Report Number: wucse-2009-73

2009

Open Workflows: Context-Dependent Construction and Execution in Mobile Wireless Settings

Louis Thomas, Justin Wilson, Grui-Catalin Roman, and Christopher Gill

Existing workflow middleware executes tasks orchestrated by rules defined in a carefully handcrafted static graph. Workflow management systems have proved effective for service-oriented business automation in stable, wired infrastructures. We introduce a radically new paradigm for workflow construction and execution called open workflow to support goal-directed coordination among physically mobile people and devices that form a transient community over an ad hoc wireless network. The quintessential feature of the open workflow paradigm is dynamic construction and execution of custom, context-specific workflows in response to unpredictable and evolving circumstances by exploiting the knowledge and services available within a given spatiotemporal... **Read complete abstract on page 2.**

Open Workflows: Context-Dependent Construction and Execution in Mobile Wireless Settings

Louis Thomas, Justin Wilson, Grui-Catalin Roman, and Christopher Gill

Complete Abstract:

Existing workflow middleware executes tasks orchestrated by rules defined in a carefully handcrafted static graph. Workflow management systems have proved effective for service-oriented business automation in stable, wired infrastructures. We introduce a radically new paradigm for workflow construction and execution called open workflow to support goal-directed coordination among physically mobile people and devices that form a transient community over an ad hoc wireless network. The quintessential feature of the open workflow paradigm is dynamic construction and execution of custom, context-specific workflows in response to unpredictable and evolving circumstances by exploiting the knowledge and services available within a given spatiotemporal context. This work introduces the open workflow approach, surveys open research challenges in this promising new field, and presents algorithmic, architectural, and evaluation results for the first practical realization of an open workflow management system.

2009-73

Open Workflows: Context-Dependent Construction and Execution in Mobile Wireless Settings

Authors: Louis Thomas, Justin Wilson, Gruia-Catalin Roman, and Christopher Gill

Web Page: http://mobilab.wustl.edu/projects/openworkflow/

Abstract: Existing workflow middleware executes tasks orchestrated by rules defined in a carefully handcrafted static graph. Workflow management systems have proved effective for service-oriented business automation in stable, wired infrastructures. We introduce a radically new paradigm for workflow construction and execution called open workflow to support goal-directed coordination among physically mobile people and devices that form a transient community over an ad hoc wireless network. The quintessential feature of the open workflow paradigm is dynamic construction and execution of custom, context-specific workflows in response to unpredictable and evolving circumstances by exploiting the knowledge and services available within a given spatiotemporal context. This work introduces the open workflow approach, surveys open research challenges in this promising new field, and presents algorithmic, architectural, and evaluation results for the first practical realization of an open workflow management system.

Type of Report: Other

Open Workflows: Context-Dependent Construction and Execution in Mobile Wireless Settings

Louis Thomas, Justin Wilson, Gruia-Catalin Roman, and Christopher Gill
Department of Computer Science and Engineering
Washington University in St. Louis
{thomasl,wilsonj,roman,cdgill}@cse.wustl.edu

***Abstract*—Existing workflow middleware executes tasks orchestrated by rules defined in a carefully handcrafted static graph. Workflow management systems have proved effective for service-oriented business automation in stable, wired infrastructures. We introduce a radically new paradigm for workflow construction and execution called *open workflow* to support goal-directed coordination among physically mobile people and devices that form a transient community over an ad hoc wireless network. The quintessential feature of the open workflow paradigm is dynamic construction and execution of custom, context-specific workflows in response to unpredictable and evolving circumstances by exploiting the knowledge and services available within a given spatiotemporal context. This work introduces the open workflow approach, surveys open research challenges in this promising new field, and presents algorithmic, architectural, and evaluation results for the first practical realization of an open workflow management system.**

I. Introduction

With the development of small, powerful wireless devices, computing must embrace the frequent, transient, ad hoc interactions inherent to mobile environments. As computing and communication become more and more integrated into the fabric of our society, new kinds of enterprises and new forms of social interactions will continue to emerge. We ask the fundamental question (which we explored first in [1]): how can ad hoc communities of people and their personal devices coordinate to solve problems? 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 share 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. For instance, consider a construction worker discovering a mercury spill. While there is a prescribed response, it is his supervisor who has the needed expertise and training. She initiates the response, but access to the spill is made difficult by a support structure whose dismantling requires special intervention which only the chief engineer can manage. The result is a series of frantic phone calls and the dispatching of various workers and equipment to execute what might 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 the 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, e.g., to print photos, reserve tickets, or make a bid in an online auction. The workflow is a directed acyclic graph with vertices denoting tasks and edges defining an execution order along with the flow of data and control. Each task is a specification for a service to be discovered and invoked by the workflow engine. What makes the workflow paradigm successful is the high degree of decoupling that it exhibits at multiple levels: between the user's need and the workflow required to satisfy it, between the task specifications and the services that implement them, and between the workflow engine that invokes a service and the service provider that executes it.

Despite workflow middleware being well established, efforts toward using it in ad hoc wireless environments are relatively new. Our previous research in this area includes the development of workflow execution engines targeted to small portable devices [2], and techniques for executing workflows in mobile wireless networks [3]. These studies reveal the need for a major reevaluation of the way one thinks about workflow middleware: hosts may move, service availability may depend upon which hosts are within communication range, user needs tend to be situational, and one cannot anticipate the range of responses demanded by changing circumstances. These observations suggest that in ad hoc wireless settings it is desirable to tailor or generate workflows dynamically.

Starting with this premise, we pose the question of how workflow middleware might be reshaped for use *in the absence of any wired connectivity*. In this research, we explore whether workflow middleware can become a coordination mechanism for activities that are carried out in an ad hoc setting.

We use the term *open workflow* to denote a workflow specification, construction, and execution paradigm that is shaped by the dynamics and constraints of an activity whose underlying infrastructure is a mobile ad hoc wireless network. We assume a set of participants (people and the host devices they carry) that are dedicated towards a common purpose and that move about and interact with each other and with the

real world. The participants form a transient community that evolves over time. In our approach, one of the members of a community identifies a need for action, which then results in the dynamic construction of a workflow to satisfy the need and the execution of that workflow in a distributed and cooperative manner. An important feature of the open workflow paradigm is the workflow construction process: workflow fragments encoding individual knowledge distributed across the set of participants are assembled into a custom workflow both automatically and contextually. In doing so, we also consider the available resources (expressed as services offered by the participants) along with the mobility of the participants and their willingness to commit to being present at a specific place and time and to delivering results to any other dependent participants. The latter highlights another feature of the open workflow paradigm, its sensitivity to the time and location considerations necessary when performing activities in the real world.

