda,P} = \{\langle s_{10}, \ldots, s_{n0} \rangle\}$; and $\mathcal{H}^{k+1}_{\Lambda,P}$ is the set of $k+1$-action long system histories of the form $h \cdot (\langle a_1, \ldots, a_m \rangle, a) \cdot \langle s_1^{k+1}, \ldots, s_n^{k+1} \rangle$, such that:

- $h \in \mathcal{H}^k_{\Lambda,P}$ (notice that $|[h]| = k$);
- $\langle s_1^k, \ldots, s_n^k \rangle$ is the last configuration in h;
- $\lambda_i|_{adv_P(h,i)+1} = \lambda'_i \cdot a_i$, for every $i \in \{1, \ldots, m\}$, that is, a_i (i.e., the pending request of agent $\mathcal{T}_i$) is the $(adv_P(h, i) + 1)$-th action in $\mathcal{T}_i$'s agent trace λ_i.
- $P_a(h, \langle a_1, \ldots, a_m \rangle) = \langle a, w \rangle$, for some $w \in \{1, \ldots, n\}$, that is, the scheduler states that the action a at system history h should be executed in device $\mathcal{D}_w$;
- $(s_w^k, a, s_w^{k+1}) \in \delta_w$, that is, device $\mathcal{D}_w$ may evolve from its current state s_w^k to state s_w^{k+1} (by executing action a).;
- $s_i^{k+1} = s_w^k$, for all $i \neq w$, that is, devices other than $\mathcal{D}_w$ remain still.

Finally, $\mathcal{H}^\omega_{\Lambda,P} = \{h \mid h|_\ell \in \mathcal{H}^\ell_{\Lambda,P}, \text{for every } \ell > 0\}$ is the set of all *infinite* system histories h whose finite prefixes $h|_\ell$, for any ℓ, are prescribed by P. Observe that, in defining the runs prescribed by P, no constraints are enforced between functions P_t and P_a. Obviously, to account for a solution, these two functions ought to work together, as shown next.

The Solution We say that a *scheduler* $P = (P_a, P_t)$ *fairly realizes the target system trace* $\Lambda = \langle \lambda_1, \ldots, \lambda_m \rangle$ for $\mathcal{S}_t$ iff

1. for all system histories $h \in \mathcal{H}^\ell_{\Lambda,P}$, with $\ell \geq 0$, we have:
   (a) there exists $h' \in \mathcal{H}^{\ell+1}_{\Lambda,P}$ such that $h \sqsubseteq h'$, that is, P is able to extend h one more step;
   (b) $[h]$ is an interleaving of the agent traces $\lambda'_1, \ldots, \lambda'_m$, where $\lambda'_i = \lambda_i|_{adv_P(h,i)}$ or $\lambda'_i = \lambda_i|_{adv_P(h,i)+1}$;
2. for all infinite system histories $h \in \mathcal{H}^\omega_{\Lambda,P}$, $[h]$ is a *fair interleaving* of the agent traces in Λ, that is, P_t selects each target agent infinitely many often in h.

Condition (1b) binds P_t and P_a, forcing P_a to return actions that are pending requests of *some* agent, and constraining P_t so as to advance an agent only if its previous pending request has been delegated (by P_a) to some device. Note also that P has no control on the actual action requested by the agents: each autonomous agent decides its own execution trace.

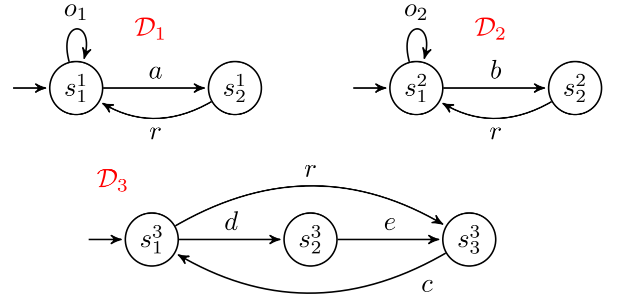
A scheduler $P = (P_a, P_t)$ is a concurrent composition *of the target system $\mathcal{S}_t$ in the available system $\mathcal{S}_a$ iff P fairly realizes every possible target system trace Λ for $\mathcal{S}_t$.*

Intuitively, a scheduler is a solution if it is able to intelligently arrange the execution of actions in the devices in a way that each target is guaranteed to have its desired execution trace realized.

Examples

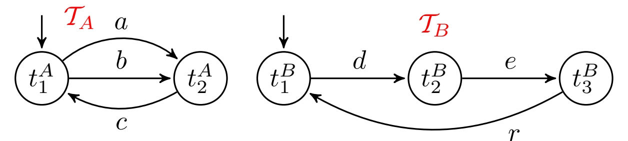
In this section, we illustrate the setting and scheduling problem of interest through examples in an incremental way. We begin with a two-agents scenario in which there are not many choices to be made and where a successful scheduling, i.e., a concurrent composition, is easy to generate.

So, we imagine the availability of the following three, deterministic, and hence fully controllable, devices:



Devices $\mathcal{D}_1$ and $\mathcal{D}_2$ are (structurally) similar; they differ only on the options available in their initial states, either a and o_1, or b and o_2, respectively.

