Automated Situation-Aware Service Composition in Service-Oriented Computing

S. S. Yau, Arizona State University, USA

H. Davulcu, Arizona State University, USA

S. Mukhopadhyay, Utah State University, USA

D. Huang, Arizona State University, USA

H. Gong, Arizona State University, USA

P. Singh, Arizona State University, USA

F. Gelgi, Arizona State University, USA

ABSTRACT

150 words or less

Keywords: *α-calculus; α-logic; agent synthesis; automated situation-aware service composition; service-based systems; situation-aware workflow*

INTRODUCTION

Service-oriented architecture (SOA) (W3C, 2004b) has the major advantage of enabling rapid composition of distributed applications, regardless of the programming languages and platforms used in developing and running the applications. SOA has been adopted in many distributed systems like Grid and Global Information Grid (GIG) (U.S. Department of Defense, 2002), in various application domains, such as collaborative research and development, e-business, health care, environmental control,

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

military applications, and homeland security. The systems based on SOA are considered as *service-based systems* (SBS). In SBS, capabilities are provided by various organizations as services, which are software/hardware entities with well-defined interfaces to provide certain capability over wired or wireless networks using standard protocols, such as HTTP and Simple Object Access Protocol (SOAP). For a user to effectively utilize available services in an SBS to achieve the user's goal, it is necessary to provide the capability for the user to *compose* appropriate services into higher-level functionality. Such higher-level functionality is considered as a *workflow*, which is a series of cooperating and coordinated activities. Since a large-scale SBS usually consists of thousands of services, it is desirable that the SBS can allow users to declaratively specify their goals and automate the service composition based on the specified goals.

Control flow graph (Georgakopoulos et al., 1995; Hsu, 1993; WFMC, 1996) is a common type of framework for depicting the local execution dependencies among the key activities in a service composition. It is a good way to visualize the overall flow of control among the milestones that a workflow should satisfy. Control flow graphs are the primary specification means in most commercial implementations of workflow management systems. A typical graph specifies the initial and final activities in a workflow, the successor-activities for each activity in the graph, and the execution of all the successors concurrently or only a selected successor.

On the other hand, dependencies among service invocations in a workflow are often based on situation changes in highly dynamic environments, where users often have different requirements in various situations or services cannot provide desirable QoS due to attacks, system failures or overload. Such dependencies can be effectively captured by *situational constraints*, which are the restrictions on what and how services should be invoked in various situations. A *situation* is a set of contexts in a system over a period of time that affects future system behavior for specific applications, and a *context* is any instantaneous, detectable, and relevant property of the environment, the system, or the users (Yau, Wang, & Karim, 2002; Yau et al., 2002). It is necessary to specify situational constraints in a modular and reusable fashion so that a service composition system can discover appropriate services based on situations and users' goals, compose services into a control flow, and coordinate their execution adaptively based on situation changes. We consider such a process as *situation-aware service composition*.

So far, the most widely used specification language for Web services, WSDL (WSDL, 2001), is not expressive enough to specify these situational constraints. Other frameworks for modeling and executing workflows in Web service based systems, such as BPEL4WS (Andrews et al., 2003) and OWL-S (W3C, 2004a), are not expressive enough to model services with side effects (i.e., services can change states of themselves or other services), and do not provide any facilities for automated service composition.

In this article, we will present an approach to automated situation-aware service composition which is based on our α-logic, α-calculus, and declarative model for situation awareness (SAW) (Yau et al., 2005a; Yau et al., 2005b). Our approach will include an efficient and constructive agent synthesis algorithm based on α-logic proof theory that can automate the synthesis of executable agents which satisfy the control flow and situational constraint specifications for situation-aware service composition.

In the following sections, we will summarize the key concepts and features of our SAW model, α-logic and calculus. We will show how to use our SAW model and α-logic to specify situational constraints and control flow graphs. We will present an algorithm for automated situation-aware service composition based on α-logic proof theory. We will also show how situation-aware workflow agents described in α-calculus terms are synthesized from the workflows generated using our service composition algorithm. These agents will be used to monitor and execute the workflows.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

An accident response scenario will be used to illustrate our approach.

RELATED WORK

Our approach to automated situation-aware service composition involves several areas, including planning and Web service composition, Web services and workflow specification languages, and situation awareness. In this section, we will provide a summary of related work in these areas.

Planning and Web Service Composition

Substantial research has been done in the areas of planning and Web service composition. Planning goals can be expressed as temporal formulas in TLPlanner (Bacchus & Kabanza, 1996). TLPlanner then generates plans using a forward chaining engine which generates finite linear sequences of actions. As these linear sequences are generated, the paths are incrementally checked against the temporal goals. The approach used in TLPlanner is sound and complete. However, this approach, in the worst case, performs an exhaustive search of the state space, which is often referred to as the *state-explosion problem*. A technique for semi-automatic service composition in the context of Web services using semantic description of the services was presented in Sirin, Hendler, and Parsia (2003). In this technique, nonfunctional properties are used to reduce the set of services satisfying service composition requirements. In Rao, Kungas, and Matskin (2003), deduction in linear logic is used to compose Web services, but no execution model is provided for the synthesized composition of Web services. In Ponnekanti and Fox (2002), the first order logic is used to declaratively specify Web-service-based systems, and deduction is used to synthesize service compositions from the declarative first order logic specifications. However, this approach cannot deal with SBS, because services often have side effects. In Woodman et al. (2004), composition and verification of Web services are specified using π-calculus (Milner, 1999), but the π-calculus terms from declarative specifications of the clients' requirements cannot be automatically synthesized. Besides, the π-calculus terms representing service compositions, which are manually generated by developers, cannot be reconfigured under changing environments and cannot be used to specify nonfunctional QoS goals, like situation awareness, deadlines, and so forth.

A problem closely related to service composition is the workflow scheduling problem. Existing workflow schedulers are passive. Passive schedulers receive sequences of events from an external source, such as a workflow or a services coordinator, and validate that these sequences satisfy all constraints, possibly after rejecting, suspending, and reordering some events in the sequences. Several such schedulers for workflows which are specified using *event-condition-action rules* (ECA rules) (Dayal, Hsu, & Ladin, 1990), constraint languages, such as *intertask dependencies* (Attie, Singh, Sheth, & Rusinkiewicz, 1993) and *Event Algebra* (Singh, 1995), are described in Singh (1995), Attie et al. (1993), and Gunthor (1993). To validate a particular sequence of events, each of these schedulers takes at least quadratic time of the number of events. However, in these approaches, an unspecified external system is used to do the consistency checking to ensure the liveliness of the scheduling strategy and to select the event sequences for execution. The complexity of the algorithms for these tasks is exponential in the worst case.

In contrast to passive scheduling, our approach is pro-active. In particular, we do not rely on any external system. Instead, we synthesize an explicit control flow graph of all allowed executions, that is, executions that are known to satisfy all situational constraints. This representation can be used to enumerate all allowed executions at linear time per execution path (linear in the size of the path). In this way, at each stage in the execution of a workflow, the scheduler knows all the events that are eligible to start and can initiate their execution. There is no need to validate the specified situational constraints at run time since the constraints are "compiled into" the synthesized control

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

flow structure. Similarly, our approach detects inconsistency in workflow specifications during the static analysis and compilation phase.

Web Service and Workflow Specification Languages

WSDL (WSDL, 2001), BPEL4WS (Andrews et al., 2003), and OWL-S (W3C, 2004a) are the most widely used languages for specifying Web services and business processes composed of Web services. However, WSDL can only model services, and does not provide constructs for modeling processes. While BPEL4WS and OWL-S are suitable for describing, modeling, and executing workflows corresponding to business processes in Web service-based systems, it is difficult to use these languages to describe systems involving sensors and other physical devices with complex dynamic behavior for continuously accessing information, monitoring the environment, and reacting to changes in systems. Both BPEL4WS and OWL-S lack a satisfactory programming model with formal operational semantics, which makes it difficult to understand, reason with, and verify the behavior of systems described in such languages. BPEL4WS and OWL-S are not expressive enough to specify workflows with side effects. Furthermore, they do not provide facilities for automated service composition based on user requirements. Hence, processes in BPEL4WS or service models in OWL-S have to be specified manually by developers.