In [1] we began to explore the challenges of building a workflow on the fly from available contextual knowledge, i.e., the open workflow paradigm, and built a platform for further experimentation with that approach. In this article, we expand the approach by presenting in Section II an enhanced formalism for describing open workflow construction that supports *parameterized* tasks for better context sensitivity. Section III explains how we achieve collaborative construction, allocation, and execution of open workflows and extends the construction and allocation algorithms presented in [1] to capture the dynamics of service availability. In Section IV, we present our updated open workflow management system and discuss its architecture. In Section V, we evaluate its performance and discuss directions for future work. Section VI highlights related research and contrasts it with this work. We provide conclusions in Section VII.

II. Problem Definition

A. 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 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. 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.

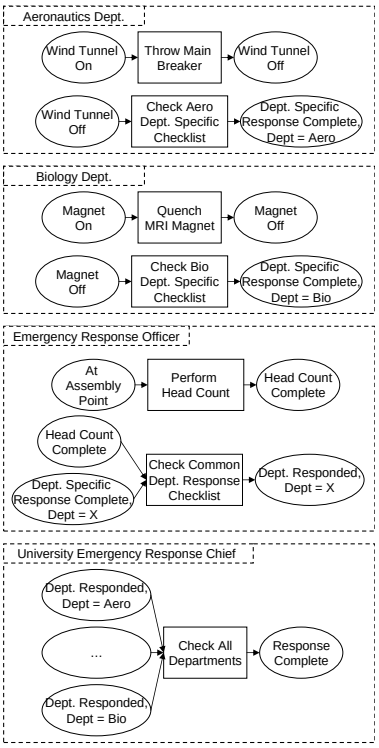


Fig. 1. Task knowledge in an emergency response scenario.

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 knowledge contained on the mobile devices owned by the members of the emergency preparedness team. As shown in Figure 1, the chief's PDA 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.

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,

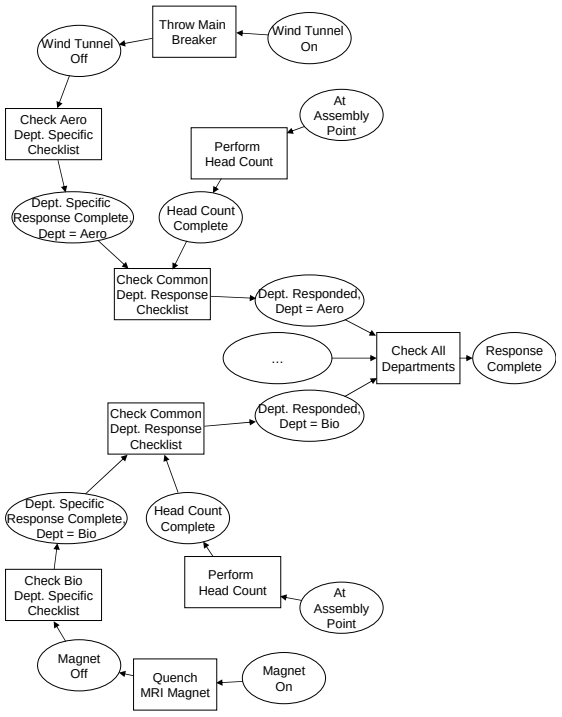


Fig. 2. A customized emergency response workflow.

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 Figure 2.

The open workflow engine then searches for participants that are able to perform the activities indicated by the workflow. The aeronautics professor's PDA 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 biology department's checklist would already be met and there would be no need to add a task to turn it off.

A 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 PDA to include the carnival as another "department". Even people not normally considered 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.

A traditional static workflow would be difficult to maintain in the face of the 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. 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.

B. Formalization

A *community* consists of a set of *participants* (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. However, we make the simplifying assumption that the community is stable for each problem instance and that all participants are mutually reachable with no changes in connectivity or resource availability while the community is creating a plan (i.e., a workflow) to solve the problem. This assumption is reasonable if the time taken to generate the plan is sufficiently short.

A *workflow* is defined as a collection of interlinked abstract *tasks*. A task represents a single abstract behavior or accomplishment without completely specifying how it must be performed. A *service* is a concrete implementation of a task and may involve a computation by the device, an activity performed by the user, 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, different tasks may be performed in sequence or in parallel by one actor or by multiple actors. Each task has *preconditions* that must be met before the task can be performed, and *postconditions* that describe the results of performing the task. Abstractly, we can enable the performance of a given task by performing one or more preceding tasks

whose postconditions taken together ensure the preconditions necessary for the given task. The order, timing, and executors of the preceding tasks are unconstrained so long as the given task's preconditions hold when it is to be performed. As the postcondition of one task can be the precondition of another, we can treat them uniformly as *conditions*. We assume that a task is either *conjunctive*, requiring all of its preconditions, or *disjunctive*, requiring only one of its preconditions, and that a task produces or establishes all of its postconditions.

Tasks can be joined together by exactly matching preconditions to postconditions. Conditions and tasks within a workflow thus may be considered nodes in a bipartite directed acyclic graph. We begin with the simplifying assuming that each condition and task in the graph can be represented by a simple semantic *label*, where each label has a distinct meaning.

A workflow has the additional constraints that (1) all sources (nodes without any incoming edges) and all sinks (nodes without any outgoing edges) are conditions, (2) a condition can have at most one incoming edge, and (3) 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 yields 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.

A workflow is constructed in response to an expressed need. In general, this need is stated in terms of a *specification* S: a *predicate* that indicates whether or not a workflow is *satisfactory*. The *inset* and the *outset* of a workflow are its sources and sinks respectively. We assume S is of the form

$$S \in \mathcal{P}(Conditions) \times \mathcal{P}(Conditions) \mapsto Boolean$$

A workflow W with inset $W.in$ and outset $W.out$ then satisfies a specification S if and only if $S(W.in, W.out)$ is true.

Composing workflow fragments may produce a workflow that cannot satisfy a specification S only due to the existence of extra sinks or sources. We can prune a workflow to remove unnecessary data flows, subject to the following constraints which ensure the result remains a valid workflow: (1) task outputs that are sinks can be pruned so long as every task has at least one output, (2) task inputs that are sources can be pruned for disjunctive tasks so long as every task has at least one input, and (3) tasks can be pruned so long as any task inputs that are sources and any task outputs that are sinks are also pruned.

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. 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 in K 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 participants move around in space, the knowledge available to the community changes with its membership and their experiences. For the same specification, different communities may respond differently or may be unable to construct an appropriate workflow.