Next, imagine that, using the above system $\mathcal{S}_a = \langle \mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \rangle$, we are to realize the target system $\mathcal{S}_t = \langle \mathcal{T}_A, \mathcal{T}_B \rangle$ built from the following two (simple) agents:

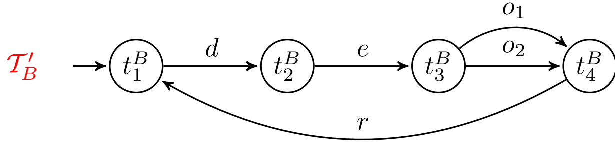


It is not hard to see how such agents can be realized using the above three devices. One scheduler $\bar{P}$ that is a concurrent composition is as follows. Initially, $\mathcal{T}_A$ requests a or b, and $\mathcal{T}_B$ requests action d. The scheduler selects $\mathcal{T}_A$'s request: it delegates a or b to either $\mathcal{D}_1$ or $\mathcal{D}_2$, respectively, and allows $\mathcal{T}_A$ to advance, thus requesting action c. After that, the scheduler selects $\mathcal{T}_B$'s request: it delegates d to device $\mathcal{D}_3$ and allows $\mathcal{T}_B$ to move on and request e next. Then, $\bar{P}$ continues by instructing $\mathcal{D}_3$ to perform e, fulfilling thus $\mathcal{T}_B$'s request and allowing it to continue. At this point, agent $\mathcal{T}_A$ is in state t_2^A requesting action c, whereas agent $\mathcal{T}_B$ is in state t_3^B requesting r. In addition, the device $\mathcal{D}_1$ ($\mathcal{D}_2$) ought to be in its state s_2^1 (s_1^2), if $\mathcal{T}_A$'s initial request was a, or in its state s_1^1 (s_2^2), if $\mathcal{T}_A$'s initial request was b. The scheduler $\bar{P}$ then continues by instructing $\mathcal{D}_3$ to perform action c, thus fulfilling agent $\mathcal{T}_A$, which in turn will next be able to request either a or b, again. Finally, $\bar{P}$ delegates the execution of

action r to either $\mathcal{D}_1$ or $\mathcal{D}_2$, whichever appropriate, allows $\mathcal{T}_B$ to progress, and finally repeats itself.

Observe how different devices may contribute to different agents at different points in time (e.g., $\mathcal{D}_3$ is used to fulfill $\mathcal{T}_B$'s requests for d and e and, but also to fulfill $\mathcal{T}_A$'s request for c).

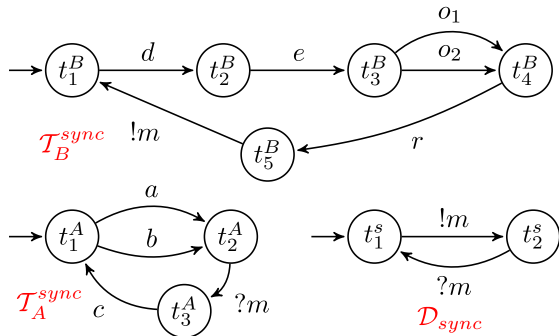
Next, consider an enriched version of $\mathcal{T}_B$ in which the agent may choose between actions o_1 and o_2 *after* having performed action e:



Realizing agents $\mathcal{T}_A$ and $\mathcal{T}'_B$ using the three devices now requires more care. In particular, the scheduler should "hold" agent $\mathcal{T}_A$ until it is known whether agent $\mathcal{T}_B$ would like to execute action o_1 or o_2. Otherwise, if agent $\mathcal{T}_A$ is immediately fulfilled (using $\mathcal{D}_1$ or $\mathcal{D}_2$), the scheduler may risk the ability to fully satisfy the autonomy of agent $\mathcal{T}_B$. To see why, suppose that $\mathcal{T}_A$ requests a first and agent $\mathcal{T}_B$, after having its actions d and e realized, happens to ask for o_1—action o_1 cannot be implemented at this point.

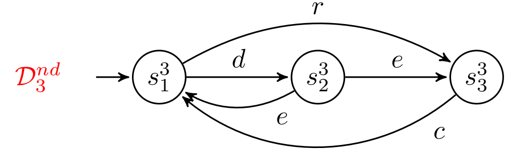
So, a concurrent composition ought to first use device $\mathcal{D}_3$ to fulfill $\mathcal{T}_B$'s requests for d and e. After that, either $\mathcal{D}_1$ or $\mathcal{D}_2$ is used to implement the next choice of $\mathcal{T}_B$, either o_1 or o_2. Then, the scheduler may continue easily as before. Observe, again, how agent $\mathcal{T}_A$ has to "wait" for two execution cycles to get its action finally performed.