Situation Awareness

Situation awareness has been studied in artificial intelligence, human-computer interactions, and data fusion community. Existing research on situation awareness may be classified into two categories. One focuses on modeling and reasoning SAW (Matheus, Kokar, & Baclawski, 2003; Matheus et al., 2003; McCarthy, 2001; McCarthy & Hayes, 1969; Pinto, 1994; Plaisted, 2003), and the other on providing toolkit, framework, or middleware for development and runtime support for SAW (Chan & Chuang, 2003; Dey & Abowd, 2001; Ranganathan & Campbell, 2003; Roman et al., 2002; Yau, Wang, & Karim, 2002; Yau et al., 2002; Yau et al., 2005a; Yau et al., 2005b; Yau et al., 2006). In the first category, situation calculus and its extensions (McCarthy, 2001; McCarthy & Hayes, 1969; Pinto, 1994; Plaisted, 2003) were developed for describing and reasoning how actions and other events affect the world. A situation is considered as a complete state of the world, which cannot be fully described and leads to the well-known frame problem and ramification problem (Pinto, 1994). A core SAW ontology (Matheus, Kokar, & Baclawski, 2003; Matheus et al., 2003) refers a situation as a collection of situation objects, including objects, relations, and other situations. However, it does not address how to verify the specification and perform situation analysis. In the second category, Context Toolkit (Dey & Abowd, 2001) provides architectural support for context-aware applications, but it does not provide analysis of complex situations. GAIA (Ranganathan & Campbell, 2003; Roman et al., 2002) provides context service, space repository, security service, and other QoS for managing and interacting with active spaces. MobiPADS (Chan & Chuang, 2003) is a reflective middleware designed to support dynamic adaptation of context-aware services based on which runtime reconfiguration of the application is achieved. Reconfigurable Context-Sensitive Middleware (Yau, Wang, & Karim, 2002; Yau et al., 2002) provides the capabilities of context data acquisition, situation analysis, and situation-aware communication management, and a middleware-based situation-aware application software development framework. Recently, a declarative SAW model for analyzing and specifying SAW requirements and algorithms for generating software agents for situation analysis from SAW requirements have been developed for SBS (Yau et al., 2005a; Yau et al., 2005b; Yau et al., 2006). The declarative SAW model will be used in our approach and discussed in the section entitled Background.

BACKGROUND

As discussed in the Introduction section, our approach to automated situation-aware service composition is based on our α-logic, α-calculus,

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

and declarative model for SAW (Yau et al., 2005a; Yau et al., 2005b). In this section, we will summarize the α-calculus, the α-logic, and our declarative model for SAW.

α-Calculus

Process calculi have been used as programming models for concurrent (May & Shepherd, 1984) and distributed systems (Caromel & Henrio, 2005); α-calculus is based on classical process calculus (Milner, 1999). It provides a formal programming model for SBS, which has well-defined operational semantics involving interactions of external actions and internal computations for assessing the current situation and reacting to it (Yau et al., 2005b). The external actions include communication between processes and logging in and out of groups/domains (Yau et al., 2005b). The internal computations involve invocation of services as well as internal control flow (Yau et al., 2005b).

For the sake of completeness, we summarize part of the syntax of α-calculus in Table 1 which will be used in this article. Similar to classical process calculus, a system in α-calculus can be the parallel composition of two other systems or a recursive or nonrecursive process. A recursive or nonrecursive process can be an inactive process, a nominal identifying a process, a process performing external actions, a process performing internal computations, a service exporting a set of methods, or the parallel composition of two other processes. The methods are described by the preconditions describing the constraints on the inputs accepted by the methods and postconditions describing the constraints on the outputs provided by the methods. Continuation passing (Appel, 1992) is used to provide semantics of asynchronous service invocations. In Table 1, $I{:}l_i(y)\hat{}cont$ denotes the invocation of the method l_i exported by I with parameter y and continuation *cont*. External actions involve input and output actions on named channels with types as in the ambient calculus (Cardelli & Gordon, 2000). Internal computation involves beta reduction, conditional evaluation for logic control, and invocation of public methods exported by a named service or private methods exported by the process itself.

α-Logic

α-logic is a hybrid normal modal logic (Blackburn, deRijke, & Venema, 2003) for specifying SBS (Yau et al., 2005b). The logic has both temporal modalities for expressing situation information as well as modalities for expressing communication, knowledge, and service invocation. It provides atomic formulas for expressing relations among variables and nominals for identifying agents. The α-logic supports developers to declaratively specify situation awareness requirements. Models for the logic are (annotated) processes in the α-calculus. These processes provide constructive interpretations for the logic. Following a Curry-Howard style isomorphism (Sorensen & Urzyczyn, 2006) in which proofs are interpreted as processes, a novel proof system of α-logic can support the synthesis of α-calculus terms from declarative α-logic specifications.

Here, we will only summarize the parts of syntax, semantics, and proof system of α-logic, which will be used in this article, and provide some intuitive explanations to the logic. Table 2 shows the part of the syntax of α-logic.

In Table 2, we assume that every variable x has a type. Intuitively, the nominals act as identifiers to processes. The knowledge formula intuitively states that after a process receives the item named u from another process, the process satisfies φ. The modality $serv(x;u;\sigma;\varphi)$ indicates that a process invoking service σ with parameter x receives u as the result, and then satisfies φ. The formula $<u>\varphi$ describes the behavior of a process after sending out u. The α-logic is a hybrid modal logic in the sense that nominals, which refer to processes, form primitive formulas.

The following modalities, which will be used in this article, can be defined in terms of the primitive connectives and modalities defined in Table 2:

$\Diamond\varphi := E(\boldsymbol{T}\ \mathrm{U}\ \varphi)$	eventually
$\varphi \prec \sigma := \Diamond(\varphi \wedge \Diamond\sigma)$	φ before σ

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Table 1. A partial syntax of α*-calculus*

(System)	
S::=	
fix I=P (recursive or nonrecursive process)	*N ::=*
$S \Vert S$	*x* (name variable)
(parallel composition of two systems)	*n* (name)
(Processes)	
P::=	
	(External actions)
0	*E::=*
(inactive process)	*K* (communication actions)
$P \Vert P$	
(parallel composition)	*K::=*
I	*Ch(x)*
(identifier)	(input)
E.P	*Ch<Str>*
(external action)	(output)
C.P	
(internal computation)	*Ch::=*
$P\{l_1(x_1),\ldots l_k(x_k);\ldots l_n(x_n)\}$	*N* (named channel with type)
(method export)	
$l_1,\ldots l_k$ are private methods that can be invoked by *P* itself only while	
$l_{k+1},\ldots l_n$ are public methods that can be invoked by other processes.	
(Internal computations)	
C::=	
let x=D instantiate P(beta reduction)	$pre::=\sigma[y] \wedge \rho[y]$
if C(x) then P else P'(conditional evaluation)	
ρ	$post::= (\sigma[x] \wedge \rho[x])\ x$
(constraint)	
ε	σ*::=*
(no-computation)	*b*
tt	(base type)
(constant true)	σ→σ
ff	(function type)
(constant false)	
	ρ*::=*
D::=	$x \geq y+c$
$I{:}l_i(y)\hat{}cont$ (method invocation)	$x > y+c$
	$x \leq y+c$
$I{:}l_i::=$ $pre_i::post_i[y]$ (method definition)	$x < y+c$

We do not describe the full semantics of the α-logic in this article, but illustrate only the following salient features with the α-logic formulas interpreted over systems or processes decorated with atomic formulas:

$S \models I$ if *S* is the system *fix I=P*

$S \models {<}u{>}\,\varphi$ if *S* is the system *fix I=P* and there exist processes *Q, R, S,* and *T,* such that $P \equiv {<}u{>}Q$, $R \equiv (x).S, T \equiv P \Vert R$ and $Q \models \varphi$

$S \models pred(u1,\ldots,un)$ if *S* is decorated with *pred(u1,…,un)*

$P \models K(u;\varphi)$ if $P \equiv (x).Q$ and $Q[u/x] \models \varphi$

$P \models serv(x;u;I;\varphi)$ if $P \equiv$ *let* $y=I{:}l_i(x)$ *instantiate* *Q* for some method l_i exported by some process identified by *I* and $Q[u/y] \models \varphi$

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Table 2. A partial syntax of α-logic

Φ1, φ2 ::=	formula	
T	true	
0	inactivity	
U	nominal	
$pred(x_1, \ldots, x_n)$	atomic formula	
x ~ c	atomic constraint	*// ~::=> | <| ≤| ≥, c is a natural number*
φ1 ∨ φ2	disjunction	
┐*φ*	negation	
E(φ1 υ φ2)	until	
E(φ1 s φ2)	since	
φ1|| φ2	parallel composition	
K(u; φ)	knowledge of *u*	
serv(x;u;σ;φ)	invocation of service σ using input *x* by *φ* and returning *u*	
∃t φ	existential quantification on time	
<u> φ	behavior after sending message	
Φ1 ∧ φ2	conjunction	

The part of the proof system of α-logic used in this article consists of a set of axioms along with the following rules:

- Modus Ponens (MP): $\vdash \varphi \wedge (\varphi \rightarrow \psi) \rightarrow \psi$
- Substitution: There are two substitution rules:
  - **Substitution A:** If φ is a valid formula and ψ is a subformula of φ, ψ is an atomic formula, and τ is a formula in the α-logic, then infer φ[τ/ψ].
  - **Substitution B:** If φ and τ are valid formulas and ψ is a subformula of φ, and ¬ψ is not a formula of φ, then infer φ[τ/ψ].
- Generalization: There are two types of generalization rules for modalities and quantifiers respectively.
  - Generalization A: $\vdash \varphi \rightarrow \ \varphi$

    $\vdash \varphi \rightarrow \Theta\varphi$ $\vdash \varphi \rightarrow K(u, \varphi)$ for any *u*, $\vdash \varphi \rightarrow serv(x;u;\sigma;\varphi)$ for any *x, u* and σ
  - Generalization B: $\vdash \varphi \rightarrow \forall x\, \varphi$

The set of axioms includes all axioms of propositional normal modal logic along with the following self-duality axioms:

D1: $serv(x;u;\sigma;\varphi) \rightarrow \neg serv(x;u;\sigma;\neg\varphi)$

D2: $K(u;\ \varphi) \rightarrow \neg K(u;\ \neg\varphi)$

Our Declarative SAW Model

In general, SAW requirements for a service composition include situations related to the goal of the composition and relations among the situations and services in SBS. Developers need to analyze users' SAW requirements to capture the above information and record it precisely.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Formal specifications are unambiguous and concise, but are often difficult to use. Hence, we have developed a declarative model for SAW to facilitate developers in analyzing SAW requirements and rapidly modeling the requirements with the following constructs (Yau et al., 2005a; Yau et al., 2005b; Yau et al., 2006):

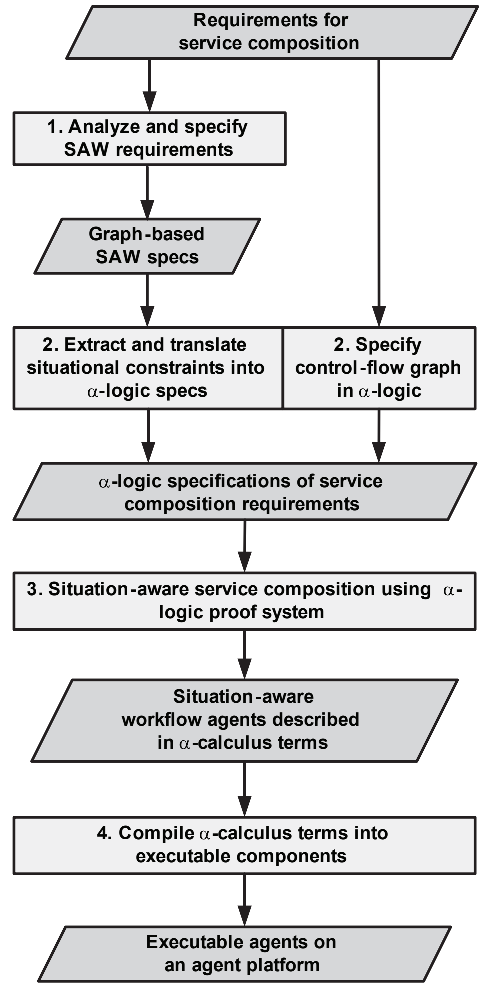
- *Context*. A context has a context name and a context type. A context value domain is defined for each context type. Context operators are defined on context value domains. A context operator is a method of a service which can either return a context value or preprocess context values.
- *Atomic situation*. An atomic situation is a situation defined using context operators and arguments and cannot be decomposed to any other atomic situations. An argument can be a constant, a variable of specific context type, or a context value at a particular time.
- *Composite situation*. A composite situation is a situation recursively composed of atomic situations and other composite situations using logical and/or temporal operators.
- *Relations* among situations and services. Four types of relations are used to describe an action's precondition and effect, and a system's reactive behaviors and knowledge sharing.

Graphical representations for the constructs in our SAW model and a GUI tool based on the graphical representations have been developed to facilitate developers to model SAW requirements visually. Later, we will discuss how the GUI tool is used to automatically extract the situational constraints for service composition from the graphical representations of users' SAW requirements and translate them to α-logic specifications.

OVERVIEW OF OUR APPROACH

Figure 1 depicts our approach to automated situation-aware service composition in SBS.

Figure 1. Our approach to automated agent synthesis for situation-aware service discovery and composition


Requirements for service composition
1. Analyze and specify SAW requirements
Graph-based SAW specs
2. Extract and translate situational constraints into α-logic specs
2. Specify control-flow graph in α-logic
α-logic specifications of service composition requirements
3. Situation-aware service composition using α-logic proof system
Situation-aware workflow agents described in α-calculus terms
4. Compile α-calculus terms into executable components
Executable agents on an agent platform

Our approach consists of the following four major components:

1. Use our declarative model for SAW to analyze and specify SAW requirements.
2. Extract the situational constraints, which are the restrictions on what and how services should be invoked in various situations, from the specified SAW requirements and specify a control flow graph in α-logic

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

as the user's goal for the service composition.

3. Conduct situation-aware service composition using the α-logic proof system and automated synthesis of situation-aware workflow agents described in α-calculus terms for runtime execution.
4. Compile α-calculus terms to executable components on an agent platform.

α-logic and α-calculus provide the logical foundation of our approach. Automated situation-aware service composition is achieved by our approach with the construction of a proof for the user's service composition goal using the α-logic proof system. The constructed proof satisfies the given control flow and situational constraint specifications and corresponds to a workflow that coordinates available services in the SBS to achieve the user's goal. Situation-aware workflow agents described in α-calculus terms are synthesized from the constructed proof to monitor and execute the workflow. We have constructed a compiler for the α-calculus terms to executable agents on an agent platform, the Secure Infrastructure for Networked Systems (SINS) platform (Bharadwaj, 2003), which is used in our demonstration.

The declarative model for SAW facilitates developers to analyze and specify situational constraints of service compositions in a SBS in a hierarchical and graph-based manner. With our declarative model for SAW, developers do not need to learn or understand the complex syntax and semantics of the logical foundation of our approach. Situational constraints for service composition are extracted from the graphical representations and translated to the corresponding α-logic specifications. We will present the four major components of our approach in the following two sections.

SAW REQUIREMENTS FOR SERVICE COMPOSITION

As discussed in the last section, the requirements of a service composition are expressed in a control flow graph and a set of situational constraints. In this section, we will discuss how control flow graphs and situational constraints are specified in α-logic and how situational constraints are extracted from the user's SAW requirements using our graph-based SAW model.

For illustration purposes, let us consider an SBS that connects the 911 call center at the Police Department (*PD*), the Fire Department (*FD*), the hospitals with emergency services, and Ambulance Services (*AMS*) for coordinating various first responders (*PD*, *FD,* and *AMS*) in handling serious traffic accident situations. *PD*, *FD* and *AMS* provide various capabilities as services in the system. The following simplified *accident response scenario* illustrates how an automatically generated workflow coordinates the field rescue operations:

0. A 911 call center receives a report that there is a serious accident at location *L* on a road and notifies nearby police patrol cars, fire stations,and hospitals for emergency services. At least one police patrol car (*CAR*), a fire engine (*FE*), and an ambulance (*AMB*) were sent to *L*.
1. Upon arriving at *L*, the police officers set up a perimeter to secure the accident site and inform the *FE* and *AMB* that the perimeter has been set up and to ask for firefighters and an ambulance.
2. Upon arriving at *L*, the firefighters start to search for passengers involved in the accident and rescue the passengers.
3. Once the passengers are rescued, the paramedics on the *AMB* assess the status of the injured passengers to determine the appropriate medical care for them.
4. After assessing the status of the injured passengers, the *AMB* takes the injured passengers to a nearby hospital.

Figure 2 depicts a control flow graph representing a template of the service composition, which should be satisfied when responding to an accident. The control flow graph depicts the requirements that whenever the 911 call center receives an accident report, a *CAR* will

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

be first sent to the accident site, and then a *FE* and an *AMB* will be sent to the accident site to rescue any injured passengers involved in an accident.

The following situational constraints need to be satisfied when the above workflow in the accident response scenario is executed:

- In a low visibility situation, upon arriving at the accident location, the police officers should set up a perimeter to secure the accident site and then inform the *FE* and *AMB* that the perimeter has been set up.
- If the injured passengers are in critical condition and it will take a relatively long time to the nearest hospital with emergency service, a helicopter *HELI* should be requested to take the critically injured passengers quickly to the hospital.

