

Specification, decomposition and agent synthesis for situation-aware service-based systems

S.S. Yau*, H. Gong, D. Huang, W. Gao, L. Zhu

Arizona State University, Tempe, AZ 85287-8809, USA

Available online 29 February 2008

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, e-business, health care, military applications 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 is presented. This approach is based on AS³ calculus and logic, and our declarative model for SAW. Evaluation results of our approach are also presented.

© 2008 Elsevier Inc. All rights reserved.

Keywords: Service-based systems; Situation awareness; Decomposition; Agent synthesis; AS³ 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 (Booth et al., 2004). SBS have been applied in many areas, such as collaborative research and development, e-business, health care, military applications and homeland security. In SBS, a service is a self-contained, loosely-coupled software entity with discoverable and invocable interface, providing certain functionality over networks using well-defined and standard protocols, such as HTTP and SOAP. However, in a dynamic environment, services may be unavailable or

unable to provide desirable QoS due to distributed denial-of-service attacks, system failures, or system overloads. It is desirable that SBS can adapt itself for satisfying users' needs under various situations. Situation awareness (SAW) is the capability of being aware of situations and adapting the system's behavior based on situation changes (Yau et al., 2004, 2006b). SAW is often needed for system monitoring, adaptive service coordination and flexible security policy enforcement (Yau and Yao, in press). A *situation* is a set of contexts in a system over a period of time that affects the future system behavior for specific applications, and a *context* is any instantaneous, detectable property of the environment, the system, or the users relevant to the system behavior (Yau et al., 2002a,b).

A large-scale SBS usually 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 software components providing SAW capability in SBS. To greatly reduce the development effort of situation-aware

* Corresponding author.

E-mail addresses: yau@asu.edu (S.S. Yau), haishan.gong@asu.edu (H. Gong), dazhi.huang@asu.edu (D. Huang), w.gao@asu.edu (W. Gao), luping.zhu@asu.edu (L. Zhu).

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 acquisition of relevant contexts, analysis of situation changes, and 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 an 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 decompose the tasks for achieving SAW capability to distributed SAW agents in SBS. Such decomposition can be achieved by partitioning the set of situations to be analyzed to multiple subsets of situations and assigning different SAW agents to analyze individual subsets of situations. However, to perform such decomposition process manually is time-consuming and error-prone. 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 AS³ calculus and logic for rapid development of Adaptable Situation-Aware Secure Service-Based (AS³) systems (Yau et al., 2007a,b). SAW requirements are analyzed and graphically specified using our SAW model and a Graphic User Interface (GUI) tool, and automatically translated to declarative AS³ logic specifications. An algorithm is presented for partitioning the generated AS³ 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 AS³ logic specifications, an SAW agent described in AS³ calculus terms will be automatically synthesized to perform the necessary tasks to meet the corresponding subset of SAW requirements. Experimental evaluation results for our specification tool, decomposition and agent synthesis algorithms will also be presented.

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 and Hayes, 1969; Pinto, 1994; Kifer et al.,

1995; Reiter, 2001; Chen et al., 2003; Matheus et al., 2003), and the other on providing toolkit, framework or middleware for development and runtime support for SAW (Dey and Abowd, 2001; Roman et al., 2002; Chan and Chuang, 2003; Ranganathan and Campbell, 2003; Yau et al., 2004, 2006b).

In the first category, Situation Calculus (McCarthy and Hayes, 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 and Hayes (1969) considers a situation as a complete state of the world, while 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 (McCarthy and Hayes, 1969) because a situation cannot be fully described. Reiter’s definition makes a situation totally determined by executed actions. GOLOG (Levesque et al., 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) (Kifer 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. Yau et al. presented a declarative SAW model (Yau et al., 2005a, 2006b), which provide graphical representations of SAW requirements that can be automatically translated to formal specifications based on AS³ logic (Yau et al., 2005b, 2007b). This declarative SAW model and AS³ logic are used in our approach, and will be outlined in Section 3.

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 decomposition of SAW requirements and automated synthesis of reusable and autonomous software components for achieving SAW capability in service-oriented computing environment.

3. Background

In this section, we will highlight the architecture of our AS³ systems (Yau et al., 2007a,b), 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 AS³ calculus and logic (Yau et al., 2005b, 2007b), which are used in the development of our agent synthesis approach.