In the above examples, the order in which the final request for actions c (for agent $\mathcal{T}_A$) and r (for agent $\mathcal{T}_A$) are fulfilled is irrelevant. That is, there are solutions where c is first executed and solutions where r is executed first. Imagine now that the two agents $\mathcal{T}_A$ and $\mathcal{T}_B$ want to synchronize their final actions so that action c is executed *only after* action r has been performed. Synchronization can be achieved through the use of devices. To that end, let us extend the system to include a distinguished "synchronization" device $\mathcal{D}_{sync}$, with special actions $!m$ (post message m) and $?m$ (read message m), and adapt our agents for explicit coordination. By posting ($!m$) and reading ($?m$) a message to and from the synchronization device $\mathcal{D}_{sync}$, agents $\mathcal{T}_A^{sync}$ and $\mathcal{T}_B^{sync}$ are now able to synchronize their last moves: any scheduling composition solution may satisfy $\mathcal{T}_A^{sync}$'s request for c only after having fulfilled $\mathcal{T}_B^{sync}$'s request for action r.



More sophisticated forms of synchronization, involving several agents and possibly several synchronization-coordination devices, can be captured similarly.

Finally, notice that all devices that we have used so far are *deterministic*. Due to lack of information on their actual behaviors, the scheduler may only know an incomplete description of their logic, which shows up as nondeterminism (and hence partial controllability) in the corresponding TSs For instance, imagine the following nondeterministic version of device $\mathcal{D}_3$:



Imagine then the task of realizing the original target system $\mathcal{S}_t$ by using the available system $\langle \mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3^{nd} \rangle$. When using device $\mathcal{D}_3^{nd}$, the scheduler has to cope with the fact that it is not known, a priori, whether instructing $\mathcal{D}_3^{nd}$ to perform action e would result in the device evolving to state s_3^3 or state s_1^3. (It can observe though the resulting state after e is executed.) As a matter of fact, the original scheduler $\bar{P}$ would not be a concurrent composition anymore: the realization would be "broken" if device $\mathcal{D}_3^{nd}$ happened to evolve to state s_1^3 (and not to s_3^3) upon a request for executing action e.

Nonetheless, the target system *can still be realized* by a more sophisticated scheduling strategy. In contrast with $\bar{P}$, the new scheduler has to be "smarter" after device $\mathcal{D}_3^{nd}$ evolves to state s_1^3 when performing e. Suppose that the last action fulfilled for agent $\mathcal{T}_A$ was a (an analogous argument applies for the case of b). First, the scheduler instructs device $\mathcal{D}_3^{nd}$ to execute action r first and then action c, thus allowing agents $\mathcal{T}_B$ and $\mathcal{T}_A$ to advance to states t_1^B and t_1^A, respectively. At this point, the whole system is as initially, except for $\mathcal{D}_1$ being left in state s_2^1. The trouble comes if agent $\mathcal{T}_A$ happens to request action a. The only device able to do a, namely $\mathcal{D}_1$, is not able to realize such action from where it was left (state s_2^1). The scheduler then, leaves $\mathcal{T}_A$ alone, and fulfills the next three requests of agent $\mathcal{T}_B$, namely actions d and e, by using device $\mathcal{D}_3^{nd}$, and action r, by using $\mathcal{D}_1$. At this point the scheduler can indeed fulfill $\mathcal{T}_A$'s pending request for a by using $\mathcal{D}_1$. Now, $\mathcal{D}_3^{nd}$ is either in state s_3^3 or again in state s_1^3. If in s_3^3 thing can proceed easily for another loop; if in s_1^3, the scheduler has to apply the strategy above again.

Reactive Synthesis in LTL

Linear Temporal Logic (LTL) is a well-known logic used to specify dynamic or temporal properties of programs, see e.g., (Vardi 1996). *Formulas* of LTL are built from a set $\mathcal{P}$ of atomic propositions and are closed under the boolean operators, the unary temporal operators $\bigcirc$ (*next*), $\Diamond$ (*eventually*), and $\Box$ (*always*, from now on), and the binary temporal operator *until* (which in fact can be used to express both $\bigcirc$ and $\Box$, though it will not be used here). LTL formulas are

interpreted over infinite sequences π of propositional interpretation for $\mathcal{P}$, i.e., $\pi \in (2^{\mathcal{P}})^\omega$. If π is an interpretation and i a natural number, and ϕ a propositional formula, we denote by $\pi, i \models \phi$ the fact that ϕ is true in the i-th propositional interpretation of π. Such interpretation is extended to the temporal operators as follows (we omit *until* for brevity).

$\pi, i \models \bigcirc\phi$ iff $\pi, i{+}1 \models \phi$;
$\pi, i \models \Diamond\phi$ iff for some $j \geq i$, we have that $\pi, j \models \phi$;
$\pi, i \models \Box\phi$ iff for all $j \geq i$, we have that $\pi, j \models \phi$.

An interpretation π satisfies ϕ, written $\pi \models \phi$, if $\pi, 0 \models \phi$. Standard logical tasks such as satisfiability or validity are defined as usual, e.g., a formula ϕ is *satisfiable* if there exists an interpretation that satisfies it. Checking satisfiability or validity for LTL is PSPACE-complete.

