Situation-aware composition and execution in dynamic environments by automated planning

Qiang Lu\textsuperscript{a,b,*}, Justin Wilson\textsuperscript{c}, Yixin Chen\textsuperscript{c}, Christopher Gill\textsuperscript{c}, Louis Thomas\textsuperscript{c}, Gruia-Catalin Roman\textsuperscript{d}, Guoliang Chen\textsuperscript{b}

\textsuperscript{a} College of Information Engineering, Yangzhou University, 196 Huayang Road, Yangzhou, Jiangsu, 225127, China
\textsuperscript{b} School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China
\textsuperscript{c} Department of Computer Science & Engineering, Washington University in Saint Louis, St. Louis, MO 63130, USA
\textsuperscript{d} School of Engineering, The University of New Mexico, Albuquerque, NM 87131, USA

\begin{abstract}
Existing workflow management systems execute complex sequences of tasks by invoking corresponding services identified at execution time and ordered according to a graph that captures the workflow execution rules and data dependencies. The technology is mature and is used across the Internet to support a wide range of business transactions. However, by taking advantage of a relatively stable communication and service infrastructure, this paper extends the applicability of workflow technology by considering settings in which service availability is highly variable due to the mobility of people and devices and due to frequent and extended disconnections from the wired infrastructure. This paper extends workflow technology to support goal-directed coordination among physically mobile people and devices that form a transient community over a mobile ad hoc network (MANET). Our approach is based on the notion of open workflows, i.e., workflows that are constructed dynamically using available knowledge and services. Because the workflow construction and execution are situation aware, different workflows are built under different circumstances even when the need being addressed is unchanged. We also introduce a framework that co-designs workflow allocation and host scheduling in a unified way, which can handle communication range limitations in a mobile environment while previous methods cannot.

Specifically, we transform the collaboration problem into a temporal planning model and then solve it using automated planners. We integrate this framework into the CiAN workflow management system, which was developed in our previous work. Results of experiments we have conducted show that our approach significantly broadens the scope of previous research by solving the challenges of availability, mobility, and communication constraints in MANET domains.

\end{abstract}

\section{Introduction}

As the increasing usage of small, powerful wireless devices, computing will embrace the frequent, transient, ad hoc interactions of mobile environments. Since computing and communication become more and more integrated into the fabric of our society, there will be large demands for new kinds of enterprises and new forms of social interactions. In this paper, we propose a new framework to support arbitrary collaborations among groups of people (and their personal devices) in a mobile ad hoc network (MANET). Application domains that motivate or even require this form of interaction include low profile military operations, emergency responses to major natural disasters, scientific expeditions in remote parts of the globe, field hospitals, and large construction sites. These application domains exhibit several key features: ad hoc interactions among people, high levels of mobility, the need to respond to unexpected developments, the use of locally available resources, prescribed rules of operation, and specialized knowhow.

The coordination problem in this MANET environments usually results in how to arrange a series of activities or tasks according to a specific order to achieve a high level goal, which can be seen as a workflow that is reactive, opportunistic, composite, and constrained by the set of participants present on the site along with their knowledge and resources. Current workflow middleware allows people to initiate complex goal-oriented activities that leverage services made available by a wide range of service-oriented portals. In a typical scenario, a user employs a web browser...
to make a request to a workflow engine responsible for executing a predefined workflow that can satisfy the specific user need. While the workflow itself is fixed and defined ahead of time, its execution is adaptive as services are selected at runtime through a discovery and matching process facilitated by the service specifications that are part of the workflow and by the service directories and service providers.

However, consider the new features in MANET domains, the stability assumptions under which such workflow technology operates need to be reexamined. In many situations, people and devices are agents that deliver services interacting with the physical world. This leads to an extended view of what a service is and induces a high level of variability into the system. People are not always ready to help and, similarly, devices can be busy, exhibit independent behavior, or be subject to environmental controls. Generally speaking, different hosts have different capabilities, availability, and other resources. In a dynamic community which members can join and leave at will in a MANET environment, there is no way to know which resources can be used before the participants have been fixed. Thus, there will be no fixed server that can store all resources. In such an environment, the availability of resources is distributed across all participants.

Designing a workflow management system (WfMS) for these domains faces many challenges, as hosts may move and service availability may depend upon which hosts are within communication range. To extend the use of workflow technology to these highly dynamic environments, it is also important to consider the idea of building workflows on the fly in response to a specific need being expressed in a particular setting. At one extreme one might consider for each user the availability of a fixed set of workflows among which one is selected automatically based on existing resources and other considerations. At the other extreme one could employ sophisticated automatic reasoning systems to construct a workflow totally from scratch. Our approach assumes that knowledge about how to respond to certain basic needs is available in the form of small workflow fragments distributed across a set of hosts or accessible from servers. Workflow fragments are merely small workflows (possibly even a single task) that are intended to be composed into larger workflows at a later time. As new and unanticipated needs are identified complex workflows are constructed by composing existing workflow fragments in a way that pays attention to the availability of resources across the relevant spatiotemporal domain.

These challenges make most existing workflow management algorithms inadequate since they do not consider the mobility and communication constraints in the ad hoc settings. Standard workflow management systems, such as ActiveBPEL (Active Endpoints, 2008), Oracle Workflow Engine (Oracle Inc., 2007), JBoss (JBoss Labs, 2007), and BizTalk (Microsoft Corp, 2007), are designed to work in fully wired environments, such as corporate LANs or across the Internet. Reliance on centralized control and reliable communication means that such solutions cannot robustly operate under the constraints of dynamic mobile environments. The work on federating separate execution engines running independent workflows (Omicini et al., 2006) removes the requirement of centralized control. Kumar et al. (2004) investigate decentralized orchestration of a single workflow by partitioning the workflow at build time and using message passing at run time. However, both approaches still assume reliable communication and a fixed group of hosts. A closely related WfMS is WORKPAD (Mecella et al., 2006), which was designed to support collaborative work of human operators in emergency/disaster scenarios. WORKPAD requires a stable wireless connected community where all devices would not leave or join in and at least one member of the MANET to be connected with a central coordinating entity that orchestrates the workflow and shoulders any heavy computational loads.

While in MANET domains the workflow is generated on the fly to match the present situation, all above related systems assume that a fully specified workflow already exists. A related work which supports dynamic workflows is Worklets (Adams et al., 2006). It adopts a service oriented architecture which supports flexible work practices based on accepted ideas of Activity Theory which focuses on understanding human activities and work practices (Nardi, 1996).

Some recent research efforts targeted toward the MANET domains (Sen et al., 2007, 2008; Thomas et al., 2009; Haitjema et al., 2010) still confront some limitations. For example, CiAN adopts a simple heuristic allocation algorithm which divides the workflow allocation into sub-problems recursively (Sen et al., 2007, 2008). Instead of using a recursive allocation algorithm, Haitjema solves this problem by transforming it to a numeric temporal planning problem and calling a temporal planner to find a feasible allocation (Haitjema et al., 2010). These works de-couple workflow allocation from host scheduling which largely simplifies the workflow problems. However, they may fail to find a feasible solution that supports a given workflow allocation due to the simplification, even if there exists a feasible solution under another workflow allocation.

In our previous work (Thomas et al., 2009), we addressed this problem by dividing it into two independent phases: constructing a feasible workflow plan and allocating all tasks in the workflow plan to hosts. The workflow construction phase first builds a workflow graph which consists of all feasible workflows and then selects one feasible workflow by using an iterative pruning algorithm. Then in the allocation phase, it simply determines how and to whom to distribute each task in the constructed workflow according to each host’s task availability information. When it assigns a task to a host, it allows the host to decide independently how and when to move to the task specific location, when to execute the task, and how to exchange data and coordinate with other hosts. Splitting the original problem into these two phases largely simplifies the planning problem. However, it makes two assumptions: (1) the constructed workflow plan can be successfully allocated and (2) all hosts will find feasible movement schedules to satisfy the host communication constraints. In real-world applications, especially in dynamic MANET environments, such assumptions may be violated because they do not consider the host movement and communication constraints when allocating the workflow. In other words, by decoupling workflow allocation from host scheduling, the approach may fail to find a feasible solution that supports a selected workflow allocation, even if there exists a feasible solution under another workflow allocation solution. Even if all tasks can be allocated in the first phase, it still cannot guarantee that two hosts are in communication range at a future time when they need to communicate with each other. Moreover, considering the feasible workflow plan, workflow plan allocation, and host schedule together gives a larger decision space which may lead to more preferable (e.g. shorter) plans. The co-design of workflow plan, workflow allocation, and host scheduling, which considering the dependencies, communication, and temporal constraints, is beyond the capability of the allocation algorithm in CiAN, or any other existing workflow algorithms we know of.

In order to address these challenges for collaboration in MANETs, in this paper we extend the open workflow scheme introduced in our previous work (Thomas et al., 2009). Fig. 1 gives an overview of our co-design open workflow (COW) scheme and our previous open workflow scheme. In these schemes, we both first use the open workflow scheme to construct workflow problems dynamically. Specifically, once a new task requirement is given, we first collect relevant knowledge from participants and construct a workflow problem represented by a graph. After that,
our previous scheme divides the remaining problems into three independent processes, workflow plan, workflow allocation, and host scheduling, and solves them step by step. As a contrast, in our new scheme we consider the workflow plan, workflow allocation, and host scheduling together as a COW problem and introduce a temporal planning approach to solve the co-design problem in a unified framework.

To show the advantages of our COW scheme, we summarize the supporting features of the related WfMSs discussed in this section in Table 1. We can see that the (Thomas et al., 2009) and COW WfMSs are the only two which fully support MANET features: decentralized control, mobile ad hoc network, dynamically generated workflow specification, and dynamic hosts.

In summary, our contributions are as follows:

1. We start with an open workflow scheme that dynamically constructs workflow problems using available knowledge and services. Specifically, the workflow problem is built on the fly in response to a specific user need through the composition of a set of pre-existing services and in a way that ensures the availability of all needed services during its execution, which may be subject to specific spatiotemporal constraints.

2. Based on constructed workflow problems, we automatically translate the co-design problem to a temporal planning problem, and call a temporal planner to find a feasible workflow plan, a workflow allocation, and suitable host schedules. Our model thus co-designs the workflow plan, workflow allocation, and host scheduling, and can handle dynamic initial and goal states to support online workflow construction.

