Automatic Workflows Composition of Mobile Services

Giuseppe De Giacomo   Massimiliano de Leoni   Massimo Mecella   Fabio Patrizi

SAPIENZA – Università di Roma
Dipartimento di Informatica e Sistemistica
Via Salaria 113 – 00198 Roma, ITALY
{degiacomo,deleoni,mecella,patrizi}@dis.uniroma1.it

Abstract

Pervasive computing environments are nowadays more and more used as a supporting tool for cooperative workflows, e.g., in emergency management. A typical problem in these scenarios is the synthesis of workflows in presence of sets of services (hosted on mobile devices) with constrained behaviors, just before the collaborating team is dropped off in the operation field.

In this paper, we propose a technique able to automatically synthesize distributed orchestrators, each one coordinating a service and synchronizing with the other orchestrators, given a target generic workflow to be carried out and a set of behaviorally-constrained services.

1. Introduction

Pervasive computing environments are nowadays more and more used as a supporting tool for cooperative workflows, e.g., in emergency management [7].

Each team member is typically equipped with handheld devices (PDAs) and communication technologies (e.g., WiFI for constituting a Mobile Ad hoc NETwork – MANET), and, through the interplay with the software running on the device, can execute specific actions. The team member and his device offer a service towards the other members, and an overall workflow coordinates the actions of all the services. On the other hand, actions offered by such mobile services are typically constrained; as an example, if a service $\mathcal{A}$ is instructed to take some photos, then it needs to be instructed to forward them to another storage device $\mathcal{B}$ (and no other photos can be taken until the forwarding is executed), as the device offering $\mathcal{A}$ has not enough storage space to keep multiple photos (this is quite common with current handheld devices). Moreover, the effects of such actions can not be foreseen, but can be observable afterwards.

Typically, as demonstrated in many research projects [1],

generic workflows for the different teams are designed a-priori, and then, just before a team is dropped off in the operation field, they need to be instantiated on the basis of the currently available services offered by the mobile devices and operators effectively composing the team.

Moreover, the effective workflow to be enacted by the team, through the offered services, cannot be *centrally* orchestrated, as in general devices may not be powerful enough, and continuous connection with this central orchestrator would be not guaranteed. Conversely, decentralized orchestrators (one for each device/service) should *distributively* coordinate the workflow, through the appropriate exchange of messages, conveying synchronization information and the outputs of the performed actions by the services.

The problem addressed in this paper is how to synthesize the distributed orchestrators in presence of services with constrained behaviors. We propose a novel technique, sound, complete and terminating, able to *automatically* synthesize such distributed orchestrators, given *(i)* the target generic workflow to be carried out, in the form of a finite transition system, and *(ii)* the set of behaviorally-constrained services, again in the form of (non deterministic) finite transition systems.

This issue has some similarities with the one of automatically synthesizing composite services starting from available ones [11, 12, 10, 16, 2]. In particular, [2] considers the issue of automatic composition in the case in which available services are behaviorally constrained, and [4] in the case in which the available services are behaviorally constrained and the results of the invoked actions cannot be foreseen, but only observable afterwards. All the previous approaches consider the case in which the synthesized orchestrator is centralized.

On the other side, the issue of distributed orchestration has been considered in the context of Web service technologies [1, 5], but with emphasis on the needed run-time architectures. Our work can exploit such results, even if they need to be casted into the mobile scenario (in which service

---

providers are less powerful).

The remainder of this paper is as follows. In Section 2, the general framework is presented. Section 3 presents a complete example, in which a target workflow, possible available services and the automatically synthesized orchestrators are shown. Section 4 presents the proposed technique, and finally Section 5 concludes the paper by presenting some discussion and future work.

2. Conceptual Architecture

As previously introduced, we consider scenarios in which a team consists of different operators, each one equipped with PDAs or similar handheld devices, running specific applications. The interplay of *(i)* software functionalities running on the device and *(ii)* human activities to be carried out by the corresponding operator, are regarded as *services*, that suitably composed and orchestrated form the *workflow* that the team need to carry out. Such a workflow is enacted, during run-time, by an *orchestrator* (a.k.a. workflow management system).

The service behavior is modeled by the possible sequences of actions. Such sequences can be nondeterministic; indeed nondeterministic sequences stem naturally when modeling services in which the result of each action on the state of the service can not be foreseen. Let us consider as an example, a service that allows taking photos of a disaster area; after invoking the operation, the service can be in a state `photo OK` (if the overall quality is appropriate), or in a different state `photo bad`, if the operator has taken a wrong photo, the light was not optimal, etc. Note that the orchestrator of a nondeterministic service can invoke the operation but cannot control what is the result of it. In other words, the behavior of the service is partially controllable, and the orchestrator needs to cope with such partial controllability. Note also that if the orchestrator observes the status in which the service is after an operation, then it can understand which transition, among those nondeterministically possible in the previous state, has been undertaken by the service. We assume that the orchestrator can indeed observe states of the available services and take advantage of this in choosing how to continue in executing the workflow.

The workflow is specified on the basis of a *set of available actions* (i.e., those ones potentially available) and a *blackboard*, i.e., a conceptual shared memory in which the services provide information about the output of an action (cfr. complete observability wrt. the orchestrator). Such a workflow is specified a-priori (i.e., it encodes predefined procedures to be used by the team, e.g., in emergency management), without knowing which effective services are available for its enactment.