Here we are interested in a different kind of logical task, which is called *realizability*, or *Church problem*, or simply *synthesis* (Vardi 1996; Pnueli & Rosner 1989). Namely, we partition $\mathcal{P}$ into two disjoint sets $\mathcal{X}$ and $\mathcal{Y}$. We assume to have *no control* on the truth value of the propositions in $\mathcal{X}$, while we can control those in $\mathcal{Y}$. The problem then is: *can we control the values of $\mathcal{Y}$ such that for all possible values of $\mathcal{X}$ a certain LTL formula remains true?* More precisely, interpretations now assume the form $\pi = (X_0, Y_0)(X_1, Y_1)(X_2, Y_2)\cdots$, where (X_i, Y_i) is the propositional interpretation at the i-th position in π, now partitioned in the propositional interpretation X_i for $\mathcal{X}$ and Y_i for $\mathcal{Y}$. Let us denote by $\pi_{\mathcal{X}}|_i$ the interpretation π projected only on $\mathcal{X}$ and truncated at the i-th element (included), i.e., $\pi_{\mathcal{X}}|_i = X_0 X_1 \cdots X_i$. The *realizability problem* checks the existence of a function $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$ such that for all π with $Y_i = f(\pi_{\mathcal{X}}|_i)$ we have that π satisfies the formula ϕ. The *synthesis problem* consists in actually computing such a function. Observe that in realizability/synthesis we have no way of constraining the value assumed by the propositions in $\mathcal{X}$: the function we are looking for only acts on propositions in $\mathcal{Y}$. This means that the most interesting formulas for the synthesis have the from $\varphi \rightarrow \psi$, where φ captures the "relevant" assignments of the propositions in $\mathcal{X}$ (and $\mathcal{Y}$) and ψ specifies the property we want to assure for such relevant assignments. The realizability (and actual synthesis) are 2EXPTIME-complete for arbitrary LTL formulas (Pnueli & Rosner 1989). However, recently, largely inspired by research in discrete-event control (Ramadge & Wonham 1993), several well-behaved patterns of LTL formulas have been identified, for which efficient procedures based on model checking technologies applied to game structures[6] can be devised. Here, we shall focus on one of the most general well-behaved pattern, called "*general reactivity (1)*" or *GR(1)* (Piterman, Pnueli, & Sa'ar 2006). Such formulas have the form $\varphi \rightarrow \psi$, with φ and ψ of the following shape

φ: $\phi[\mathcal{X}, \mathcal{Y}] \wedge \bigwedge_j \Box\phi_j[\mathcal{X}, \mathcal{Y}, \bigcirc\phi[\mathcal{X}]] \wedge \bigwedge_k \Box\Diamond\phi_k[\mathcal{X}, \mathcal{Y}]$;
ψ: $\phi[\mathcal{X}, \mathcal{Y}] \wedge \bigwedge_j \Box\phi_j[\mathcal{X}, \mathcal{Y}, \bigcirc\phi[\mathcal{X}, \mathcal{Y}]] \wedge \bigwedge_k \Box\Diamond\phi_k[\mathcal{X}, \mathcal{Y}]$,

where $\phi[\mathcal{Z}]$ stands for any boolean combination of symbols from $\mathcal{Z}$. Notice that: *(i)* with the first conjunct we can express initial conditions; *(ii)* with the second (big) conjunct we can express transitions —and we have the further constraint that in doing so within φ we cannot talk about the next value of the propositions in $\mathcal{Y}$; and *(iii)* with the third (big) conjunct we can express *fairness* conditions of the form "it is always true that eventually something holds." For such formulas we have the following result.

Theorem 1 (Piterman, Pnueli, & Sa'ar 2006) *Realizability (and synthesis) of GR(1) LTL formulas $\varphi \rightarrow \psi$ can be determined in time $O((p * q * w)^3)$, where p and q are the number of conjuncts of the form $\Box\Diamond\phi$ in φ and ψ, respectively,[7] and w is the number of possible value assignments of $\mathcal{X}$ and $\mathcal{Y}$ under the conditions of $\varphi \rightarrow \psi$.*

Synthesis of the Scheduler

We are now ready to investigate how to check for the existence of (and actually compute) a scheduler that realizes a multi-agent target system.

We reduce our problem to realizability (and synthesis) of a GR(1) LTL formula Φ. Below, we shall illustrate the reduction using informal concepts, such as "*phases*" and "*stages*", whose only purpose is to guide the reader through the construction of the LTL formula Φ. The reader should keep in mind that, although the reduction can be informally understood as a set of constraints on the strategy to get the solution, its formal justification is simply Theorem 2, stating its soundness and completeness, and Theorem 3 stating its optimality with respect to computational complexity.

The intuition behind the reduction is as follows. We make the scheduler operate on *phases*, each of them formed of $m+1$ *stages*. At the beginning of each phase, there are m actions pending to be served, one per target agent. In each of the m first stages, the scheduler may select a (pending) action and an available device, and instruct the device to execute the action. Finally, in the last stage of each phase, called the "synchronization" stage, the scheduler matches one-to-one actions that have been executed during the phase with target agents that were requesting those actions. The targets that were matched are allowed to request new actions (within their capabilities), and a new phase begins.