AS³ 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 AS³ system, in which organizations publish their capabilities as services. Each service provides a set of methods as "actions" in the AS³ system. SAW Agents collect context data periodically, analyze situations based on context data and execution results of actions in the system, trigger appropriate actions in the system based on the situations, 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

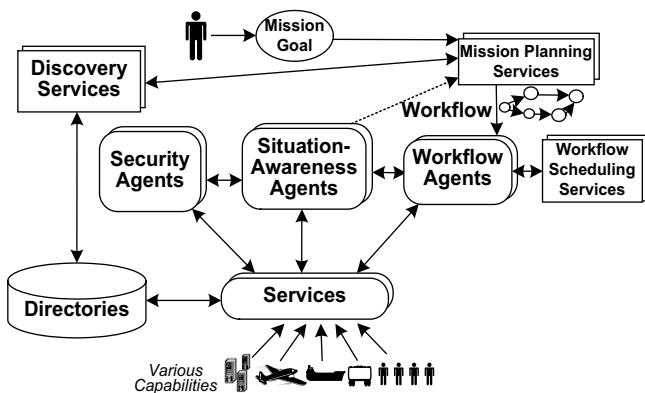


Fig. 1. The architecture of an AS³ system.

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, shown in Fig. 2, 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.

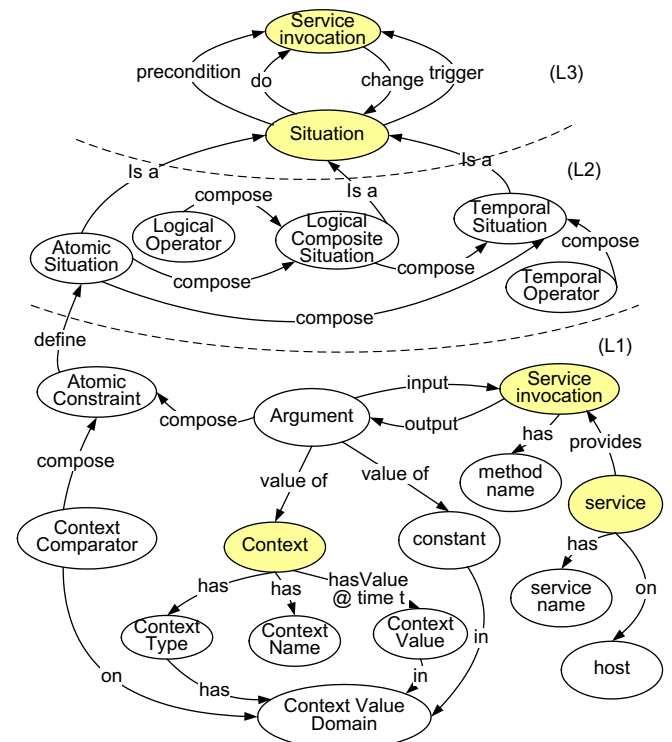


Fig. 2. An ontology for SAW in service-based systems.

- 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 *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).
- 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 as 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 context data can be retrieved by invoking one or more services provided by the system platform or developed by various service providers.

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

- 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.
- 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.
- 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.
- From the situations obtained in (iii), developers identify atomic situations if the situations contain any temporal or logical operators.
- For each identified atomic situation, developers use service invocations, contexts, constants, and context comparators identified in (ii) to specify the atomic situation. Developers then use the specified atomic situations, and temporal or logical operators to further specify the situations identified in (iii).

Our SAW model is language-independent and can be translated to specifications written in 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. Fig. 3 illustrates the graphical representation of

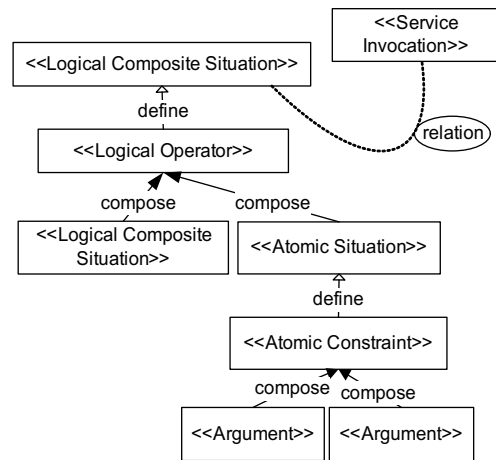


Fig. 3. Partial graphical representation of SAW requirements.