The issue is then how to compose (i.e., realize) such a workflow by suitably orchestrating available services. In the proposed scenario, such a composition of the workflow is useful when a team leader, before arriving on the operation field, by observing *(i)* the available devices and operators constituting the team (i.e., the available services), and *(ii)* the target workflow the team is in charge of, need to derive the orchestration.

At run-time (i.e., when the team is effectively on the operation field), the orchestrator coordinates the different services in order to enact the workflow. The communications between the orchestrator and the services are carried out through appropriate middleware, which offers broadcasting of messages and a possible realization of the blackboard [13].

Indeed the orchestrator is distributed, i.e., there is not any coordination device hosting the orchestrator; conversely, each device, besides the service, hosts also a local orchestrator. All the orchestrators, by appropriately communicating among them, carry on the workflow in a distributed fashion. Also the blackboard, from an implementation point of view, is realized in a distributed fashion.

3. A Case Study

Let's consider a scenario where a disastrous event (e.g., an earthquake) breaks out. After giving first assistance to people involved in the affected area, a team of civil protection is sent on the spot. Team members, equipped with mobile devices, need to document damage directly on a situation map so that following activities can be scheduled (e.g., reconstruction jobs). Specifically their work is supposed to be focused on three buildings A, B and C. For each building a report has to be prepared. Those reports should contain: *(i)* a preliminary questionnaire giving a description of the building and *(ii)* some photos about the conditions of buildings. Filling questionnaires does not require to stay very close to buildings, whereas taking photos does.

Suppose the team is composed of three mobile services MS_1, MS_2, MS_3, whose capabilities include compiling questionnaires and taking/evaluating building pictures, in addition to a repository service RS, which is able to forward the documents (questionnaires and pictures) produced by mobile units to a remote storage in a central hall. Services can read and write some shared boolean variables, namely $\{$`qA,qB,qC,pA,pB,pC,available`$\}$, held in a blackboard, which represent relevant environment properties that can be accessed by all members, for reading/writing. For example, variable `qA` set to `T` corresponds to the availability of questionnaire `A`.

Each service has its own capabilities and limitations, basically depending on technological, geographical and historical reasons – e.g., a team member who, in the past, visited building A, makes its respective unit able to compile questionnaire A; a unit close to building B can move there, and so on. Mobile services are described by state-transition diagrams where non-deterministic transitions are allowed. Diagrams of Figures 1(a) − 1(d) describe, respectively, units $MS_1 − MS_3$ and RS. An edge outcoming from a state s is labeled by a triple $E[C]/A$, where both $[C]$ and A are optional, with the following semantics: *when the service is in state s, if the set of events E occurs and condition C holds, then: i) change state according to the edge and ii) execute action A.* In this context, a set of events represents a set of requests assigned to the service, which can be satisfied

only if the condition (or guard) holds (is true). Actions correspond to writing messages on the blackboard, while the actual fulfillment of requests is implicitly assumed whenever a state transition takes place. In other words, each set of events represents a request for some tasks, which are actually performed, provided the respective condition holds, during the transition. Moreover, blackboard writes can be possibly performed.

For instance, consider Figure 1(a). Initially (state S0), MS_1 is able to serve requests: {compile_qB} (compile questionnaire about building B), {read_pC} (get photo of building C from repository), {move_A} (move to, or possibly around, building A) and {req_space} (ask remote storage for freeing some space). In all such cases, neither conditions nor actions are defined, meaning that, e.g., {move_A} simply requires the unit to reach, i.e., actually moving to, building A, independently of any condition and without writing anything on the blackboard. After building A is reached (S1), a photo can be taken ({take_pA}). A request for this yields a non-deterministic transition, due to the presence of two different outgoing edges labeled with the same event and non-mutually-exclusive conditions (indeed, no condition is defined at all). Note that, besides possibly leading to different states (S2 or S3), a non-deterministic transition may, in general, give raise to different blackboard writes, as it happens, e.g., if a request for {eval_pC} is assigned when the service is in state S5. State S2 is reached when, due to lack of light, the photo comes out too dark. Then, only photo modification ({modify_pA}, which makes it lighter) is allowed. On the other hand, state S3 (the photo is quite fine) gives also the possibility to ask the repository for additional space while photo modification is being performed ({modify_pA,req_space}). In such case, {available=T} is written on the blackboard, which announces that some space is available in the repository and, thus, additional data can be stored there. Moreover, state S3 allows for serving a {write_pA} request, which has the effect of writing the taken photo into the remote storage. Such task can be successfully completed only if there is available space, as required by condition [available], and, in such case, it is to be followed by action {pA=T}, in order to announce the availability, in the storage, of a picture of building A. Now, consider the request for {read_pC} outgoing from state S0. Such task gets a photo of building C, if any, from the remote storage, and forces a service transition to state S5. Then, {evaluate_pC} can be requested with the aim of checking whether or not the photo captures relevant aspects of building C and consequently accepting or rejecting it. Recall that the photo could be not in the storage. If so, a {pC=F} write is performed. Otherwise, either {pC=T} or {pC=F} can be written on the blackboard, depending on whether the picture is accepted or not. Finally, we complete the description of the service by observing that task {write_qB} can be requested in order to write a filled questionnaire in the remote storage, assuming it is small enough to be written without satisfying any additional space condition.
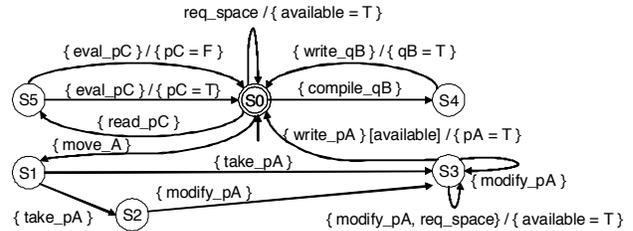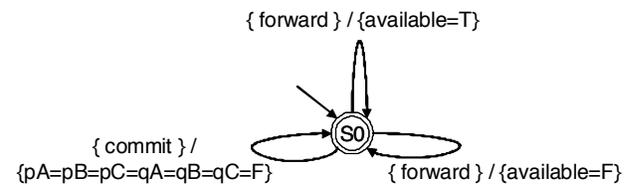