In the remainder of this paper we introduce and explore the open workflow paradigm, the temporal planning framework for the co-design problem, and report on the results of building a platform for experimentation with this approach. We introduce the temporal planning problems, workflow formalization, and a motivating example to show the possibilities and advantages of the open workflow paradigm in Section 2. Section 3 explains how we construct open workflow problems through capturing the dynamics of service availability, solve the co-design problem by using temporal planning techniques, and allocate and execute tasks according to the workflow plan and host schedules generated from the solution plan of the planning problem. In Section 4, we present our open workflow management system and discuss its architecture. In Section 5, we evaluate the performance of our system. Section 6 highlights related research and contrasts it with this work. We provide conclusions and discuss directions for future work in Section 7.

2. Background

In this section, we first formally define the temporal planning problems and the workflow problem model for dynamic workflow construction, and then introduce a motivating example.

(a) Previous open workflow scheme (b) co-design open workflow scheme.

Fig. 1. An overview of the previous open workflow scheme and co-design open workflow scheme. The main differences between the two schemes are highlighted using shaded boxes.
2.1. **PDDL formalization**

Our method applies to temporal planning problems defined in a STRIPS (Stanford Research Institute Problem Solver) (Fikes and Nilsson, 1971) formalization of PDDL2.1 (Planning Domain Definition Language) (Fox and Long, 2003) and PDDL2.2 (Edelkamp and Hoffmann, 2004). In these problems, actions have starting and ending conditions, overall preconditions, starting and ending effects, all of which are conjuncts of propositions. Each action has a duration that is defined by a positive rational number.

We now formally define temporally planning problems. A fact $f$ is an atomic proposition that can be either true or false; we use $f_t$ to represent the fact $f$ at time $t$. A state $S$ is a set of fact propositions. We use $S_t$ to represent the state at time $t$. For each fact $f_t$, $S_t$ can only include $f_t$ or $\neg f_t$. For convenience, we assume $f_t(S_t) = (\neg f_t(S_t))$.

**Definition 1** (Durative action). A durative action $a$ is defined by a tuple $(\rho, \pi, \pi_\flat, \pi_\sharp, \alpha_\flat, \alpha_\sharp)$, where $\rho$ is the duration of $a$; $\pi$, $\pi_\flat$, $\pi_\sharp$ are precondition fact sets that must be true at the start and at the end of $a$, respectively; $\alpha_\flat$, $\alpha_\sharp$ are the effect fact sets at the start and the end of $a$, respectively.

In this paper, we assume that action durations are rational number where $\rho(a) > 0$. Given a durative action $a$, we use $\pi$ to represent $\pi(a)$. The same abbreviation applies to $\pi_\flat$, $\pi_\sharp$, $\alpha_\flat$, and $\alpha_\sharp$. In PDDL2.1, the annotations of temporal precondition and overall facts are: (1) $\pi:\{\text{at start } f\}$, (2) $\pi_\flat$ : $\{\text{at start } f\}$ and (3) $\pi_\sharp$ : $\{\text{at end } f\}$. The annotations of effects are: (1) $\alpha_\flat$ : $\{\text{at start } f\}$ and (2) $\alpha_\sharp$ : $\{\text{at end } f\}$.

Given an action $a$ and a sequence of states $[S_0, S_1, \ldots, S_n]$, a plan is valid at time $t$ (denoted as $a_t$) if the following conditions are satisfied: (a) $\forall f \in \pi, f_t(S_0)$; (b) $\forall f \in \pi_\flat, f_t(S_0)$; and (c) $\forall f \in \pi_\sharp, f_t(S_0)$.

A timed initial literal like $(at f_t)$ or $(at f_t$ (not $f_t$)), which is a feature of PDDL2.2 (Edelkamp and Hoffmann, 2004), represents the fact $f$ that becomes true or false at a certain time $t$.

**Definition 3** (Solution plan). Given a temporal planning problem $\Pi = (\mathcal{F}, \mathcal{A}, I, G)$, a plan $P = (p_0, p_1, \ldots, p_n-1)$ is a sequence of action sets, where each action set $p_i \subseteq A$ indicates the actions executed at time $t$. $P$ is a solution plan if there exists a state sequence $S_0, S_1, \ldots, S_n$ satisfying:

1. $S_0 = I$;
2. for each action $a_t \in p_t$, $a_t$ is valid at time $t$, and $S_{t+1} = S_t \cup f_t$; and (3) for all $f \in G, f_t(S_n)$.

**Definition 4** (Solution makespan). Given a plan $P = (p_0, p_1, \ldots, p_{n-1})$, the makespan is defined as the time duration of the plan:

$$t = \max \{\max \{t | (p(0) - 1)\}.$$

The formulation of the temporal planning is a subset of PDDL2.1 and PDDL2.2. Specifically, our temporal planner supports PDDL2.1 features, including predicate representations, typed representations, untyped representations, grounded representations, and negative preconditions, and PDDL2.2 feature timed initial literals. The PDDL features which are not supported include object fluent representations, schematic representations, ADL conditions, conditional effects, universal effects, derived predicates, and numeric state variables.

2.2. **Workflow formalization**

Workflows can be defined as a set of related tasks that are arranged according to a specific order and structure to accomplish a higher level goal in a collaborative manner. A workflow represents a single abstract behavior or accomplishment without completely specifying how it must be performed.

**Definition 5** (Workflow task). A workflow task is defined as $t = (C_{pre}, C_{post}, J, L, d)$, where $C_{pre}$ is the set of preconditions that must be met before the task can be performed, $C_{post}$ is the set of postconditions that describe the results of performing the task, $J$ indicates the junction relation of the preconditions, $L$ is the set of available locations (one or more) where the task can be executed, and $d$ is the time duration of executing the task.

As the postcondition of one task can be the precondition of another, we can treat them uniformly as conditions, denoted as $C$. For each task, the junction of preconditions is either conjunctive (denoted as $\land$), requiring all of its preconditions, or disjunctive (denoted as $\lor$), requiring only one of its preconditions. We assume that a task produces or establishes all of its postconditions. Similar to the PDDL fact and action definitions, each condition and task is represented as a semantic label, where each label has a distinct meaning.
Fig. 2(a) shows some workflow tasks used in a university-wide emergency response system. For example, task “throw main breaker” has the precondition “wind tunnel on” and postcondition “wind tunnel off”. This task can only be executed in the location “aeronautics dept”. For the task “check common dept. response checklist”, the preconditions include two conjunctive conditions “head count complete” and “dept. specific response complete dept=x”, the postcondition is “dept. responded dept=x”, and the executing location is “at assembly point”.

**Definition 6 (Workflow problem).** A workflow problem is defined as $W = (T, C, W_{in}, W_{out})$, where $T$ is the task set, $C$ is the set of all conditions, $W_{in}$ is initial condition set, and $W_{out}$ is the goal condition set.

Tasks can be joined together by exactly matching preconditions, postconditions, junction, and available locations. Conditions and tasks within a workflow problem thus may be considered nodes in a directed graph.
Definition 7 (Workflow graph). A workflow graph is defined as a directed graph \( G_W = (T, C, E, W_{in}, W_{out}) \), where \( T \) is a task node set, \( C \) is a condition node set, \( E \) is a directed edge set, \( W_{in} \subseteq C \) is a set of source nodes (nodes without any incoming edges) and \( W_{out} \subseteq C \) is a set of sink nodes (nodes without any outgoing edges). For each precondition \( c_{pre} \) of a task \( t \), there is an edge \( e = (c_{pre}, t) \in E \). For each postcondition \( c_{post} \) of a task \( t \), there is an edge \( e = (t, c_{post}) \in E \).

Note that we assume there are no duplicate nodes (nodes with the same label) in the graph. This definition allows us to compose two workflows by merging (a) identical sinks from one workflow with the corresponding sources from the other workflow and (b) identical sources in both workflows. Two workflows are composable if and only if matching sinks and sources yield a valid workflow. For instance, a workflow \( W_{1} \) with sources \( \{a, b, c\} \) and sinks \( \{d, e, f\} \) and a workflow \( W_{2} \) with sources \( \{c, d, e\} \) and sinks \( \{g, h\} \) can be composed into a new workflow \( W \) with sources \( \{a, b, c\} \) and sinks \( \{f, g, h\} \). Workflow fragments are merely small workflows (possibly even a single task) that are intended to be composed into larger workflows at a later time.

In a workflow graph \( G_W \), we call a condition reachable from a condition set \( C \) when it is in \( C \) or when it denotes the output of a task which is reachable from \( C \); a task is reachable from \( C \) when all necessary input conditions are reachable from \( C \).

A workflow graph is constructed in response to an expressed need. In general, this need is stated in terms of a specification \( S \): a tuple of condition sets \((t, \omega)\) where \( t \) is the triggering condition set and \( \omega \) is the condition set that represents the goal. Given a specification \( S(t, \omega) \), a workflow \( W \) is satisfiable if and only if there exists a feasible plan \( P_W \) satisfying: (1) each task in \( P_W \) is reachable from \( t \) and (2) \( \omega \subseteq \{c \in C_{post}(t), t \in P_W\} \).

Definition 8 (Workflow plan). Given a workflow problem \( W \), a workflow plan \( P_W \) is a sequence of task sets \((T_0, T_1, ..., T_n)\) which satisfy the following rules (let \( C_{i+1} \) be the condition set after executing all tasks in \( T_i \)):

1. For each task \( t \in T_0 \), \( C_{pre}(t) \subseteq W_{in} \).
2. For each task \( t \in T_i \), \( C_{post}(t) \subseteq C_i \) and all tasks in \( T_i \) can be performed in parallel.
3. \( W_{out} \subseteq C_{n+1} \).

Note that each \( T_i \) may consist of one or multiple tasks.

Definition 9 (Workflow plan length). Given a workflow plan \( P_W = (T_0, T_1, ..., T_n) \), the workflow plan length is defined as the total number of tasks in the plan:

\[
\sum_{i=0}^{n} |T_i|
\]

As the plan is formed, tasks must be allocated to hosts who will eventually execute corresponding services. A service is a concrete implementation of a task and may involve a computation by a certain device, an activity performed by a certain host, or some combination of the two. Execution of a task thus consists of the invocation of a service satisfying the respective task specification. Within a workflow problem, different tasks may be performed in sequence or in parallel by one actor or by multiple actors.

The availability of services and resources within the community determines to whom tasks are allocated. Service availability is determined by whether any host can commit to providing a service: that is, (1) whether the host is capable of performing the service, (2) whether the host has time available, (3) whether the host can travel to the necessary location to perform the service, (4) whether the host can gather the necessary inputs and distribute any outputs in a timely manner, and (5) whether the host is willing (according to their preferences) to perform the service.