As the plan is formed, tasks must be *allocated* to participants who will eventually *execute* corresponding services. The *availability* of services and resources within the community determines to whom tasks are allocated. Service availability is determined by whether any participant can commit to providing a service: that is, (1) whether the participant is *capable* of performing the service, (2) whether the participant has time available, (3) whether the participant can travel to the necessary location to perform the service, (4) whether the participant can gather the necessary inputs and distribute any outputs in a timely manner, and (5) whether the participant is willing (according to their preferences) to perform the service. If the community is stable and all participants are mutually reachable, it is easy to guarantee that the participants supporting the execution of tasks that depend upon each other are able to communicate the needed results in a timely fashion. More sophisticated routing techniques and analysis [4] may be needed if the movement of participants results in temporary disconnections. Once a participant has made a commitment, it is responsible for ensuring the service is executed as agreed. A participant 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 participants 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.

We can expand the expressiveness of our workflow representation by relaxing the assumption that conditions are represented as simple labels. Let the representation of each condition be a set of *attributes*, where each attribute is a 3-tuple consisting of a *name*, a *type*, and a *value-set*. The name of an attribute identifies its semantics, and the value-set of an attribute is a nonempty set of elements of the attribute's type. Attributes are *metadata* describing the actual data flow or control flow semantics of conditions. By tracking multiple attributes for each condition, we can more closely model realistic behavior. For example, consider a task that delivers a load of bricks. Instead of having a postcondition of "bricks delivered", we can have the more descriptive postcondition of "label: bricks delivered, location: St. Louis, quantity: 50", which is significantly different from the task with the postcondition "label: bricks delivered, location: St. Charles, quantity: 100". In this case, each condition has three attributes, namely 'label', 'location', and 'quantity'. We use value-sets rather than single values as it allows us to indicate that a

metadata attribute has multiple acceptable values, whether represented as a numeric range or a set of enumerated values. For example, "shipping: {overnight, 2-day}" indicates that both overnight and 2-day shipping are acceptable, but ground shipping not acceptable.

In order for tasks to be joined, their preconditions and postconditions still must match exactly. When they do not match, it may still be possible to join the tasks by *specializing* them, further constraining the tasks' preconditions and postconditions when there are multiple acceptable values until they do match. The required specialization can be determined by *intersecting* the conditions; if the intersection is empty, the tasks cannot be connected. Starting at the bottom of the hierarchy, two attributes (with the same name and type) may be intersected to produce a new attribute (with the same name and type) whose value-set is the intersection of the original attributes' value-sets. However, if the intersection of the value-sets is the empty set, then the intersection of the attributes is *empty* — there is no attribute corresponding to the intersection. Next, two conditions may be intersected to produce a new condition containing the attributes resulting from intersecting the attributes of the original conditions on a name-and-type by name-and-type basis. If any of the attribute intersections are empty, then the intersection of the conditions is considered empty. For example, if the postcondition of task "Buy Animal" is the condition "animal: {pig, chicken}" and the precondition of task "Raise Animal" is "animal: {cow, chicken}", then the tasks cannot be safely connected as we might buy an animal that we can't raise. However, the intersection of the conditions is "animal: chicken", meaning that both tasks agree that a chicken would be acceptable, so a specialized version of the "Buy Animal" task that only produced chickens and a specialized version of the "Raise Animal" task that only consumed chickens *could* be connected together. If the precondition of task "Raise Animal" was "animal: {dog, cat}", the intersection would be empty and there would be no way to specialize these tasks to make them compatible. If there are any attributes in one condition that are not in the other, then the intersection of the conditions is also considered empty — that is, there is a semantic mismatch and the two conditions cannot safely be connected. For example, there is no way to specialize a task with the postcondition "animal: goose" and a task with the precondition "animal: goose, lays-golden-eggs: true" to make them compatible. Even though both tasks involve a goose, a task that can produce a goose is unlikely to be able to produce a goose that lays golden eggs, and a task that requires a goose that lays golden eggs is unlikely to be able to establish its postconditions with a regular goose.

To correctly specialize a task, it is important to maintain the same relationships among the preconditions and postconditions of the new task that the original task establishes among its preconditions and postconditions. We extend the definition of a task to include a set of *relations*. A relation is a formula establishing a relationship between two or more individual attributes of task. We make the simplifying assumption that a relation always establishes the *equality* of two attributes. For example, consider the task "Raise Animal" which has precondition "label: baby animal, animal: {cow, chicken}" and the postcondition "label: adult animal, animal {cow, chicken}". If we specialize this task so that the postcondition is an adult cow, we need to correspondingly change the precondition lest we claim that we can turn a baby chicken into an adult cow.[1] We add the relation

$$precondition.animal = postcondition.animal$$

to our task definition to make the relationship explicit. Relations can establish equality between any two attributes, whether one is in the precondition and one is in the postcondition, both are in the precondition, or both are in the postcondition. For example, when assembling a car it may be important to establish the relation that the color of received car door be the same as the color of the received car body.

Changing condition labels to attribute sets also requires a clarification of the workflow composition rules. Duplicate condition nodes are still prohibited in the graph, but the condition "animal: {pig, chicken}" and the condition "animal: chicken" are considered distinct. A "Buy Animal" task with the postcondition condition "animal: {pig, chicken}" should also be distinct from a "Buy Animal" task with the postcondition "animal: chicken", so we change the rule to say that a task's uniqueness is determined by is preconditions, postconditions, and label taken as a whole. Thus, two different specializations of the same task are not considered duplicate tasks, and we can use a task more that once in a workflow *so long as* the contexts (as represented by the specializations of the tasks) are distinct. A general task may be performed at two different locations or produce two different sized outputs in the same workflow because the instances are two distinct specializations.[2]

With the introduction of attribute sets and task specialization, we need new rules to specify which services implement a task specification. Here we consider each task *as it is used in the final workflow*, meaning that it may have adjusted attributes due to specialization and that unnecessary preconditions and postconditions will have been removed. A service implements a task's specification if it meets the following four requirements.

1) The service's label must exactly match the label of the task, and the service and the task must both be conjunctive or both be disjunctive.
2) The service's relations must exactly match the task's relations.[3]
3) The service's preconditions must *dominate* the tasks preconditions. That is, the service must accept *at least* all of the possible input values that the task accepts,

[1]The original task indicates that it *might* perform this feat, but not that it *must*. A service that just raises the animal normally still conforms to the behavior specified by the original task.

[2]The careful reader may have noticed that the workflow in Figure 2 violates this rule. Adding a "location" attribute to the "At Assembly Point" condition would make the workflow valid and be very helpful to the officer trying to perform the head count.

