Specification, Decomposition and Agent Synthesis for Situation-Aware Service-based Systems

S. S. Yau, H. Gong, D. Huang, W. Gao, and L. Zhu
Arizona State University, Tempe, AZ 85287-8809, USA
{yau, haishan.gong, dazhi.huang, w.gao, luping.zhu}@asu.edu

ABSTRACT

Service-based systems are distributed computing systems with the major advantage of enabling rapid composition of distributed applications, such as collaborative research and development, scientific computing, e-business, health care and homeland security, regardless of the programming languages and platforms used in developing and running various components of the applications. In dynamic service-oriented computing environment, situation awareness (SAW) is needed for system monitoring, adaptive service coordination and flexible security policy enforcement. To greatly reduce the development effort of SAW capability in service-based systems and effectively support runtime system adaptation, it is necessary to automate the development of reusable and autonomous software components, called SAW agents, for situation-aware service-based systems. In this paper, a logic-based approach to declaratively specifying SAW requirements, decomposing SAW specifications for efficient distributed situation analysis, and automated synthesis of SAW agents from decomposed specifications is presented. This approach is based on AS^3 *calculus and logic, and our declarative model for SAW. Evaluation results of our approach are also presented.*

Keywords: Service-based systems, situation awareness, decomposition, agent synthesis, AS^3 calculus and logic.

1. INTRODUCTION

Service-Based Systems (SBS) are distributed computing systems with the major advantage of enabling rapid composition of distributed applications, regardless of the programming languages and platforms used in developing and running different components of the applications. SBS have been applied in many areas, such as collaborative research and development, e-business, health care, environmental control, military applications and homeland security (Booth, et al., 2004). In these systems, situation awareness (SAW), which is the capability of being aware of situations and adapting the system's behavior based on situation changes (Yau, et al., 2004; Yau, et al., 2006b), is often needed for system monitoring, adaptive service coordination and flexible security policy enforcement (Yau, et al., 2007). 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, et al., 2002ab).

A large-scale SBS often needs to support various applications simultaneously. These applications often need to share and reuse situation information in the system for providing better QoS. Hence, it is necessary to provide reusable SAW capability in SBS. To greatly reduce the development effort of situation-aware application software in SBS as well as supporting runtime system adaptation, it is necessary to automate the development of reusable and autonomous software components, called *SAW agents*, for performing various tasks in runtime to achieve SAW capability. These tasks include the acquisition of relevant contexts, the analysis of situation changes, and the decision making on triggering proper actions in response to situation changes.

Due to efficiency and dependability considerations, such tasks should not be performed by a centralized SAW agent in a large-scale SBS since SBS often involves a large number of contexts, situations, and services distributed over networks. On the other hand, performing these tasks on distributed SAW agents in a large-scale SBS requires proper coordination of the SAW agents so that the entire system can have a consistent and complete view of situation changes in the system. Communication overhead incurred from such coordination may have significant impact on system performance. Hence, it is necessary to properly distribute the tasks for achieving SAW to distributed SAW agents in SBS. Manually decomposing the situations into subsets and specifying which SAW agent should analyze which subset of situations is time-consuming and error-prone. Furthermore, such a manual process is tedious and very difficult for developers to produce SAW agents with good performance of distributed situation analysis. Hence, it is desirable that the decomposition can be automatically done in such a way that the SAW agents can perform distributed situation analysis efficiently.

In this paper, we will present an approach to logic-based specification, automated decomposition and agent synthesis for situation-aware SBS. Our approach is based on our declarative SAW model (Yau, et al., 2005a), and AS^3 calculus and logic for rapid development of Adaptable Situation-Aware Secure Service-Based (AS^3) systems (Yau, et al., 2007). SAW requirements are analyzed and graphically specified using our SAW model and a Graphic User Interface (GUI) tool, and automatically translated to declarative AS^3 logic specifications. We have developed an algorithm to decompose the generated AS^3 logic specifications to appropriate subsets based on the distribution of context sources, system and network status, as well as the composition relations among situations. For each subset of AS^3 logic specifications, an SAW agent described in AS^3 calculus terms will be automatically synthesized to perform the necessary tasks to meet the corresponding subset of SAW requirements.

2. CURRENT STATE OF THE ART

Substantial research has been done on SAW in artificial intelligence, human-computer interactions and data fusion community. Existing approaches may be divided in two categories: One focuses on modeling and reasoning SAW (McCarthy, et al., 1969; Pinto, 1994; Reiter, 2001; Lausen, et al., 1995; Matheus, et al., 2003; Chen, et al. 2003), and the other on providing toolkit, framework or middleware for development and runtime support for SAW (Yau, et al., 2004; Yau, et al., 2006b; Dey and Abowd, 2001; Roman, et al., 2002; Ranganathan and Campbell, 2003; Chan and Chuang, 2003).

In the first category, Situation Calculus (McCarthy, et al., 1969) and its variants (Pinto, 1994; Reiter, 2001) are used to represent dynamic domains, but the definitions of "situation" used in Situation Calculus and its variants are quite different. McCarthy (McCarthy, et al., 1969) considers a situation as a complete state of the world, while Reiter *et al.* (Reiter, 2001) considers a situation as a state of the world resulting from a finite sequence of actions. McCarthy's definition leads to the Frame problem because a situation cannot be fully described. Reiter's definition makes a situation totally determined by executed actions. GOLOG (Levesque 1997) is a logic programming language, and allows programs to reason about the state of the world and to consider the effects of various possible courses of action before committing to a particular behavior. However, it only works with completely known initial situations. Frame Logic (abbr., F-Logic) (Lausen, et al., 1995) was developed by Kifer et al., and has the modeling capabilities of object-oriented concepts. It can be used for specifying and reasoning SAW requirements. Matheus *et al.* presented a core ontology for SAW (Matheus, et al., 2003) to provide a basis for building situations. A situation here is considered as a collection of situation objects, including objects, relations and other situations. Temporal and spatial relationships of situations can be specified using it. CoBrA Ontology (Chen, et al. 2003) is intended for modeling context knowledge and enabling knowledge sharing in intelligent spaces. It defines a set of vocabularies for describing people, agents, places, etc. in an intelligent meeting room system. However, these ontologies are limited to representing and reasoning SAW requirements.

In the second category, Context Toolkit (Dey and Abowd, 2001) provides a set of ready-to-use context processing components (called widgets) and a distributed infrastructure that hosts the widgets for developing context-aware applications. GAIA (Roman, et al., 2002; Ranganathan and Campbell, 2003), which is a distributed middleware infrastructure provides development and runtime support for context-aware applications in ubiquitous computing environment. It manages the resources and services that are used by applications, provides a component-based application framework for constructing, running or adapting applications. MobiPADS (Chan and Chuang, 2003) is a reflective middleware designed to support dynamic adaptation of context-aware services based on application's runtime reconfiguration. Services are configured and chained together to provide augmented services to mobile applications. RCSM (Yau, et al., 2004, 2006b) provides the capabilities of context acquisition, situation analysis and situation-aware communication management, and a middleware-based situation-aware application software development framework. However, no existing approaches can have automated synthesis of software components for runtime support for SAW in service-oriented computing environment.

3. BACKGROUND

In this section, we will highlight the architecture of our AS[3] systems (Yau, et al., 2007), where SAW agents are used to provide runtime support for context acquisition and situation analysis (Yau, et al., 2005a). We will also summarize the key concepts of our declarative SAW model (Yau, et al., 2005a), and AS[3] calculus and logic (Milner, 1999), which are used in the development of our agent synthesis approach.

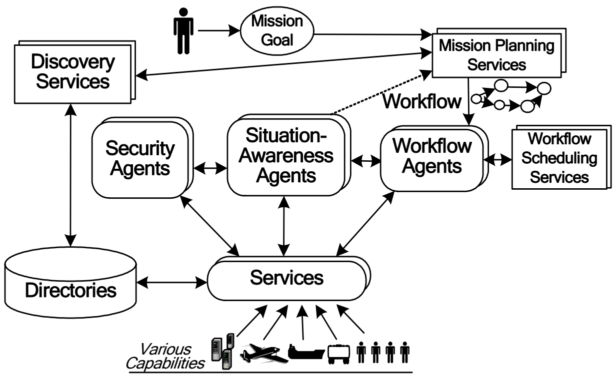

Fig. 1. The architecture of an AS[3] System.

AS[3] systems are collections of services, users, processes and resources, which act to achieve users' goals under dynamic situations without violating their security policies. Fig. 1 shows the architecture of an AS[3] system, in which organizations publish their capabilities as services. Each service provides a set of methods as "actions" in the AS[3] system. *SAW Agents* collect context data periodically, analyze situations based on context data and executed action results, trigger appropriate actions based on situations to provide reactive behavior of the system, and provide situational information to other agents for situation analysis, service coordination, and security policy enforcement. *Security Agents* enforce relevant security policies in a distributed manner based on the current situation. *Mission Planning Service* and *Workflow Scheduling Service* generate and schedule workflows to achieve users' goals based on security policies, situations and available resources. *Workflow Agents* coordinate the execution of workflows based on situational information.