Specifying Control Flow Graphs

Since α-logic has connectives for service invocation (*serv*), eventuality (◊), parallel composition (||), and disjunction (∨) as well as atomic constraints, it is straightforward to specify control flow graphs using α-logic formulas. For example, the control flow graph in Figure 2 can be specified in α-logic as follows:

(◊serv(;L,T,'accident';U;911CallCenter) ≺ ◊serv(L,T;'car_sent';X;911CallCenter) ≺ (◊serv(L,T;'fe_sent';Y;911CallCenter) || ◊serv(L,T;'amb_sent';Y;911CallCenter)) ≺ ◊serv(L,T;'rescued';Z;911CallCenter))

Specifying Situational Constraints

Similar to the coordination events in Klein (1991), we define the following five types of situational constraints for service composition:

a. **Allowance constraint:** A service can be invoked in a certain situation.
b. **Prohibition constraint:** A service cannot be invoked in a certain situation.
c. **Ordering constraint:** Service invocations should follow a specific order in a certain situation.
d. **Existential constraint:** In a certain situation, the invocation of a service causes another service to be invoked eventually.
e. **Set constraint:** In a certain situation, a set of services should be invoked together if they are concatenated by "∧", or one of them should be invoked if they are concatenated by "∨".

These situational constraints can be specified using α-logic as follows:

a. *Allowance constraint* is represented as

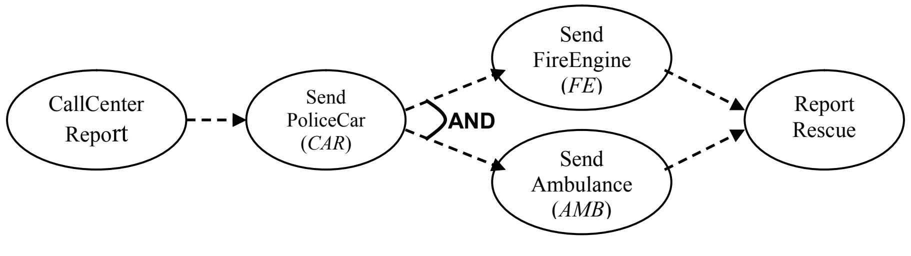
   situation → service invocation,

   where *service invocation* is described by modality *◊serv(Input;Output;Service;Agent)*. It denotes that *service* should eventually be invoked by *Agent* in *situation*.

b. *Prohibition constraint* is represented as

   situation → ¬ service invocation.

Figure 2. The overall goal for the accident response scenario service composition



Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Similar to allowance constraints, it denotes that *service* should not be invoked in *situation*.

c. *Ordering constraint* is represented as

situation $\rightarrow$ *service*$_1$ *invocation* $\prec$ *service*$_2$ *invocation* $\prec$ … $\prec$ *service*$_m$ *invocation*,

where operator "$\prec$" means before. It describes that, in *situation*, *service*$_1$ should be invoked first, followed by *service*$_2$ to be invoked, until *service*$_m$ to be invoked.

d. *Existential constraint* is in the format

situation $\rightarrow$ (*service invocations* $\rightarrow$ *service invocation)*,

where *service invocations* are invocations of multiple services concatenated by conjunction ($\wedge$) and disjunction ($\vee$). It states that in *situation*, *service invocations* cause *service* to be eventually invoked.

e. *Set constraint* is in the format

situation $\rightarrow$ *service invocations*.

For example, our accident response scenario with the requirement, "If the injured passengers are in critical conditions, a helicopter HELI should be requested to bring them quickly to the hospital," can be expressed by the following situational constraint:

serv(L,T;'critical';AMB;911CallCenter) $\rightarrow$ $\Diamond$serv(L,T;'helicopter_sent', HeliID;AMB;911CallCenter),

where *serv(L,T;'critical';AMB;911CallCenter)* represents that the *911CallCenter* agent detects critically injured passengers in location *L* at time *T* by invoking a service provided by *AMB*.

Extracting Situational Constraints from SAW Requirements

The complex syntax and semantics of α-logic make them difficult for developers to specify situational constraints directly in α-logic. In addition, it would be time-consuming for developers to manually specify all situational constraints for service composition since dynamic SBS may consist of many services and its behavior may be affected by various situations. Hence, we have developed an algorithm to automatically extract all situational constraints from the users' SAW requirements. The situational constraint extraction algorithm involves the following two major steps:

Step (1) translate the graphical representations of the users' SAW requirements to corresponding α-logic specifications, and

Step (2) extract the situational constraints from the α-logic specifications obtained in Step (1).

The final output of the algorithm is the α-logic specifications of situational constraints, which will be the input to our α-logic proof system for service composition. Step (1) of the algorithm is quite straightforward following a direct mapping from the constructs in our declarative SAW model to the formulas in α-logic as follows:

- α-logic formula for service specification

  A method of a service is described in α-logic as follows:

  Method(Input;Output;Service;Agent) $\rightarrow$ *serv(Input;Output;Service;Agent)*,

  where *Method(Input;Output;Service;Agent)* states a method signature.

  Modality *serv(Input;Output;Service;Agent)* describes an event indicating that *Agent* invokes the *Method* exported by the *Service* with *Input* tuple as the input and receives *Output* tuple as the result. For

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

example, in the accident response scenario, the fact that 911CallCenter invokes the "get_injury_status" method exported by AMB service to determine the status of injured passengers at location L and time T is stated as follows:

get_injury_status (L,T;Status;AMB;911CallCenter)
$\rightarrow \Diamond$serv(L,T;'critical';AMB;911CallCenter)

get_injury_status (L,T;Status;AMB;911CallCenter)
$\rightarrow \Diamond$serv(L,T;'minor';AMB;911CallCenter)

In the above specifications, different constants are used in the place of *Output*. The first formula indicates that a possible result of invoking *get_injury_status* is *critical*. The second formula shows that another possible result is *minor*. Such specifications are used to specify nondeterministic result of the invocation of a service, which is of particular interest to users of the service. Variables can be used in the specification of an output when there is no need to specify nondeterministic result of a service.

- α-logic formula for situation specification

The following α-logic formula describes how *Situation* is defined:

Definition $\rightarrow \Diamond$*(k([Contexts, Situation], Agent))*,

where *Contexts* are the related contexts, *Agent* is the name of the agent monitoring *Situation*, and *Definition* contains a set of operations for collecting related contexts and analyzing *Situation*.

The above α-logic formula means that *Agent* will eventually have the knowledge about *Situation* if *Agent* performs all the operations specified in *Definition*. *Definition* in this formula is described using α-logic as follows:

1. For the situation operators in our model, $\neg$, $\wedge$ and $\vee$ are provided in α-logic; *P* (sometimes) and *H* (always) are defined using the existential and universal quantifications on time in α-logic; and *Know* is represented using *k(Output; Agent)*.
2. For context acquisition and processing, *serv(Input;Output;Service;Agent)* and atomic constraints are used to state the required service invocations and condition evaluations.

When a specified situation is referred in a situational constraint, we use *Definition* in the specification of the situation so that our service composition system can evaluate the truth value of the situation using the current sequence of service composition.

- α-logic formulas for relation specification

Relations among services and situations in our SAW model are specified using atomic formulas in α-logic. A *precondition* relation is specified as an atomic formula *precondition(S, M)*, which indicates that a situation *S* is the precondition of invoking service *M*. Similarly, an atomic formula *do(M, S)* specifies a *do* relation, which indicates that the invocation of *M* makes *S* true; and an atomic formula *trigger(M, S)* specifies a *trigger* relation, which indicates that the situation *S* should immediately lead to the invocation of *M*. The *precondition* and *do* relations are sufficient for specifying the conditions and possible effects of service invocations. The *trigger* relation is sufficient for specifying the reactive behaviors of an SBS in various situations. The *tell* relation is not used in our service composition because agent communications are automatically synchronized during the service composition process.

By directly mapping the model constructs for services, situations, and relations in our SAW

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

model to the corresponding α-logic formulas, Step (1) of our situational constraint extraction algorithm translates the model representation of SAW requirements for a service composition to the α-logic specifications of these SAW requirements. From these α-logic specifications of SAW requirements, Step (2) of our situational constraint extraction algorithm extracts situation constraints for a service composition. Due to the complexity of this step, we will first briefly describe the extraction of allowance and existential constraints using the following process, and then highlight the main ideas for extracting prohibition, ordering and set constraints (see Box 1).

The extraction of prohibition constraints from SAW requirements is based on the analysis of conflicting situations. If a situation $S_1 \rightarrow \neg S_2$, and *precondition*(S_2, *Method*) are defined, a prohibition constraint $S_1 \rightarrow \neg \Diamond Method$ will be generated.