Let $\mathcal{S}_a = \langle \mathcal{D}_1, \ldots, \mathcal{D}_n \rangle$ be an available system, where $\mathcal{D}_i = (\mathcal{A}, S_i, s_{i0}, \delta_i)$, with $i \in \{1, \ldots, n\}$, are the available devices (over the shared actions $\mathcal{A}$), and let $\mathcal{S}_t = \langle \mathcal{T}_1, \ldots, \mathcal{T}_m \rangle$ be a target system, where $\mathcal{T}_i = (\mathcal{A}, T_i, t_{i0}, \hat{\delta}_i)$, with $i \in \{1, \ldots, m\}$, are the target agents (again, over the same shared actions $\mathcal{A}$). We start building the GR(1) LTL formula $\Phi = \varphi \rightarrow \psi$ by specifying the sets of uncontrolled and controlled propositions $\mathcal{X}$ and $\mathcal{Y}$, and then build the assumption formula φ and the requirement formula ψ.

[6]Interestingly, realizability is the logical task at the base of the logics ATL and ATL* (Alura, Henzinger, & Kupferman 2002), whose semantics is based on an alternating multi-agent game played by a team of cooperating agents against the other agents in the systems. While general algorithms for ATL* are indeed 2EXPTIME-hard and difficult to implement, practical tools, based on model checking of game structures, exist for the simpler ATL (Alur *et al.* 1998).

[7]We assume that both φ and ψ contain at least one conjunct of such a form, if not, we vacuously add the trivial one $\Box\Diamond\texttt{true}$.

Uncontrolled and controlled propositions $\mathcal{X}$ and $\mathcal{Y}$ As the set of *uncontrolled* propositions $\mathcal{X}$, we have:

- one atomic proposition s for each state $s \in S_i$ in $\mathcal{D}_i$ and each $i \in \{1, \ldots, n\}$, denoting that device $\mathcal{D}_i$ is in state s;
- one atomic proposition t for each state $t \in T_i$ in $\mathcal{T}_i$ and each $i \in \{1, \ldots, m\}$, denoting that target $\mathcal{T}_i$ is in state t;
- one atomic proposition a_i for each action $a \in \mathcal{A}$ and target $\mathcal{T}_i$, stating that action a is pending for target $\mathcal{T}_i$;
- atomic propositions $nopend_k^a$, for each action $a \in \mathcal{A}$ and each $k \in \{0, \ldots, m\}$, denoting that there are k actions a's pending to be satisfied.

Similarly, as the set of *controlled* propositions $\mathcal{Y}$, we have:

- atomic propositions $Exec_{ia}$, for each $i \in \{1, \ldots, n\}$ and $a \in \mathcal{A}$, denoting that a is executed in device $\mathcal{D}_i$;
- atomic propositions $Full_i$, for each $i \in \{1, \ldots, m\}$, denoting that target $\mathcal{T}_i$ has been fulfilled (within the phase);
- atomic propositions $at_1, \ldots, at_m$ and $atSync$, used to denote the $m+1$ stages within one phase.

Assumption formula φ Next, we build the formula of the form $\varphi = \varphi_{init} \wedge \varphi_{trans}$ that is ought to capture the *assumptions* on the overall framework the scheduler is acting on. For technical convenience, we introduce two auxiliary notation that will be used below. First, we use $\Delta_{\mathcal{B}}(s)$ to denote the set of actions that behavior $\mathcal{B}$ can legally perform when in state s, i.e., $\Delta_{\mathcal{B}}(s) = \{a \mid \exists s'.(s, a, s') \in \delta\}$. Second, when Σ is a set of formulas and $0 \leq k \leq |\Sigma|$, we use $\mathsf{Count}(\Sigma, k)$ to denote the (exponential in Σ) formula which states that exactly k formulas in Σ are true.

The formula φ_{init} is a propositional formula which characterizes the (legal) initial states of the overall system:

$$\varphi_{init} = \bigwedge_{j=1}^{n} s_{j0} \wedge \bigwedge_{j=1}^{m} [t_{j0} \wedge \bigvee_{a \in \Delta_{\mathcal{T}_j}(t_{j0})} a_j].$$

That is, initially, every device and target agent is in its initial state s_{j0} and t_{j0}, respectively, and each target $\mathcal{T}_i$ is requesting some action among the possibles in its initial state.

The formula φ_{trans} is an LTL formula which characterizes the assumptions on how the overall system may evolve:

$$\varphi_{trans} = \bigwedge_{j=1}^{m} \Box\phi_j^{agt} \wedge \bigwedge_{j=1}^{n} \Box\phi_j^{dev} \wedge \Box\phi_{pend}.$$

We build formula ϕ_i^{agt}, which represents the operation of the *target agent* $\mathcal{T}_i$, with $i \in \{1, \ldots, m\}$, as the conjunction of (for each state $t \in T_i$ and action $a \in \mathcal{A}$):

- $atSync \wedge t \wedge Full_i \wedge a_i \rightarrow \bigcirc t'$, for the unique transition $(t, a, t') \in \hat{\delta}_i$, stating that, at the last (synchronization) stage $atSync$ of the phase, the target agent evolves, from sate t to t' if its requested action a has been fulfilled;
- $(\neg atSync \vee \neg Full_i) \wedge t \rightarrow \bigcirc t$, for each transition $(t, a, t') \in \hat{\delta}_i$, forcing the agent to remain still outside the synchronization stage and when not fulfilled;
- $t \rightarrow \bigvee_{a \in \Delta_{\mathcal{T}_i}(t)} a_i$, stating that the agent always requests actions accordingly to its capabilities, that is, when in state t the agent is requesting an action in $\Delta_{\mathcal{T}_i}(t)$;
- $(\neg atSync \vee \neg Full_i) \wedge a_i \rightarrow \bigcirc a_i$, stating that the agent's requested action persists until the synchronization stage, and also between phases if it has not been fulfilled;
- $(\bigvee_{a \in \mathcal{A}} a_i) \wedge \bigwedge_{a \neq b \in \mathcal{A}} \neg(a_i \wedge b_i)$, stating that the target agent is always requesting one and only one action.