Definition 10 (Task availability commitment). A task availability commitment is defined as \( a' = (h, t, t_1, l_1, l_2, l_3) \), where \( h \) is a host, \( t \) is a task, \( t_1 \) and \( l_1 \) are a starting time and a starting location, and \( t_2 \) and \( l_2 \) are an ending time and an ending location, respectively.

The task availability shows that host \( h \) is available to execute task \( t \) in the time window between \( t_1 \) and \( t_2 \). The starting location indicates where the host will be when it becomes available at the starting time. The ending location indicates where the host must be to meet its next commitment at the ending time. Once a host has made a commitment, it is responsible for ensuring that the service is executed as agreed. A host is thus free to move about and requires no further communication with the community except possibly for previously agreed upon meetings to gather inputs or distribute outputs. As individual hosts execute their assigned services from the dynamically constructed workflow, the community as a whole thus performs the activities necessary to satisfy the specification and achieve the original goal.

Note that the workflow patterns supported by our workflow management system include sequence, parallel split, synchronization, simple merge, multi-choice, structured synchronizing merge, multi-merge, local synchronizing merge, arbitrary cycles which supports simple cycles, and persistent trigger (Aalst et al., 2003). We currently only support static loops in our workflows. For example, if we have three tasks \( t_1, t_2, t_3 \), \( C_{pre}(t_1, t_2, t_3) = (\{a\}, \{b\}, \{c\}) \), and \( C_{post}(t_1, t_2, t_3) = (\{b\}, \{c\}, \{d\}) \), then the constructed workflow graph has a loop \((t_1, t_2, t_3)\). Given an initial condition \( W_{in} = \{a\} \) and a goal condition \( W_{out} = \{c\} \), our solver can find a feasible solution \((t_1, t_2)\). Such loops are static loops that are found as solutions to the planning problem. We currently cannot support dynamic loops that need to check a condition during execution time (i.e., tasks that have to be repeated several times until a condition is met). To support that, we will need to use enriched planning models, such as conformant planning (Hoffmann and Brafman, 2006), in which there are contingent actions that output certain “states” after its execution. We will investigate this in our future work.

2.3. Motivating example

To highlight the possibilities and advantages of the open workflow paradigm, consider the demands placed on a university-wide emergency response system. The university must be prepared to cope with a wide variety of disasters, such as a fire in a building, a tornado, an earthquake, or violence on campus. While the same emergency response system will be used for each of these events, the proper reaction varies significantly: evacuating the building during a fire versus heading to the basement during a tornado. The knowledge (workflow fragments) of how to react to each emergency is decentralized and location specific, as each building will have a different evacuation route and an individual will only be interested in the evacuation route for the building they are in. The university can easily centralize some workflow fragments that are known a priori to be applicable to disaster response scenarios in general, but it is not practical to try to centralize all of the specific workflow fragments that everyone knows because there are any number of mundane ones, some of which may be important and others which may not, again depending on the specific situation that actually arises.

Furthermore, different university departments may have department specific knowledge that will influence the plan. The aeronautics department might have a wind tunnel that must be
shut down during a building evacuation for the safety of emergency personnel. The department emergency coordinator will be aware of all the special facilities of the department, such as the wind tunnel, but only the professor and a few students will be able to properly shut down the wind tunnel, demonstrating that knowledge and ability can be independent. Devices can participate in the community as well. The wind tunnel may have a small monitoring device (similar to a warning light) that can participate in the workflow to contribute the knowledge that the wind tunnel is active and needs to be shut down, even though the monitoring device itself does not have the ability to power down the wind tunnel.

Suppose a fire breaks out in a building on this campus. When the fire alarm sounds, the emergency response chief wants to make sure that proper procedures are followed and everyone gets out safely. She requests a disaster response workflow from the open workflow system on her mobile device, specifying that there has been a fire. The open workflow engine begins by collecting workflow fragments contained on the mobile devices owned by the members of the emergency preparedness team. As shown in Fig. 2(a), the chief's mobile device knows that her task is to coordinate with the other departments in the university. Another officer has been developing a general response plan consisting of a set of tasks that can be used with any department. The coordinators for each department know the tasks that are specific for their department, such as shutting down a dangerously strong MRI magnet or high velocity wind tunnel fan.

In Fig. 2, each task (action) is shown as a rectangle and each condition is an oval. The oval pointing into a rectangle is a precondition and pointing out from a rectangle is a postcondition. The dashed lines are the workflow fragment boundaries (including who the source of the fragment is).

Using the knowledge gathered from throughout the community, the open workflow engine searches for a set of tasks that can be connected into a workflow that meets the conditions and requirements given by the emergency response chief. There may be many possible workflows, and some tasks may or may not be used. The engine may need to specialize general tasks, such as the common department plan, to adapt them to the context in which they will be used. An example of such a workflow that meets the requirements for responding to a fire is shown in Fig. 2(b).

The open workflow engine then searches for hosts that are able to perform the activities indicated by the workflow. The aeronautics professor’s mobile device will notify him that he needs to shut down the wind tunnel, while an emergency response officer (or other suitably responsible person) will be directed to each assembly point to do a head count. The allocation of tasks by the open workflow engine is sensitive to abilities (the officer would not be sent to shut down the wind tunnel) and to spatiotemporal constraints (the officer would not be told to do two head counts at the same time, and would not be directed to run to the other side of campus if another officer was nearer). If the chief had given different conditions and requirements (e.g., due to a tornado), the resulting workflow could be very different (e.g., people would be directed to go to the basement rather than exit the building). The constructed workflow is also sensitive to the state of the environment: if the wind tunnel was already off, all the requirements for the aeronautics department’s checklist would already be met and there would be no need to add a task to turn it off.

A workflow plan to handle a campus emergency must be adaptive to the dynamic campus environment. For example, if a week-long spring carnival is being put on by the student body in a large campus quadrangle, the emergency response plan will have to adjust during this time. The emergency assembly point that usually occupies the quad will have to move, and the carnival itself will be another group with which to coordinate. Notice that in our example, only a small change is needed to incorporate the carnival into the university wide emergency response system: the emergency response chief would only need to update her “Check All Departments” task in her mobile device to include the carnival as another “department”. Even people not normally considered a part of the campus community will impact the emergency response plan. Visitors will need clear directions to an emergency assembly point and may need assistance rejoining their hosts. Construction on campus will involve new groups to coordinate and new needs and activities during an emergency.

Note that it is desirable to create a workflow plan on the fly since even though the workflow fragments may be easily known in advance, the diversity of different disaster situations that could arise means that it is only when something specific happens that an appropriate workflow can be constructed. In general if a predefined plan is likely to be applicable in many scenarios, then it can be stored, either as a collection of modular workflow fragments which would be applicable to a wider range of situations, or as one or more larger fragments which would apply to fewer situations but which could potentially produce a plan more quickly.

A traditional static workflow would be difficult to maintain in the face of such a continuously changing community. To be sensitive to the variety of emergency situations and the individual capabilities and dynamic availability of the members of the campus community, the workflow would need to contain a large number of conditional branches which must be carefully crafted and assiduously maintained. Such a static workflow cannot respond rapidly to new resources or changes in the environment. In contrast, an open workflow system can construct a workflow customized to the state of the campus community at the time the emergency occurs, and each person’s device can guide them and give them specific directions customized to the current emergency. Sensitivity to context, in the form of knowledge, capabilities, and availability, is the driving force behind the creation of our open workflow system.

3. Co-design open workflow framework

Algorithm 1. Co-design open workflow framework.

```
Input S(t, ω), K
while True do
    GW ← Workflow_Construction (S(t, ω), K);
    if GW = No solution then
        return No solution;
    end
    P ← Transform_to_Temporal_Planning_Problem (GW);
    K ← SGPlan(P);
    if K = False then
        return No solution;
    end
    PW, Schedule ← Generate_Workflow_Plan_and_Schedule (P);
    if Task_Allocation(PW, Schedule) = True then
        if Task_Execution(PW, Schedule) = True then
            return Problem solved;
        end
    end
else
    return S(t, ω), K;
end
```

Q. Lu et al. / Engineering Applications of Artificial Intelligence 35 (2014) 215–236
We now formulate the workflow plan, workflow allocation, and host scheduling together as a co-design problem. Our overall algorithm is referred to as a co-design open workflow (COW) framework, shown in Algorithm 1. We start from a workflow specification $S(t, o)$, where $t$ is the triggering (initial) condition set and $o$ is the goal condition set, and a set of workflow fragments $K$, construct a workflow graph (Line 2) and encode the COW problem as a temporal planning problem (Line 6), either prove it unsatisfiable (Line 8–9) or find a solution which can generate a workflow plan and host schedules (Line 11). According to the workflow plan and host schedules, we try to allocate tasks to each host (Line 12) and execute workflow tasks and some other actions like move the host and communicate between hosts (Line 13). If the task allocation or task execution fails, we update the current initial host and communicate between hosts (Line 13). If the task allocation or task execution fails, we update the current initial condition set $t$ and goal condition set $o$, recollect the workflow fragments $K$ if necessary (Line 16–22), and restart from constructing the workflow graph (Line 2).

Note that our framework can handle dynamics such as community dynamics and task allocation and execution errors. For example, the departure of a host or another change in availability during the end of the allocation phase may cause the allocation to fail. When such a change is detected, an alternative workflow that does not require the lost resources will be reconstructed (Line 21). A failure during execution will result in a revised or repaired workflow (Line 17), which requires reconstruction, reallocation, and compensating execution. However, the handling of dynamics such as non-determinism of workflow fragments, data flow between workflow fragments by the open workflow paradigm is still an open problem for future research. Extending the current implementation with more feedback mechanisms between the construction, allocation, and execution phases seems like a promising approach. Developing an appropriate commitment and execution state model that allows the hosts to accomplish these activities in a mobile ad hoc setting is a focus for future work.

In this section, we first present the dynamic workflow construction algorithm, then we introduce the encoding method that transforms the COW problem to a temporal planning problem. Further, we discuss the task allocation and execution procedures according to the workflow plan and host schedules generated from the solution plan found by solving the planning problem.

### 3.1. Dynamic workflow construction

A community consists of a set of hosts (people and their mobile devices) willing to work cooperatively to solve problems. A community is dynamic in that its membership and thus the knowledge, capabilities, availability, and other resources provided by its members are not fixed a priori. The resources available to solve one problem may be significantly different from the resources available to solve the next problem, and the resources may be different even between two instances of the same problem, leading to significantly different solutions.

While there are many ways to maintain a community and share knowledge within that community, we chose an approach that places few restrictions on the members. We constrain our definition of a dynamic local community to one whose hosts are within communication range of the host who generates the specification, and announce their willingness to participate. This assumption guarantees that the host can collect all hosts’ service availabilities when performing workflow allocation. Note that the community is dynamic as members can join and leave at will.