3.1 A Declarative Situation Awareness (SAW) Model

In our declarative SAW model, an ontology is defined for the essential entities for representing SAW and the relations among these entities (Yau, et al., 2005a, 2006b). The advantages of the ontology are that it describes an abstract and application-independent view of SAW, and can be easily shared or extended to model SAW requirements in different application domains. The ontology contains the following entities:

- A *context* has a unique *context name*, a *context type* and a *context value* at a time.
- A *context comparator* is a binary operator returning a Boolean value.
- A *service* has a unique service name, and is on a *host*.
- A *service invocation* is provided by a service, and has a unique method name, accepts inputs as arguments and returns outputs as context values.
- An *argument* can be a constant in the context value domain, or a context variable whose value is obtained through service invocations at runtime.
- An *atomic constraint* is used for comparing two arguments using a context comparator.
- A *situation* can be an atomic situation, a logical composite situation or a temporal situation. The value of a situation is a Boolean value.
- An *atomic situation* is a situation defined using a set of service invocations and an atomic constraint, and cannot be decomposed into any other atomic situations.
- A *situation operator* is a logical operator or a temporal operator.
- A *temporal operator* is either *P* (had been true over a period time in the past), or *H* (was true sometime in the past) defined over a period of time in the past.
- A *logical composite situation* is a situation recursively composed of atomic situations or other logical composite situations or temporal situations using *logical operators*, such as $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation).

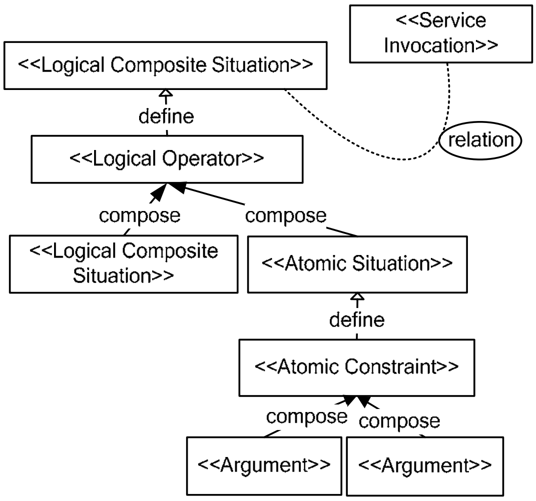

Fig. 2. Partial graphical representation of SAW requirements

- A *temporal situation* is a situation defined by applying a *temporal operator* on a situation over a period of time. The situation used to define a temporal situation can be either an atomic situation or a logical composite situation, which is not composed by any temporal situations.

Three basic relations, *precondition*, *do*, and *trigger*, are defined among situations and service invocations. Relation *precondition* describes a situation is a *precondition* of a service invocation. Relation *do* describes the effect of a service invocation. Relation *trigger* represents a reactive behavior of the system. In SBS, we assume that there are services available for monitoring and providing context values. Hence, contexts can be done through service invocations.

Based on our SAW model, developers can analyze the SAW requirements of an application as follows:

i) Based on the functionality of the application required by users and the specifications of the services available in SBS, developers identify the services to be used in the application.

ii) Developers identify the contexts and all the methods (service invocations) provided by the services found in (i), as well as constants and context comparators used in the application.

iii) Following the basic relations in our SAW model, developers identify the situations relevant to the service invocations identified in (ii), and identify the relations among these situations and the service invocations.

iv) Developers extract atomic situations from the situations identified in (iii) if the identified situations contain any situation operators.

v) Developers construct atomic situations using the service invocations, contexts, constants, and context comparators identified in (ii).

Our SAW model is language-independent and can be translated to specifications of various formal languages, such as F-Logic and AS^3 logic. To facilitate the specification of SAW requirements, we have developed a graphical representation for the constructs in our SAW model and implemented them in a GUI tool. Fig. 2 illustrates partial graphical representation of the constructs in our SAW model. Boxes represent the entities in the model. The type of an entity is quoted by “<<” and “>>”. A solid line with a solid arrowhead from one entity to another entity represents that the starting entity is used by or composes the terminating entity. A solid line with a non-solid arrowhead represents that its starting entity is used to define the terminating entity. A dotted line with a relation encircled by an ellipse is used to connect a situation and a service invocation with the relation between them. The attributes associated with entities, such as context types and termination conditions of situation analysis, are, however, not represented in Fig. 2. These attributes are required for synthesizing SAW agents.

3.2 AS^3 Calculus and Logic

Process calculi have been used as programming models for concurrent (May and Shepherd, 1984) and distributed systems (Caromel and Henrio, 2005). AS^3 calculus (Yau, et al., 2006a; Yau, et al., 2005b) is based on classical process calculus (Appel, 1992). 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 (Milner, 1999). The external actions include communication among processes, logging in and out of groups/domains. The internal computations involve invocation of services as well as internal control flow.

For the sake of completeness, we summarize part of the syntax of AS^3 calculus in Table 1 which will be used in this paper. Similar to classical process calculus, a system in AS^3 calculus can be the parallel composition of two other systems, or a recursive or non-recursive process. A recursive or non-recursive 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 defined by the preconditions describing the constraints on the inputs accepted by the methods and post-conditions describing the constraints on the outputs provided by the methods. Continuation passing (Cardelli and Gordon, 2000) is used to provide semantics of asynchronous service invocations. In Table 1, $I{:}l_i(y)\text{\textasciicircum}cont$ 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 (Huth and Ryan, 2004). 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.

Table 1
Part of the syntax of AS^3 calculus

P::=	**//Processes**	E::=	**//External actions**
zero	(inactive process)	*ch*(*x*)	(input from a named channel)
P *par* P	(parallel composition of processes)	*ch*<*x*>	(output to a named channel)
I($x_1, \ldots, x_n$)	(process identifier with parameters)	C::=	**//Internal computations**
E.P	(external action)	let *x*=D instantiate P	(beta reduction)
C.P	(internal computation)	if *exp* then P else P'	(conditional evaluation)
P_1 *plus* P_2	(nondeterministic choice)		
time t.P	(timeout)	D::= $I{:}l_i(y)\text{\textasciicircum}cont$	(method invocation)

Table 2
Part of the syntax of AS^3 logic

φ_1, φ_2 ::=	formula	E(φ1 *U* φ2)	until
T	true	E(φ1 *S* φ2)	since
U	nominal	k(u; φ)	knowledge of u
pred($x_1,\ldots,x_n$)	atomic formula	serv(x;u;σ;φ)	invocation of service σ using input x by φ and returning u
x ~ c	atomic constraint // ~::= > | < | ≤ | ≥ | =, c is a natural number	∃t φ	existential quantification on time
		<u> φ	behavior after sending message
φ1 ∨ φ2	disjunction	φ1 ∧ φ2	conjunction
┐φ	negation		

AS^3 logic (Yau, et al., 2005b, 2006a) is a hybrid normal modal logic (Blackburn, et al., 2003) for specifying SBS (Milner, 1999). 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 AS^3 logic supports developers to declaratively specify situation awareness requirements. Models for the logic are processes in the AS^3 calculus. These processes provide constructive interpretations for the logic. Following a Curry-Howard style isomorphism (Sorensen and Urzyczyn, 2006), in which proofs are interpreted as processes, a novel proof system of AS^3 logic can support the synthesis of AS^3 calculus terms from declarative AS^3 logic specifications.

Here, we will only summarize the parts of syntax of AS^3 logic, which will be used in this paper, and provide some intuitive explanations to the logic. Table 2 shows the part of the syntax of AS^3 logic.

In the above table, 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;σ;φ)* indicates that a process invoking service σ with parameter *x* receives *u* as the result, and then satisfies φ. The formula <*u*>φ describes the behavior of a process after sending out *u*. The AS^3 logic is a hybrid modal logic in the sense that nominals, which refer to processes, form primitive formulas (Blackburn, et al., 2003).

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

- Eventually: diam(φ) := E(T *U* φ)
- Universal quantification on time: ∀t φ := ¬∃t¬φ

4. OVERVIEW OF OUR APPROACH

As mentioned before, the tasks for achieving SAW capability in a SBS include relevant context acquisition, distributed situation analysis and triggering proper actions in response to situation changes at runtime. To develop SAW capability in SBS, the following major issues need to be addressed:

- *Specify SAW requirements*. SAW requirements from users are described in natural languages and cannot be processed algorithmically. Such descriptions are normally ambiguous. Hence, developers need to have effective tools to support generation of precise specification of the SAW requirements, which is machine processable.
- *Decompose the specifications*. Specifications need to be properly decomposed to distribute the tasks to distributed SAW agents so that they can efficiently achieve SAW capability.
- *Synthesize SAW agents*. To greatly reduce the development effort and support runtime system adaptation, SAW agents need to be automatically synthesized.

In this section, we will present an overview of our approach to logic-based specification, automated decomposition and agent synthesis for situation-aware SBS.

4.1 Architecture of Our Approach

The architecture of our approach is depicted in Fig. 3. The development of SAW capability in SBS consists of the three steps described in the three boxes in the middle of the figure, each with a set of techniques identified in the dashed boxes on the left-hand side. The parallelograms and the dotted-line box on the right-hand side contain the outputs of these steps.

Step 1) *Specifying SAW requirements.* SAW requirements are first represented graphically using our GUI tool, then translated to formal specifications in AS^3 logic. Using the GUI tool, developers can easily generate AS^3 logic specifications without any knowledge of the AS^3 logic. We assume that the consistency and redundancy of the specifications have been checked by developers or some automated tools.

Step 2) *Decomposing SAW specifications.* Given consistent and concise SAW specifications, situations need to be decomposed into multiple subsets, each of which is assigned to an SAW agent for collecting contexts, analyzing the situations and triggering system's reactive behavior under these situations. In this step, situations are grouped to subsets by a decomposition algorithm based on a set of inputs and two decomposition factors. The inputs are SAW requirement specifications and domain knowledge specifications with network topology and the communication bandwidth between each pair of hosts in the system (see Sec. 6). The decomposition of situations ensures that the communication cost among the SAW agents for analyzing these situations can be greatly reduced, and SAW agents can be easily re-synthesized when SAW requirements are reconfigured at runtime.