We build formula ϕ_i^{dev}, which encodes the operation of the *available device* $\mathcal{D}_i$, $i \in \{1, \ldots, n\}$, as the conjunction of (for each state $s \in S_i$ and action $a \in \mathcal{A}$):

- $s \wedge Exec_{ia} \rightarrow \bigcirc[\bigvee_{(s,a,s') \in \delta_i} s']$. These assertions encode that if the available device $\mathcal{D}_i$ is in state s, and $\mathcal{D}_i$ is selected for the execution of an action a (i.e., $Exec_{ia}$ is true), then the device evolves to one of its successor states accordingly to its transitions.
- $s \wedge (\bigwedge_{a \in \mathcal{A}} \neg Exec_{ia}) \rightarrow \bigcirc s$. This states that the device remains still in s if it has not been chosen for execution.

Finally, we build the formula ϕ_{pend}, which models the "counting" of pending requested actions (per action), as the conjunction of (for each $a \in \mathcal{A}$ and $k \in \{0, \ldots, m\}$):

- $nopend_k^a \rightarrow \neg nopend_{k'}^a$, for every $k' \in \{0, \ldots, m\} \setminus \{k\}$, stating that the $m+1$ propositions to model the counter for action a are disjoint;
- $at_1 \rightarrow [nopend_k^a \leftrightarrow \mathsf{Count}(\bigcup_{j=1}^{m} a_j, k)]$, which forces the counter for a to be *assigned* (or initialized) at the first stage at_1 of a phase;
- $(\bigvee_{j=1}^{n} Exec_{ja}) \wedge nopend_k^a \rightarrow \bigcirc nopend_{k-1}^a$, for $k > 0$, stating that the counter for action a is decremented if the action has just been executed in some device;
- $(\bigwedge_{j=1}^{n} \neg Exec_{ja}) \wedge \neg atSync \wedge nopend_k^a \rightarrow \bigcirc nopend_k^a$, which states that the current value of the counter is propagated if the action was not executed anywhere or we are not in the last synchronization stage of a phase—in which case, the counter will be next re-calculated at stage at_1.

Requirement Formula ψ We now turn to the formula capturing the *requirements* for the module to be synthesized—the scheduler. We first define some useful abbreviations:

- $norequested_k^a \stackrel{\text{def}}{=} \mathsf{Count}(\bigcup_{j=1}^{m} a_j, k)$, where $a \in \mathcal{A}$ and $k \in \{0, \ldots, m\}$—action a is being requested k times;
- $assigned_k^a \stackrel{\text{def}}{=} \mathsf{Count}(\bigcup_{j=1}^{m} \{Full_j \wedge a_j\}, k)$, where $a \in \mathcal{A}$ and $k \in \{0, \ldots, m\}$—action a has been satisfied (i.e., assigned to a target requesting a) k times;
- $Exec^- \stackrel{\text{def}}{=} \bigwedge_{a \in \mathcal{A}} \bigwedge_{j=1}^{n} \neg Exec_{ja}$—no action is being currently performed in any device.

The formula to be constructed for capturing the necessary requirements is of the form $\psi = \psi_{init} \wedge \psi_{trans} \wedge \psi_{ev}$. The formula $\psi_{init} = at_1$ states that the only requirement for the initial state is that the scheduler starts in the first stage. The

formula ψ_{trans} is an LTL formula stating the constraints on how the actions can be executed in devices and how target agents are "fulfilled," namely:

$$\psi_{trans} = \Box\phi_{stages} \wedge \Box\phi_{Full} \wedge \Box\phi_{Exec}.$$

We build formula ϕ_{stages}, which models the $m+1$ stages $at_1, \ldots, at_m, atSync$ of a phase, as the conjunction of:

$$\begin{array}{l} at_1 \vee \ldots \vee at_m \vee atSync, \\ at_i \rightarrow \bigcirc at_{i+1}, \text{ for each } i \in \{1, \ldots, m-1\}, \\ at_m \rightarrow \bigcirc atSync,\ atSync \rightarrow \bigcirc at_1, \\ \bigwedge_{i \neq j \in \{1,\ldots,m\}} \neg(at_i \wedge at_j), \\ \bigwedge_{i \in \{1,\ldots,m\}} \neg(at_i \wedge atSync). \end{array}$$

Formula ϕ_{Full}, which represents how targets *may be fulfilled* within each phase, is built as the conjunction of:

- $\neg atSync \rightarrow \bigwedge_{j\in\{1,\ldots,m\}} \neg Full_j$, denoting that no target can be satisfied in a stage other than the last one;
- $atSync \rightarrow \bigvee_{j\in\{1,\ldots,m\}} Full_j$, stating that at least one target ought to be satisfied at the last $atSync$ stage;
- $atSync \rightarrow [assigned^a_k \leftrightarrow norequested^a_r \wedge nopend^a_p]$, for each $a \in \mathcal{A}$, $r, p \in \{0, \ldots, m\}$, and $k = (r - p)$, stating that an action is assigned to targets exactly as many times as they were executed in the devices during the phase.