[3]This is a sufficient though not strictly necessary condition to guarantee a service meets a task specification. It is only necessary that a service's relations establish at least all the conditions the tasks's relations establish, but this quickly becomes undecidable once the relations besides simple equality are allowed.

though it may accept inputs that the task would not. We say that one attribute dominates another if they have the same type and the dominant attribute's value-set contains at least all the values in the subordinant attribute's value-set. A condition dominates another if they have the same set of attribute names and all of the dominant condition's attributes dominate the subordinant condition's attributes on a name-by-name basis. For a disjunctive task, the service must have a precondition that dominates the task's one precondition (and any extra service preconditions are disregarded). For a conjunctive task, the task and the service must have the same number of preconditions and there must be a one-to-one mapping from task precondition to service precondition such that each service precondition dominates its corresponding task precondition.

4) The tasks's postconditions must dominate the service's postconditions, when the service's preconditions are specialized to match the task's preconditions. That is, the service must produce *at most* any of the possible output values that the task produces, though it may never produce some outputs that the task would. The service must have at least as many postconditions as the task and there must be a one-to-one mapping from task postcondition to service postcondition such that each task postcondition dominate the service postcondition. (Any extra service postconditions are disregarded.)

III. Collaborative Construction, Allocation, and Execution

A. Construction

We begin this section by introducing a construction algorithm for open workflows under the simplifying assumption that conditions are represented by simple labels. We assume a participant has identified a need for action and generated a specification S of the form

$$W.in \subseteq \iota \wedge W.out = \omega$$

where ι and ω are sets of conditions with ι being the triggering conditions and ω being the conditions that represent the goal. The participant is in contact with the other members of a community and can collect from each a set of workflow fragments. For the purposes of illustration, we start with the simplifying assumption that the participant initially collects all the fragments in the community to create the set K. Using the gathered information, the participant runs our algorithm to find a feasible workflow — a workflow composed of fragments from K (subject to pruning) that satisfies S — if one exists. We only consider initially the issue of generating one feasible workflow, although there are potentially many ways of combining fragments in K to satisfy S. While our algorithm chooses arbitrarily among equivalent options, any heuristic may be incorporated to direct the search toward more favorable solutions.

Our algorithm is based on graph traversal and graph coloring, and takes its inspiration from spanning tree algorithms and routing algorithms such as AODV [5]. Our strategy is to combine all workflow fragments from K into one large graph, henceforth called the workflow *supergraph* G. The supergraph represents a unified view of all possible actions represented in the set K, however it is not necessarily 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 G to identify one feasible workflow within this graph. We start by coloring the nodes corresponding to set ι of the specification S. Following the data flows, we explore the graph, growing the colored section as we identify which tasks and conditions are *reachable* from ι. We call a condition reachable when it is in ι or when it denotes the output of a reachable task; a task is reachable when all necessary input conditions are available for its execution via some path starting from ι.

Once we have reached all the elements of ω, 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 ω. The pruning phase removes cycles, ensures only one task produces each output, and excludes undesirable outputs. Once the second color has swept all the way back to ι, 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 pseudo-code in Algorithm 1. For purposes of the algorithm, we annotate every node and edge in G with a $color$ (initially $uncolored$) and every node with a $distance$ (initially ∞) from a source on the graph. Nodes are marked $green$ for reachability during the exploration phase and $blue$ for workflow membership during the pruning phase; $purple$ identifies nodes on the boundary of the blue region. Condition nodes are considered disjunctive. The algorithm selects nodes nondeterministically; any node may be processed next so long as it matches the guard condition.

We offer a proof sketch of the correctness of our algorithm by highlighting several key invariants. First, we claim that every green node is reachable starting from ι, and all of its prerequisites have a smaller distance. A node is reachable when it is in ι, or when its prerequisites are reachable. The invariant holds after every step of the algorithm because we start with the nodes in ι with distance 0 and we work outward one edge at a time, coloring a node n green only when n's prerequisites are already green (reachable) and assigning n a distance greater than any of its prerequisites.

Second, once ω is colored blue, we claim that after every even number of iterations, the graph of blue nodes and blue edges is a valid workflow. At each step we choose a node n which is in the inset of the blue portion of the supergraph as it has no blue parents. Once we color the prerequisites of n blue, n is no longer a member of the inset but the prerequisite nodes are now members, so n and thus n's dependents are still reachable from the inset. On an odd iteration we color a task, and on the even iteration we color its prerequisite conditions. Thus, after each pair of steps, the sinks and sources of the graph will be conditions and the graph will be a valid workflow.

Algorithm 1 Workflow Construction (given ι, ω, and K)

— *Construct Supergraph* —
$G \leftarrow \emptyset$
for all fragments $F \in K$ **do**
  for all nodes $n \in F$ **do** **if** $n \notin G$ **then** $G \leftarrow G \cup \{n\}$
  end if **end for**
  for all edges $e \in F$ **do** **if** $e \notin G$ **then** $G \leftarrow G \cup \{e\}$
  end if **end for**
end for

— *Exploration Phase* —
Track the set of *greenNodes* (initially empty).
for all $n \in \iota$ **do** $(n.color,\, n.distance) \leftarrow (green,\, 0)$ **end for**
until $\omega \subseteq greenNodes \vee$ none of the following cases apply,
for some $n \in G$ **do**
  if n is disjunctive $\wedge$ any of n's parents are $green$ **then**
    $d \leftarrow \min\{p \in n\text{'s parents} \vee p.color = green \,|\, p.distance\}$
    if $(n.color = uncolored \vee (n.color = green \wedge n.distance > d+1))$ **then**
      $(n.color,\, n.distance) \leftarrow (green,\, d+1)$
    end if
  else if n is conjunctive $\wedge$ all of n's parents are $green$ **then**
    $d \leftarrow \max\{p \in n\text{'s parents} \vee p.color = green \,|\, p.distance\}$
    if $(n.color = uncolored \vee (n.color = green \wedge n.distance > d+1))$ **then**
      $(n.color,\, n.distance) \leftarrow (green,\, d+1)$
    end if
  end if
end until
if $\neg(\omega \subseteq greenNodes)$ **then** there is no solution — exit.

— *Pruning Phase* —
Track the set of *purpleNodes* (initially empty).
for all $n \in \omega$ **do** $n.color \leftarrow purple$ **end for**
until $purpleNodes = \emptyset$ **for** some $n \in purpleNodes$ **do**
  if $n.distance = 0$ **then**
    $requiredParents \leftarrow \emptyset$
  else if n is disjunctive **then**
    $requiredParents \leftarrow$ {the parent of n with minimum $distance$}
  else if n is conjunctive **then**
    $requiredParents \leftarrow n$'s parents
  end if
  for all $p \in requiredParents$ **do**
    $edge(p,\, n).color \leftarrow blue$
    if $p.color = green$ **then** $p.color \leftarrow purple$ **end if**
  end for
  $n.color \leftarrow blue$
end until
The set of nodes and edges colored $blue$ is the constructed workflow.