The extraction of ordering constraints from SAW requirements is based on the analysis of *trigger* relations. If *precondition*(S_0, M_1), *do*(M_1, S_1) and *trigger*(M_2, S_1) are defined, an ordering constraint $S_0 \rightarrow \Diamond(M_1 \prec M_2)$ will be generated.

Finally, a set of constraints will be generated when a situation S_0 triggers multiple service invocations (concatenated by "$\wedge$"), or appears in multiple allowance constraints.

Box 1.

```
0      For each precondition($S_x$, Method)
           Find serv(Input; Output; Service; Agent) for Method from service specification,
           and generate an allowance constraint $S_x \rightarrow \Diamond$serv(Input; Output; Service; Agent)
1        If $S_x$ is an effect of the invocation of a service,
1a         Find do($M_x$, $S_x$) and precondition(Situation, $M_x$) from the relation specifications
1b         Find serv($Input_x$; $Output_x$; $Service_x$; $Agent_x$) for $M_x$ from service specification
           Generate an existential constraint as follows:
1c         Situation $\rightarrow$ (serv($Input_x$; $Output_x$; $Service_x$; $Agent_x$) $\rightarrow \Diamond$serv(Input; Output; Service; Agent)),
           where the left hand side of "$\rightarrow$" is a service invocation for $M_x$, and the right hand side is a
           service invocation for Method
1d         Go back to 0
2        If $S_x$ is an effect of two or more service invocations,
2a         Decompose $S_x$ into a set of atomic situations aSituSet.
2b         For each atomic situation $aSitu_i$ in aSituSet
2bi            Find do($M_i$, $aSitu_i$) and precondition ($S_i$, $M_i$) as described in 1a, put $S_i$ in a set of situa-
               tions situSet
2bii           Find service invocation $serv_i$ for $M_i$ as described in 1b, put $serv_i$ in a set of service invoca-
               tions servSet.
2c         Compose all situations in situSet into Situation and compose all service invocations in servSet
           into service invocations, in the way that is same as atomic situations in aSituSet compose $S_x$
           Generate an existential constraint as follows:
2b         Situation $\rightarrow$ (service invocations $\rightarrow \Diamond$serv(Input; Output; Service; Agent)),
2c         Go back to 0
```

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

AUTOMATED SYNTHESIS OF SITUATION-AWARE WORKFLOW AGENTS

Service Composition with α-Logic

Our service composition approach aims at the synthesis of noniterative services compositions without loops, which means that we cannot automate the synthesis of workflows with repetitive invocations of certain services until a particular condition is satisfied. Fortunately, such behavior can be captured by defining the termination condition as a situation *S*, and the *trigger* relations that invoke certain services when *S* is not true. Hence, the implementation of such behavior can be automatically done using our technique presented in Yau et al. (2006). However, further investigations are needed to properly integrate this technique with our service composition approach presented in this article.

Given an α-Logic formula $\boldsymbol{G}$ describing a control flow graph and a set of situational constraints $\boldsymbol{C}$, we have developed an algorithm for synthesizing a workflow $\boldsymbol{G_C}$, which is described by an α-logic formula and represents the class of executions of $\boldsymbol{G}$ satisfying $\boldsymbol{C}$, that is $\boldsymbol{(G \wedge C)}$. Although $\boldsymbol{(G \wedge C)}$ represents all the executions of $\boldsymbol{G}$ satisfying $\boldsymbol{C}$, it is not an explicit set of instructions to the workflow scheduler for executing the workflow. In order for $\boldsymbol{(G \wedge C)}$ to be executable, it must be a formula corresponding to an explicit workflow control flow specification. Hence, "synthesizing" $\boldsymbol{(G \wedge C)}$ means that we need to obtain a formula $\boldsymbol{G_C}$ which does not contain conjunctions and satisfies $\boldsymbol{G_C \models (G \wedge C)}$. The above service composition problem can be formulated as follows:

Given a control flow graph *G* and a set of situational constraints *C*, construct an executable control flow graph $G_C \models (G \wedge C)$.

Now, we will present our algorithm, called *Enforce*, which takes $\boldsymbol{G}$ and situational constraints $\boldsymbol{C}$ as inputs and produces a conjunction-free control flow specification $\boldsymbol{G_C}$ through a series of transformations. Our algorithm includes a proof procedure based on forward-chaining natural deduction presented in detail in the "Background" section to enforce existential constraints and a procedure to enforce ordering and other constraints. In order to enforce situational constraints at design time, during each step of the forward chaining proof procedure, the left-hand side (or the body) of each situational constraint is evaluated using the current sequence of service composition. If the body of a constraint is evaluated to be true, the right-hand side (or the head) of the constraint is enforced using the Enforce Procedure found in Box 2.

To illustrate the Enforce Procedure, consider the service composition in our accident response scenario discussed before. In order to demonstrate the interactions between the control flow specification and situational constraints, let us consider the following simple control flow stating a sequential composition template made up of invocations of the ambulance service (*AMB*) and the reporting of successful rescue operations to the *911CallCenter*:

(◊serv(L,T;'amb_sent';Y;911CallCenter)≺◊serv(L,T;'rescued';Z;911CallCenter))

with the following situational constraints:

◊serv(L,T;'amb_sent';AMB;U) → ◊serv(L,T;Status;AMB;V)

◊serv(L,T;'critical';AMB;V) → ◊serv(L,T;'helicopter_sent';AMB;V)

The proof procedure uses the two steps of natural deduction with Modus Ponens (MP) (see the "Background" section) to discover the corresponding agents and the services composition plan.

◊ (serv(L,T;'amb_sent';Y;911CallCenter)≺ serv(L,T;'rescued';Z;911CallCenter))

The Enforce Procedure synthesizes the workflow goal and natural deduction enables services discovery shown in Box 3.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Box 2.

Allowance constraint:

$Enforce(\Diamond serv(I;O;M;A), Goal) \equiv Goal \,||\, \Diamond serv(I;O;M;A)$, if *M* is new

Composite constraints:

$Enforce(C1 \wedge C2 ,Goal) \equiv Enforce(C2, Enforce(C1, Goal))$

$Enforce(C1 \vee C2 ,Goal) \equiv Enforce(C1, Goal) \vee Enforce(C2, Goal)$

Ordering constraint:

$$Enforce(\Diamond(serv(I1;O1;M1;A1) \prec serv(I2;O2;M2;A2)), Goal)$$
$$\equiv Synch(Enforce(\Diamond serv(I2;O2;M2;A2), Enforce(\Diamond serv(I1;O1;M1;A1), Goal))$$

where *Synch* replaces every occurrence of $\Diamond serv(I1;O1;M1;A1)$ with $\Diamond serv(I1;O1;M1;A1).<\xi>$ and every occurrence of $\Diamond serv(I2;O2;M2;A2)$ with $k(\xi ;A2).serv(I2;O2;M2;A2)$ in the *Goal*.

Existential constraint:

$Enforce(C1 \rightarrow C2, Goal) \equiv Enforce(C1, Goal) \vdash Enforce(C2, Goal)$

Prohibition constraint:

$Enforce(\neg\Diamond serv(I;O;M;A), Goal) \equiv Enforce(\Diamond serv(I;O;M;A) \rightarrow\bot, Goal)$

Box 3.

$$\Diamond(serv(L,T;\text{'amb\_sent'};AMB;911CallCenter).<\xi> \;||\; k(\xi).serv(L,T;\text{'rescued'};AMB;911CallCenter))$$

$$\begin{tikzcd}
\Diamond(serv(L,T;\text{'amb\_sent'};AMB;911CallCenter).<\xi> \arrow[d] & k(\xi).serv(L,T;\text{'rescued'};AMB;911CallCenter)) \arrow[d] \\
dispatch\_amb(L,T;\text{'amb\_sent'};AMB;911Callcenter) & amb\_send\_hospital(L,T;\text{'rescued'};AMB;911CallCenter)
\end{tikzcd}$$

Hence, the body of the following existence constraint becomes true:

$$\Diamond serv(L,T;\text{'AMB\_sent'},AmbID;AMB;U) \rightarrow \Diamond serv(L,T;Status;AMB;V),$$

which causes an insertion and leads to the following new workflow goal:

$$\Diamond(serv(L,T;\text{'AMB\_sent'},AmbID;Y;911CallCenter).<\xi>.\Diamond serv(L,T;Status;AMB;V) \;||\; k(\xi).\Diamond serv(L,T;\text{'rescued'};Z;911CallCenter))$$

Because *get_injury_status* is a method having nondeterministic results, the natural deduction over the specification of *get_injury_status* branches out to two distinct workflow goals as seen in Box 4.