Once a problem has been identified and a specification given, the knowhow (in the form of workflow fragments) and capabilities (in the form of services) of the local community are synthesized to form a plan by constructing a workflow problem. The construction problem is defined as follows. Given a workflow specification $S$ and a set of workflow fragments $K$, find a set of workflow fragments $W$ which may be composed (subject to pruning) into a workflow $W$ that satisfies $S$ — we say that $W$ is feasible given $S$ and $K$. It is important to note that the defining features of the open workflow paradigm rest with the fact that the specification $S$ can be generated dynamically in response to a new need, context change, or other event, and that the set $K$ represents the combined knowledge of the community as a whole. $K$ is distributed and dynamic. As hosts move around in space, the knowledge available to the community changes with its membership and their

---

**Algorithm 2. Workflow construction**

```
Input: $t$, $o$, and $K$

/* Construct Supergraph */
$G \leftarrow \emptyset$
forall the fragments $F \in K$
do
forall the nodes $n \in F$
do
| if $n \notin G$ then $G \leftarrow G \cup \{n\}$
end
forall the edges $e \in F$
do
| if $e \notin G$ then $G \leftarrow G \cup \{e\}$
end
end
/* Exploration Phase: track the set of greenNodes (initially empty) */
greenNodes $\leftarrow \emptyset$; $U \leftarrow \emptyset$
forall the $n \in t$
do
n.color $\leftarrow$ green;
greenNodes $\leftarrow$ greenNodes $\cup \{n\}$
forall the $m \in n$.children do
| if $m$.color = uncolored then $U \leftarrow U \cup \{m\}$
end
end
while $U \neq \emptyset$ do
```
get \( n \) from \( U, U ← U \backslash \{ n \} \);
if \( n \) is conjunctive \( \land \) all of \( n \)’s parents are green then
| \( n \).color ← green;
end
else if \( n \) is disjunctive \( \land \) any of \( n \)’s parents are green then
| \( n \).color ← green;
end
if \( n \).color = green then
| greenNodes ← greenNodes \cup \{ n \};
forall the \( m \) ∈ \( n \)’s children do
| if \( m \).color = uncolored then \( U ← U \cup \{ m \} \); end
end
if \( \neg (\omega \subseteq \text{greenNodes}) \) then return No solution;

// Pruning Phase: track the set of redNodes (initially empty) */
redNodes ← \( \emptyset \), \( U ← \emptyset \);
forall the \( n \) ∈ \( \omega \) do
| \( n \).color ← red;
| redNodes ← redNodes \cup \{ n \};
| \( U ← U \cup \{ n \} \);
end
while \( U \neq \emptyset \) do
| get \( n \) from \( U, U ← U \backslash \{ n \} \);
forall the \( p \) ∈ \( n \)’s parent nodes do
| if \( p \).color = green then \( p \).color ← red; redNodes ← redNodes \cup \{ p \}; \( U ← U \cup \{ p \} \);
end
end
if \( \neg (\iota \subseteq \text{redNodes}) \) then return No solution;

// The set of nodes and edges colored green and red is the constructed workflow graph */

experiences. For the same specification, different communities may respond differently or may be unable to construct an appropriate workflow.

We now introduce the construction algorithm for open workflow problems. A host is ready to begin workflow construction once it: (1) has identified a need and generated a specification \( S(\iota, \omega) \), where \( \iota \) is the triggering condition set and \( \omega \) is the condition set that represents the goal; (2) is in contact with the other members of a community; and (3) can collect from each of those members a set of workflow fragments. For the purposes of illustration, we start with the simplifying assumption that the host initially collects all the fragments in the community to create the set \( K \). Using the gathered information, the host called as the \textit{initiator} runs our algorithm to find a feasible workflow — a workflow composed of fragments from \( K \) (subject to pruning) that satisfies \( S \) — if one exists.

Our algorithm is based on graph traversal and graph coloring. Our strategy is to combine all workflow fragments from \( K \) into one large graph, henceforth called the workflow supergraph \( GW \). The supergraph represents a unified view of all possible actions represented in the set \( K \), however it is not necessarily for a valid workflow since it may have cycles, outputs produced by multiple tasks, unavailable inputs, or undesired outputs. We use a node coloring process on the supergraph \( GW \) to identify one feasible workflow within this graph. We start by coloring the nodes corresponding to set \( \iota \) of the specification \( S \). Following the data flows, we explore the graph, growing the colored section as we identify which tasks and conditions are \textit{reachable} from \( \iota \).

Once we have reached all the elements of \( \omega \), we prune the reachable set down to a valid workflow. Working backwards with a new color, we identify only those paths which are actually required to reach \( \omega \). The pruning phase removes undesirable outputs and unnecessary tasks and conditions. Once the second color has swept all the way back to \( \iota \), we have fully identified \( W \), a valid workflow that satisfies specification \( S \) and that is composed only of fragments in \( K \) that have been pruned of unneeded outputs and paths.

With this general strategy in mind, we present the full pseudocode in Algorithm 2. For purposes of the algorithm, we annotate every node and edge in \( GW \) with a \textit{color} (initially \textit{uncolored}) on the graph. Nodes are marked green for reachability during the exploration phase and \textit{red} for workflow membership during the pruning phase. The algorithm selects nodes nondeterministically; any node may be processed next so long as it matches the guard condition.

In Algorithm 2, tasks reachable from \( \iota \) are colored green in the exploration phase (Line 10–32). In the pruning phase, we identify and color all the tasks that can eventually reach \( \omega \) (Line 34–45). Only nodes that are flagged both green and red can be in the solution, which reduces the number of nodes we need to consider during workflow construction. Note that there may exist many feasible plans in the constructed workflow graph. If the workflow of the supergraph does not satisfy \( S \), then there is definitely no satisfying workflow to be constructed from the available tasks.

We observe that the coloring process requires only local knowledge. Thus, we can relax the assumption that all of the workflow fragments are collected from the community before the coloring process begins. In our implementation, we build the supergraph incrementally, drawing from the community only the fragments that we need to extend the supergraph along the boundaries of the colored region.
Note that our construction algorithm adopts a more relaxed pruning compared with the pruning algorithm in our previous open workflow scheme (Thomas et al., 2009). Here we only prune task nodes that will not be selected in any workflow plans while our previous algorithm (Thomas et al., 2009) uses a strict pruning which directly constructs a workflow plan and deletes all other task nodes.

3.2. Temporal planning for solving COW problem

The goal of workflow allocation is to determine if a feasible plan can be obtained from the workflow super-graph and all tasks in the plan can be successfully allocated. A successful allocation is a mapping of tasks to hosts such that each host can execute all its assigned tasks. In this section, we will solve the allocation problem in a co-design framework.

3.2.1. Constraints of workflow allocation and host schedules in MANET

In order for a host to execute a task \( t \), the host must be in the location specified by \( t \) executing the predecessor services of \( t \) in the plan, and have enough free time (larger than the duration of \( t \)) to execute the task. Note that the requirement that a host receives results for all the dependencies before executing its task guarantees that the final task plan can be correctly completed.

In a MANET environment, workflow allocation is complicated by the need for host scheduling. Task allocation is sensitive to the spatiotemporal constraints of the hosts. Each service has a specified location and duration determined from the service and task metadata during construction, and we assume that each host can tell us how long it takes for that host to travel between two locations. To determine whether one or more service allocations will fit into an availability window, there are three kinds of constraints that the host must satisfy.

1. **Availability constraint**: The host has enough time to execute the service for the specified duration.
2. **Mobility constraint**: The host has enough time to travel from its starting location to the first service location or from a service location to the subsequent service location and travel from the last service location to their ending location (to meet their next commitment).
3. **Communication constraint**: After a service is executed, the postconditions will need to be communicated to the host (s) executing the subsequent service(s) in the workflow. This transfer of conditions is normally done over a network. But, since the geographic area (e.g., for a construction site) may be reasonably large, there may be no network infrastructure and we assume that two people can only communicate when their mobile devices are within the communication range of one another. This means that even if a host can execute a task, we must consider whether or not the host will be able to pass the result onto the host we have chosen to perform the subsequent task.

All of these constraints can make the process of choosing a host for a given task a non-trivial problem. In fact, deciding whether or not there is an allocation of tasks to hosts such that the workflow could possibly complete in this manner is NP-hard (Haitjema et al., 2010).

3.2.2. Temporal planning problem encoding

To overcome the limitations of the existing allocation algorithms (Sen et al., 2008; Haitjema et al., 2010) discussed in Section 1, we formulate the host/workflow co-design problem (workflow plan search, workflow plan allocation, and host scheduling) into a temporal planning problem in PDDL and use state-of-the-art temporal planners to solve it. Using the Planning Domain Definition Language (PDDL) 2.1 (Fox and Long, 2003) and 2.2 (Edelkamp and Hoffmann, 2004), we define the planning model for the co-design problem as follows.

Given a workflow graph \( GW = (T, C, F, W\ in, W\ out) \) (created by Algorithm 2) and a task availability set \( A^t = (a^t(a^t = (h, t, l_s, l_e, l_j),) \), we can construct a temporal planning problem \( P = (A, F, A_I, G) \), where \( F \) is the set of facts, \( A \) is the set of durative actions, \( I \subseteq F \) is the set of initial state facts, and \( G \subseteq F \) is the set of goal state facts.

To define fact set \( F \), we first define four objects: host, task, location, and condition-token. Each host can execute certain tasks at certain locations. Each condition-token is associated with a condition which is a precondition or postcondition of a task. A host must have all the required tokens for a task (any precondition token for a disjunctive task or all precondition tokens for a conjunctive task) before executing it.

From \( GW \) and \( A^t \), we first collect the information of all participating hosts and all possible locations for each host. Then we build the fact set \( F \) where each fact \( f \in F \) is specified by one of the following four kinds of predicates.

1. **1. **at \( ?h \) – host \( ?l \) – location: a host \( h \) is at location \( l \).
2. **2. **free \( ?h \) – host: a host \( h \) is free, which means it can execute a task, move to other locations, or communicate with another host.
3. **3. **done \( ?c \) – condition: a condition \( c \) is established.
4. **4. **has-token \( ?h \) – host \( ?c \) – condition-token: a host \( h \) has a condition token \( c \), which means the host \( h \) knows that \( c \) has been established.

We consider three kinds of durative actions: a host may execute a task at a given location by a given host, a host may move between two locations, and a host may communicate with another host. These three kinds of durative actions are defined as follows.