Finally, we claim that the coloring of blue nodes will eventually terminate, and upon termination the graph formed by the blue nodes and edges will be a workflow satisfying specification S. From the first invariant, every node n with distance greater than 0 must have prerequisites with distance strictly less than n's distance. Every time a node n in the inset is replaced with its prerequisites, the distance of the nodes added to the inset is strictly less than the distance of the node removed. Eventually the inset will consist solely of nodes with distance 0 (thus nodes in ι) and the algorithm will terminate. As the inset is a subset of ι and the outset is equal to ω, the workflow consisting of the blue nodes and edges satisfies S.

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 community to one whose participants are within communication range of each other and announce their willingness to participate. The community is dynamic as members can join and leave at will.

We observe that the coloring process requires only local knowledge. Thus, we relax the assumption that all of the workflow fragments are collected from the community before the coloring process begins. In our implementation, the member constructing the workflow builds the set of workflow fragments K and thus the supergraph G incrementally by querying other members of the community for workflow fragments that can be used to extend G. Members joining after the algorithm has started can still contribute knowledge, and the departure of a member does not affect the knowledge already collected in the supergraph.

We next tackle the challenge of constructing a valid workflow when configurations are represented as attribute sets rather than simple labels. If we were to directly apply the preceding algorithm, we would need to create a supergraph node for every possible specialization of every task that may appear in the workflow, which quickly becomes intractably large. Instead, we place each task into the supergraph once, using a special representation of the conditions. Consider two tasks that can be connected in a workflow by specializing their conditions. Their corresponding task nodes should be correspondingly connected (through a condition node) in the supergraph. However, if the tasks' original conditions do not match, the two task nodes would be connected to two different condition nodes with no connection between them. We combine those two condition nodes into a *condition bin* node, where a condition bin represents a *generalization* rather than a specialization of a condition. We say that a condition fits into a bin if (1) the condition and the bin have the same number of attributes and the same set of attribute names, and (2) for each attribute, the bin contains at least one value that the condition contains. Our supergraph is now a bipartite graph of task nodes and condition bin nodes.

As new workflow fragments are received from the community to be added to the supergraph, we must incrementally update the condition bins with the new conditions. We maintain the invariants that every condition in K fits into exactly one bin in the supergraph, and that every pair of conditions with a non-empty intersection fall into the same bin. If a

new condition fits into none of the condition bins in the supergraph, a new condition bin is created that has the same attribute sets as the condition. If the new condition fits into one existing bin, the bin is updated by adding any new values from the condition's value-sets to the bin's value-sets, maintaining the invariants by ensuring that any subsequently encountered condition with these values will also fit into this bin. Finally, if the new condition fits into more than one existing bin, all matching condition bins must be merged. Condition bins are merged by (1) creating a new bin containing an attribute-by-attribute union of all the value-sets, and (2) pointing the edges incident on all the old bins to the new bin. The merging of two condition bins in a supergraph can force further merging with other related bins in order to maintain the invariants.

The resulting supergraph is a useful, simplified representation of the workflow construction problem. If there is no workflow in the supergraph that satisfies S while ignoring task specialization and relations, then there is definitely no satisfying workflow to be constructed from the available tasks while honoring task specialization and relations. The inverse does not hold, but we use the detection of a simplified workflow in the supergraph as an indication that we can start searching for a complete workflow.

There is more useful information we can glean from the supergraph. As stated earlier, tasks reachable from ι are colored green in the supergraph. We can also use the supergraph to identify and color all the tasks that can eventually reach ω. We use a similar traversal algorithm going in the opposite direction to identify these tasks. As we now want a node to possibly have two colors, we will say that this reverse traversal *flags* nodes with the color *red*. 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.

We can construct valid workflow satisfying S by building backward from ω to ι just as in the blue coloring process. Each time a task is added to the workflow under construction, both the new task and the existing task in the workflow may need to be specialized before the new task can be joined to the existing task. When a parent task must be chosen for a condition, the supergraph constrains the search space to only those tasks connected to the corresponding condition bin. However, we do not know which of those parent tasks, if any, will lead to a feasible workflow, so we may need to use backtracking to try the alternatives. The construction of the workflow can traverse cycles in the supergraph, so long as a particular specialization of a task in the cycle does not occur more than once in the final workflow.

At this point, while we can construct a valid workflow satisfying S, we do not have enough information to guarantee that the workflow we construct will be *allocatable*. We discuss out approach to this problem in the next section.

B. Allocation

If we do not take into account the availability of services as provided by the members of the community, we may construct a feasible workflow that the community cannot execute. Once the supergraph indicates that a task may be in the final workflow (that is, the task is flagged as both green and red), we query the community to find the availability of services corresponding to this task. A participant's availability information for a task consists of (1) the set of services provided by the participant that implement the task and (2) a list of the participant's *availability windows* for each service, where each window consists a starting time, a starting location (where the participant will be when it becomes available), an ending time, and an ending location (where the participant must be to meet its next commitment). A window may also be open ended (no end time or end location) if the participant has no further commitments. The supergraph allows us to collect only the relevant details from the extensive availability information distributed amongst the participants. We make the simplifying assumption that every service available within the community has at least one open ended availability window.

For each task, there may may be no service availability, or there may be multiple implementing services available from multiple participants. To ensure that we only consider tasks with available corresponding services during construction, we create a new supergraph from the service definitions (which we denote the *service supergraph* to distinguish it from the *task supergraph*). We execute our previously described construction algorithm using the service supergraph, producing a valid workflow consisting of *service invocations* that satisfies the specification S using only services that are available within the community. We use the term service invocation to describe a single usage of a service in a workflow, since a service may be invoked multiple times but with different contexts (specializations) as previously described. In our initial implementation of this algorithm, we perform an iterative deepening depth first search through the service supergraph to construct our final workflow.

The final step of the allocation phase is to allocate each service invocation in the workflow to a participant in the community. For each service invocation in the constructed workflow that has no unallocated predecessors, we allocate it to the first available time slot in our collection of participant availability windows. Our allocation is sensitive to the spatiotemporal constraints of the participants. We assume that each service invocation will have a specified location and duration determined from the service and task metadata during construction, and we assume that each participant can tell us how long it takes for that participant to travel between two locations. To determine whether a service allocation will fit into an availability window, we verify that the participant has enough time to (1) travel from their starting location to the service location, (2) execute the service for the specified duration, and (3) travel from the service location to their ending location (to meet their next commitment). By the assumption of at least one open ended availability window, we are guaranteed to find at least one suitable participant. We allocate the service invocation to that participant, and iterate until all of the service invocations in the workflow have been allocated to the community.