One of the subgoals satisfies the body of another situational constraint:

$$serv(L,T;\text{'critical'};AMB;V) \rightarrow \Diamond serv(L,T;\text{'helicopter\_sent'};AMB;V)$$

which causes another insertion into the control flow graph on the *critical* branch:

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Box 4.

$$\begin{tikzcd} & \Diamond \text{serv(L,T;Status;AMB;V)} \arrow[dl] \arrow[dr] & \\ \Diamond \textit{serv(L,T;'critical';AMB;V)} & & \Diamond \textit{serv(L,T;'minor';AMB;V)} \end{tikzcd}$$

(◊(serv(L,T;'critical';AMB;V).◊serv(L,T;'heli copter_sent',HeliID;AMB;V)))

Successive applications of natural deduction will complete the control flow graph by finding and inserting all the necessary services and terminates the proof procedure.

Cycle Detection and Simplification

The above transformation may yield a proof tree which may have subformulas, where the *send* and *receive* primitives introduced during the application of the Synch function cause a cyclic wait. The problem with such *cyclic blocks* is that, when they exist, the specification of the derived control flow graph may be inconsistent and hence not executable. Fortunately, a variant of depth- first search procedure on the control flow graph generated from our *Enforce* algorithm can identify all executable cyclic blocks and remove all cyclic blocks in time $O(|G|^3)$. This procedure, called *unblock,* is shown below and yields a cyclic-block-free and executable service composition satisfying the original workflow goal made up from a conjunction of the control flow template and the situational constraints (see Box 5).

Representing Situation-Aware Workflow Agents Using α-Calculus

To enable the automated synthesis of situation-aware workflow agents, we use α-calculus as the programming model for agents to develop a deductive technique to synthesize α-calculus terms defining situation-aware workflow agents from α-logic specification automatically.

Note that situation-aware workflow agents are distributed autonomous software entities and have the following capabilities of supporting situation analysis and service coordination:

C1. Participant service management, including monitoring the status of participant services and invoking appropriate services when needed.

C2. Context acquisition and situation analysis, including collecting contexts from its participant services and analyzing situations

Box 5.

```
procedure unblock(G)
1. for each cycle_i in G do using DFS
2.   for each input action K(ξ_ij;σ) in cycle_i do
     //σ is an arbitrary process
3.      W_ij = remove(W, K(ξ_ij;σ))
4. return ∨W_ij
```

```
procedure remove(W, K(ξ;σ))
1. W' = (W ∧ ¬ K(ξ;σ))
2. S = S(W) – S(W')
 //S(W)={<ξ> | <ξ> is an output action in W}
3. if S ≠ 0 then
4.   for each <ξ> in S do
5.      W' = remove(W', K(ξ;σ))
6. else return W'
```

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

continuously based on its configuration.

C3. Communication among agents, including communication with other agents to exchange context and situation information, service status, requests, and responses for service invocation.

The above capabilities of situation-aware workflow agents can be represented using α-calculus as follows:

C1. Management of participant services by situation-aware workflow agents is modeled by internal computations and external communications in α-calculus. In SBS, collecting information on the status of participant services is done by invoking certain methods provided by participant services. Such service invocation is represented by the method invocation in α-calculus. Similarly, invoking appropriate participant services is also represented by the method invocation in α-calculus. The condition that determines when a service should be invoked is represented by the conditional evaluation and constraint evaluation in α-calculus. For example, assume that we want to represent a ticketing agent, which checks the reservation service of an airline to find out the availability of tickets on the flight XYZ for a particular date and reserves a ticket if there are still tickets left. Such an agent is represented using α-calculus as follows:

Fix Ticketing :=

let integer x = Reservation:checkAvailability(flightNo, date) instantiate
//collecting information from the participant service, "TicketingService"

if x > 0

//conditional evaluation

then Reservation:reserveTicket(flightNo, date, 1) //service invocation

else string ch<"This flight is booked full." >

C2. Similarly, service invocation for context acquisition and processing context information by situation-aware workflow agents is represented by the method invocation in α-calculus. Situation analysis process in situation-aware workflow agents is represented by conditional evaluation, input, and output actions in α-calculus.

C3. Communication among agents can be represented by the input and output actions in α-calculus. One restriction imposed on communication between two agents is that the agents must run in parallel, that is if $Agent_1 = (u).T$, and $Agent_2 = <u>.T$, $Agent_1$ can receive u only when the calculus term $Agent_1 \mid\mid Agent_2$ is satisfied in the system.

In addition, the recursive process, concatenation and parallel composition of processes and conditional evaluation in α-calculus, can be used to represent complex control structures such as loops and conditional branches for more complex behavior of situation-aware workflow agents.

Automated Synthesis of α-Calculus Terms Defining Situation-Aware Workflow Agents

From the proof generated using our algorithm for service composition based on α-logic, we can extract executable α-calculus terms in a straightforward way following the lines of Waldinger (2000). For example, from the proof process of the following workflow goal

$\Diamond(serv(L,T;'AMB\_sent',AmbID;Y;911CallCenter) \prec serv(L,T;'rescued';Z;911CallCenter)),$

which we have shown previously, the following 911CallCenter agent can be synthesized as seen in Box 6.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Box 6.

```
fix 911CallCenter=                                                          (line 1)
  integer ch1(l, t).                                                        (line 2)
    let string r1=AMB:dispatch_amb(l, t)^c_dispatch instantiate             (line 3)

      let string status=AMB:get_injury_status(l, t)^c_get instantiate       (line 4)
        if status=='critical' then                                          (line 5)
          let string r2=AMB:dispatch_heli(l, t)^c_heli instantiate          (line 6)
            let string result=AMB:heli_send_hospital()^c_sendIf instantiate (line 7)
         string o<string result>                                            (line 8)
        else let string result=AMB:amb_send_hospital()^c_sendElse instantiate (line 9)
          string o<string result>                                           (line 10)
```

In these calculus terms, *c_dispatch*, *c_get*, *c_heli*, *c_sendIf*, and *c_sendElse* are identifiers for continuation passing (see the "Background" section) of different service invocations.

Compilation of Synthesized α-Calculus Terms into Executable SINS Agents

The synthesized α-calculus terms encode a workflow generated by the α-logic proof system. To deploy and execute the workflow, the synthesized α-calculus terms need to be compiled to executable components on an agent platform, such as the SINS (Bharadwaj, 2003) and Ajanta (2002). We selected the SINS as the agent platform due to the following reasons:

1. SINS platform comprises SINS Virtual Machines (SVM), which provide various support for agents running on SVMs, such as marshalling and demarshalling data for service invocations, instantiating agents, and secure and reliable group communication among agents.
2. Agents running on SINS are specified using Secure Operation Language (SOL), which is a platform-independent high-level synchronous programming language (Bharadwaj, 2002) and can be verified and compiled to other programming languages, such as Java.

Hence, compiling α-calculus terms to SOL programs is easy because there is no need to handle any platform-dependent low-level details. The generated programs can be migrated to different systems supporting SINS. We have already developed a compiler from α-calculus to SOL.

EVALUATION RESULTS

Complexity Analysis of Our Service Composition Approach

Since both the cycle detection and simplification procedure and the process of synthesizing α-calculus terms from the proof have polynomial time complexity, the complexity of our service composition approach is primarily the complexity of our *Enforce(C, G)* algorithm. The worst-case complexity of our *Enforce(C, G)* algorithm is exponentially increasing with the number of constraints in *C*, which turns out to be inherent in the satisfiability problem since given a concurrent control flow graph *G* and a set of constraints *C*, determining whether $(G \wedge C)$ is executable is NP-complete (Hofstede, Orlowska, & Rajapakse, 1998).

Theorem *(complexity)*: Let *G* be a control flow graph and *C* be a set of situational constraints in the conjunctive normal form. Let $|G|$ denote the number of nodes in *G*, *N* the number of constraints in *C*, and *d* the largest number of disjunctions in a constraint in *C*, then the

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

worst-case complexity of our *Enforce(C, G)* algorithm is $O(d^N \times |G|)$.

Proof: *Based on the description of the Enforce procedure, the algorithm is linear in the size of G, except for the disjunctive case. In the disjunctive case, where* $Enforce(C_1 \vee C_2, G) \equiv Enforce(C_1, G) \vee Enforce(C_2, G)$, *various possible branches in G are created. Since the number of possible branches resulting from the enforcement of a constraint is the same as the number of disjunctions in the constraint, our Enforce(C, G) algorithm runs in* $O(d^N \times |G|)$.

Interactions Between Search Space and Situational Constraints

Our approach to automated situation-aware service composition is based on a deductive proof technique described in the previous section. Our deductive proof system searches the state-space in a backward chaining manner using natural deduction and enforces constraints using the *Enforce* algorithm in a forward chaining manner. Since in real-life scenarios, both the initial information and the goal state information are usually incomplete, backward and forward chaining searches have identical performance (Ghallab, Nau, & Traverso, 2004). Both search methods progress to the next state while preserving the consistency of the situational constraints.