(a) Mobile Service MS_1
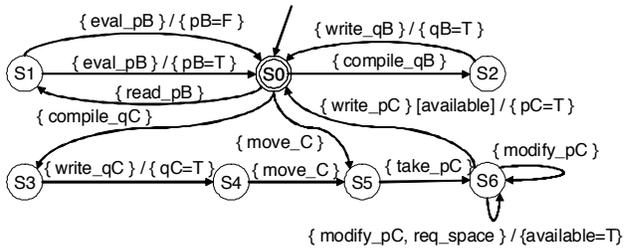
(b) Mobile Service MS_2
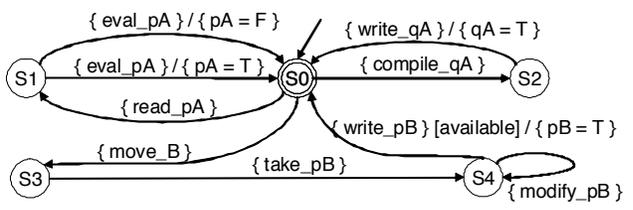
(c) Mobile Service MS_3

(d) Repository Service RS
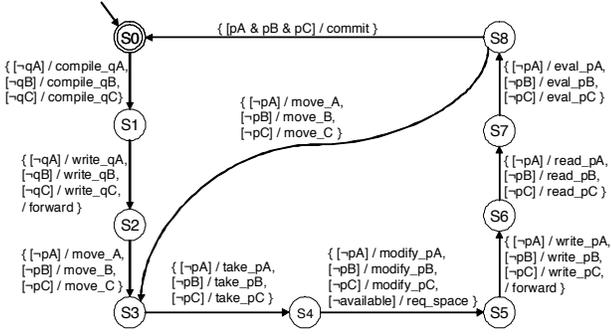
Figure 1. Mobile services

Figure 2. The target workflow

Semantics of other actions, e.g. `write_qA`, is straightforward and, consequently, diagrams of units MS_2, MS_3 and RS can be similarly interpreted. RS is a service representing an interface between mobile units and the communication channel used for sending data to remote storage. In fact, task `forward` must be performed by RS whenever a mobile unit is asked for writing (e.g. `write_pC` or `write_qB`) some data. *Forwarding* means receiving data from mobile services and writing it to remote storage. For security reasons, only mobile services are trusted systems which can ask the storage for freeing space (`req_space`) and can access the storage for reading (e.g., `read_pC`), while sending data can be performed only by the repository service.

After each forwarding, it may happen that the storage becomes full. This is why the `forward` task is non-deterministic and may yield either a {`available=T`} or a {`available=F`} write on the blackboard. On the other hand, a mobile service performing a {`req_space`} guarantees that remote storage will free some space, consequently it is deterministic and yields a {`available=T`} write on the blackboard. Finally, RS is allowed to send the remote storage a `commit` message, which asks the storage for compressing last received data and consequently makes files no longer available for reading.

The goal of the team is to collect both questionnaires and photos about all buildings. In Figure 2, a graphical representation of the desired workflow is shown where, initially: *(i)* all services are assumed to be in state `S0` and *(ii)* blackboard state is {`qA=qB=qC=pA=pB=pC=F, available=T`}. Edges outcoming from each state are labeled by sets of pairs $[C]/T$, whith the following semantics: *if, in current state, condition (guard) C holds, then task T must be assigned to some service.* Hence, each state transition may require, in general, the execution of a set of tasks. Observe that the target workflow is deterministic, that is, no two guards appearing inside different sets which label different edges outcoming from the same state can be true at the same time. Intuitively, after having filled all questionnaire and taken one photo per building, the target workflow requires services to iterate between states `S3`-`S8` until a a *good* photo for each building has been sent to the remote storage. Then,

the team must be ready to perform the operation again. In order to guarantee that pictures actually capture relevant aspects of the buildings, a sort of *peer review* strategy is adopted, i.e., each photo a unit writes in the remote storage must be read, evaluated and approved/rejected by a second unit. Both approval and rejection are publicly announced by writing a proper message on the blackboard (indeed, it is sufficient {`pC=F`} or {`pC=T`}). When all documents are sent (questionnaires are not subject to review process) a `commit` message is sent to the remote storage and the team can start a new iteration.