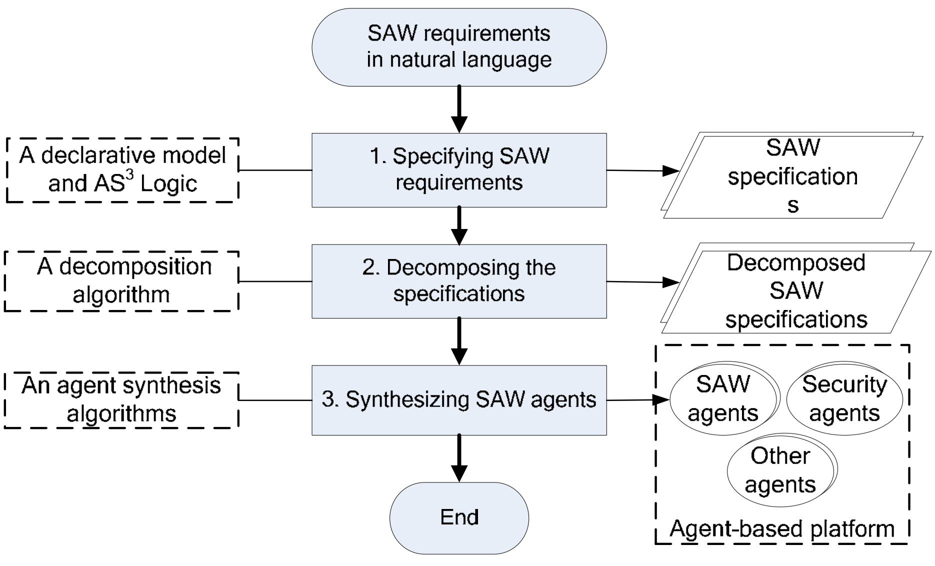

Fig. 3. Architecture of our approach

Step 3) *Synthesizing SAW agents.* From the decomposed situations and related specifications, SAW agents are automatically synthesized with AS^3 calculus terms using our agent synthesis algorithm (see Sec. 7). The synthesized SAW agents will be compiled into executable codes using an existing compiler. The executable SAW agents will run on a distributed agent execution platform, e.g. the Secure Infrastructure for Networked Systems (SINS) (Bharadwaj, 2003), to provide SAW capability for SBS. These SAW agents can also work with other agents, such as security agents for flexible security policy enforcement and workflow agents for adaptive workflow coordination.

In our approach, a system special service was developed to facilitate the analysis of temporal situations by SAW agents. The system special service has the following four methods:

- *appendHistory(SituName, SituData, Timestamp)* stores situational information and removes outdated data.
- *chkSituP(SituName,* ω, ε*)* checks whether the situation was true sometime within [*CurrentTime*-ω, *CurrentTime*-$\omega+\varepsilon$], where *CurrentTime* is the present time, ω is an offset from *CurrentTime*, and ε is the length of the time period to be checked.
- *chkSituH(SituName,* ω, ε*)* checks whether the situation was always true within [*CurrentTime*-ω, *CurrentTime*-$\omega+\varepsilon$].
- *retrieveRelatedData(SituName,* ω, ε, *Type)* retrieves related data of the situation.

At runtime, the contexts and situational information of temporal situations and the situations used to define temporal situations will be periodically retrieved from and recorded in the system special service by invoking aforementioned four methods.

4.2 An Illustrative Example

Consider a SBS, which has access to a set of services, including a rescue center, rescue ships, helicopters and medical ships, for various sea rescue operations. The following "sea rescue" example is presented to show how situational information is used for coordinating execution of a service-based system, and to illustrate our approach:

1) The rescue center (*rc*) receives an SOS message from a ship (*bs*) indicating that *bs* has an accident and some passengers are seriously injured.
2) Upon detecting such a situation, *rc* is responsible for locating proper services to rescue the injured passengers.
3) If there are injured passengers in a 'critial' status, and the weather is safe for a helicopter to perform rescue operation, and *bs* is within a helicopter's flight range, *rc* will notify a helicopter *heli* (by triggering *dispatch_heli* method) to pick up the injured passengers and take them to a nearby hospital.
4) Otherwise, *rc* will notify a nearby medical ship *mShip* to go to *bs* to provide emergency medical treatment for injured passengers. Also, *heli* will return to its base if it is on the way to *bs*.

In this example, a precondition of *dispatch_heli* action is that wind velocity near *bs* has been lower than 1000 feet per minute for 15 time units. Developers can analyze the SAW requirements using our SAW requirement analysis steps identified in Sec. 3.1. Due to limited space, we only illustrate the analysis of partial SAW requirements in this example as follows:

i) Identify the following services used in the application: *rc*, *bs*, *heli*, and *mShip*.
ii) In order to invoke *dispatch_heli* method provided by *heli* service, the following contexts, constants and context comparator should be considered:
    a. Contexts: location of *bs*, wind velocity near *bs*, and passenger injury status (collected by invoking *get_injuryStatus* method of *rc* service).
    b. Constants: 15, 1000, and 'critial'
    c. Context comparator: = and <
iii) Method *dispatch_heli* should be *triggered* by *rc* under a situation (called *readyToDispatchHeli* situation), which means that there are passengers in critical status (called *criticalInjuryFound* situation), and that *heli* is able to perform the rescue operation on *bs* (called *canUseHeli* situation). Situation *canUseHeli* is true when *bs* is within *heli*'s flight range (called *withinRange* situation) and wind velocity near *bs* has been lower than 1000 feet per minute (called *lowWindVelocity* situation) for over 15 time units (called *lowWindVelocityForAWhile* situation).
iv) Extract atomic situations *criticalInjuryFound*, *withinRange,* and *lowWindVelocity* from the situations identified in (iii).

v) Construct the atomic situation *criticalInjuryFound* using *get_injuryStatus* method of *rc* service, context of injury status, constant 'critical' and context comparator '<'.

5. SPECIFYING SAW REQUIREMENTS

After requirement analysis, developers can construct the graphical representations of these SAW requirements, and generate AS^3 logic specifications from the graphical representations using our GUI tool without any knowledge of AS^3 logic (see Sec. 3.2). The generation of AS^3 logic specifications for SAW requirements can be easily done following a mapping between our model constructs and AS^3 logic formulas shown in Table 3. In the following, we will discuss various specifications of SBS. Fig. 4 shows partial SAW specifications in the "sea rescue" example.

Table 3
Specifying SAW requirements in AS^3 logic

Specification	Syntax
Service invocation	$m(a; b; \sigma) \rightarrow serv(x; u; \sigma)$
Atomic situation	$serv(x_1; u_1; \sigma_1), \ldots, serv(x_n; u_n; \sigma_n), arg_1\ op_c\ arg_2$ $\rightarrow diam(k([u_1, \ldots, u_n], s, monitor\_until(f, cond)))$
Logical composite situation	$k([u_1, \ldots, u_k], s_1) \wedge k([u_{k+1}, \ldots, u_n], s_2) \mid k([u_1, \ldots, u_k], s_1) \vee k([u_{k+1}, \ldots, u_n], s_2) \mid$ $\neg k([u_1, \ldots, u_k], s_1)$ $\rightarrow diam(k([u_1, \ldots, u_n], s, monitor\_until(f, cond)))$
Temporal situation	$\forall$ Time currentTime-$\omega \leq$Time$\leq$currentTime-$\omega+\varepsilon$, s' | $\exists$ Time currentTime-$\omega \leq$Time$\leq$currentTime-$\omega+\varepsilon$, s' $\rightarrow diam(k([x_1, \ldots, x_n], s, monitor\_until(f, cond)))$
Relation among situations and service invocations	trigger(m, s) precondition(m, s) $do(m, s_1, s_2)$

- ***Specifying Services***

A method *m* of service σ with input *a* and output *b* is denoted by method signature $m(a; b; \sigma)$, and the invocation of service σ with input *x* returns *u* as output is denoted by the modality $serv(x; u; \sigma)$, where *a, b, x* and *u* are typed variables. In particular, *a* and *b* are of platform-specific data types, while *x* and *u* are of platform-independent context type. Hence, a service specification provides a mapping between high-level platform-independent service implementation_to low-level platform-specific service implementation. For example, the following specification describes a method of service *rc* for collecting a context of "*injStat*" type:

get_injuryStatus([int(ALoc)]; [string(IStatus)]; rc)→ serv([loc(ALoc)]; [injuryStatus(IStatus)]; rc)

In the above service specification, the variables *ALoc* and *IStatus* used in the modality *serv* are typed using context types *loc* and *injuryStatus*, whereas the same variables used in the method signature of *get_injuryStatus* are typed using the data types *int* and *string*. This allows developers to map the context types, which are platform-independent and only are used for high-level reasoning on SAW, to the actual data types supported by the low-level execution platform.