Automated situation-aware service composition can be formulated as a constraint-based planning problem (Carman, Serafini, & Traverso, 2003). In practice, the complexity of plan generation may change, depending on the number of constraints due to constraint interactions. When the number of constraints is small, the constraint-based planning problem reduces to backward chaining search in the almost unconstrained search-space. On the other hand, depending on the applications, a large number of constraints may reduce the size of search-space dramatically so that interleaved steps of natural deduction and constraint enforcer can achieve the solution in fewer steps.

A Case Study with Ordering Constraints

The accident response scenario presented earlier illustrates our service composition approach in a relatively small scale problem. In order to illustrate the exponential decrease of complexity with the increasing number of constraints and services, we opted to present a case study on *Single Machine Total Weighted Tardiness Problem* (SMTWT) using a forward chaining planner (Gelgi, 2005) because the performance of this planner is expected to be close to our proof system and the results in Ghallab et al. (2004). The general characteristics of the SMTWT problem allow us to synthesize multiple instances of constrained service composition problems in large scale with varying number of services and constraints.

In the SMTWT problem, n services have to be processed on a single machine. Associated to each service j is a processing time p_j, a due date d_j and a weight w_j. The services are all available for processing from the start. The tardiness of service j is

$$T_j = max\{0, com_j - d_j\},$$

where com_j is the completion time of service j in the current service sequence, and the total weighted tardiness is given by $\sum w_j T_j$.

The goal of the SMTWT is to find a service sequence which minimizes the sum of the weighted tardiness. We have used the formulation of linear ordering variables (Dyer & Wolsey, 1990) for this scheduling problem.

For the experimental setup, given n (the number of services), u (the interval $(0, u]$), and the number of ordering constraints, we randomly generated the ordering constraints. Processing time for each service was selected randomly in the interval $(0, u]$, whereas weights were selected in the interval $(0, 10u]$[1]. Deadlines were selected randomly in the interval $(0, \sum_j p_j)$.

We randomly generated 40 SMTWT problem instances by varying the following parameters:

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

- **Number of services:** 20, 40, 60, 80, and 100 services.
- **Processing times:** Two intervals *(0, 10]* and *(0, 100]*.
- **Number of constraints:** Each problem with $n^2/3$, $n^2/5$, $n^2/10$ and *0* constraints, where *n* is the number of services.

We allowed a maximum of 15 minutes to run the forward chaining planner to generate the optimum solution for each problem. The problem solving times were given in Table 3 for the problem instances which have the processing times for each service in the intervals *(0, 10]* and *(0, 100]*. Figure 3 illustrates the results graphically. To show the characteristics of the results, we only depicted the results up to one minute in the figures; otherwise, we would not be able to see the variation of most execution times clearly, which is of the main interest. Both sections (a) and (b) of Figure 4 depict similar characteristics of the execution times and the execution times are very close for the problems with the same number of services and constraints. That means the process time interval for each service does not affect the execution time in our formulation. The results in Table 3 and Figure 3 show that the increasing number of constraints leads to faster generation of the solutions for the problems. As expected, *time/constraints* and *time/service* ratios are exponential. Without constraints, it takes almost 30 minutes to solve a problem with only 20 services.

CONCLUSION AND FUTURE WORK

In this article, we have presented an approach to automated situation-aware service composition in SBS. Our approach is based on α-logic and α-calculus and a declarative model for SAW. An accident response scenario was given to illustrate how situational constraints for situation-aware service composition are specified, how the proof system of α-logic is used to compose a workflow using available services, and how to synthesize α-calculus terms representing situation-aware workflow agents that can monitor and execute the composed work-

Table 3. Planner execution times for SMTWT problems with (a) $p_j \in (0, 10]$ and (b) $p_j \in (0, 100]$. Execution times are given in seconds.

Interval(u)=(0, 10]	*Number of services* (*n*)					
		20	40	60	80	100
Number of constraints	$n^2/3$	0.02	0.29	2.00	4.36	11.42
	$n^2/5$	0.04	1.43	5.31	6.65	33.02
	$n^2/10$	0.45	15.76	783.36	-	-
	0	-	-	-	-	-

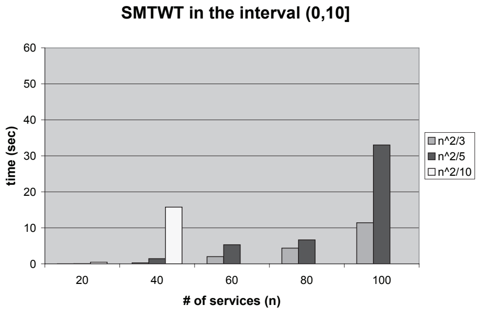
(a)

Interval(u)=(0, 100]	*Number of services* (*n*)					
		20	40	60	80	100
Number of constraints	$n^2/3$	0.02	0.35	2.02	4.48	12.26
	$n^2/5$	0.05	1.52	5.67	7.75	37.24
	$n^2/10$	0.68	19.70	400.40	-	-
	0	-	-	-	-	-

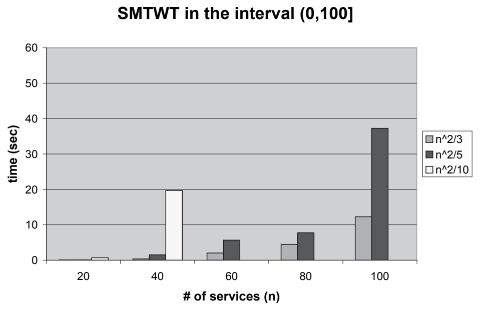
(b)

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 3. The planning times for the SMTWT problems with (a) $p_j \in (0, 10]$ and (b) $p_j \in (0, 100]$. Bars with the same color have the same number of constraints



(a)



(b)

flow. We have also presented the worst-case complexity analysis of our service composition algorithm and the results of a set of preliminary experiments, which used a forward chaining planner to illustrate the exponential decrease of complexity with the increasing number of constraints. To make this approach more useful, future research will focus on developing a dynamic proof system to deal with various situations requiring workflow adaptation and to incorporate security and real-time in the service composition.

ACKNOWLEDGMENT

The work reported here was supported by the DoD/ONR under the Multidisciplinary Research Program of the University Research Initiative under Contract No. N00014-04-1-0723. The authors would like to thank Ramesh Bharadwaj and James Kirby, Jr. of the U.S. Naval Research Laboratory for many helpful discussions. The authors would like to thank anonymous reviewers for many valuable comments for improving this article.

REFERENCES

Ajanta. (2002). Ajanta project Web site. Retrieved September 1, 2007, from http://www.cs.umn.edu/Ajanta/

Andrews, T., et al. (2003). *Business process execution language for Web services* (Version 1.1). Retrieved September 1, 2007, from ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

Appel, A.W. (1992). *Compiling with continuations*. Cambridge University Press.

Attie, P., Singh, M.P., Sheth, A.P., & Rusinkiewicz, M. (1993, August). Specifying and enforcing intertask dependencies. In *Proceedings of the International Conference on Very Large Databases (VLDB)* (pp.134-145).

Bacchus, F., & Kabanza, F. (1996). Using temporal logic to control search in a forward chaining planner. In M. Ghallab & A. Milani (Eds.), *New directions in planning* (pp. 141-153). IOS Press.

Bharadwaj, R. (2002). SOL: A verifiable synchronous language for reactive systems. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'02)*. Retrieved September 1, 2007, from http://chacs.nrl.navy.mil/publications/CHACS/2002/2002bharadwaj-entcs.pdf

Bharadwaj, R. (2003, May). Secure middleware for situation-aware naval C^2 and combat systems. In *Proceedings of the 9th International Workshop on Future Trends of Distributed Computing System (FTDCS'03)* (pp. 233-240).

Blackburn, P., deRijke, M., & Venema, Y. (2003). *Modal logic*. Cambridge University Press.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Cardelli, L., & Gordon, A.D. (2000). Mobile ambients. *Theoretical Computer Science*, *240*(1), 177-213.

Carman, M., Serafini, L., & Traverso, P. (2003, June). Web service compositions as planning. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*.

Caromel, D., & Henrio, L. (2005). *A theory of distributed objects*. Springer Verlag.

Chan, A.T.S., & Chuang, S.N. (2003). MobiPADS: A reflective middleware for context-aware computing. *IEEE Transactions on Software Engineering*, *29*(12), 1072-1085.

Dayal, U., Hsu, M., & Ladin, R. (1990, May). Organizing long running activities with triggers and transactions. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (pp. 204-214).