Finally, in Figure 3 a solution to the distributed composition problem is presented which consists of a set of local orchestrators which, upon execution, coordinate the services in order to realize the target workflow of Figure 2. Recall that each mobile service is attached to a *local orchestrator* which is able to both assigning tasks to the service itself and broadcasting messages. In order to accomplish their task, that is, realizing workflow transitions by properly assigning a subset of workflow requests to the respective services, local orchestrators need to access, for each transition: *(i)* the set of workflow requests and *(ii)* the whole set of messages other orchestrators sent. For this reason, both workflow requests and orchestrator messages are broadcasted. Each orchestrator transition is labeled by a pair I/O, which means: *if, in current state, I occurs, then perform O*, where $I = \langle A, M, s \rangle$ and $O = \langle A', M' \rangle$ with the following semantics: A is the set of tasks the workflow requests, M is the set of (broadcasted) messages the orchestrator received (including its own messages), s is the state reached by the attached service after tasks assigned by the orchestrator (A', see below) have been performed, $A' \subseteq A$ is the subset of actions the local orchestrator assigns to the attached service and M' is the set of messages the orchestrator broadcasts after the service performed A'. Notation has been compacted by introducing some shortcuts for set representation. In details, *(i)* "..." stands for "any set of elements": for instance, in the transition between states `S0` and `S1` of local orchestrator for MS_1 (Figure 3(a)), the set {`... commit`} represents any set (of tasks) containing `commit`; *(ii)* an element with the prefix "-" stands for "anything but the element, possibly nothing": for instance, in the first (from top) transition between states `S4` and `S5` of Figure 3(a), the set {`... modify_pA, -req_space`} stands for "any set (of tasks) not including `req_space` and including `modify_pA`".

Observe that local orchestrators are deterministic, that is, at each state, no ambiguity holds on which transition, if any, has to be selected. In general, this is due to the presence of messages, which are useful for selecting which tasks are to be assigned to each service. As an example, observe that third and fourth transitions of Figure 3(a) can be performed when a same set of tasks ({`... req_space, modify_pA`}) is requested by the workflow. The choice of which one is to be assigned to attached service depends on the messages the orchestrator received, which somehow represent other services current capabilities. So, in state `S4`, when the set of requested tasks includes both `req_space` and

modify_pA: *(i)* if received messages include m_3^1 (that is, the message local orchestrator for MS_1 sends when the service reaches state S3 from S1), then the orchestrator assigns tasks {modify_pA, req_space} to the service; *(ii)* otherwise, the set of assigned tasks is {modify_pA} and, consequently, there will be another local orchestrator assigning a set of tasks including req_space to its respective service, basically depending on the messages it received.

The orchestrators for MS_2 and MS_3 are roughly similar. The only noticeable difference is in transition between state $S4$ and $S5$ where the local orchestrator for MS_3 assigns the same action modify_pB for the attached service, independently of the other actions to be assigned. Indeed, orchestrators MS_1 and MS_2 makes this assignment dependent of the actions which are to be assigned to other services.

4 The Proposed Technique

The formal setting. A *Workflow Specification Kit (WfSK)* $\mathcal{K} = (\mathcal{A}, \mathcal{V})$ consists of a finite set of actions $\mathcal{A}$ and a finite set of variables $\mathcal{V}$, also called *blackboard*, that can assume only a finite set of values. Actions have known (but not modeled here) effects on the real world, while they do not change directly the blackboard.

Using a WfSK $\mathcal{K}$ one can define workflows over $\mathcal{K}$. Formally a workflow $\mathcal{W}$ over $\mathcal{K}$ is defined as a tuple: $\mathcal{W} = (S, s_0, G, \delta_{\mathcal{W}}, F)$, where:

- S is a finite set of workflow states;

- $s_0 \in S$ is the single initial state;

- G is a set of guards, i.e., formulas whose atoms are equalities (interpreted in the obvious way) involving variables and values.;

- $\delta_{\mathcal{W}} \subseteq S \times G \times 2^{\mathcal{A}-\{\emptyset\}} \times S$ is the workflow transition relation: $(s, g, A, s') \in \delta_{\mathcal{W}}$ denotes that in the state s, if the guard g is true in the current blackboard state, then the set of (concurrent) actions $A \subseteq \mathcal{A}$ is executed and the service changes state to s'; we insist that such a transition relation is actually *deterministic*: for no two distinct transitions (s, g_1, A_1, s_1) and (s, g_2, A_2, s_2) in $\delta_{\mathcal{W}}$ we have that $g_1(\gamma) = g_2(\gamma) = \mathtt{true}$, where γ is the current blackboard state;

- finally, $F \subseteq S$ is the set of states of the workflow that are final, that is, the states in which the workflow can stop executing.

In other words a workflow is a finite state program whose atomic instructions are sets of actions of $\mathcal{A}$ (more precisely invocation of actions), that branches on conditions to be evaluated on the current state of the blackboard $\mathcal{V}$.

What characterizes our setting however is that actions in the WfSK do not have a direct implementation, but instead are realized through *available services*. In other words action executions are not independent one from the other but they are constrained by the services that include them. A

service is essentially a program for a client (actually the orchestrator, as we have seen). Such a program, however, leaves the selection of the set of actions to perform next to the client itself (actually the orchestrator). More precisely, at each step the program presents to the client (orchestrator) a choice of available sets of (concurrent) actions; the client (orchestrator) selects one of such sets; the actions in the selected set are executed concurrently; and so on.