some constructs in our SAW model. The 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 not represented in Fig. 3. These attributes are required for synthesizing SAW agents. A GUI tool was developed using Java and a third-party software package JGraph to implement the above graphical representation. It contains two major parts: a drawing area and a tab area. Developers can drag and drop the boxes, lines and arrows onto the drawing area to graphically specify SAW requirements. The tab area consists of a set of tabs for developers to input detail information for each entity.

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., 2005b, 2006a, 2007b) is based on classical process calculus (Milner, 1999; Cardelli and Gordon, 2000). 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. 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 and part of the syntax of AS^3 logic in Table 2 which are 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

Table 1
Part of the syntax of AS³ calculus

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

Table 2
Part of the syntax of AS³ logic

$\phi_1, \phi_2 ::=$	Formula	$E(\phi_1 u \phi_2)$	Until
\mathbf{T}	True	$E(\phi_1 s \phi_2)$	Since
U	Nominal	$k(u; \phi)$	Knowledge of u
$\text{pred}(x_1, \dots, x_n)$	Atomic formula	$\text{serv}(x; u; \sigma; \phi)$	Invocation of service σ using input x by ϕ and returning u
$x \sim c$	Atomic constraint ²		
	$// \sim ::= > < \leq \geq =,$ c is a natural number	$\exists t \phi$	Existential quantification on time
$\phi_1 \vee \phi_2$	Disjunction	$\langle u \rangle \phi$	Behavior after sending message
$\neg \phi$	Negation	$\phi_1 \wedge \phi_2$	Conjunction

[2] The constraints in AS³ logic are restricted to a subset of linear arithmetic constraints as shown in Table 2, so that AS³ logic is decidable.

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. *time t.P* denotes a process that will start execution after a timeout of t units. 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 (Appel, 1992) is used to provide semantics of asynchronous service invocations. In Table 1, $I: I_i(y)\hat{cont}$ denotes the invocation of the method I_i exported by I with parameter y and continuation $cont$. The continuation $cont$ is an explicit form for passing control between a method being invoked and the process invoking the method. 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. AS³ logic (Yau et al., 2005b, 2006a, 2007b) is a hybrid normal modal logic¹ (Blackburn et al., 2003) for specifying SBS. 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 AS³ logic supports developers to declaratively specify situation awareness requirements. Models for the logic are processes in the AS³ 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 AS³ logic can support the synthesis of AS³ calculus terms from declarative AS³ logic specifications.

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 $\text{serv}(x; u; \sigma; \phi)$ indicates that a process ϕ invoking service σ with parameter x receives u as the result. The formula $\langle u \rangle \phi$ describes the behavior of a process after sending out u . The AS³ logic is a hybrid normal 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: $\text{diam}(\phi) := E(T U \phi)$.
- Universal quantification on time: $\forall t \phi := \neg \exists t \neg \phi$.

4. Overview of our approach

As mentioned before, the tasks for achieving SAW capability in an SBS include relevant context acquisition, distributed situation analysis and triggering proper actions in response to

¹ Blackburn et al. define hybrid logics as the logics “use one sort of atoms called nominals to refer to states which are regarded as first class citizens”. A normal modal logic is “a set of formulas that contains all tautologies, $\Box(\phi \rightarrow \sigma)$, $(\Box\phi \rightarrow \sigma)$, and $\Diamond\phi \leftrightarrow \neg\Box\neg\phi$, and is closed under uniform substitution, modus ponens, and generalization” (Blackburn et al., 2003).

situation changes at runtime. To develop SAW capability in SBS, the following major issues need to be addressed:

- (1) *Specifying 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.
- (2) *Partitioning the set of specified situations.* The set of specified situations need to be properly partitioned to decompose the tasks to distributed SAW agents so that they can efficiently achieve SAW capability.
- (3) *Synthesizing 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 partition of the set of specified situations and agent synthesis for situation-aware SBS. In Section 5, we will present the specification of SAW requirements in AS³ logic. Automated partition of the set of specified situations and synthesis of SAW agents will be presented in Sections 6 and 7, respectively.