- ***Specifying Atomic Situations***

In atomic situation specifications, each atomic situation *s* consists of a set of service invocations $serv(x_1; u_1; \sigma_1), \ldots, serv(x_n; u_n; \sigma_n)$ for collecting context values $u_1, \ldots u_n$ and an atomic constraint $arg_1\ op_c\ arg_2$ for comparing arguments arg_1 and arg_2 using context comparator op_c. Argument arg_1 is always a context variable, whose value is one of $u_1, \ldots u_n$. Argument arg_2 can either be one of $u_1, \ldots, u_n$ or be a constant in the context value domain. The atomic constraint determines the value of situation *s*. Attribute *f* denotes that situation *s* should be analyzed every *f* time units. Attribute *cond* is the termination condition of *s*. It means that whenever *cond* becomes true, stop analyzing *s*. For example, an atomic situation *criticalInjuryFound* with the meaning of "an injured passenger is in critical status" should be analyzed every 10 time units until the situation *rescueSuccess* becomes true. AS^3 logic specification for this atomic situation is below:

serv([loc(ALoc)];[injuryStatus(IStatus)]; rc), *IStatus = 'critical'*
→ diam(k([loc(ALoc), injuryStatus(IStatus)], criticalInjuryFound, monitor_until(10,rescueSuccess)))

	/ service specifications */*
SERV1)	get_windVelocity([int(ALoc), int(Time)]; [int(Vel)]; rc) → serv([loc(ALoc), int(Time)]; [windVel(Vel)]; rc)
SERV2)	withinFlightRange([int(ALoc)]; [bool(Result)]; heli) → serv([loc(ALoc)]; [bool(Result)]; heli)
SERV3)	backToBase([ ]; [ ]; heli; SAW_heliAgent) → serv([ ]; [ ]; heli)
SERV4)	detect_accident([ ]; [int(ALoc)]; rc) → serv([ ]; [loc(ALoc)]; rc)
	/ atomic situation specifications */*
AS1)	serv([loc(ALoc), int(Time)]; [windVel(Vel)]; rc) ∧ Vel < 1000 → diam(k([loc(ALoc), windVel(Vel)], lowWindVelocity, monitor_until(10, rescueSuccess)))
AS2)	serv([loc(ALoc)]; [bool(Result)]; heli) ∧ Result = true → diam(k([], withinRange, monitor_until(50, rescueSuccess)))
AS3)	serv([]; [loc(ALoc)]), ALoc >0 → diam(k([loc(ALoc)], accident_detected, monitor_until(50, rescueSuccess))) */* temporal situation specifications*/*
TS)	∀Time CurrentTime-15 < Time < CurrentTime ∧ k([loc(ALoc), windVel(Vel)], lowWindVelocity) → diam(k([loc(ALoc), windVel(Vel)], lowWindForAWhile, monitor_until(10, rescueSuccess)))
CS1)	*/* logical composite situation specifications*/* k([loc(ALoc), windVel(Vel)], lowWindForAWhile]) ∧ k([], withinRange)) → diam(k([int(ALoc), windVel(Vel)], canUseHeli, monitor_until(10, rescueSuccess)))
CS2)	k([loc(ALoc), windVel(Vel)], canUseHeli) ∧ k([loc(ALoc), injuryStatus(IStatus)], criticalInjuryFound) → diam(k([loc(ALoc), windVel(Vel), injuryStatus(IStatus)], readyToDispatchHeli, monitor_until(10, rescueSuccess)))
RB1)	*/* reactive behavior specifications */* trigger(k([int(ALoc), windVel(Vel)], not(canUseHeli)), serv([ ];[ ]; heli)

Fig. 4. Partial SAW specifications in the example

In the above specification, the modality *serv(loc(ALoc);injuryStatus(IStatus);rc)* corresponds to a service invocation *get_injuryStatus*, which returns the injury status of a passenger in the accident, given the accident location *ALoc*. Atomic constraint *IStatus = 'critical'* is used for comparing a context variable *IStatus* with a constant *'critical'* using context comparator *'='*.

- ***Specifying Temporal Situations***

In AS^3 logic, temporal operators *P* (sometimes in the past) and *H* (had been true over a period of time in the past) are defined using ∃ (existential) and ∀ (universal) quantifications over a time range. The time range is defined as [$CurrentTime\text{-}\omega$, $CurrentTime\text{-}\omega+\varepsilon$], where ω is an offset from the present time *CurrentTime*, and ε is the length of the time period to be checked. For example, a temporal situation *lowWindForAWhile* with the meaning of "wind velocity in the accident location has always been low in the past 15 time units" is specified as follows:

∀Time CurrentTime-15 ≤ Time ≤ CurrentTime, k([int(Time), windVel(Vel)], lowWindVelocity)
→ diam(k([windVel(Vel)], lowWindForAWhile, monitor_until(10, rescueSuccess)))

A temporal situation cannot be used to define another temporal situation because the conflict or overlap of two time ranges can make the defined situation meaningless.

- ***Specifying Logical Composite Situations***

In logical composite situation specifications, each logical composite situation *s* is composed by atomic situations, temporal situations and/or other logical composite situations using logical operators $\wedge$, $\vee$, and/or $\neg$. For example, a logical composite situation *canUseHeli* with the meaning of "a helicopter can be used only when the accident location is within its reachable range and there has been low wind velocity in the accident location for a while" should be analyzed every 10 time units until the situation *rescueSuccess* becomes true. AS[3] logic specification for this situation is given below:

k([windVel(Vel)], lowWindForAWhile) $\wedge$ *k([], withinRange)*
$\rightarrow$ *diam(k([loc(ALoc), windVel(Vel)], canUseHeli, monitor_until(10, rescueSuccess)))*

- ***Specifying Relations Among Situations and Service Invocations***

The "trigger" relation in our SAW model represents the reactive behavior of the system. Specification of a trigger relation in AS[3] logic is a simple formula in the format *trigger(m, s)*, where method *m* is triggered when situation *s* is true. Similarly, "*precondition*" relation is represented as *precondition(m, s)*, where situation *s* is the precondition of method *m*. "do" relation is represented as "*do(m,* s_1*,* s_2*)*", which means that invoking *m* under situation s_1 will cause situation s_2 becomes true.

6. AUTOMATED DECOMPOSITION OF SAW SPECIFICATIONS

The analysis of a situation can be done by a single SAW agent or multiple SAW agents distributed on multiple hosts collaboratively. A host *h* is considered the *sink point* of a situation *s* if the final value of *s* is calculated on *h*. Due to various system size and network bandwidth among hosts, different selections of *sink points* for situations in SBS will have different impacts on the performance of situation analysis. Furthermore, reconfiguration of SAW requirements in runtime will require re-synthesis of affected SAW agents. In particular, changes in the specification of a situation *s* are most likely affect the situations used to define *s* or the situations defined using *s*. Hence, to reduce the effort of re-synthesizing SAW agents, it is desirable to let an SAW agent process as many related situations as possible. Hence, the purpose of our automated decomposition of SAW specifications is to determine the appropriate *sink point* for each situation and group the related situations together for SAW agents to perform situation analysis efficiently.

6.1 Domain Knowledge and Decision Factors for Automated Decomposition of Situations

Decomposition requires domain knowledge of network topology and communication bandwidth between each pair of hosts in the system. The network topology specification describes which service is on which host. In AS[3] logic, network topology and communication bandwidth are specified as follows:

- *serviceHost(s, h)*: Service *s* is deployed on host *h*.
- *bw(*h_1*,* h_2*, b)*: The bandwidth between host h_1 and host h_2 is *b*. When $h_1 = h_2$, $b = \infty$.

Generally, domain knowledge specification is provided by domain experts. Based on the SAW requirement specifications and domain knowledge specification, the decomposition of our approach depends on the following two factors:

Factor 1) Communication cost

The communication cost for analyzing situation *s* when host h_k is selected as the sink point is denoted as *cost(s,* h_k*)*, which is given by

$$cost(s,h_k)=\begin{cases} 0,\ H=\{h_k\} \\ \sum_{i=1,\ i\neq k}^{n}(n_x+n_y)\times\frac{1}{bw(h_k,h_i)},\ n>1,\ s\notin TS,\ s \not\rtimes TS \\ \frac{1}{fr\times bw(h_k,h_{sys})}+\sum_{i=1,\ i\neq k}^{n}(n_x+n_y)\times\frac{1}{bw(h_k,h_i)},\ n>1,\ s\notin TS,\ s \not\rtimes TS \\ \frac{1}{bw(h_k,h_{sys})}+\sum_{i=1,i\neq k}^{n}n_y\times\frac{1}{bw(h_k,h_i)},\ n>1,\ s\in TS \end{cases}$$

where *H* denotes a set of unique hosts related to situation *s* by providing either the contexts or situational information for analyzing *s*, or the service invocations which should be triggered under *s* for reactive behaviors of the system; *TS* denotes a set of temporal situations for the system; h_{sys} is the host, where the system special service locates; $s \triangleright TS$ denotes that *s* is used to define a temporal situation; and $s \not\triangleright TS$ denotes that *s* is not used to define any temporal situation. *cost(s, h_k)* is calculated in the following four cases:

1) *H* contains only one element, which is h_k. In this case, situation *s* will be assigned to h_k with no choice. Hence, *cost(s, h_k) = 0*.

2) $s \notin TS$ and $s \not\triangleright TS$. In this case, situation *s* is not a temporal situation and not used to define any temporal situation. If *s* is an atomic situation, then n_x is the number of interactions between h_k and h_i for collecting context values for *s* from h_i. If *s* is a logical composite situation, then n_x is the number of interactions between h_k and h_i for collecting situational information for *s* from h_i. Regardless of the type of *s*, n_y is the number of interactions between h_k and h_i for triggering service invocations, which are provided by services on h_i.

3) $s \notin TS$ and $s \triangleright TS$. In this case, situation *s* is not a temporal situation, but is used to define a temporal situation. The communication cost for analyzing *s* is calculated in the same way as 2). In addition, the communication cost for recording the information of *s* in the system special service is $\frac{1}{fr \times bw(h_k, h_{sys})}$, where *fr* denotes the frequency of analyzing *s*, and h_{sys} is the host where the system special service locates.

4) $s \in TS$. In this case, situation *s* is a temporal situation. The communication cost has two parts: a) $\frac{1}{bw(h_k, h_{sys})}$, the communication cost for retrieving situational information from the system special service, b) $\sum_{i=1, i \neq k}^{n} n_y \times \frac{1}{bw(h_k, h_i)}$, the communication cost for triggering service invocations for *s*,.

It is obvious that the final selection of *sink point* for situation *s* should be the host that requires the minimum communication cost, compared to all other related hosts.