C. Execution

When a participant is allocated a service invocation, it adds a commitment to its schedule that contains all the necessary information to execute the service. The participant is free to roam, but is responsible for meeting its commitments. Thus the execution phase of an open workflow proceeds in a fully decentralized, distributed manner. To meet a commitment, the participant must (1) acquire the required inputs for the service from the executor of the preceding tasks, (2) be at the required location for executing the service, and (3) execute the service at the required time. The participant 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 participant's final responsibility is to communicate the service's outputs to any other participants that require them.

IV. System Architecture

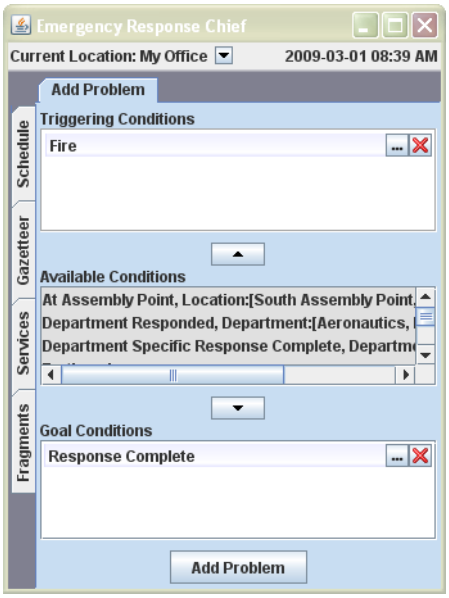
A. An Open Workflow Management System

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 [6].

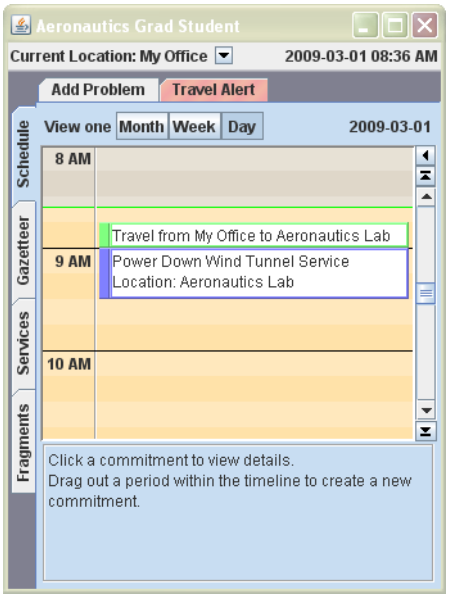
The basic steps in deploying an application using our open workflow management system are (1) installing the program on the users' devices, (2) adding knowhow in the form of workflow fragments, and (3) adding service descriptions. In our implementation, we use XML configuration files to provide the task and service definitions for each device. Once this initial configuration has been completed, any participant 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.

Figure 3 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. Figure 3(a) shows the form that allows the user to create a problem specification by entering information about the triggering conditions and goal. In Figure 3(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



(a) Add Problem tab



(b) Schedule tab

Fig. 3. Screenshots from community members participating in an emergency response open workflow.

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.

B. 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

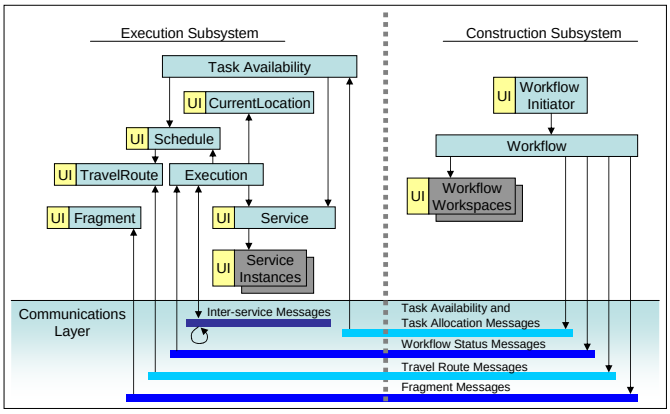


Fig. 4. System architecture.

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 Figure 4. We first observe that for a particular open workflow problem, one host acts as the initiator while all hosts (including the initiator) may act as participants. 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.

a) Construction Subsystem.: The Workflow Initiator is responsible for interacting with the user to define the trigger conditions and goal conditions for the 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 participants and listens to confirm that each allocation is properly committed ("Workflow Status Messages").

b) 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 participant'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. (We direct the Workflow Manager's queries to the Execution Manager rather than the Schedule Manager because we plan in future work to expose further status information beyond just whether a service invocation has been allocated.) The Service Manager maintains the list of services exposed by this host and 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, Schedule Manager, etc.) 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 two ways from the architecture presented in [1]. The primary 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 [3], with the Task Availability Manager which implements the approach described in this paper. The other change was to encapsulate location and travel time information management within the Current Location Manager and the Travel Route Manager. These refinements were achieved without significant impact to other subsystems due to the modularity and flexibility of our architecture overall.

V. Evaluation

We use a combination of simulation and empirical evaluation to test our system and demonstrate the viability of

the open workflow paradigm. We focus on characterizing the performance of the system in terms of three variables that have the greatest impact on the scalability of our architecture: the number of participants in the community, the number of tasks known to the entire community, and the difficulty of the problem being solved which we characterize by the size of the resulting workflow.

Our experimental set up is as follows. Given the number of hosts, the global number of tasks, and the 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 some host.

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 the graph is strongly connected. From this single supergraph we can then draw a large number of guaranteed-satisfiable specifications by randomly picking triggering and goal conditions. We use only disjunctive task nodes in order to maintain the guarantee of satisfiability during our automated evaluations. 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 $\frac{1}{n}$th of the entire supergraph, so the hosts must cooperate to solve the posed problem. For each test run, the test driver randomly choses a path of the desired length through the supergraph, and the initial and final label nodes of the path are used as the specification for that test run. In all of the figures below, the results for each path length are the average of one hundred runs.

For the simulations, all the hosts were run within in a single JVM and communicate solely through a simulated network. The simulations were run on a Windows XP workstation with a 2.8 GHz Intel Xeon processor and 2.75 GB of memory, running the Java 1.6.0_16 HotSpot Client VM.