4.1. Architecture of our approach

The architecture of our approach is depicted in Fig. 4. 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 the corresponding technique(s) 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, and then translated to formal specifications in AS³ logic. Using the GUI tool, developers can easily generate AS³ logic specifications without any knowledge of the AS³ logic. We assume that the consistency and redundancy of the specifications have been checked by developers or some automated tools.

Step (2) *Partitioning the set of specified situations.* Given consistent and concise SAW specifications, the set of situations need to be partitioned into multiple subsets, each of which is assigned to an SAW agent for collecting contexts, analyzing the situations in the subset and triggering system’s reactive behavior under these situations. We have developed an algorithm to perform the partitioning based on multiple inputs, including SAW requirement specifications, and the system-specific knowledge of network topology and the communication bandwidth between each pair of hosts in the system (see Section 6). Our algorithm will find a proper way to partition the set of situations specified in Step (1) so that the communication cost among the SAW agents for analyzing these situations and the cost for re-synthesizing SAW agents due to runtime changes of SAW requirements can be greatly reduced.

Step (3) *Synthesizing SAW agents.* From the results of Step (2), SAW agents are automatically synthesized with AS³ calculus terms using our agent synthesis algorithm (see Section 7). We have developed an AS³ calculus to Java compiler to compile the AS³ calculus terms to Java codes, which can be compiled to executable codes using a Java compiler. The executable SAW agents will run on a distributed agent execution platform, e.g. the Secure Infrastructure for

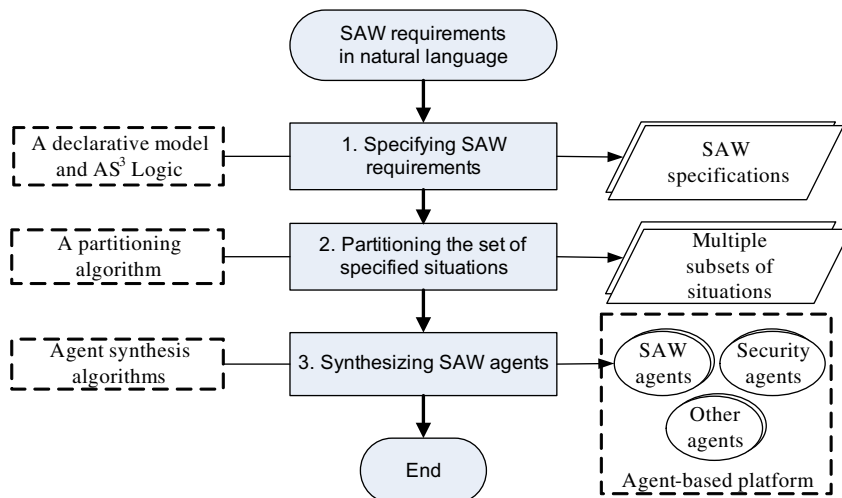


Fig. 4. Architecture of our approach.

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

- *appendHistory(SituName, SituData, Timestamp)* stores situational information *SituData* for situation *SituName* and removes outdated data.
- *chkSituP(SituName, ω , ϵ)* checks whether the situation was true sometime within $[CurrentTime - \omega, CurrentTime - \omega + \epsilon]$, 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 $[CurrentTime - \omega, CurrentTime - \omega + \epsilon]$.
- *retrieveRelatedData(SituName, ω , ϵ , Type)* retrieves related data of the situation, where *Type* is an integer used to describe which type of data with timestamps within $[CurrentTime - \omega, CurrentTime - \omega + \epsilon]$ should be retrieved. If *Type* is zero, the latest data that makes the situation true will be retrieved. If *Type* is one, the latest data that makes the situation false will be retrieved. If *Type* is two or three, all the data that makes the situation true or false will be retrieved, respectively.

At runtime, the contextual and situational information of temporal situations and the situations used to define temporal situations will be periodically retrieved from and recorded by invoking the above four methods. For the simplicity of prototype implementation, this system service currently stores data in a centralized repository, which is subject to the single point of failure. However, such a problem can be solved utilizing existing techniques for distributed data management (Thuraisingham, 1996).