Factor 2) Situation composition tree

A *situation composition tree* is a tree that reflects the composition relation of a set of situations used in defining another situation. Leaf nodes correspond to atomic situations. The edge between a parent node and its child node represents the definition or composition relation. For a logical composite situation cs_i, its child nodes are the situations used to compose cs_i. For a temporal situation ts_i, its child node is the situation used to define ts_i. Every situation belongs to a situation composition tree. If the situation is the root of the tree, it means that the situation is not used to define any other situation. Otherwise, the situation is used to define other situations. Situations on the same tree are more likely to be affected by the SAW requirement reconfiguration in runtime. Hence, situations on the same tree should be grouped together as much as possible, in order to minimize the effort of re-synthesizing SAW agents.

6.2 Decomposition algorithm

The decomposition of specified situations is conducted in the following two steps: 1) determine the *sink point* for each situation, and 2) decompose the situations with the same *sink point* to subsets based on their situation composition trees. Situation composition trees can be easily constructed based on situation definitions. Our decomposition algorithm is shown as follows:

***Decomposition* algorithm:**

Require: a list of situations *sList*, a list of hosts *hList*, a list of situation composition trees *treeList*, SAW specifications and network topology specifications, the system special service is provided by $host_{sys}$

Initialize a list *L* = {}

for each situation s_i in *sList* **do**

Initialize a list $hostList_i$ = {} for s_i

4: **if** s_i is an atomic situation defined by a set of service invocations Ω **then**
5: **for** each service invocation ν_i in Ω_i **do**
6: Get h_p of a service σ_i, such that *serviceHost*(σ_i, h_p) and σ_i provides ν_i, put h_p into $hostList_i$, and record the number num_p of h_p in $hostList_i$
7: Get h_p' of a service *x*, which provides external context values for ν_i , and x $\notin \Omega_i$, put h_p' into $hostList_i$ and record the number num_p of h_p' in $hostList_i$
8: **end for**
9: **else if** s_i is a logical composite situation defined by a set of situation *S* **then**
10: **for** each situation s_x in *S* **do**
11: Get *host*(s', h_p) from *L* for s' and put h_p into $hostList_i$
12: **end for**
13: **else if** s_i is a temporal situation defined by a situation s' **then**
14: Get *host*(s', h_p) from *L* for s' and put h_p into $hostList_i$
15: **end if**
16: Find all *trigger*(s_i, *a*) and *host*(*a*, h_k), and insert h_k into $hostList_i$, and record the number num_k of h_k in $hostList_i$
17: **for** each unique h_p in $hostList_i$ **do**
18: Calculate the cost for analyzing s_i and triggering service invocations under s_i,
19: **if** s_i is not a temporal situation and is used to define a temporal situation **then**
20: Calculate the cost for recording s_i in the system special service on $host_{sys}$
21: **else if** s_i is a temporal situation
22: Calculate the cost for retrieving value of s' from the system special service on $host_{sys}$
23: **end if**
24 Calculate the total cost *cost*(s_i, h_i) for s_i if the saw agent for s_i is deployed on h_p
25: **end for**
26: Get sink point h_i for s_i, such that $cost(s_i, h_i) = min(cost(s_i, h_t))$, $h_t \in HostList_i$. If there are multiple hosts can ensure the minimum cost, choose the one that has fewer situations assigned
27: Insert a formula *sinkPoint*(s_i, h_i) into *L*
28: **end for**
29: **for** each unique host h_k in *L* do
30: Initialize a set of empty lists $AgentList_k$, each empty list agent_{ki} in $AgentList_k$ corresponding to a situation composition tree $tree_i$ in *treeList*
31: **end for**
32: **for** each situation *s* in *L*, where *sinkPoint*(*s*, h_k) **do**
33: **if** $s \in tree_1 \cap \ldots \cap tree_n$ **then**
34: Find agent_{ki}, where $1 \le i \le n$, $agent_{ki} \in AgentList_k$, and agent_{ki} contains the minimum situations
35: Insert *s* into $agent_{ki}$
36: **end if**
37: **end for**

In this algorithm, determining the *sink points* for situations is done in Lines 1-28 and decomposing situations with the same sink point is done in Lines 29-37.

Now, let us use the "*sea rescue*" example to illustrate this algorithm. Suppose service *rc* is provided by host $host_{rc}$, and service *heli* is provided by host $host_{heli}$. The bandwidth between $host_{rc}$ and $host_{heli}$, *bw*($host_{rc}$, $host_{heli}$) is assumed to be 30 megabits per second.

First, we initialize an empty list *L* (Line 1 of ***Decomposition*** algorithm). For atomic situation *lowWindVelocity*, initialize an empty host list $hostList_{lwv}$ (Line 3 of ***Decomposition*** algorithm). From the specifications in Fig. 4, we know that situation *lowWindVelocity* is determined by comparing context value of *Vel* and a constant 1000. Value *of Vel* is returned by method *get_windVelocity* of service *rc* on host $host_{rc}$. Hence, we insert $host_{rc}$ into $hostList_{lwv}$, and initialize the count of $host_{rc}$ in $hostList_{lwv}$ to be 1 (Line 6). Because *ALoc* is an external variable used by method *get_windVelocity*, and *ALoc* is provided by service *rc* on $host_{rc}$, we increase the count of $host_{rc}$ in $hostList_{lwv}$ to 2 (Line 7). No service invocation should be triggered under situation *lowWindVelocity.* The *sink point* of situation *lowWindVelocity* is $host_{rc}$ because $hostList_{lwv}$ only contains $hostList_{rc}$. We insert *sinkPoint*(*lowWindVelocity*, $host_{rc}$) in *L* (Lines 18-27).

Similarly, the *sink point* for atomic situation *accidentDetected* is $host_{rc}$, the *sink point* for situation *withinRange* is $host_{heli}$. For temporal situation *lowWindForAWhile*, initialize an empty list $hostList_{faw}$. We get $sinkPoint(lowWindVelocity, host_{rc})$ from *L* and insert $host_{rc}$ into $hostList_{faw}$ (Line 14). Because no service invocation should be triggered under *lowWindForAWhile*, $hostList_{faw}$ only contains $host_{rc}$. Hence, the *sink point* of situation *lowWindForAWhile* is also $host_{rc}$. We insert $sinkPoint(lowWindForAWhile, host_{rc})$ in L. In Fig. 4, logical composite situation *canUseHeli* is composed of *withinRange* and *lowWindForAWhile*. No service invocation should be trigger under *canUseHeli*. Hence, we can have that the host list for *canUseHeli* contains $host_{rc}$ with count of 1, and $host_{heli}$ with count of 1 too. The communication cost for choosing $host_r$ as the *sink point* for situation *canUseHlei* and the communication cost for choosing $host_{heli}$ as the *sink point* for situation *canUseHlei* are calculated as follows:

$cost(canUseHeli, host_{rc}) = 1*1/bw(host_{rc}, host_{heli}) = 1/30 \approx 0.033$

$cost(canUseHeli, host_{heli}) = 1*1/bw(host_{rc}, host_{heli}) = 1/30 \approx 0.033$

Because $host_{rc}$ has more situations than $host_{heli}$, the *sink host* for situation *canUseHeli* is $host_{heli}$ (Line 26). We insert $sinkPoint(canUseHeli, host_{heli})$ in *L*. Choosing *sink point*s for other situations can be done in the same way. Then, we decompose situations with the same *sink point* based on their situation composition trees. In this example, three situations *accidentDetected* (AS3 in Fig. 4), *lowWindVelocity* (AS1 in Fig. 4) and *lowWindForAWhile* (CS1 in Fig. 4) have the same *sink point* $host_{rc}$. Based on their definitions, the two situations *lowWindVelocity* and *lowWindForAWhile* belong to the same situation composition tree, and hence the two situations are grouped together. Therefore, *accidentDected* is analyzed by an SAW agent, and the two situations *lowWindVelocity* and *lowWindForAWhile* are analyzed by another SAW agent.

6.3 Complexity analysis of the decomposition algorithm

To analyze the complexity of our decomposition algorithm, we first give the following two definitions:

The length of an atomic situation (LAS) is the number of service invocations used to collect contexts for analyzing the atomic situation. *The length of a logical composite situation (LLCS)* is the number of situations used to compose the logical composite situation in the logical composite situation's definition.

Theorem 1 (complexity of the decomposition algorithm): Given *p* situations, and *q* services, *w* hosts, the complexity of situation decomposition is $O(p*q+w*k)$, where *k* is the number of situation composition trees in the system.

Proof: Assume that there are *x* atomic situations, *y* logical composite situations, *z* temporal situations, the maximum *LAS* is l_{as}, the maximum *LLCS* is l_{lcs}, and under a situation at most *r* service invocations can be triggered. To find the *sink point* for each atomic situation, at most $q*(l_{as}+r)^2$ steps are needed. To find the *sink point* for each logical composite situation, at most $q*(l_{lcs}+r)^2$ steps are needed. To find the *sink point* for each temporal situation, at most $q*r^2$ steps are needed. Decomposing situations on *w* sink points based on *k* situation composition trees, it takes at most $w*k$ steps. Because l_{as}, l_{lcs} and *r* are usually small numbers, the total complexity is $O(x*q*(l_{as}+r)^2+y*q*(l_{lcs}+r)^2+z*q*r^2+w*k)= O((x+y+z)*q+w*k) = O(p*q+w*k)$.