Formally, a *service* $\mathcal{S}$ is a tuple $\mathcal{S} = (S, s_0, G, C, \delta_{\mathcal{S}}, F)$ where:

- S is a finite set of states;

- $s_0 \in S$ is the single initial state;

- G is a set of guards, as described for workflows;

- C is a set of partial variable assignment for $\mathcal{V}$, that is used to update the state of the blackboard;

- $\delta_{\mathcal{S}} \subseteq S \times G \times 2^{\mathcal{A}-\{\emptyset\}} \times C \times S$ is the service transition relation, where $(s, g, A, c, s') \in \delta_{\mathcal{S}}$ denotes that in the state s, if the guard g is true in the current blackboard state and it is requested the execution of the set of actions $A \subseteq \mathcal{A}$, then the blackboard state is updated according to c and the service changes state to s';

- finally, $F \subseteq S$ is the set of states that can be considered final, that is, the states in which the service can stop executing, but does not necessarily have to.

Observe that, in general, services are *nondeterministic* in the sense that they may allow more than one transition with the same set A of actions and compatible guards evaluating to the same truth value [2]. As a result, when the client (orchestrator) instructs a service to execute a given set of actions, it cannot be certain of which choices it will have later on, since that depends on what transition is actually executed – nondeterministic services are only partially controllable.

To each service we associate a *local orchestrator*. A local orchestrator is a module that can be (externally) attached to a service in order to control its operation. It has the ability of activating-resuming its controlled service by instructing it to execute a set of actions. Also, the orchestrator has the ability of broadcasting messages from a given set of $\mathcal{M}$ after observing how the attached service evolved w.r.t. the delegated set of actions, and to access all messages broadcasted by the other local orchestrators at every step. Notice that the local orchestrator is not even aware of the existence of the other services: all it can do is to access their broadcasted messages. Lastly, the orchestrator has full observability on the blackboard state.

A *(messages extended) service history* $h_{\mathcal{S}}^+$ for a given service $\mathcal{S} = (S, s_0, G, C, \delta_{\mathcal{S}}, F)$, starting in a blackboard state γ_0, is any finite sequence of the form $(s^0, \gamma^0, M^0) \cdot A^1 \cdot (s^1, \gamma^1, M^1) \cdots (s^{\ell-1}, \gamma^{\ell-1}, M^{\ell-1}) \cdot A^\ell \cdot (s^\ell, \gamma^\ell, M^\ell)$, for some $\ell \geq 0$, such that for all $0 \leq k \leq \ell$ and $0 \leq j \leq \ell - 1$:

---

[2]Note that this kind of nondeterminism is of a *devilish* nature – the actual choice is out of the client (orchestrator) control.

(a) Local orchestrator for MS_1

$<\{ ... commit \}, \{ ... \}, \{ ... \} > / <\{ \}, \{ \} >$

$<\{ ... eval\_pC \}, \{ ... m_5{}^1 \}, S0 > / <\{ eval\_pC \}, \{ m_0{}^1 \} >$

$<\{ ... compile\_qB \}, \{ ... m_0{}^1 \}, S4 > / <\{ compile\_qB \}, \{ m_4{}^1 \} >$

$<\{ ... read\_pC \}, \{ ... m_0{}^1 \}, S5 > / <\{ read\_pC \}, \{ m_5{}^1 \} >$

$<\{ ... write\_qB \}, \{ ... m_4{}^1 \}, S0 > / <\{ write\_qB \}, \{ m_0{}^1 \} >$

$<\{ ... move\_A \}, \{ ... m_0{}^1 \}, S1 > / <\{ move\_A \}, \{ m_1{}^1 \} >$

$<\{ ... move\_A \}, \{ ... m_0{}^1 \}, S1 > / <\{ move\_A \}, \{ m_1{}^1 \} >$

$<\{ ... write\_pA \}, \{ ... m_3{}^1 \}, S0 > / <\{ write\_pA \}, \{ m_0{}^1 \} >$

$<\{ ... - req\_space\ modify\_pA \}, \{ ... m_2{}^1 \} \cup \{ ... m_3{}^1 \}, S3 > / <\{ modify\_pA \}, \{ m_3{}^1 \} >$

$<\{ ... req\_space \}, \{ ... m_0{}^1 \}, S0 > / <\{ req\_space \}, \{ m_1{}^1 \} >$

$<\{ ... take\_pA \}, \{ ... m_1{}^1 \}, S2 > / <\{ take\_pA \}, \{ m_1{}^1 \} >$

$<\{ ... req\_space, modify\_Pa \}, \{ ... m_3{}^1 \}, S3 > / <\{ modify\_pA, req\_space \}, \{ m_3{}^1 \} >$

$<\{ ... take\_pA \}, \{ ... m_1{}^1 \}, S3 > / <\{ take\_pA \}, \{ m_3{}^1 \} >$

$< \{ ... req\_space, modify\_pA \}, \{ ... m_2{}^1, m_6{}^2 \}, S3 > / <\{ modify\_pA \}, \{ m_3{}^0 \} >$

$<\{ ... req\_space, modify\_pA \}, \{ ... m_2{}^1, m_6{}^2 \}, S3 > / <\{ modify\_pA \}, \{ m_3{}^1 \} >$

(b) Local orchestrator for MS_2

$<\{ ... commit \}, \{ ... \}, \{ ... \} > / <\{ \}, \{ \} >$

$<\{ ... compile\_qC \}, \{ ... m_0{}^2 \}, S3 > / <\{ compile\_qC \}, \{ m_3{}^2 \} >$

$<\{ ... read\_pB \}, \{ ... m_0{}^2 \}, S1 > / <\{ read\_pC \}, \{ m_1{}^2 \} >$