In Figure 5, we show the average time for each path length from a supergraph with 25 task nodes as the number of participating hosts varies from 2 to 16. 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 were to broadcast requests rather than using pairwise communication, the processing of responses by the initiating host would still require time linear in the number of hosts in the community.

In Figure 6, we show the average time for each path length for 2 participating hosts as the number of task nodes in the supergraph varies from 25 to 100. As we increase the length of the solution paths, we see the average time will be dominated by the time taken to search through the service supergraph to construct the workflow. The rate of increase grows with the number of task nodes because the Workflow Manager 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 path lengths greater than 10 in the small 25 task supergraph.

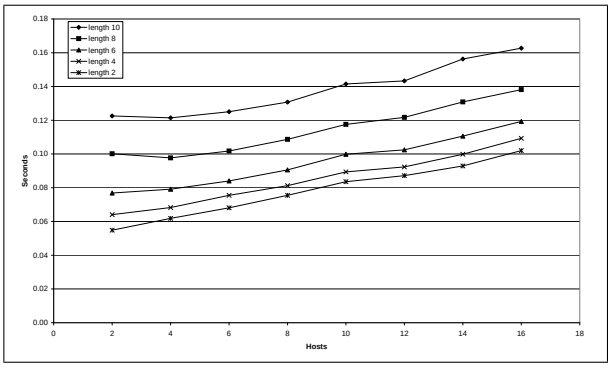


Fig. 5. Simulation of 25 task nodes partitioned across different numbers of hosts.

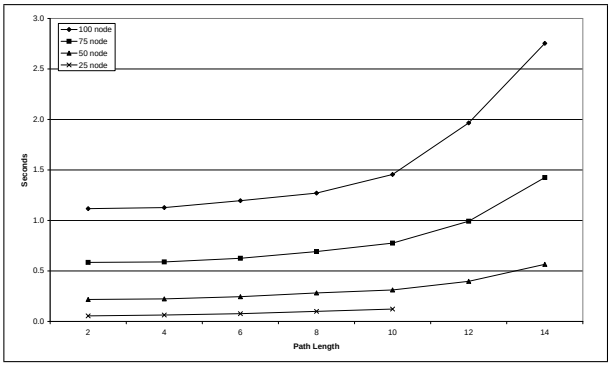


Fig. 6. Simulation of different numbers of task nodes partitioned across 2 hosts.

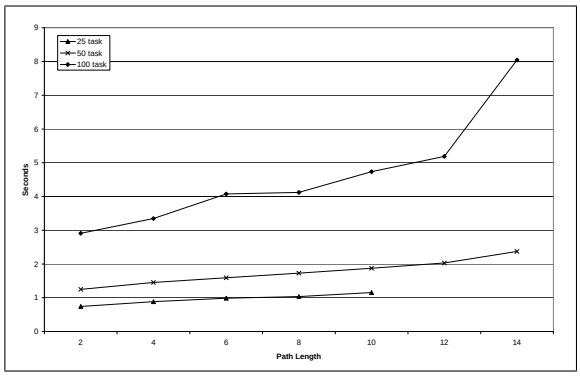


Fig. 7. Empirical performance of ad hoc wireless networking for different numbers of task nodes partitioned across 4 hosts.

After the simulations, we performed empirical evaluation of our application using four Fit-2PCs connected by an ad hoc wireless network using 802.llg (54Mbit/s). The hosts were all running Ubuntu Linux 8.04 with a 1.6 GHz Intel Atom processor and 1 GB of 533 MHz DDR2 memory. All hosts were running the Java 1.6.0_16 HotSpot Client VM. Connectivity among the hosts was verified before the measurements were started. The timing results for workflow graphs with 25, 50, and 100 task nodes are shown Figure 7.

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 fifty tasks to explore, and a solution path length of fourteen, our system finds and allocates a solution in about two and a half seconds on average.

A. Directions for Future Work

These encouraging results demonstrate that our system is ready to be evaluated against real-world problems. In order to accomplish this, we will seek a community to serve as a source of realistic benchmarks. We expect to face new issues when adapting our system to the rigors and challenges posed by our sample community.

One such concern for future research is the representation specifications. Weakening our initial assumption that a specification only involves the inset and outset would allow specifications that include constraints on all aspects of the workflow graph, such as path length, task preferences, and external temporal and spatial constraints. Furthermore, the specification can be expanded to influence the allocation and execution phases. A specification, for example, could minimize the set of participants or restrict the locations of certain tasks. As the the sophistication of our formalism increases, more advanced planning techniques need to be brought into play. Solving problems with hundreds of tasks and large workflow path lengths will require further optimizations to our basic iterative deepening depth first search that take further advantage of the inherent structure of valid workflow graphs.

The handling of errors, community dynamics, and changes in the environment by the open workflow paradigm is another area for future research. For example, the departure of a participant or a other 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 could be constructed. A failure during execution should result in a revised or repaired workflow, which requires reconstruction, reallocation, and compensating execution. 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 participants to accomplish these activities in a mobile ad hoc setting is a focus for 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 participants 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.

VI. Related Work

In this research, we have focused on overcoming the challenges of bringing workflows to transient communities connected by mobile ad hoc networks. Standard workflow management systems, such as ActiveBPEL [7], Oracle Workflow Engine [8], JBoss [9], and BizTalk [10], are designed to work in fully wired environments, such as corporate LANs or across the Internet. Reliance on centralized control and reliable communication mean such solutions cannot successfully operate under the constraints of dynamic mobile environments.

Several workflow systems have been developed which extend the realms in which workflows may operate. The work on federating separate execution engines running independent workflows by Omicini, et al., [11] removes the requirement of centralized control. Chafle, et al., [12], investigate decentralized orchestration of a single workflow by partitioning the workflow at build time and using message passing at run time. Both approaches still assume reliable communication and a fixed group of participants. MoCA [13] uses proxies for distributed control and has some design features that support mobile environments while Exotica/FDMC [14] describes a scheme to handle disconnected mobile hosts. In AWA/PDA [15], the authors adopt a mobile agent based approach based on the GRASSHOPPER agent system. WORKPAD [16] is designed to meet the challenges of collaboration in a peer-to-peer MANET involving multiple human users, however WORKPAD retains the requirement that at least one member of the MANET be connected with a central coordinating entity that orchestrates the workflow and shoulders any heavy computational loads. Sliver [2] brings a full BPEL execution engine to a single cell phone, however that phone still acts as the sole coordinator. Finally, CiAN [3] presents a workflow management system which eliminates the need for a central arbiter by distributing not only service execution but also the task allocation problem across multiple hosts.