7. AUTOMATED SYNTHESIS OF SAW AGENTS

7.1 Representing SAW agents using AS^3 calculus

Instead of directly synthesizing SAW agents in platform-dependent programming languages, such as C++, Java and C#, our automated agent synthesis approach first synthesizes the AS^3 calculus terms, which define SAW agents. The main advantage of using AS^3 calculus is to provide us platform-independent models of the agents, which capture the essential processes of context acquisition, situation analysis and reactive behavior triggering. These models can later be used to verify the synthesized agents by a model checker. Platform-specific compilers can be developed to compile AS^3 calculus terms to executable code on different platforms. We have developed a compiler to compile AS^3 calculus terms to agents in Java on SINS platform (Bharadwaj, 2003). Here, we will focus on the synthesis algorithms of SAW agents in AS^3 calculus terms.

Before presenting our SAW agent synthesis algorithms, we first need to examine how SAW agents are defined using AS^3 calculus. Fig. 5 depicts the specifications of the SAW agent, *saw_heliAgent*, in our "sea rescue" example. The *saw_heliAgent* monitors three situations *withinRange* (AS2 in Fig. 4), *canUseHeli* (CS1 in Fig. 4), and *readyToDispatchHeli* (CS2 in Fig. 4). The main process of *saw_heliAgent* is defined by L16-L18 in Fig. 5. L17 instantiates three sub-processes, *withinRange_Agent*, *canUseHeli_Agent* and *readyToDispatchHeli_agent,* in parallel to analyze AS2, CS1 and CS2, respectively. An input action for collecting the information of *accidentDetected* situation is performed in L17 before instantiating *winthinRange_Agent*. L18 recursively executes the *saw_heliAgent*.

```
L1   fix withinRange_Agent(integer ALoc) =
L2     let bool Result = heli:withinRange(integer ALoc) instantiate
L3       if Result = true
L4         then ch withinRange<true>
L5         else ch withinRange<false>.
L6     ( time 50. withinRange_Agent(integer ALoc)
L7      plus ch rescueSuccess(string Status) . zero)
     ||
L8   fix canUseHeli_Agent =
L9     ch lowWindForAWhile(bool S1) par ch withinRange(bool S2).
L10    if S0=true && S1 = true && S2 = true
L11      then ch canUseHeli<integer ALoc, integer Vel, true>
L12      else {ch canUseHeli<integer ALoc, integer Vel, false> . heli:backToBase()}.
L13     { time 10.canUseHeli_Agent(integer ALoc, bool S0)
L14      plus ch rescueSuccess(string Status) . zero }
     ||
L15  fix readyToDispatchHeli_Agent =
       ... …
     ||
L16  fix saw_heliAgent =
L17    {ch accidentDetected(integer ALoc, bool S0) . withinRange_Agent(integer ALoc) } par
     canUseHeli_Agent  par readyToDispatchHeli.
L18    saw_heliAgent
```

Fig. 5. An example SAW agent in AS^3 calculus

The sub-process *canUseHeli_Agent* is defined by L8-L14. It first collects information on situations *lowWindForAWhile* (S1) and *withinRange* (S2) in L9. Then, the result of analyzing situation *canUseHeli* is generated based on the truth value of S1 and S2 (L10-L12). In addition, method *backToBase* is triggered in L12.

This example illustrates the following important aspects of defining SAW agents using AS^3 calculus:

(a) The input and output actions in AS^3 calculus are used to represent communications among SAW agents. When an SAW agent determines the value of a situation *s*, it sends all the related contexts and the value of *s* through a communication channel also named *s*. All other agents interested in *s* will receive the information from channel *s*. Hence, SAW agents can be easily reused since new applications can obtain situational information based on the names of situations.

(b) The parallel composition and non-deterministic choice (see Table 1) in AS^3 calculus are used when multiple input actions need to be performed by an SAW agent without predefined execution orders. Which operator should be used is determined by our agent synthesis algorithms.

(c) The method invocation and atomic constraint evaluation in AS^3 calculus are used to represent operations on contexts.

(d) The timeout and recursive processes in AS^3 calculus are used to represent periodical context acquisition and situation analysis.

7.2 The SAW agent synthesis algorithms

Given a set of SAW specifications, our SAW agent synthesis process will do the following:

1) For each specified situation *s*, if *s* is an atomic situation, synthesize a sub-process for *s* using ***SynAtom*** algorithm. If *s* is a logical composite situation, synthesize a sub-process for *s* using ***SynComp*** algorithm. If s is a temporal situation, synthesize a sub-process for *s* using ***SynTemporal*** algorithm.

2) For each SAW agent, synthesize its main process to initialize the sub-processes for all the situations processed by the SAW agent using ***SynMain*** algorithm.

These algorithms are given below:

***SynAtom* algorithm:**

Require specification of an atomic situation aS_i in the format of

$Def_i \rightarrow k([x_0, \ldots, x_n]\ aS_i, monitor\_until(f_i, cond_i))$

1: Initialize an empty list aL_i to store the operations for analyzing aS_i, and two empty lists $reqL_i$ and $acqL_i$ to store the required and acquired variables of aS_i.

2: **for** each atomic formula T_j in Def_i **do**

3: **if** T_j is $serv(I_j;O_j;S_j)$ **then**

4: Find the method signature M_j from the specification of service S_j by matching I_j and O_j, and add M_j to aL_i. Append I_j and O_j to $reqL_i$ and $acqL_i$ respectively.

5: **else if** T_j is $K(O_j;SM_j)$, where SM_j is a service name concatenated with a method name **then**

6: Add a an input action to aL_i, and append O_j to $acqL_i$

7: **else if** T_j is an atomic constraint **then**

8: Generate an *If-then-else* statement, in which the condition is a constraint evaluation for T_j, an output action *ch* $aS_i(x_0, \ldots, x_n,$ ***true***) is in the *then* branch, an output action *ch* $aS_i(x_0, \ldots, x_n,$ ***false***) in the "*else*" branch

9: Iterate reactive behavior specifications to find actions to be triggered in aS_i or $\neg aS_i$, and add the method invocations to the "*then*" or "*else*" branch, and append it to aL_i

10: **end if**

11: **end for**

12: Get input perimeters for instantiating this sub-process by removing all variables in $acqL_i$ from $reqL_i$

13: Append *(time* f_i **.**$aS_i\_agent(req_i)$ for recursion to aL_i

14: **if** aL_i is used to define a temporal situation **then**

15: Get system's current time *Now* and append *.appendHistory(*aS_i*, SituData, Now)* to aL_i, where *SituData* contains $x_0, \ldots, x_n$ and aS_i's value

16: **end if**

17: Append *plus ch* $cond_i$*(bool Status) . zero)* to the end of aL_i

***SynComp* algorithm:**

Require specification of an logical composite situation cS_i in the format of:

$Def_i \rightarrow k([x_0, \ldots, x_n], cS_i, monitor\_until(f_i, cond_i))$

1: **for** each formula $k([c_0, \ldots, c_j], S_j)$ in Def_i **do**

2: Generate an input action *ch* $S_j\ (x_0, \ldots, x_n, S_j\_result)$ to get the information of S_j

3: **if** S_j is the name of a situation **then**

4: Generate a condition expression in the format of ($S_j\_result = true$)

5: **else if** S_j is in the form not(S_j'), where S_j' is the name of a situation **then**

6: Generate a condition expression in the format of ($S_j\_result = false$)

7: **end if**

8: **end for**

9: **if** a conjunction ($\wedge$) in Def_i is used **then**

10: The corresponding input actions are concatenated using "par", and the condition expressions are concatenated using "and"

11: **else if** a disjunction ($\vee$) in Def_i is used **then**

12: The corresponding input actions are concatenated using "plus", and the condition expressions are concatenated using "or"

13: **end if**

14: Generate *if-then-else* statements with the generated conditional evaluations, and placed them after all the input actions as line 8 in *SynAtom*

15: Output actions for sending the situation analysis result and actions to be triggered are added on proper branches as line 9 in *SynAtom*

16: Generate statement for recursion and termination as lines 13-17 in ***SynAtom***

***SynTemporal* algorithm:**

Require specification of a temporal situation tS_i in the format of

$\forall T$ (or $\exists T$), $CurrentTime - \varpi < T < CurrentTime - \varpi + \varepsilon$, $k([c_0, \ldots, c_j], S_j)$
$\rightarrow k([x_0, \ldots, x_n], tS_i, monitor\_until(f_i, cond_i))$

Generate statement for invoking service *chkSituP*(S_j, ω, ε) or *chkSituH*(S_j, ω, ε)

Generate statement for invoking service *retrieveRelatedData*(S_j, ω, ε)

Generate *if-then-else* statements with the generated conditional evaluations, and placed them after all the input actions as line 8 in ***SynAtom***

Output actions for sending the situation analysis result and actions to be triggered are added on proper branches as line 9 in ***SynAtom***

Generate statement *(time f_i . tS_i_agent(req_i) plus ch $cond_i$(bool Status) . zero)*

***SynMain* algorithm:**

Require a list of situations *L* for agent $agent_i$

for each situation *s* in *L*
    if *s* needs input perimeters $p_1, \ldots, p_n$ for instantiating its corresponding sub-process **then**
        Find a set of situations $S = \{s_k, \ldots, s_j\}$ from situation specifications, such that they provide $\{p_1, \ldots, p_n\}$ as outputs
        for each s' in S
            Generate an output action *ch s'(contextType p_i, … , contextType p_m, bool S')*
        end for
        Concatenate output actions using "par"
        Generate a statement of . *s_agent(*contextType p_1, … , contextType p_n*)*
    else
        Generate *s_agent*
    end if
end for
Concatenate statements using *par*
Generate a statement of . $agent_i$ for recursion

We will again use the "sea rescue" example to illustrate the above process. Based on decomposition results, *saw_heliAgent* monitors three situations *withinRange* (AS2 in Fig. 4), *canUseHeli* (CS1 in Fig. 4), and *readyToDispatchHeli* (CS2 in Fig. 4). Hence, sub-process *withinRange_Agent* for analyzing situation *winthinRange* is synthesized using ***SynAtom***.