1. **1. **execute \( ?h \) – host \( ?l \) – location \( ?t \) – task. Since we use the STRIPS formalization of PDDL which does not support disjunctive preconditions, we first split each disjunctive task into a sequence of conjunctive tasks, where each of them only has one precondition. For instance, we split a disjunctive task \( t = (c_1, c_2), C_{post, disjunctive, L, d} \) into two conjunctive tasks \( t_1 = (c_1), C_{post, conjunctive, L, d} \) and \( t_2 = (c_2), C_{post, conjunctive, L, d} \). Then for each conjunctive task \( t \), we find out all hosts who can execute \( t \), and all possible locations for a certain host and task. We define an “execute” action for each conjunctive task as: (1) **duration**: a positive rational number indicating the executing time of task \( t \). (2) **Preconditions**: (at start (at \( h l \)), (at start (free \( h ))) \), (over all \( (at \ h ) l \)), and (at start (has-token \( c \) ) for all precondition tokens \( c \) of task \( t \). (3) **Effects**: (at start (not (free \( h ))) \), (at end (free \( h ))) \), (at end (done \( c ))) \), and (at end (has-token \( c \) )) for all postcondition tokens of task \( t \). The last two effects represent that postcondition tokens of \( t \) have been established and host \( h \) has these tokens.
2. **2. **move \( ?h \) – host \( ?l_1 \) ?l_2 – location: (1) **duration**: a positive rational number indicating the time for host \( h \) to move from \( l_1 \) to \( l_2 \). It equals the distance between \( l_1 \) and \( l_2 \) divided by the speed of the host. Note that all the move actions together encode the map information of the environment. (2) **Preconditions**: (at start (at \( h l_1 \)). (3) **Effects**: (at start (not (at \( h l_1 ))) and (at end (at \( h l_2 )))\).
3. **3. **communicate \( ?h_1 \) h_2 – host \( ?h_1 \) ?h_2 – location \( ?c \) – condition-token: (1) **duration**: a positive rational number indicating the time for transferring a message (token \( c \)). (2) **Preconditions**: (at start (at
tasks are different in different hosts. To solve this problem, we need two actions at the same time. Note that since dependent actions cannot move to other locations when executing a task or communicating with another host. The precondition (at (start (free h)) and effect (at (start (not (free h)))) guarantee that the host h cannot perform two actions at the same time. Note that since dependent tasks are different in different execution orders, we cannot define all execute actions in an unordered way. Therefore, all actions are grounded in the domain definition of PDDL.

The initial state of the problem includes two kinds of predicates. We use timed initial literals like (at n (free h)) which is a feature of PDDL2.2 (Edelkamp and Hoffmann, 2004) to represent facts that become true or false at certain time points.

1. Timed initial literals for each availability window (ts, l, ts, ls) of a host h: (1) If ts is not null which means it has a starting time, we add (at ts (free h)) which indicates that host h will be free at time ts (a positive rational number). We also add (at ts (h l)) if ls is not null which means host h will be at location l at time ts; (2) If ts is null, we add (free h) and (at h l) which indicates that host h is free at the initial time and the initial location of h is l; (3) If ts is null which means it gives an ending time, we add (at (not (free h))) which indicates that host h will be not free at time ts (a positive rational number). We also add (at ts (h l)) if ls is not null which means host h will be at location l at time ts.

2. Initial condition tokens: (done c) and (has-token h c) which indicate that condition c is established in the initial state and h has the token c.

During the execution of a plan, when a set of new task requirements comes in, a host may be executing some tasks currently and will be free at future time n. If the host is required for a task, we need to add the host as an object in the planning problem. Thus we use the timed initial literals to represent the initial status of such hosts. By exploiting the timed initial literals in PDDL2.2 planning, our approach can support dynamic planning in response to new tasks during execution.

The goal state includes predicates (done c) for all required condition tokens c. Again, our approach can support dynamic planning. When new problems (initial and goal conditions) are added during execution, we generate a new planning problem with the dynamic initial and goal state discussed above, and then call a planner to solve it.

Based on the above PDDL model, any solution plan P = (P0, P1, ..., Pn−1) found by a temporal planner will give a coarse-grained solution. The solution can generate a workflow plan in the constructed workflow graph and specify the workflow allocation as well as a schedule of movements and communications for each host which satisfy the availability, mobility, and communication constraints. Specifically, since each “execute” action (execute 7h – host l – location 2t – task) specifies the host h to execute the task t in the location l, we can generate a workflow plan P0 which has the sequence of workflow task sets in the constructed workflow graph and a workflow allocation which satisfies the availability constraint. The host h has enough time to execute the service t for the specified duration. Since each “move” action (move 7h – host l1 – task) indicates the starting location l1 and the ending location l2 of the travel of the host h, and each “communicate” action (communicate 7h1, h2 – host l1, l2 – location 2c – condition-token) indicates the sending host h1 and the location l1, the receiving host h2 and the location l2, and the transferring condition token c, the generated schedule of movements and communications for each host can guarantee that: (1) the host has enough time to finish each travel, such as from their starting location to the first service location, from a service location to the subsequent service location, and from the last service location to their ending location (to meet their next commitment), and (2) after a service has been executed, the postconditions will send to the host(s) executing the subsequent service(s) in the workflow. This ensures that the workflow allocation satisfies the mobility and communication constraints.

We use an example to help explain how our planning method guarantees the workflow allocation and host schedules satisfy the availability, mobility, and communication constraints.

Example 1. Consider a workflow problem W = (T, C, W, in, W, out), where T = {t1, t2, t3}, C = {c1, c2, c3}, W, in = {c1}, W, out = {c2}, f0 = (c1, c1, l1, l1, 5.00), and t1 = (c1, c2, l1, l1, 10.00). There are three locations (h0, h1, l0, l1, l2) where the distances between each two locations are dis (h0, h1) = 20 m, dis (h1, l0) = 20 m, and dis (h0, l2) = 35 m. Suppose the communication range is 25 m which means two hosts cannot communicate with each other between l0 and l2. Given two task availability commitments a0 = (h0, t0, 0, h0, 100, l1) and a2 = (h1, t1, 0, l2, 100, h2), the temporal planning will find a solution plan shown as follows.

0.000: (execute-h0-10-t0) [5.000]
5.000: (move-h0-10-11) [12.000]
17.002: (communicate-h0-h1-11-12-c1) [1.000]
18.003: (execute-h1-12-t1) [10.000]

Once the solution has been found, our method will generate three commitments (execute, h0, l0, 0.000, 5.000), (move, l0, t1, 5.001, 12.000), (send, h1, l2, c1, 17.002, 1.000) for the host h0, and two commitments (receive, h0, c1, l1, 17.002, 1.000) and (execute, t1, l2, 18.003, 10.000) for the host h1. It means that we need host h0 to execute task t0 at location l0 at time point 0.000, move from l0 to l1 at time point 5.001, and send the token c1 to h1 at location l1 at time point 17.002, and host h1 to receive the token c1 from h0 at location l2 at time point 17.002 and execute task t1 at location l2 at time point 18.003. These schedules tell host h0 where to move after executing task t0 to make sure it can send the token c1 to host h1 notifying task t0 has been finished and the precondition of task t1 has been satisfied. After receiving the token c1, host h1 can execute task t1 to reach the goal (c2).

Note that the resulting temporal problem is temporally simple without required concurrency (Cushing et al., 2007; Lu et al., 2013). A temporal problem has required concurrency (called temporally expressive) when, in any solution, there exist two actions a1 and a2 that need to be executed concurrently. Our PDDL2.2 domain is a temporally simple domain. Note that temporally simple problems are typically more tractable than temporally expressive ones. Hence, our formulation can be efficiently solved using existing temporally simple planners. Although the problem is temporally simple, it is essential to exploit the durative and concurrent nature of actions in our model in order to generate efficient plans with short makespan. In real-world applications, this temporal feature is very important because users usually want to finish all tasks as soon as possible.
3.3. Task allocation and execution

Once the planner finds a solution which contains a sequence of actions and the times at which they occur, for each action in the solution, we make a commitment for a certain host and a certain type (executing task, moving host, or communicating condition token). Then, we subscribe these commitments to certain hosts, and wait for the feedback signals indicating whether the hosts accept the commitments. The allocation process is successfully finished if all hosts accept all commitments produced by the solution.

Once the planning process is completed, the execution of the workflow begins. When a host is assigned an action, it adds a commitment to its schedule that contains all the necessary information including action type, executing time, location, duration, and others, to execute the appropriate service as directed by the manager. The agent is free to roam, but is responsible for completing assigned tasks, including moving to certain locations, executing tasks, and exchanging condition token information. The execution phase of this workflow is thus conducted in a decentralized and distributed manner. To meet a commitment, the host must (1) acquire the required precondition tokens for the task from the executor of the preceding tasks, (2) be at the required location for executing the task, and (3) execute the task at the required time. The host monitors these conditions and, based upon their knowledge of their location and the travel times involved, travels and communicates as necessary to meet the conditions and successfully execute the service. Once the service has been executed, the host’s final responsibility is to communicate the service’s outputs to any other hosts that require them.

3.4. Replanning support

The flexibility of such a dynamic domain is much higher than normal workflow problems due to the high variability of agent speeds and task completion probability. The change of speed may cause an agent to be unable to arrive at a required location in time for executing tasks or communicating with other hosts. A failed execution of a task may break the dependencies of other tasks and cause the whole plan to crash. On the other hand, a host may reject a commitment in the allocation process due to accidental events, such as a change in the availability of a task. Thus, the ability to handle these exceptions is very important for the workflow management system. In these cases, we need to unsubscribe unfinished commitments, reconstruct the workflow graph (Line 2 in Algorithm 1), and rebuild the planning problem by recollecting the current task availability information, and repeat the planning, task allocation and execution. To make sure handling these problems correctly, our COW’s replanning process works as follows.

1. Decide the new initiator host: For the case of task allocation exception, the current initiator host is the new initiator. For case of task execution exception, the host who failed to execute a task will send a message to notify the hosts who require the postconditions of the failed task. If the host cannot notify other hosts due to some extreme conditions such as loss of communication, these hosts waiting for the postconditions will automatically mark that the task has been failed to execute if they do not receive the postcondition token at the appointed time. In any case, these “waiting” hosts will become the candidates of the new initiator host. For each candidate host, it generates a new workflow problem ID by using a hash function of the triple (old_problem_ID, host_ID, wall_time) which conforms with the follow rules: (1) if wall_time < wall_time2, then problem_ID1 < problem_ID2; (2) if wall_time = wall_time2 and host_ID1 < host_ID2, then problem_ID1 < problem_ID2, wall_time is the real world time. Each candidate host will send the generated problem ID to other hosts and compare the received IDs with its own. The host generating the smallest problem ID will be the new initiator. Note that for the problem generated by identifying a need to a host (see Section 3.1), we use the triple (S(i, ω), host_ID, wall_time) to generate the problem ID.