While our system builds upon CiAN's model of distributed workflow allocation and execution, all these systems assume that a thoughtfully designed and fully specified workflow already exists. Open workflow is designed for settings where the availability of resources and the range of responses demanded by changing circumstances cannot be anticipated. The workflow to be executed must be generated on the fly to match the present situation.

The automatic composition of services has been explored using a variety of AI planing engines, including Golog [17], Workflow Prolog [18], and PDDL [19]. A review of further automated service composition methods may be found in [20]. Ponnekanti and Fox create workflows by rule-based chaining in SWORD [21], and discuss situations in which the resulting workflows may not produce the desired results due to the preconditions and postconditions of each task not being sufficiently specified. Fantechi and Najm [22] present an approach for ensuring correct service composition by using a more

detailed formal specification of the service behavior. While the initial open workflow construction algorithm we present is a simplified alternative to the powerful techniques presented in these papers, it also addresses a new problem specific to the mobile ad hoc environment. All these systems assume that the knowledge base from which to build the workflow already exists. 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 knowhow and capabilities available within the community.

VII. Conclusions

In this research, we have introduced the open workflow paradigm. We began in [1] by presenting the first algorithm for building a workflow on the fly from available contextual knowledge and constructing a platform for further experimentation with that approach. In this article we have presented important extensions to this work: an enhanced formalism for describing workflow construction that supports parameterized tasks for better context sensitivity and extended construction and allocation algorithms to capture the dynamics of service availability in an ad hoc community. These proposed advances have been implemented and evaluated in our open workflow platform.

Taken together, the novel open workflow paradigm explored by this research enables the development of new classes of applications that are designed to exploit community knowledge in solving real world problems that arise unexpectedly and can be addressed only through the coordinated exploitation of capabilities distributed among the members of the community. The open workflow paradigm presents significant new challenges for the middleware, MANET, workflow, planning, and human-computer interaction research communities. The work presented here represents only the first steps toward characterizing and addressing these concerns.

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 participants to construct and execute responses to needs identified by the participants. 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.

Acknowledgments. This paper is based upon work supported in part by the National Science Foundation (NSF) under grant No. IIS-0534699. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] L. Thomas, J. Wilson, G.-C. Roman, and C. Gill, "Achieving coordination through dynamic construction of open workflows," in *(to appear) Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware 2009)*, Urbana Champaign, IL, USA, November 30, 2009.

[2] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman, "Sliver: A BPEL workflow process execution engine for mobile devices," in *LNCS*, vol. 4294, 2006, pp. 503–508. [Online]. Available: http://dx.doi.org/10.1007/11948148_47

[3] R. Sen, G.-C. Roman, and C. D. Gill, "CiAN: A workflow engine for MANETs," in *COORDINATION*, ser. Lecture Notes in Computer Science, D. Lea and G. Zavattaro, Eds., vol. 5052. Springer, 2008, pp. 280–295.

[4] R. Handorean, C. D. Gill, and G.-C. Roman, "Accommodating transient connectivity in ad hoc and mobile settings," in *Pervasive*, ser. Lecture Notes in Computer Science, A. Ferscha and F. Mattern, Eds., vol. 3001. Springer, 2004, pp. 305–322.

[5] C. E. Perkins and E. M. Belding-Royer, "Ad-hoc on-demand distance vector routing," in *WMCSA*. IEEE Computer Society, 1999, pp. 90–100.

[6] Mobilab Group, "Open workflow project web site," http://mobilab.wustl.edu/projects/openworkflow/.

[7] Active-Endpoints, "ActiveBPEL engine," http://www.active-endpoints.com/active-bpel-engine-overview.htm.

[8] Oracle Inc., "Oracle workflow," http://www.oracle.com/technology/products/integration/workflow/workflow_fov.html.

[9] JBoss Labs, "JBoss application server," http://www.jboss.com/docs/index.

[10] Microsoft Corp., "The BizTalk server," http://www.microsoft.com/biztalk/.

[11] A. Omicini, A. Ricci, and N. Zaghini, "Distributed workflow upon linkable coordination artifacts," in *COORDINATION*, ser. Lecture Notes in Computer Science, P. Ciancarini and H. Wiklicky, Eds., vol. 4038. Springer, 2006, pp. 228–246.

[12] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda, "Decentralized orchestration of composite web services," in *Proc. of the 13th Intl. WWW Conference*, 2004, pp. 134–143.

[13] V. Sacramento, M. Endler, H. K. Rubinsztejn, L. D. S. Lima, K. Gonçalves, and G. A. Bueno, "An architecture supporting the development of collaborative applications for mobile users," in *Proc. of WETICE '04*, 2004, pp. 109–114.

[14] G. Alonso, R. Gunthor, M. Kamath, D. Agrawal, A. E. Abbadi, and C. Mohan, "Exotica/FDMC: A workflow management system for mobile and disconnected clients," *Parallel and Distributed Databases*, vol. 4, no. 3, 1996.

[15] H. Stormer and K. Knorr, "PDA- and agent-based execution of workflow tasks," in *Proceedings of Informatik 2001*, 2001, pp. 968–973.

[16] M. Mecella, M. Angelaccio, A. Krek, T. Catarci, B. Buttarazzi, and S. Dustdar, "WORKPAD: an adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios," *Collaborative Technologies and Systems, International Symposium on*, vol. 0, pp. 173–180, 2006.

[17] S. McIlraith and T. C. Son, "Adapting golog for composition of semantic web services," in *Proceedings of the 8th International Conference on Knowledge Representation and Reasoning(KR2002)*, 2002, pp. 482–493.

[18] S. Gregory and M. Paschali, "A prolog-based language for workflow programming," in *COORDINATION*, ser. Lecture Notes in Computer Science, A. L. Murphy and J. Vitek, Eds., vol. 4467. Springer, 2007, pp. 56–75.

[19] D. McDermott, "Estimated-regression planning for interactions with web services," in *Proceedings of the 6th International Conference on AI Planning and Scheduling*. AAAI Press, 2002, pp. 204–211.

[20] J. Rao and X. Su, "A survey of automated web service composition methods," in *In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004*. Springer-Verlag, 2004, pp. 43–54.

[21] S. R. Ponnekanti and A. Fox, "SWORD: A developer toolkit for web service composition," in *Proceedings of the 11th World Wide Web Conference*, Honolulu, Hawaii, USA, May 2002. [Online]. Available: http://www2002.org/CDROM/alternate/786/

[22] A. Fantechi and E. Najm, "Session types for orchestration charts," in *COORDINATION*, ser. Lecture Notes in Computer Science, D. Lea and G. Zavattaro, Eds., vol. 5052. Springer, 2008, pp. 117–134.