We construct formula ϕ_{Exec}, representing how devices *may be selected* for action execution, as the conjunction of:

- $at_1 \rightarrow \neg Exec^-$, stating that at least one device must be activated in the first stage of every phase;
- $atSync \rightarrow Exec^-$, forcing no device to be activated in the last synchronization stage of phases;
- $\bigwedge_{a\in\mathcal{A}} \bigwedge_{i\neq j\in\{1,\ldots,n\}} \neg(Exec_{ia} \wedge Exec_{ja})$, denoting that no action can be delegated to two (or more) devices;
- $(\bigvee_{i\in\{1,\ldots,n\}} Exec_{ia}) \rightarrow \neg nopend^a_0$, for each $a \in \mathcal{A}$, stating that if an action is delegated to some device, then such action ought to be pending (at least once).

Finally, the formula ψ_{ev} encodes the fairness conditions that must be guaranteed by the scheduler, namely:

$$\psi_{ev} = \bigwedge_{i\in\{1,\ldots m\}} \Box\Diamond Full_i.$$

Intuitively, it imposes the requirement that one is *always* able to (*eventually*) fulfill every single target agent, by actually performing its requested action in some available device. Thus, this constraint encodes the "fairness" property of successful schedulers (see second condition in the solution).

We can now state our main result. Checking the realizability of Φ is a sound and complete technique for concurrent composition, as stated by the following theorem.

Theorem 2 (Soundness & Completeness) *There exists a scheduler that is a concurrent composition of the target system $\mathcal{S}_t$ in the system $\mathcal{S}_a$ iff the LTL formula Φ, constructed as above, is realizable.*

It is easy to see that $\Phi = \varphi \rightarrow \psi$ is indeed a GR(1) LTL formula. Considering Theorem 1 and analyzing the structure of Φ, we get that: *(i)* φ contains no subformulas of the form $\Box\Diamond\phi$; *(ii)* ψ contains m such subformulas; and finally *(iii)* the number of possible value assignments of $\mathcal{X}$ and $\mathcal{Y}$ under the conditions of $\varphi \rightarrow \psi$ is $O(|\mathcal{A}| * u^{m+n})$, where u is the number of states of the largest TS among the target agents and the devices (observe that variables that represent states in a TS are pairwise disjoint). As a consequence we get:

Theorem 3 (Complexity upperbound) *Checking the existence of a scheduler that is a concurrent composition of the target system $\mathcal{S}_t = (\mathcal{T}_1, \ldots, \mathcal{T}_m)$ in the available system $\mathcal{S}_a = (\mathcal{D}_1, \ldots, \mathcal{D}_n)$ can be done in $O(m * |\mathcal{A}| * u^{m+n})$, where $u = max\{|T_1|, \ldots, |T_m|, |S_1|, \ldots, |S_n|\}$.*

We stress that by solving realizability with the techniques in (Piterman, Pnueli, & Sa'ar 2006) we do get an actual synthesized scheduler, not merely verify its existence.

We close this section by observing that when we have a single target, we essentially get the composition problem in (De Giacomo & Sardina 2007), and in fact the EXPTIME-hardness result in (Muscholl & Walukiewicz 2007) applies to our case as well. As a result, we obtain a complete computational complexity characterization of the problem.

Theorem 4 (Complexity characterization) *Checking the existence of a scheduler that is a concurrent composition of a target system $\mathcal{S}_t$ in a system $\mathcal{S}_a$ is EXPTIME-complete.*

Conclusion

In this paper, we looked at the *concurrent composition* problem, that is, the problem of realizing multiple virtual agents by making use of a set of available shared devices. Solving such a problem amounts to synthesizing a scheduler that implements agents' action requests by delegating them to the concrete existing devices—possibly accommodating the interleaving among the agents—in a way that agent autonomy is fully preserved.

The synthesis technique we devised is based on a reduction to realizability of a special kind of LTL formulas for which practical algorithms exist. As a result, we can make use of current LTL-based state-of-the-art synthesis systems such as TLV[8] and Anzu[9] to solve the concurrent composition problem.

There are clear analogies between concurrent composition and classical scheduling (Lawler *et al.* 1993): the set of all agents' requests form the *activities* to be carried on, whereas the available devices stand for the so-called *resources*. However, the set of activities and the set of resources are not specified at the outset but are dynamic: the set of activities, as well as the set of resources to which activities can be assigned, change dynamically as agents and devices progress on their executions. A natural question that arises is how substantially different the concurrent composition problem is compared to classical scheduling problems, e.g., Job-Shop-Scheduling (JSS). Here, we just notice that JSS, as all classical scheduling problems, is NP-complete,

[8] www.cs.nyu.edu/acsys/tlv/

[9] www.ist.tugraz.at/staff/jobstmann/anzu/

while the concurrent composition problem is EXPTIME-complete. As a consequence, any reduction from concurrent composition to classical scheduling must be exponential (assuming NP differs from EXPTIME). This is a strong indication that the link between the two is non-trivial.