$<\{ ... write\_qC \}, \{ ... m_3{}^2 \}, S4 > / <\{ write\_qC \}, \{ m_4{}^2 \} >$

$<\{ ... move\_C \}, \{ ... m_4{}^2 \}, S5 > / <\{ move\_C \}, \{ m_5{}^2 \} >$

$<\{ ... move\_C \}, \{ ... m_4{}^2 \}, S5 > / <\{ move\_C \}, \{ m_5{}^2 \} >$

$<\{ ... write\_pC \}, \{ ... m_6{}^2 \}, S0 > / <\{ write\_pA \}, \{ m_0{}^2 \} >$

$<\{ ... - req\_space\ modify\_pC \}, \{ ... m_6{}^2 \}, S6 > / <\{ modify\_pC \}, \{ m_6{}^2 \} >$

$<\{ ... req\_space, modify\_Pa \}, \{ ... m_3{}^2 \}, S6 > / <\{ modify\_pC \}, \{ m_6{}^2 \} >$

$<\{ ... take\_pC \}, \{ ... m_4{}^2 \}, S6 > / <\{ take\_pC \}, \{ m_6{}^2 \} >$

$< \{ ... req\_space, modify\_pA \}, \{ ... m_2{}^1, m_6{}^2 \}, S6 > / <\{ req\_space \}, \{ m_6{}^2 \} >$

$<\{ ... req\_space, modify\_pC \}, \{ ... m_2{}^1, m_6{}^2 \}, S6 > / <\{ req\_space, modify\_pC \}, \{ m_6{}^2 \} >$

(c) Local orchestrator for MS_3

$<\{ ... commit \}, \{ ... \}, \{ ... \} > / <\{ \}, \{ \} >$

$<\{ ... eval\_pA \}, \{ ... m_1{}^3 \}, S0 > / <\{ eval\_pA \}, \{ m_0{}^3 \} >$

$<\{ ... compile\_qA \}, \{ ... m_0{}^3 \}, S2 > / <\{ compile\_qB \}, \{ m_3{}^3 \} >$

$<\{ ... read\_pA \}, \{ ... m_0{}^3 \}, S1 > / <\{ read\_pA \}, \{ m_1{}^3 \} >$

$<\{ ... write\_qA \}, \{ ... m_2{}^3 \}, S0 > / <\{ write\_qA \}, \{ m_0{}^3 \} >$

$<\{ ... move\_B \}, \{ ... m_0{}^3 \}, S3 > / <\{ move\_B \}, \{ m_3{}^3 \} >$

$<\{ ... move\_B \}, \{ ... m_0{}^3 \}, S3 > / <\{ move\_B \}, \{ m_3{}^3 \} >$

$<\{ ... write\_pB \}, \{ ... m_4{}^3 \}, S0 > / <\{ write\_pB \}, \{ m_0{}^3 \} >$

$<\{ ... take\_pB \}, \{ ... m_3{}^3 \}, S4 > / <\{ take\_pB \}, \{ m_4{}^3 \} >$

$<\{ ... modify\_pB \}, \{ ... m_4{}^3 \}, S4 > / <\{ modify\_pB \}, \{ m_4{}^3 \} >$

(d) Local orchestrator for RS

$<\{ ... commit \}, \{ ... \}, S0 > / <\{ commit \}, \{ m_0{}^4 \} >$

τ

$<\{ - commit \}, \{ ... \}, S0 > / <\{ \}, \{ m_0{}^4 \} >$

$<\{ ... forward \}, \{ ... \}, S0 > / <\{ forward \}, \{ m_0{}^4 \} >$

$<\{ ... forward \}, \{ ... \}, S0 > / <\{ forward \}, \{ m_0{}^4 \} >$

τ

$\tau = <\{ ... \}, \{ ... \}, \{ ... \} > / <\{ \}, \{ \} >$

Figure 3. Local orchestrators

- $s^0 = s_0$;

- $\gamma^0 = \gamma_0$;

- $A^k \subseteq \mathcal{A}$;

- $(s^j, g, A^{j+1}, c, s^{j+1}) \in \delta_i$ with $g(\gamma^j) = \mathtt{true}$ and $c(\gamma^j) = \gamma^{j+1}$ that is, service $\mathcal{S}$ *can* evolve from its current state s^j to state s^{j+1} while updating the backboard state from γ^j to γ^{j+1} according to what specified in c;

- $M^0 = \emptyset$ and $M^k \subseteq \mathcal{M}$, for all $k \in \{0, \dots, \ell\}$.

The set $\mathcal{H}_{\mathcal{B}}^+$ denotes the set of all *service histories* for $\mathcal{S}$.

Formally, a *local orchestrator* $\mathcal{O} = (P, B)$ for service $\mathcal{S}$ is a pair of functions of the following form:

$$P : \mathcal{H}_{\mathcal{B}}^+ \times 2^{\mathcal{A}} \to 2^{\mathcal{A}}; \qquad B : \mathcal{H}_{\mathcal{B}}^+ \times 2^{\mathcal{A}} \times S \to 2^{\mathcal{M}}.$$

Function P states what actions $A' \subseteq A$ to delegate to the attached service at local service history $h_{\mathcal{B}}^+$ when actions A were requested. Function B states what messages, if any, are to be broadcasted under the same circumstances and the fact that the attached service has just moved to state s after executing actions A'. We attach one local orchestrator $\mathcal{O}_i$ to each available service $\mathcal{S}_i$. In general, local orchestrators can have infinite states.