2. Notify other hosts to unsubscribe unfinished commitments: After deciding the new initiator, the new initiator will send a message to other hosts who have unfinished commitments to unsubscribe these commitments. Note that we use a lazy strategy here which means the new initiator does not wait for the confirmation of other hosts. If a host is not reachable, it will not be subscribed any new commitment in the replanning process since it will not be included in the new recollected task availability information. However, continuing its commitments may trigger another replanning for the same old problem (ID) since the host may wait for some unsubscribed task’s post-conditions. We will use a replan confirm message queue technique to avoid duplicate replanning and conflict commitment subscription for the same workflow problem (ID). We will discuss it in the following paragraph.

3. Generate a new workflow problem and solve it. In the case of allocation exception, we reconstruct the workflow graph, and rebuild the planning problem by recollecting the current task availability information of reachable hosts (Line 21 in Algorithm 1), and repeat the solving and allocation procedures until all hosts accept the commitments. In our system model, we can handle the execution exception since our workflow construction algorithm (Section 3.1) supports dynamic workflow graph constructing, and the temporal planning algorithm (Section 3.2.2) supports online planning by exploiting the timed initial literals. We first collect all established conditions and task availability windows of reachable hosts to generate a new triggering condition set r, and the unfinished goal conditions to generate a new goal condition set α' (Line 17 in Algorithm 1). Then we generate a new workflow graph GW with the specification S(r, α'). Given the new workflow graph GW, we generate a new planning problem with the dynamic timed initial state and goal state, and call a planner to solve it. If the planner cannot find a solution, the new initiator will notify the WMS that the problem cannot be solved and terminate the replanning process. Otherwise, to avoid duplicate replanning for the same old problem (ID), the new initiator will send a replan notice message defined as (old_problem_ID, problem_ID, notice) to all other reachable hosts in the community notifying that the old problem (ID) is going to be replanned. The host receiving the message will check its replan confirm message queue and send back a no-conflict message if there does not exist a message having the same old_problem_ID. Otherwise, it will send back a conflict message to tell the new initiator that the old problem (ID) has been replanned before. If the new initiator receives a conflict message, it will stop the replanning process. Otherwise, it will send a replan confirm message defined as (old_problem_ID, problem_ID, new_initiator_ID) to all other hosts notifying that the old problem (ID) has been replanned by the new initiator and then continue the task allocation and execution processes. The host receiving the replan confirm message will store it in its replan confirm message queue.

4. System architecture

4.1. An open workflow management system

We have designed and implemented a complete open workflow management system in Java. Our approach offers an intuitive
calendar-like interface, behind which integrated goal specification, communication, and service invocation features combine to enable construction and execution of sophisticated open workflows. Source code and executables for the application are available as open source software at our web site (Mobilab Group, 2009).

The basic steps in deploying an application using our open workflow management system are (1) installing the program on the users’ devices, (2) adding knowhow in the form of workflow fragments, and (3) adding service descriptions. In our implementation, we use XML configuration files to provide the task and service definitions for each device. Once this initial configuration has been completed, any host can use their device to create a problem specification. In response, the system will automatically construct, allocate, and (by prompting the users) execute an appropriate workflow.

Fig. 4 shows two screenshots from community members participating in an open workflow. The tabs on the left are for reviewing static knowledge. On the top are tabs for dynamic activities and alerts. Fig. 4(a) shows the form that allows the user to create a problem specification by entering information about the triggering conditions and goal. In Fig. 4(b), the Schedule tab allows the user to view their schedule of commitments. The necessary travel time is also blocked out in the schedule, and the system has added an alert tab to notify the user that they must soon begin traveling to meet their scheduled commitment. The remaining tabs allow the user to configure the list of workflow fragments (knowhow), the list of local services (capabilities), and other system settings.

The system invokes services by loading the Java class named in the XML configuration file, passing it a map of the inputs received from the preceding workflow tasks and configuration parameters from the XML configuration file, and receiving a map of outputs to send to the subsequent workflow tasks. Thus the system can feasibly invoke any computational service that can be called from Java. The system also directly supports services that require user action. We provide a sample service implementation that interacts with the user by presenting a simple form, where the form is defined in the XML configuration file. Human-oriented “services” can be implemented by presenting the user with a form for data entry or even just brief instructions and a button to click when the activity is complete.

4.2. Goals, design principles, and architecture

Our goal is a system that will support the coordination and participation of devices with diverse capabilities. Further, we want to build a system robust enough and flexible enough to encourage rather than hinder innovations from future research. Consideration of these goals led us to the following two design principles. First, the architecture should break apart the major responsibilities of the system into independent components, allowing each host to provide only the components that are appropriate to the host’s physical capabilities. Second, the architecture should isolate and hide the highly variable details of the transports, protocols, and caching schemes used during communication by providing an abstract communications layer. Passing messages through an intermediary also ensures that local and remote components are accessed uniformly.

Based upon these design principles, we identified the following major responsibilities for our open workflow management system, as illustrated in Fig. 5. We first observe that for a particular open workflow problem, one host acts as the initiator while all hosts (including the initiator) may execute tasks. We therefore split the system responsibilities into two corresponding subsystems: the construction subsystem and the execution subsystem. The construction subsystem is responsible for identifying the problem to be solved, issuing queries to discover knowhow, capabilities, and availability, formulating the plan of action, and assigning work. The execution subsystem is responsible for replying to informational queries, accepting appropriate work assignments, and actually doing the processing or communicating necessary to complete the work. In Fig. 5, each horizontal bar represents a
component of the system. The arrows show the direction of message transmission between different components.

**Construction subsystem**: The Workflow Initiator is responsible for interacting with the user to define the initial conditions and goal conditions for a new problem. The Workflow Manager is the core component of the construction subsystem. The Workflow Manager creates and maintains a separate workspace for each open workflow, allowing it to work simultaneously on multiple isolated and independent problems. The Workflow Manager issues queries to discover knowhow (“Fragment Messages”) and integrates the responses into the supergraph. It queries for availability and travel time information (“Travel Route Messages”) and constructs an open workflow. Finally, it issues allocation requests to hosts and listens to confirm that each allocation is properly committed (“Workflow Status Messages”).

**Execution subsystem**: The Fragment Manager is responsible for maintaining a host’s database of workflow fragments and responding to knowhow queries during workflow construction. The Travel Route Manager is responsible for maintaining a host’s database of known locations and travel times. The Current Location Manager encapsulates detecting and tracking the host’s location. The Schedule Manager tracks the host’s schedule and scheduling preferences. It maintains a database of all commitments, primarily consisting of scheduled service invocations and their associated location and travel time details, which is the key data structure for both allocation and execution of an open workflow. The Execution Manager monitors the input, spatial, and temporal conditions required for each scheduled service invocation during the execution phase. Once an invocation’s necessary conditions are met, it triggers service execution, and publishes any output messages. It also responds to queries about the service invocations it is monitoring for execution. The Service Manager provides a uniform service invocation interface to the Execution Manager by handling parameter marshaling and any other mechanics required to actually invoke a local service during the execution phase. Finally, the Task Allocation Manager responds to availability queries by summarizing and translating the information from the Schedule Manager and the Service Manager. It also translates task allocation requests into commitments that can be added to the Schedule manager.

Our architecture permits multiple open workflows to be constructed and executed concurrently within the same community and even within the same host. The Workflow Manager maintains a separate workspace containing construction state information for each workflow. The remaining components (such as the Task Availability Manager, Fragment Manager and Schedule Manager) act at task granularity and thus handle two task-based requests from two separate workflows no differently than they handle two task-based requests from the same workflow. While multiple workflows will necessarily compete for utilization of the same resources (in the form of hosts, their capabilities, and other resources present in the environment), there is no impedance at an architectural level to constructing and executing multiple open workflows at once.

Our current system architecture varies in three ways from the architecture presented in Thomas et al. (2009). The first change is the replacement of the Auction Manager and Auction Participation Manager, which performed allocation in a manner similar to our earlier Collaboration in Ad hoc Networks (CiAN) middleware (Sen et al., 2008), with the Task Availability Manager which implements the approach described in this paper. The second change is to encapsulate location and travel time information management within the Current Location Manager and the Travel Route Manager. The third change is using co-design in finding the workflow plan, workflow plan allocation, and host scheduling instead of splitting and solving all three independently. These refinements were achieved without significant impact to other subsystems due to the modularity and flexibility of our architecture overall.

### 5. Evaluation

We use a combination of simulation and empirical evaluation to test our system and demonstrate the viability of the open workflow paradigm. In our implementation, we use a state-of-the-art temporal planner, SCPlan (Wah and Chen, 2006; Chen et al., 2006) to solve the compiled temporal planning problem in PDDL. We also compare our method with the enumeration search algorithm (ESearch) used in our previous work on the open workflow paradigm (Thomas et al., 2009). We show that ESearch may fail in some problems (finding unexecutable workflow allocations) since it cannot handle the communication constraints, while the method presented here can solve all of the problems considered in our evaluation correctly and efficiently. Furthermore, our method can solve large problems more efficiently than ESearch.

#### 5.1. Experimental setting

We focus on characterizing the performance of the system in terms of three variables that have the greatest impact on the scalability of our approach: the number of hosts in the community, the number of tasks known to the entire community, and the difficulty of the workflow problem being solved which we characterize by the workflow plan length, denoted as path length. Since it is hard to quantify whether a set of randomly generated workflow problems is difficult, our random problem generator generates a diverse range of problems based on several parameters:

1. \( n_h \): the number of hosts.
2. \( n_t \): the number of tasks.
3. \( n_l \): the number of locations which is set to 10.
4. \( \text{dis}(l_1, l_2) \): the distance between locations \( l_1 \) and \( l_2 \) which is randomly chosen from [25, 300] m. We set the communication range to 25 m which is the same as CiAN’s settings (Sen et al., 2008). If the distance of two locations is larger than 25 m, any two hosts at these two locations cannot exchange information with each other directly, which means they need to transmit messages through other hosts or one of them needs to move to the other’s communication range.
4. \( s \): the moving speed of hosts. We set it to 1.7 m/s which is close to the walking speed of human.