Initially, the list aL_2 for storing the operations for analyzing (AS2) is empty. Since the first atomic formula *serv([loc(Aloc)]; [bool(Result)]; heli)* in (AS2) matches the case in Line 4 of ***SynAtom***, the corresponding method signature shown in (SERV2) is found and appended to aL_2. The list $reqL_2$ for storing the required contexts for analyzing (AS2) and the list $acqL_2$ for storing the contexts collected by *saw_heliAgent* are also updated. Now, we have $reqL_2$ = *[loc(Aloc)]*, $acqL_2$ = *[bool(Result)]*, aL_2 = [*withinFlightRange([int(ALoc)]; [bool(Result)]; heli)*].

Since the second atomic formula *Result = true* in (AS2) matches the case in Line 7 of ***SynAtom***, an *if-then-else* statement is generated following Lines 8-9. Now, aL_2 = [*withinRange(int(ALoc); bool (Result); heli*), *if Result=true then ch withinRange<true> else ch withinRange<false>*].

Since there is no more atomic formula in (AS2), the loop from Line 2 to Line 10 ends. Since $reqL_2$ contains variable *ALoc*, which is not in $acqL_2$, an input parameter is declared for *withinRange_Agent* (L1 in Fig. 5).

Next, AS^3 calculus terms for the operations currently in aL_2 need to be generated and properly ordered. The calculus term for *withinRange([int(ALoc)]; [bool(Result)]; heli)* is the following beta reduction in AS^3 calculus:

let bool Result=heli:withinFlightRange(integer ALoc) instantiate P ,

where *P* denotes a process of subsequent operations.

In this example, the subsequent operation is the *if-then-else* statement in aL_2 since variable *Result* used in the *if-then-else* statement is the output from method *withinFlightRange*. Hence, *P* is replaced by the *if-then-else* statement, and L2-L5 in Fig. 5 are generated. Finally, since *monitor_until(50, rescueSuccess)* is specified in (AS2), L6-L7 in Fig. 5 are generated following Lines 13-17 of ***SynAtom***.

For logical composite situation "canUseHeli" (CS1 in Fig. 4), a sub-process is generated using ***SynComp*** algorithm. By scanning CS1, the following formulas are found:

- *k([], withinRange)*
- *k([loc(ALoc), windVel(Vel)], lowWindForAWhile)*

Hence, the corresponding input actions and condition expressions, which are generated following Lines 3-4 of ***SynComp***, are given below:

Input Actions	Condition Expression
ch lowWindForAWhile(bool S1)	S1 = true
ch withinRange(bool S2)	S2 = true

As shown in L9-L12 in Fig. 5**,** following Lines 1-13 of ***SynComp***, the input actions are concatenated using **par**, and the subsequent condition evaluation is generated. Finally, L13-L14 in Fig. 5 are generated since *monitor_until(10, rescueSuccess)* is specified in (CS1). Similarly, *readyToDispatchHeli_agent* can be synthesized.

After the generation of *withinRange_Agent* for (AS2), *canUseHeli_Agent* for (CS1) and *readyToDispatchHeli_agent* for (CS2), the main process of *saw_heliAgent* is synthesized using ***SynMain***.

In ***SynMain***, if a situation monitored by an SAW agent depends on the context data collected by other SAW agents, proper input actions will be generated by ***SynMain***, and the data retrieved by input actions will be used to instantiate the sub-process for monitoring the situation. The input actions and subsequent instantiation statement of sub-processes are concatenated using **par**.

For (AS2), its required input list $reqL_2$ contains variable *ALoc*. By searching the situation specifications, situation *accidentDetected* provides the value of *ALoc*. Hence, an input action in L17 in Fig. 5 is synthesized to collect *ALoc*. Then, the sub-process for analyzing situation *withinRange* (AS2) is instantiated with an input parameter (*ALoc*) in Fig. 5. Similarly, we can also generate the instantiation statement for the sub-process that monitors situation *canUseHeli* (CS1) and the sub-process that monitors situation *readyToDispatchHeli*. Finally, the instantiation statements for the sub-processes are composed using **par** in L17 in Fig. 5. A recursion statement is added at the end of *saw_heliAgent*.

7.3 Complexity analysis of the SAW agent synthesis algorithms

Theorem 2 (complexity of agent synthesis): Given *p* situations, and q services, the complexity of agent synthesis is $O((p+2q)^{*}p)$..

Proof: Assume that there are *x* atomic situations, *y* logical composite situations, *z* temporal situations, the maximum *LAS* is l_{as}, the maximum *LLCS* is l_{lcs}, the maximum number of trigger relations for a situation is *g*, and the maximum number of input parameters for a situation is *e*. For synthesizing sub-processes for *x* atomic situations, it takes $O(x*(l_{as}+g)*q)$ steps. For synthesizing sub-processes for *y* composite situations, it takes $O(y*(l_{lcs}+g*q))$ steps. For synthesizing sub-processes for *z* temporal situations, it takes $O(z*g*q)$ steps. To synthesize the main processes, it takes $p*(e*p+g*q)$ steps. Since l_{as}, l_{lcs}, *g*, *e* are usually small numbers, the total complexity is $O(x*(l_{as}+g)*q)+y*(l_{lcs}+g*q)+ z*g*q + p*(e*p+g*q) = O((p+2q)*p)$.

8. Evaluations

8.1 Evaluating our GUI tool

Experiments have been conducted to evaluate our overall approach. Evaluating the usability of our GUI tool is based on case studies. We asked a novice user and an expert user to use our SAW tool. They are required to model the SAW requirements of a situation-aware application. The average time spent for modeling different types of SAW requirements by the two users is shown in Table 4.

Table 4
Average time for modeling different types of SAW requirements

Service	Atomic situation	Logical Composite Situation	Temporal Situation	Relation
1 min/service	2 min/situation	1 min/situation	0.7 min/situation	0.5 min/relation

The time needed for modeling an atomic situation increases as *LAS* increases. The time needed for modeling a logical composite situation increases as *LLCS* increases. However, *LAS* is usually smaller than 20 because defining an atomic situation generally does not involve many service invocations. Developers can often keep *LLCS* small by reusing situations previously defined in the specifications of new situations.

8.2 Evaluating our decomposition and SAW agent synthesis algorithms

Our decomposition and SAW agent synthesis algorithms were implemented using Prolog. A test generation tool was developed using Java to randomly generate specifications of services, situations and relations in AS^3 logic. Programs were run on a desktop with Pentium D CPU 3.00 GHz and 2 G RAM.

Fig. 6 shows the time comparison of decomposing and synthesizing SAW agents for 5 to 1000 situations (*LAS* = [1, 3], *LLCS* = [2, 4]) with different percentages of logical compositions. The solid line shows that it takes about 2.5 and 22 seconds to decompose and synthesize 100 and 1,000 situations with atomic and temporal situations only, respectively. The dotted line shows that it takes less than 1 second and 10 minutes

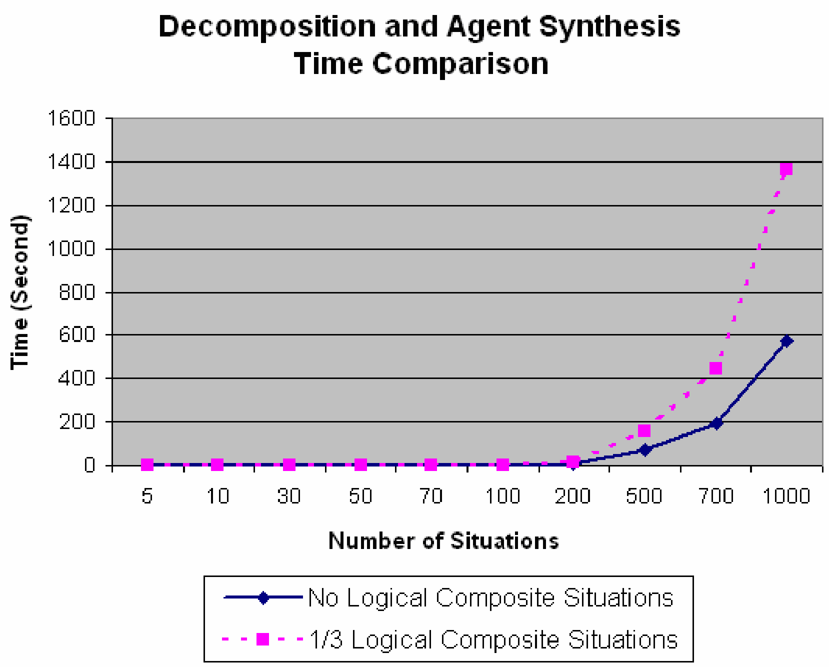

Fig. 6. Decomposition and agent synthesis time comparison

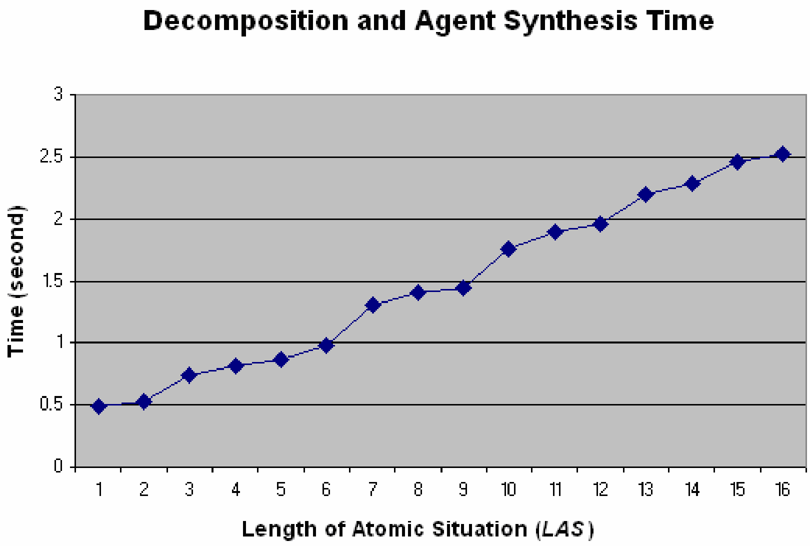

Fig. 7. Decomposition and agent synthesis time for 80 situations with different LAS

to decompose and synthesize 100 and 1000 situations, respectively, with 1/3 logical composite situations, and 2/3 atomic situations and temporal situations. It is noted that less time is needed to decompose and synthesize SAW agents for situations with logical composite situations than that without logical composite situations because the number of situation composition trees is smaller for situations with logical composition situations.