Several issues remain to be studied. For example, an interesting issue is how to drop a central scheduler in favor of many distributed ones. Ideas on distributing the scheduler to the devices can be found in (Sardina, Patrizi, & De Giacomo 2007). However, here it may be preferable to distribute the scheduler to the agents instead, given that these must be already equipped with some computational mechanism for deliberation.

Another interesting issue is the study of "weaker" notions of agent autonomy which may be sufficient in certain contexts, such as guaranteeing that agents can always carry on *at least one* action/transition within their capabilities. We believe techniques developed within LTL synthesis give us formal tools to tackle such variants of the problems as well.

In this work, we took a high-level perspective on agents and shared devices and focused on the synthesis problem only. Nonetheless, there are many other practical aspects of concern when it comes to implementing the solution. For instance, how to design such devices so that they can easily interoperate among themselves, as we assume here, is an active area of research on its own (Bordignon *et al.* 2007; Lundh, Karlsson, & Saffiotti 2007). In fact, we expect a fruitful cross-fertilization between the theoretical studies on automated synthesis of agents, as the one in the present paper, and practical work on experimenting device integration in robot ecologies and ambient intelligence.

Acknowledgments

The authors would like to thank Alessandro Saffiotti and his group for an interesting discussion on robot ecologies, which inspired this work. The authors would also like to thank the anonymous reviewers for their helpful suggestions. The first author was supported by the Australian Research Council and Agent Oriented Software (under grant LP0560702), and the National Science and Engineering Research Council of Canada under a PDF fellowship. The second author was partially supported by the the European FET basic research project FP6-7603 Thinking Ontologies (TONES).

References

Alur, R.; Henzinger, T. A.; Mang, F. Y. C.; Qadeer, S.; Rajamani, S. K.; and Tasiran, S. 1998. MOCHA: Modularity in model checking. In *Proc. of the 10th International Conference on Computer Aided Verification (CAV)*, 521–525.

Alura, R.; Henzinger, T. A.; and Kupferman, O. 2002. Alternating-time temporal logic. *Journal of the ACM* 49:672–713.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Hull, R.; and Mecella, M. 2005. Automatic composition of transition-based semantic web services with messaging. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, 613–624.

Bordignon, M.; Rashid, J.; Broxvall, M.; and Saffiotti, A. 2007. Seamless integration of robots and tiny embedded devices in a PEIS-ecology. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3101–3106.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model checking*. Cambridge, MA, USA: MIT Press.

De Giacomo, G., and Sardina, S. 2007. Automatic synthesis of new behaviors from a library of available behaviors. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1866–1871.

Demolombe, R., and Otermin Fernandez, A. M. 2006. Intention recognition in the situation calculus and probability theory frameworks. In *Proc. of the International Workshop on Computational Logic in Multi-Agent Systems (CLIMA)*, volume 3900 of *LNCS*, 358–372.

Kim, J.; Kim, Y.; ; and Lee, K. 2004. The third generation of robotics: Ubiquitous robot. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 1–7.

Lawler, E.; Lenstra, L.; Rinnooy Kan, A.; and Shmoys, D. 1993. *Sequencing and Scheduling: Algorithms and Complexity*, volume 4 of *Logistics of Production and Inventory, Handbooks in Operations Research and Management Science*. North-Holland. chapter 9, 445–522.

Lee, J., and Hashimoto, H. 2002. Intelligent space: Concept and contents. *Advanced Robotics* 16(3):265–280.

Lundh, R.; Karlsson, L.; and Saffiotti, A. 2007. Plan-based configuration of an ecology of robots. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 64–70.

Lundh, R.; Karlsson, L.; and Saffiotti, A. 2008. Automatic configuration of multi-robot systems: Planning for multiple steps. In *Proc. of the European Conference in Artificial Intelligence (ECAI)*.

McIlraith, S., and Son, T. C. 2002. Adapting Golog for programming the semantic web. In *Proc. of Principles of Knowledge Representation and Reasoning (KR)*, 482–493.

Muscholl, A., and Walukiewicz, I. 2007. A lower bound on web services composition. In *Proc. of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 4423 of *LNCS*, 274–286.

Parr, R., and Russell, S. 1998. Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems* 10:1043–1049.

Piterman, N.; Pnueli, A.; and Sa'ar, Y. 2006. Synthesis of reactive(1) designs. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, 364–380.

Pnueli, A., and Rosner, R. 1989. On the synthesis of a reactive module. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 179–190.

Ramadge, P. J., and Wonham, W. M. 1993. The control of discrete event systems. *Proc. of the IEEE* 77(1):81–98.

Saffiotti, A., and Broxvall, M. 2005. PEIS ecologies: Ambient intelligence meets autonomous robotics. In *Proc. of the International Conference on Smart Objects and Ambient Intelligence*, 275–280.

Sardina, S.; Patrizi, F.; and De Giacomo, G. 2007. Automatic synthesis of a global behavior from multiple distributed behaviors. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1063–1069.

Vardi, M. Y. 1996. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*. Springer. 238–266.