![Fig. 5. System architecture.](image-url)
5. \( d \): the duration of actions. The duration of execute is randomly chosen from \([50, 100] \) s, the duration of communicate is set to 1 s, and the duration of move is set to the distance of the two locations divided by the speed \((\text{dis}(l_1, l_2))/s\).

Our experimental setup is as follows. Given the number of hosts, the global number of tasks, and the path length of the workflow as parameters for an experiment, we configure the hosts, establish connectivity within the community, and then measure the time taken from when the specification is given to the initiating host to the time when all tasks of the resulting workflow have been successfully allocated.

To configure the hosts, we first construct a workflow supergraph of the chosen size by creating the desired number of nodes and then repeatedly adding edges between disconnected nodes until every two nodes in the graph are reachable. From this single supergraph we can then draw a large number of satisfiable specifications by randomly assigning triggers and goal conditions. Given a supergraph and a chosen number of hosts, we finish setting up the scenario by distributing the tasks randomly and evenly amongst the hosts, and independently distributing corresponding services randomly and evenly amongst the hosts. Each of the \( n \) hosts has only \( 1/n \) th of the entire supergraph, so the hosts must cooperate to solve the posed problem.

Typically, the complexity of a planning problem can be characterized by the number of facts and actions. For the temporal planning problems defined in our COW framework (see Section 3.2.2), the number of facts is \( O(n_{\text{hm}} L) \) (at \( th \) – \( ?l \) – location) + \( O(n_{\text{h}}) \) (free \( th \) – host) + \( O(n_{\text{t}}) \) (done \( ?c \) – condition) + \( O(n_{\text{hm}} T) \) (has-token \( th \) – host \( ?c \) – condition) = \( O(n_{\text{hm}} T) \) and the number of actions is \( O(n_{\text{hm}} L) \) (execute \( th \) – host \( ?l \) – location) + \( O(n_{\text{hm}} F) \) (move \( th \) – host \( ?l \) \( ?d \) – location) + \( O(n_{\text{hm}} F) \) (communicate \( th_1 \) \( ?d_2 \) – host \( ?l \) \( ?d_1 \) – location \( ?c \) – condition-token) = \( O(n_{\text{hm}} F) \). Besides the number of hosts and tasks, the path length is also an important metric for problem complexity. For an exhaustive or heuristic search method, the complexity is typically bounded by a search tree with a branching factor, denoted as \( h \), which is the average number of successor states at a given state. The number of nodes that needs to be examined can be roughly estimated by \( O(b^h) \), where \( L \) is the depth of the search tree, which is turned in the decision by the workflow plan length (path length). For each test run, the test driver randomly chooses a path of the desired path length through the supergraph, and the initial and final label nodes of the path are used as the specification for that test run. In Figs. 6, 7, 10 and 11, the results for each path length are the average of ten runs.

The running time limit for each problem instance is set to 30 s. We set this relatively low time limit in order to ensure the practicability of our approach in real-world applications, where users prefer short planning time.

5.2. Viability of our method

For the simulations, all the hosts were run within a single JVM and communicate solely through a simulated network. The simulations are run on a Linux workstation with a 2.4 GHz Intel Core Duo processor and 2.0 GB of memory, running the Java 1.6.0.22 Client VM. We use SGPlan to solve all the compiled temporal planning problems.

In Fig. 6(a) and (b), we show the average planning time, which includes the time of constructing workflow graph, solving temporal planning problem, and allocating solution tasks, for each path length from a supergraph with 300 task nodes as the number of participating hosts varies from 10 to 300. The average time grows roughly linearly with the number of hosts, as in our implementation the initiating host communicates pairwise with every member of the community during the construction and allocation phases. We note that even if we are to broadcast requests rather than using pairwise communication, the processing time of the initiating host is still linear in the number of hosts in the community. In Fig. 7(a) and (b), we show the average planning time for each path length for 20 participating hosts as the number of task nodes in the supergraph varies from 50 to 600. The average time grows roughly linearly with the number of task nodes. The rate of increase grows with the number of task nodes because the WfMS encounters more nodes during its search through the densely connected supergraph as the number of tasks increases. The longest path through the graph also increases as the size of the graph increases, which explains the absence of timings for some path lengths (e.g., those greater than 16 for the 100-node cases in Fig. 7(b)). For most of the problems with different host number and task node, the planning time grows roughly linearly with the path length (see Figs. 6(b) and 7(b)).

Given a workflow problem with a path length \( L \), suppose the probability of solving the problem by at least \( N \) replanning is \((1 - (1 - p)^N)\). Even though the probability of triggering a replanning is pretty large (see Fig. 8(a)), the probability of triggering multiple replanning for a single problem is acceptable (see Fig. 8(b)). To validate the ability of handling task allocation and execution exceptions, we generate a series of random problems that set the number of task nodes as 100 and the number of participating hosts as 20. We run each algorithm 50 times on each problem instance as the probability \( p \) varies from 0.01 to 0.04. In Fig. 9, we show the average planning time and the total number of triggering replanning for each path length. The results show that our system can handle the task allocation and execution exceptions with acceptable extra number of replanning and replanning time.

After the simulations, we perform empirical evaluation of our application using four Fit-2PCs connected by an ad hoc wireless network using 802.11g (54 Mbit/s). The hosts are all running Ubuntu Linux 8.04 with a 1.6 GHz Intel Atom processor and 1 GB of 533 MHz DDR2 memory. All hosts are running the Java 1.6.0.16 HotSpot Client VM. Connectivity among the hosts is verified before the measurements were started. The timing results for workflow graphs with 25, 50, 75, and 100 task nodes are shown in Fig. 10.

We can see from this graph that even in a realistic networking environment, our system shows the potential to solve reasonably sized problems with acceptable response times. For example, with a community knowledge base of 100 tasks to explore, and a workflow path length of 22, our system finds and allocates a solution in about 4.5 s on average.

5.3. Comparison between planning search and ESearch

We compare our method with the previous open workflow enumeration search algorithm (ESearch) (Thomas et al., 2009). We generate a series of random problems that set the number of task nodes as 100 and the number of participating hosts as 10. We run each algorithm 10 times on each problem instance. We measure the average planning time taken to find a workflow plan and
allocate it to corresponding hosts, the solution quality (makespan), and the number of executable allocations. Since ESearch does not consider the communication limitation, the solution found by ESearch may not be executable when two hosts are out of their communication range at a certain time. On the other hand, since our method solves the problem in a co-design fashion, all solutions found by our method can satisfy the availability, mobility, and communication constraints.

The average planning time, solution makespans, and the number of executable solutions are shown in Fig. 11. Fig. 11 (a) shows that SGPlan can solve all problems efficiently (within 0.26 s) and ESearch takes much longer time than SGPlan in problems with a path length greater than 14. For the problems with path length greater than 14, the solving time of ESearch grows exponentially with the path length while SGPlan grows linearly with a small rate. From Fig. 11(b), we see that ESearch has

![Fig. 6. Simulation of 300 task nodes partitioned across different numbers of hosts. (a) Planning time. (b) Planning time. (c) Path length = 10. (d) Path length = 14. (e) Path length = 18. (f) Path length = 22.](image-url)
a better solution quality in terms of makespan than SGPlan. However, due to its inability to handle communication constraints, ESearch usually cannot find a feasible allocation on problems whose path length is larger than 10 (see Fig. 11(c)), which means that the solutions with shorter makespans found by ESearch are usually unexecutable. This clearly shows the limitation of that approach and the advantage of our proposed co-design approach.

6. Related work

6.1. AI techniques for workflow management

In this paper, we use a temporal planner to solve workflow allocation and host scheduling problems. The most related works are using AI techniques in workflow engines. In our framework,
the workflow plan, workflow allocation, and host scheduling are considered as a co-design problem. An overview of early uses of AI techniques in workflow engines is presented in Myers and Berry (1999). This paper describes how techniques from the AI community, specifically reactive control, planning, and scheduling, could be leveraged to develop powerful, next-generation adaptive workflow engines that provide many of these advanced process management capabilities. Specifically, it focuses on two major areas of using AI planning and scheduling: (1) providing adaptive resource allocation and (2) providing process synthesis and repair (with a focus on replanning). Obviously, our work belongs to the former one. The work that considers workflow planning and coordination together we are aware of is Schuschel and Weske (2003). It shows that planning and coordination are closely interrelated and considering them separately may lead to deficiencies in the system’s consistency and performance. It builds an integrated workflow planning and coordination system combined planning algorithms and workflow management concepts. However, like many standard workflow management systems (see Section 1) which rely on centralized control and reliable communication, it cannot solve the coordination problems under the dynamic mobile environments considered in this paper.

In contrast, most other systems deal with the workflow plan, workflow allocation, and scheduling in separation. Jarvis et al. (1999) take a broad view of the problem of adaptive workflow systems and divide the workflow system into many layers, such as domain, process, agents, organization, and infrastructure. Then, they address each level with different AI technologies, such as dynamic capability matching, automated planning architecture, and rational capture. SWIM (Berry and Drabble, 1999), a workflow management system designed for Information Surveillance and Reconnaissance (ISR), considers the workflow managements as three roles, reactive control, scheduling, and continuous execution, and exploits different AI techniques, such as procedural reactive controllers (Firby, 1994), constraint-directed algorithms (Smith, 1994), constrained iterative repair (Zweben et al., 1994), hierarchical task network (HTN) planning (Erol et al., 1995), case-based planning (Hammond, 1989), etc., to meet the requirements for effective workflow engines. Based on the same idea of SWIM, a WfMS COSMOSS (Moreno and Kearney, 2002) was developed by a group at BT research laboratories to solve more complex domains with uncertainty in workflow task postconditions. It could automatically generate a new process library each time new products/templates are created and improve the overall system functionality by using a contingent planner, Cassandra (Pryor and Collins, 1996), to solve the transferred contingent planning problems. Note that our algorithm can also automate the generation of process library. Moreover, our method is designed to tackle a more
complex workflow problem with temporal constraints which entails a temporal planning problem that cannot be handled by COSMOSS. A summarization of the relationship between the Cow and these related WfMSs is shown in Table 2.