Dey, A.K., & Abowd, G.D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction (HCI) Journal, 16*(2-4), 97-166.

Dyer, M.E., & Wolsey, L.A. (1990). Formulating the single machine sequencing problem with release dates as a mixed integer program. *Discrete Applied Mathematics, 26*(2-3), 255-270.

Emerson, E.A. (1990). Temporal and modal logic. *Handbook of theoretical computer science (Vol. B): Formal models and semantics* (pp. 995-1072). MIT Press.

Gelgi, F. (2005). *Embedding AI planning techniques into single machine total weighted tardiness problem* (Tech. Rep.). Arizona State University. Retrieved September 1, 2007, from http://www.public.asu. edu/~fgelgi/ ai/atoc/smtwt-report.pdf

Georgakopoulos, et al. (1995). An overview of workflow management: From process modeling to workflow automation infrastructure. *Journal on Distributed and Parallel Databases*, *3*(2), 119-153.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated planning: Theory and practice*. San Francisco, CA: Morgan Kaufmann.

Gunthor, R. (1993, April). Extended transaction processing based on dependency rules. In *International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS'93)* (pp.207-214).

Hofstede, H.M., Orlowska, M.E., & Rajapakse, J. (1998). Verification problems in conceptual workflow specifications. *Data and Knowledge Engineering, 24*(3), 239-256.

Hsu, M. (1993). Special issue on workflow and extended transaction systems. *Bulletin of the IEEE Technical Committee on Data Engineering*, *16*(2).

Klein, J. (1991, February). Advanced rule-driven transaction management. In *Proceedings of IEEE COMPCON 1991* (pp. 562-567).

Matheus, C.J., Kokar, M.M., & Baclawski, K. (2003, July). A core ontology for situation awareness. In *Proceedings of 6th International Conference on Information Fusion* (pp. 545-552).

Matheus, C.J., et al. (2003, October). Constructing ruleML-based domain theories on top of OWL ontologies. In *Proceedings of 2nd International Workshop on Rules and Rule Markup Languages for the Semantic Web* (pp. 81-94).

May, D. & Shepherd, R. (1984). The transputer implementation of occam. In *Proceedings of the International Conference on Fifth, Generation Computer Systems,* Tokyo, Japan.

McCarthy, J. (2001). *Situation calculus with concurrent events and narrative*. Retrieved September 1, 2007, from http://www-formal.stanford.edu/jmc/narrative.html

McCarthy, J., & Hayes, P.J. (1969). Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence, 4,* 463-502.

Milner, R. (1999). *Communicating and mobile systems: The π -calculus*. Cambridge University Press.

Pinto, J.A. (1994). *Temporal reasoning in the situation calculus.* Ph.D. Thesis, University of Toronto, Toronto, Canada.

Plaisted, D. (2003). *A hierarchical situation calculus*. Retrieved September 1, 2007, from http://arxiv.org/abs/cs/0309053

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Ponnekanti, S., & Fox, A. (2002). Sword: A developer toolkit for Web service composition. In *Proceedings of the 11th International World Wide Web Conference (WWW 2002)*.

Ranganathan, A., & Campbell, R.H. (2003, October). A middleware for context-aware agents in ubiquitous computing environments. In *Proceedings of 5th ACM/IFIP/USENIX International Middleware Conference* (pp. 143-161).

Rao, J., Kungas, P., & Matskin, M. (2003, June). Application of linear logic to Web service composition. In *Proceedings of the 1st International Conference on Web Services (ICWS'03)* (pp. 3-9).

Roman, M., et al. (2002). A middleware infrastructure for active spaces. *IEEE Pervasive Computing, 1*(4), 74-83.

Singh, M.P. (1995, September). Semantical considerations on workflows: An algebra for intertask dependencies. In *Proceedings of the 5th International Workshop on Database Programming Languages (DBPL-5)* (pp. 5-19).

Sirin, E., Hendler, J.A., & Parsia, B. (2003, April). Semi-automatic composition of Web services using semantic descriptions. In *Proceedings of the Web Services: Modeling, Architecture and Infrastructure (WSMAI) Workshop in Conjunction with the 5th International Conference on Enterprise Information Systems (ICEIS 2003)* (pp. 17-24).

Sorensen, M.H., & Urzyczyn, P. (2006). *Lectures on the Curry-Howard isomorphism*. Elsevier.

U.S. Department of Defense. (2002). *Global information grid (GIG) overarching policy (U.S. Department of Defense Directive 8100.1)*. Retrieved September 1, 2007, from http://www.dtic.mil/whs/directives/corres/html/ 81001.htm

W3C. (2004a). *OWL Web ontology language: Overview.* Retrieved September 1, 2007, from http://www.w3.org/ TR/owl-features

W3C. (2004b). *Web services architecture*. Retrieved September 1, 2007, from http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

Waldinger, R.J. (2000, April). Web agents cooperating deductively. In *Proceedings of the 1st International Workshop on Formal Approach to Agent-Based Systems* (pp. 250-262).

Web Services Description Language (WSDL). (2001). Retrieved September 1, 2007, from http://www.w3.org/TR/wsdl

Woodman, S.J., et al. (2004, September). Notations for the specification and verification of composite Web services. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC'04)* (pp. 35-46).

WorkFlow Management Coalition (WFMC). (1996). *Terminology and glossary*. Retrieved September 1, 2007, from http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf

Yau, S.S., Wang, Y., & Karim, F. (2002, August). Development of situation-aware application software for ubiquitous computing environments. In *Proceedings of the 26th IEEE International Computer Software and Applications Conference (COMPSAC'02)* (pp. 233-238).

Yau, S.S., et al. (2002). Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing, 1*(3), 33-40.

Yau, S.S., et al. (2005a, July). Situation-awareness for adaptable service coordination in service-based systems. In *Proceedings of the 29th Annual International Computer Software and Application Conference (COMPSAC' 05)* (pp. 107-112).

Yau, S.S., et al. (2005b). *Automated agent synthesis for situation-aware service coordination in service-based systems.* Retrieved September 1, 2007, from http://dpse.eas.asu.edu /as3/papers/ASU-CSE-TR-05-009.pdf

Yau, S.S., et al. (2006, September). Automated agent synthesis for situation awareness in service-based systems. In *Proceedings of the 30th IEEE International Computer Software and Applications Conference (COMPSAC'06)* (pp. 503-510).

ENDNOTE

1 This interval was not selected on purpose since we did not have any concerns on weights. It may be changed according to the concerns.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Stephen S. Yau is currently the director of Information Assurance Center and a professor in the Department of Computer Science and Engineering at Arizona State University, Tempe, Arizona, USA. He served as the chair of the department from 1994 to 2001. He was previously with the University of Florida, Gainesville and Northwestern University, Evanston, Illinois. He served as the president of the Computer Society of the IEEE (Institute of Electrical and Electronics Engineers) and the editor-in-chief of IEEE Computer magazine. His current research is in distributed and service-oriented computing, adaptive middleware, software engineering and trustworthy computing. He received the PhD degree in electrical engineering from the University of Illinois, Urbana,. He is a life fellow of the IEEE and a fellow of American Association for the Advancement of Science.

Hasan Davulcu has been an assistant professor in the Department of Computer Science and Engineering at Arizona State University since 2002. He received a PhD in computer science from the State University of New York at Stony Brook. His research interests include logic-based workflow modeling, Web mining, and information integration.

Supratik Mukhopadhyay has been an assistant professor of computer science at Utah State University University since 2006. He received a PhD degree in computer science at the Max Planck Institute Germany in 2001. During the year 2001-2002, he did postdoctoral research at the University of Pennsylvania, and was an assistant professor in computer science at West Virginia University. His research interests include formal methods for developing dependable systems.

Dazhi Huang is a PhD student in the Department of Computer Science and Engineering at Arizona State University. His research interests include middleware, mobile and ubiquitous computing, and workflow scheduling in service-oriented computing environments. He received his BS in computer science from Tsinghua University in China.

Haishan Gong is a PhD student in the Department of Computer Science and Engineering at Arizona State University. Her research interests include situation-aware software development, and ubiquitous computing. She received her BS in computer science from Zhejiang University, China.

Prabhdeep Singh is a PhD student in the Department of Computer Science and Engineering at Arizona State University. His research interest is in collaborative systems and social networks. He completed his MS in computer science from the State University of New York at Stony Brook in 2003.

Fatih Gelgi is a PhD student in the Department of Computer Science and Engineering at Arizona State University. His research interests include automated ontology extraction in Web mining and stochastic methods in machine learning. He received his BS in computer engineering in 1999 from Middle East Technical University, Turkey.

Copyright © 2007, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.