A *distributed orchestrator* is a set $\mathcal{X} = (\mathcal{O}_1, \dots, \mathcal{O}_n)$ of local orchestrators, one for each available service $\mathcal{S}_i$.

We call *device* the pair $\mathcal{D} = (\mathcal{S}, \mathcal{O})$ constituted by a service $\mathcal{S}$ and its local orchestrator $\mathcal{O}$.

A *workflow mobile environment (WfME)* is constituted by a finite set of devices $\mathcal{E} = (\mathcal{D}_1, \dots, \mathcal{D}_n)$ defined over the same WfSK $\mathcal{K}$.

Local Orchestrator Synthesis. The problem we are interested in is the following: given n services $\mathcal{S}_1, \dots, \mathcal{S}_n$ over WfSK $\mathcal{K} = (\mathcal{A}, \mathcal{V})$ and an initial blackboard state γ_0, and a workflow $\mathcal{W}$ over $\mathcal{K}$, *synthesize a distributed orchestrator, i.e., a team of n local orchestrators, such that the workflow is realized by concurrently running all services under the control of their respective orchestrators.*

More precisely, let $\mathcal{S}_1, \dots, \mathcal{S}_n$ be the n services, each with $\mathcal{S}_i = (S_i, s_{i0}, G_i, C_i, \delta_i, F_i)$, γ_0 be the initial state of the blackboard, and $\mathcal{W} = (S_{\mathcal{W}}, s_{\mathcal{W}0}, G_{\mathcal{W}}, \delta_{\mathcal{W}}, F_{\mathcal{W}})$ the workflow to be realized.

We start by observing that the workflow (being deterministic) is completely characterized by its set of *traces*, that is, by the set of infinite action sequences that are faithful to its transitions, and of finite sequences that in addition lead to a final state. More formally, a *trace* for $\mathcal{W}$ is a sequence of pairs (g, A), where $g \in G$ is a guard over $\mathcal{V}$ and $A \subseteq \mathcal{A}$ is non-empty set of actions, of the form $t = (g^1, A^1) \cdot (g^2, A^2) \cdots$ such that there exists an execution history [3] for $\mathcal{W}$, $(s^0, \gamma^0) \cdot A^1 \cdot (s^1, \gamma^1) \cdots$ where $g^i(\gamma^{i-1}) = \mathtt{true}$ for all $i \geq 1$. If the trace

---

^{3}Analogous the execution histories defined for services except that they do not include messages.

$t = (g^1, A^1) \cdots (g^\ell, A^\ell)$ is finite, then there exists a finite execution history $(s^0, \gamma^0) \cdots (s^\ell, \gamma^\ell) \cdots$ with $s^\ell \in F_{\mathcal{W}}$.

Now, given a trace $t = (g^1, A^1) \cdot (g^2, A^2) \cdots$ of the workflow $\mathcal{W}$, we say that *a distributed orchestrator* $\mathcal{X} = (\mathcal{O}_1, \dots, \mathcal{O}_n)$ realizes the trace t iff for all ℓ and for all "system history" $h^\ell \in \mathcal{H}_{t,\mathcal{X}}^\ell$ (formally defined defined below) with $g^{\ell+1}(\gamma^\ell) = \mathtt{true}$ in the last configuration of h^ℓ, we have that $\mathsf{Ext}_{t,\mathcal{X}}(h^\ell, A^{\ell+1})$ is nonempty, where $\mathsf{Ext}_{t,\mathcal{X}}(h, A)$ is the set of $(|h| + 1)$-length system histories of the form $h \cdot [A_1, \dots, A_n] \cdot (s_1^{|h|+1}, \dots, s_n^{|h|+1}, \gamma^{|h|+1}, M^{|h|+1})$ such that:

- $(s_1^{|h|}, \dots, s_n^{|h|}, \gamma^{|h|}, M^{|h|})$ is the last configuration in h;

- $A = \bigcup_{i=1}^n A_i$, that is, the requested set of actions A is fulfilled by putting together all the actions executed by every service.

- $P_i(h|_i, A) = A_i$ for all $i \in \{1, \dots, n\}$, that is, the local orchestrator $\mathcal{O}_i$ instructed service $\mathcal{S}_i$ to execute actions A_i;

- $(s_i^{|h|}, g_i, A_i, c_i, s_i^{|h|+1}) \in \delta_i$ with $g_i(\gamma^{|h|}) = \mathtt{true}$, that is, service $\mathcal{S}_i$ can evolve from its current state $s_i^{|h|}$ to state $s_i^{|h|+1}$ w.r.t. the (current) variable assignment $\gamma^{|h|}$;

- $\gamma^{|h|+1} \in C(\gamma^{|h|})$, where $C = \{c_1, \dots, c_n\}$ is the set of the partial variable assignments c_i due to each of the service, and $C(\gamma^{|h|})$ is the set of blackboard states that are obtained from $\gamma^{|h|}$ by applying each $c_1, \dots, c_n$ in every possible order;

- $M^{|h|+1} = \bigcup_{i=1}^n B_i(h|_i, A, s^{|h|+1})$, that is, the set of broadcasted messages is the union of all messages broadcasted by each local orchestrator.