6.2. Automated service composition

The initial open workflow construction algorithm we present in this paper is closely related with automated (web) service composition. A review of automated service composition methods can be found in Rao and Su (2004). The approaches of automatic composition of services can be divided into two categories: (1) using workflow techniques and (2) using AI planning. The techniques we adopted in this paper include both dynamic workflow generation and PDDL planning techniques. In workflow-based composition, besides the dynamic workflow generation we used in this paper which both creates process model and selects atomic services automatically (e.g., eFlow Casati et al., 2000), the other approach is the static workflow generation which means that the requester needs to build an abstract process model before the composition planning starts (e.g., PPMS Schuster et al., 2000). The methods based on AI planning can be classified into four categories: (1) situation calculus, which adapts and extends the Golog (McIlraith and Son, 2002) language for automatic construction of web services (Mcilraith and Son, 2002); (2) rule-based planning, which generates composite services from high-level declarative description and uses composability rules to determine whether two services are composable (Medjahed et al., 2003; Ponnekanti and Fox, 2002); (3) theorem proving, which describes available services and user requirements in a certain language (e.g., first-order logic, Waldinger, 2000; intuitionistic logic, Lämmermann, 2002; linear logic, Rao et al., 2004) and extracts...
service composition descriptions from particular proofs generated by a theorem prover; (4) PDDL planning, which translates service composition problems to the PDDL format and exploits different planners for further service synthesis (McDermott, 2002).

While the open workflow construction algorithm we propose is actually a simplified alternative to the powerful techniques in these papers, it addresses a new problem specific to the mobile ad hoc environment. We have built upon their work by showing how to construct both the knowledge base and the derived workflow on the fly based on the knowledge and capabilities available within the community. Because of the fast development of highly efficient planning algorithms, a large variety of work following the idea of planning-based approaches to service composition have been proposed. In particular, automated composition of semantic web services based on different model language, such as OWL-S or WSML (Sirin et al., 2004; Hoffmann et al., 2007), DAML-S (Sheshagiri and Finin, 2003; Burstein et al., 2002), WS-BPEL (Bertoli et al., 2010), and WSDL (Zou et al., 2012, 2014). Compared to these works, instead of using standard industrial languages for business processes modeling and execution, we adopt a simpler model similar to the PDDL planning problem definition. A summarization of the relationship between the COW and these related automated (web) service composition systems is shown in Table 3.

6.3. WfMS with temporal constraints

Spatiotemporal constraint is an important feature in coordination problems in MANET environments. In our scheme, we introduce a temporal planning approach which is the first work that uses AI planning techniques to handle temporal constraints in workflow problems. More precisely, we use temporal actions with time duration and timed initial literals to describe the temporal constraints of workflow tasks and call a temporal planner to find a solution that satisfies spatiotemporal constraints. Most of the existing approaches are designed to derive or check certain properties of a temporally constrained system. However, they cannot be directly used to solve workflow allocation problems, nor the more complex co-design problems in MANETs.

Modeling temporal constraints in WfMS was first proposed by Marjanovic and Orlowska (1999) and Eder et al. (1999). Marjanovic introduced a concept called the “duration space” to enable visualization of some temporal constraints and a duration algorithm to calculate the shortest/longest workflow instances (Marjanovic and Orlowska, 1999). It checks the consistency of introduced temporal constraints based on the duration algorithm. Besides the “duration space”, the timed activity graph is also a powerful technique which provides primitives for expressing temporal constraints between activities and binding activity executions to certain fixed dates.

Table 2 A summarization of the relationship between the COW and related WfMSs using AI techniques.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Service/workflow construction</th>
<th>Modeling language</th>
<th>Composition technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>eFlow</td>
<td>Dynamic</td>
<td>eFlow</td>
<td>Rule-based composition</td>
</tr>
<tr>
<td>PPM</td>
<td>Static</td>
<td>PPM</td>
<td>Rule-based composition</td>
</tr>
<tr>
<td>McIlraith and Son (2002)</td>
<td>Static</td>
<td>DAML-S + Golog</td>
<td>Rule-based planning</td>
</tr>
<tr>
<td>Medjahed et al. (2003)</td>
<td>Static</td>
<td>WSDL + DAML + OIL</td>
<td>Rule-based planning</td>
</tr>
<tr>
<td>SWORD</td>
<td>Static</td>
<td>SWORD</td>
<td>Rule-based composition</td>
</tr>
<tr>
<td>Waldinger (2000)</td>
<td>Static</td>
<td>First-order logic</td>
<td>Automated deduction and program synthesis</td>
</tr>
<tr>
<td>Lammerrmann (2002)</td>
<td>Dynamic</td>
<td>Intuitionistic logic</td>
<td>Program synthesis</td>
</tr>
<tr>
<td>Rao et al. (2004)</td>
<td>Static</td>
<td>DAML-S</td>
<td>Linear logic theorem proving</td>
</tr>
<tr>
<td>McDermott (2002)</td>
<td>Static</td>
<td>PDDL</td>
<td>Estimated-regression planning (Unpop)</td>
</tr>
<tr>
<td>Sirin et al. (2004)</td>
<td>Static</td>
<td>OWL-S/WSML</td>
<td>HTN planning (SHOP2)</td>
</tr>
<tr>
<td>Hoffmann et al. (2007)</td>
<td>Static</td>
<td>DAML-S</td>
<td>Conformant planning (CFF)</td>
</tr>
<tr>
<td>Sheshagiri and Finin (2003)</td>
<td>Static</td>
<td>WS-BPEL</td>
<td>STRIPS planning</td>
</tr>
<tr>
<td>Bertoli et al. (2010)</td>
<td>Static</td>
<td>WSDL</td>
<td>Model checking</td>
</tr>
<tr>
<td>Zou et al. (2012, 2014)</td>
<td>Static</td>
<td>PDDL</td>
<td>CSTE planning (SCP), numeric planning (Metric-FF)</td>
</tr>
<tr>
<td>COW</td>
<td>Dynamic</td>
<td>PDDL</td>
<td>Temporal planning (SGPlan)</td>
</tr>
</tbody>
</table>

Table 3 A summarization of the relationship between the COW and related automated (web) service composition systems.

<table>
<thead>
<tr>
<th>WfMS</th>
<th>WfMS architecture</th>
<th>AI techniques</th>
<th>Supporting MANET features</th>
<th>Temporal constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>COW</td>
<td>Transform to classical planning</td>
<td>STRIPS planning, systematic nonlinear planning</td>
<td>Centralized control, reliable wired/wireless network, dynamically generated workflow, fixed hosts</td>
<td>No</td>
</tr>
<tr>
<td>SWIM</td>
<td>Multi-roles (reactive control, scheduling, continuous execution)</td>
<td>Rationale maintenance, planning and execution architectures, dynamic capability matching, multi-agent toolkits</td>
<td>Centralized control, reliable wired/wireless network, dynamically generated workflow, fixed hosts</td>
<td>No</td>
</tr>
<tr>
<td>COSMOSS</td>
<td>Transform to contingent planning</td>
<td>Procedural reactive controllers, constraint-directed algorithms, constrained iterative repair, HTN planning, case-based planning, etc.</td>
<td>Centralized control, reliable wired/wireless network, dynamically generated workflow, fixed hosts</td>
<td>Yes</td>
</tr>
<tr>
<td>COW</td>
<td>Transform to temporal planning</td>
<td>Contingent planning (Cassandra)</td>
<td>Decentralized control, mobile ad hoc network, dynamically generated workflow, dynamic hosts</td>
<td>Yes</td>
</tr>
</tbody>
</table>
At run time, it dynamically computes the timed activity graph which includes deadline ranges for each activity to monitor satisfiability of the remaining time constraints. Critical activity detection is another useful technique. It aims at finding a critical activity whose delay of completion directly affects the overall processing time of a workflow (Son and Kim, 2001). Son and Kim (2001) developed a method to determine the minimum number of servers (MNS) for any critical activities and maximize the number of workflow instances satisfying given deadlines.

An important area that usually involves temporal constraints is scientific workflows which models complex e-science processes. Many existing processes such as climate modeling often have a few temporal constraints globally. Unlike our approach which handles temporal constraints globally, local handling is also an effective approach. Chen and Yang (2010) systematically investigate how to localize a group of fine-grained temporal constraints so that temporal violations can be identified locally for better cost effectiveness. An interesting direction for our future work would be incorporating such analysis as heuristic guidance and pruning conditions to further improve our planning approach. A summarization of the relationship between the COW and these related WfMSs is shown in Table 4.

<table>
<thead>
<tr>
<th>WfMS</th>
<th>Temporal modeling</th>
<th>Handle temporal constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Marjanovic and Orlowska (1999)</td>
<td>Duration space</td>
<td>Shortest-path partitioning algorithm, critical path method (CPM)</td>
</tr>
<tr>
<td>Eder et al. (1999)</td>
<td>Timed activity graph</td>
<td>Critical path method (CPM)</td>
</tr>
<tr>
<td>Son and Kim (2001)</td>
<td>Timed activity graph</td>
<td>Critical activity detection</td>
</tr>
<tr>
<td>Chen and Yang (2010)</td>
<td>Timed activity graph</td>
<td>Localizing fine-grained upper bound constraints</td>
</tr>
<tr>
<td>COW</td>
<td>PDDL</td>
<td>Temporal planning (SGPlan)</td>
</tr>
</tbody>
</table>

In producing the first practical implementation of an open workflow management system, we have affected a major paradigm shift in workflow middleware. Open workflows are much more than sophisticated scripts that enable one to exploit available services — they are a coordination vehicle for social and business activities that allows cooperating hosts to construct and execute responses to needs identified by the hosts. The open workflow paradigm enables the development of an entirely new class of systems that are nimble, mobile, and supportive of this new style of coordination.

A practical workflow allocation algorithm on MANETs has many applications. Responding to catastrophes such as chemical spills, conducting geological surveys of remote areas, and even managing a community of robots exploring hazardous areas are only a few examples of many activities that would potentially benefit from this approach. We plan to explore such real applications in our future work.

We also want to investigate relaxing the current restriction that construction and allocation are performed by a single host. A middleware that supports distribution of these tasks would allow construction and allocation in the face of fragmentation of the community and support localized recovery after a failure. When location constraints prohibit a rendezvous for data transfer, the system should be extended to consider scheduling hosts into the workflow as couriers.

Finally, as with any application facing the rigors of the real world, security is critical. In addition to the usual concerns of trust, authorization, and privacy, the open workflow paradigm presents new challenges as it encourages participation across multiple administrative domains and social networks. Recognizing and handling changes in authorization and privacy due to roles and social context and resolving conflicting and competing specification ontologies are topics for future research.

Acknowledgments

This work has been supported in part by China Postdoctoral Science Foundation (No. 2013M531527), the Fundamental Research Funds for the Central Universities (No. 0110000037), China Scholarship Council, National Natural Science Foundation of China (Nos. 61033009 and 61175057), United States NSF Grants IIS-0534669, IIS-0713109, CNS-1017701, and a Microsoft Research New Faculty Fellowship.

References