Fig. 7 shows the decomposition and agent synthesis time for 80 situations containing 1/3 logical composite situations with *LLCS* = 3 and 2/3 atomic situations with *LAS* = [1, 15] and temporal situations. It takes about 2.5 seconds to decompose and synthesize 80 situations with 1/3 situations being logical composite situations and *LAS* = 15. Fig. 8 shows the decomposition and agent synthesis time for 80 situations containing 1/3 logical composite situations with *LLCS* = [2, 15] and 2/3 atomic situations with

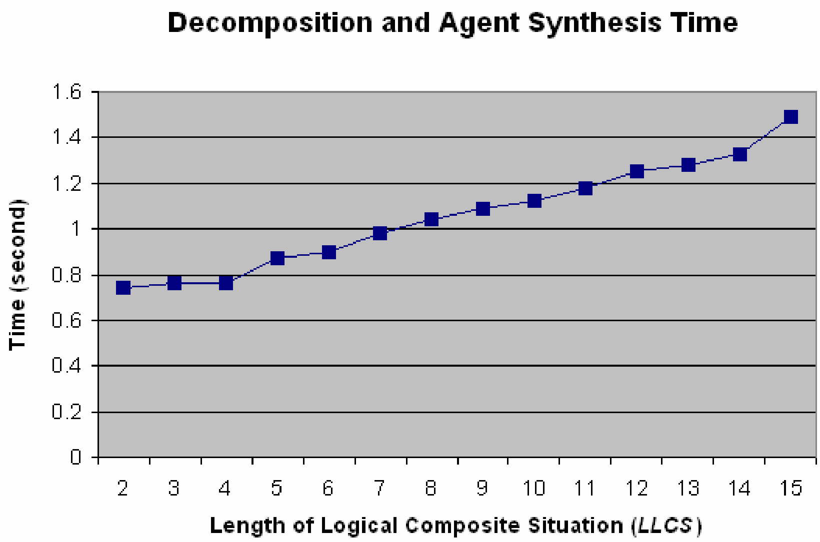

Fig. 8. Decomposition and agent synthesis time for 80 situations with different LLCS

LAS = 2 and temporal situations. It takes about 1.5 seconds to decompose and synthesize 80 situations with 1/3 situations being logical composite situations and *LLCS* = 15.

The above evaluation results show that our decomposition and agent synthesis algorithms are quite efficient. This is especially important for runtime system adaptation. When a host or some SAW agents on the host are not available or the user's QoS requirements are changed, SAW agents can be re-synthesized in a timely manner using our approach to replace the original ones.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a logic-based approach for specification, decomposition, and agent synthesis for situation-aware SBS. Our approach is based on our SAW model and AS^3 calculus and logic. SAW requirements can be analyzed and represented graphically using our SAW model and GUI tool. The graphical representation of SAW requirements can be automatically translated to declarative AS^3 logic specifications. An algorithm for decomposing SAW specifications has been developed based on network topology, communication bandwidths among various hosts, and composition relations among situations. Algorithms for automated SAW agent synthesis were also presented. Our experimental results show that our GUI tool has good usability, and the decomposition and agent synthesis algorithms are efficient. However, so far, the SAW agents are only capable of analyzing truth-value based situations. Future work includes extensions for handling fuzzy situations, semantic-based context discovery, and privacy protection in SAW.

Acknowledgment

This work was supported by the DoD/ONR under the Multidisciplinary Research Program of the University Research Initiative, Contract No. N00014-04-1-0723.

References

Appel, A., 1992. Compiling with Continuations. Cambridge University Press.

Bharadwaj, R., 2003. Secure Middleware for Situation-Aware Naval C2 and combat Systems. Proc. 9th Int'l Workshop on Future Trends of Distributed Computing System (FTDCS 2003), 233-240.

Blackburn, P., deRijke, M., Venema, Y., 2003. Modal Logic. Cambridge University Press.

Booth, D., Haas, H., McCabe, F., Newcomer, E., et al., 2004. Web Services Architecture. Available at: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

Cardelli, L., Gordon, A. D., 2000. Mobile Ambients. Theoretical Computer Science, vol. 240(1), 177-213.

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 Trans. on Software Engineering, vol. 29(12), 1072-1085.

Chen, H., Finin, T., Joshi, A., 2003. An Ontology for Context-Aware Pervasive Computing Environments. Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review, vol.18, 197-207.

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, vol. 16(2-4), 97-166.

Huth, M., Ryan, M., 2004. Logic in computer science: modeling and reasoning about systems. Cambridge University Press.

Kifer, M., Lausen, G., Wu, J., 1995. Logical foundations of object-oriented and frame-based languages'. JACM, 42(4), 741-843.

Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R., 1997. GOLOG: A Logic Programming Language for Dynamic Domains. Journal of Logic Programming, vol.31, no. 1-3, 59-84.

Matheus, C. J., Kokar, M. M., Baclawski, K., 2003. A Core Ontology for Situation Awareness, Proc. 6th Int'l Conf. on Information Fusion, 545 –552.

May, D., Shepherd, R., 1984. The Transputer Implementation of Occam. Proc. Int'l Conf. on Fifth Generation Computer Systems, 533-541.

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, PhD Thesis, University of Toronto, Toronto.

Ranganathan, A., Campbell, R. H., 2003. A Middleware for Context-aware Agents in Ubiquitous Computing Environments. Proc. ACM Int'l Middleware Conf., 143-161.
Reiter, R., 2001. Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press.
Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., Nahrstedt, K., 2002. A middleware infrastructure for active spaces. IEEE Pervasive Computing, vol. 1(4), 74-83.
Sorensen, M. H., Urzyczyn, P., 2006. Lectures on the Curry-Howard Isomorphism. Elsevier.
Yau, S. S., Karim, F., Wang, Y., Wang, B., Gupta, S. K.S., 2002. Reconfigurable Context-Sensitive Middleware for Pervasive Computing. IEEE Pervasive Computing, vol. 1(3), 33-40.
Yau, S. S., Wang, Y., Karim, F., 2002. Development of Situation-Aware Application Software for Ubiquitous Computing Environments. Proc. 26th IEEE Int'l Computer Software and App. Conf., 233-238.
Yau, S. S., Huang, D., Gong, H., Seth, S., 2004. Development and Runtime Support for Situation-Aware Application Software in Ubiquitous Computing Environments. Proc. 28th Annual Int'l Computer Software and Application Conference (COMPSAC 2004), Hong Kong, 452-457.
Yau, S. S., Huang, D., Gong H., Davulcu, H., 2005. Situation-Awareness for Adaptable Service Coordination in Service-based Systems. Proc. 29th Annual Int'l Computer Software and Application Conference (COMPSAC 2005), 107-112.
Yau, S. S., Mukhopadhyay, S., Huang, D., Gong, H., Davulcu, H., Zhu, L., 2005. Automated Agent Synthesis for Situation-Aware Service Coordination in Service-based Systems. Technical Report, ASU-CSE-TR-05-008. http://dpse.eas.asu.edu/AS3/papers/ASU-CSE-TR-05-009.pdf
Yau, S. S., Gong, H., Huang, D., Zhu, L., 2006. Automated Agent Synthesis for Situation Awareness in Service-based Systems. Proc. 30th Annual Int'l Computer Software and Application Conference (COMPSAC 2006), 503-512.
Yau, S. S., Huang, D., Gong, H., Yao, Y., 2006. Support for Situation-Awareness in Trustworthy Ubiquitous Computing Application Software. Journal of Software Practice and Engineering (JSPE), 893-921.
Yau, S. S., Mukhopadhyay, S., Davulcu, H., Huang, D., Gong, H., Singh, P., Gelgi, F., to be published. Automated Situation-Aware Service Composition in Service-oriented Computing. International Journal of Web Services Research (IJWSR).

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 Ph.D. 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. Contact him at yau@asu.edu.

Haishan Gong is a Ph.D. candidate in the Computer Science and Engineering Department 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. Contact her at Haishan.Gong@asu.edu.

Dazhi Huang is a Ph.D. 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. Contact him at Dazhi.Huang@asu.edu.

Wei Gao is a Ph.D. student in the Department of Computer Science and Engineering at Arizona State University. His research interests include mobile and ubiquitous computing, wireless network architecture, and network optimization. He received his B. E. degree in electrical engineering from University of Science and Technology, China. Contact him at W.Gao@asu.edu.

Luping Zhu is a Ph.D. candiate in the Computer Science and Engineering Department at Arizona State University. His research interests include distributed systems, software deployment. He received his BS from Xian Jiaotong University, China, MS from Zhejiang University, China, both in computer science. Contact him at Luping.Zhu@asu.edu.