The set $\mathcal{H}_{t,\mathcal{X}}^k$ of all histories that implement the first k actions of trace t and is prescribed by $\mathcal{X}$ is defined as follows:

- $\mathcal{H}_{t,\mathcal{X}}^0 = \{(s_{10}, \dots, s_{n0}, \gamma_0, \emptyset)\}$;

- $\mathcal{H}_{t,\mathcal{X}}^{k+1} = \bigcup_{h^k \in \mathcal{H}_{t,\mathcal{X}}^k} \mathsf{Ext}_{t,\mathcal{X}}(h^k, A^{k+1})$, $k \geq 0$;

In addition if a trace is finite and ends after m actions, and all along all its guards are satisfied, we have that all histories in $\mathcal{H}_{t,\mathcal{X}}^m$ end with all services in a final state. Finally, we say that a *distributed orchestrator* $\mathcal{X} = (\mathcal{O}_1, \dots, \mathcal{O}_n)$ realizes the workflow $\mathcal{W}$ if it realizes all its traces.

In order to understand the above definitions, let us observe that, intuitively, the team of local orchestrators realizes a trace if, as long as the guards in the trace are satisfied, they can globally perform all actions prescribed by the trace (each of the local orchestrators instructs its service to do some of them). In order to do so, each local orchestrator can

use the history of its service together with the (global) messages that have been broadcasted so far. In some sense, implicitly through such messages, each local orchestrator gets information on the other service local histories in order to take the right decision. Furthermore, at each step, each local orchestrator broadcasts messages. Such messages will be used in the next step by all service orchestrators to choose how to proceed.

Results. Our main technical results are summarized by the next theorem.

Theorem 4.1 *There exists a sound, complete and terminating procedure for computing a distributed orchestrator $\mathcal{X} = (\mathcal{O}_1, \ldots, \mathcal{O}_n)$ that realizes a workflow $\mathcal{W}$ over a WfSK $\mathcal{K}$ relative to services $\mathcal{S}_1, \ldots, \mathcal{S}_n$ over $\mathcal{K}$ and blackboard state γ_0. Moreover each local orchestrator $\mathcal{O}_i$ returned by such a procedure is finite state and require a finite number of messages (more precisely message types).*

Observe that we did not put any finiteness limitation on the number of states of the local orchestrators nor on the number of messages to be exchanged. This theorem, by restricting oneself to finite number of states and messages, does not loose generality.

The synthesis procedure is based on the general techniques proposed in [3, 4, 6], based on a reduction of the problem to satisfiability of a Propositional Dynamic Logic formula [8] whose models roughly correspond to orchestrators [4]. From a realization point of view, such a procedure can be implemented through the same basic algorithms behind the success of the description logics-based reasoning systems used for OWL[5], such as FaCT[6], Racer[7], Pellet[8], and hence its applicability appears to be quite promising. The reader should note that the technique is not exploited at run-time, but before the execution of the services and the local orchestrators effectively happens, therefore the requirements of mobile scenarios are not violated (e.g., just to have a concrete example, it can be run on a laptop on the jeep taking the team on the operation field).

5   Conclusion

In this paper, we have studied the workflow composition problem within a distributed general setting; the solutions proposed here are therefore palatable to a wide range of contexts, e.g., nomadic teams in emergency management, in which we have multiple independent agents and a centralized solution is not conceivable. Indeed we plan to validate the approach in the context of a research project about emergency management, namely WORKPAD.

We close the paper by observing that the kind of problems we dealt with are special forms of reactive process synthesis [14, 15]. It is well known that, in general, distributed solutions are much harder to get than centralized ones [15, 9]. This has not hampered our approach since we allow for equipping local controllers with autonomous message exchange capabilities, even if such capabilities are not present in the services that they control.

References

[1] B. Benatallah, M. Dumas, and Q. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 17, 2005.

[2] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proc. VLDB 2005*, 2005.

[3] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioural descriptions. *International Journal of Cooperative Information Systems*, 14(4):333–376, 2005.

[4] D. Berardi, D. Calvanese, G. De Giacomo, and M. Mecella. Composing web services with nondeterministic behavior. In *Proc. ICWS 2006*, 2006.

[5] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *Proc. WWW 2004 – Alternate Track Papers & Posters*, 2004.

[6] G. De Giacomo and S. Sardina. Automatic synthesis of new behaviors from a library of available behaviors. In *Proc. of IJCAI 2007*, pages 1866–1871, Hyderabad, India, Jan. 2007.

[7] F. De Rosa, A. Malizia, and M. Mecella. Disconnection prediction in mobile ad hoc networks for supporting cooperative work. *IEEE Pervasive Computing*, 4(3):62 – 70, 2005.

[8] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[9] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Proc. of LICS 2001*, page 389, 2001.

[10] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler. Information gathering during planning for web service composition. In *Proc. Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[11] S. A. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In *Proc. KR 2002*, pages 482–496, 2002.

[12] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB Journal*, 12(4):333 – 351, 2003.

[13] A. L. Murphy, G. P. Picco, and G. C. Roman. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodologies*, 15(3):279 – 328, 2006.

[14] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. POPL 1989*, pages 179–190, 1989.

[15] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. of FOCS 1990*, pages 746–757, 1990.

[16] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *Proc. ISWC 2004*, volume 3298 of *LNCS*, pages 380–394. Springer, 2004.

---

[4]We are also studying alternatives based on model checking techniques.
[5]http://www.omg.org/uml/
[6]http://www.cs.man.ac.uk/ horrocks/FaCT/
[7]http://www.sts.tu-harburg.de/ r.f.moeller/racer/
[8]http://www.mindswap.org/2003/pellet/