4.2. An illustrative example

Consider an 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 will show our approach in how situational information is used for coordinating execution of a service-based system:

- (1) The rescue center (*rc*) receives an SOS message from a ship (*bs*) indicating that *bs* has a serious accident and some of its 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 ‘critical’ status, 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, if *heli* is already dispatched and the weather changes to unsafe for *heli*, *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 the wind velocity near *bs* has been lower than 18 meters per second (m/s) for 15 time units. Developers can analyze the SAW requirements using our SAW requirement analysis steps identified in Section 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, 18, and ‘critical’.
 - (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 18 m/s (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 AS³ logic specifications from the graphical representations using our GUI tool without any knowledge of AS³ logic (see Section 3.2). The generation of AS³ logic

Table 3
Specifying SAW requirements in AS³ logic

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

specifications for SAW requirements can be easily done following a mapping between our model constructs and AS³ logic formulas shown in Table 3. In the following, we will discuss various specifications of SBS.

- 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 $serv(x_1; u_1; \sigma_1), \dots, serv(x_n; u_n; \sigma_n)$ for collecting context values u_1, \dots, 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, \dots, u_n . Argument arg_2 can either be one of u_1, \dots, 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. The AS³ logic specification for this atomic situation becomes:

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

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 AS³ logic, temporal operators P (sometimes in the past) and H (had been true over a period of time in the past) are defined using \exists (existential) and \forall (universal) quantifications over a time range. The time range is defined as $[CurrentTime - \omega, CurrentTime - \omega + \epsilon]$, 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 had 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 of atomic situations, tem-

Table 4
Partial SAW specifications in the example

<i>/* service specifications */</i>	
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)
<i>/* atomic situation specifications */</i>	
AS1)	serv([loc(ALoc), int(Time)]; [windVel(Vel)]; rc) ∧ Vel < 18 → 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)))
<i>/* temporal situation specifications*/</i>	
TS)	∀Time CurrentTime-15 < Time < CurrentTime ∧ k([loc(ALoc), windVel(Vel)], lowWindVelocity) → diam(k([loc(ALoc), windVel(Vel)], lowWindForAWhile, monitor_until(10, rescueSuccess)))
<i>/* logical composite situation specifications*/</i>	
CS1)	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)	<i>/* reactive behavior specifications */</i> trigger(k([int(ALoc), windVel(Vel)], not(canUseHeli)), serv([], [], heli))

poral 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. AS³ 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 AS³ 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. Table 4 shows partial SAW specifications in the “sea rescue” example.

6. Automated partition of the set of specified situations

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 sizes and network bandwidths 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* 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 partition of the set of specified situations 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. System-specific knowledge and decision factors for automated partition of the set of specified situations

The automated partition requires system-specific 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 AS³ logic, network topology and communication bandwidth are specified as follows:

- *serviceHost(s, h)*: Service *s* is deployed on host *h*.
- *bw(h₁, h₂, b)*: The bandwidth between host *h*₁ and host *h*₂ is *b*. When *h*₁ = *h*₂, *b* = ∞.

Generally, system-specific knowledge specifications are provided by domain experts. Based on the SAW requirement specifications and system-specific knowledge specifications, the automated partition in our approach depends on the following two factors:

6.1.1. 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

$$cost(s, h_k) = \begin{cases} 0, & H = \{h_k\} \\ \sum_{i=1, i \neq k}^n (\text{num}_x + \text{num}_y) \times \frac{1}{bw(h_k, h_i)}, & n > 1, s \notin TS, s \times TS \\ \frac{1}{fr \times bw(h_k, h_{sys})} + \sum_{i=1, i \neq k}^n (\text{num}_x + \text{num}_y) \times \frac{1}{bw(h_k, h_i)}, & n > 1, s \notin TS, s \times TS \\ \frac{1}{bw(h_k, h_{sys})} + \sum_{i=1, i \neq k}^n n_y \times \frac{1}{bw(h_k, h_i)}, & n > 1, s \in TS \end{cases}$$

where H denotes a set of unique hosts related to situation s by providing either the contextual or situational information for analyzing s , or the service invocations which should be triggered under s ; 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 in TS ; and $s \times TS$ denotes that s is not used to define any temporal situation in TS . The communication cost $cost(s, h_k)$ 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 $s \times TS$, i.e. s is not a temporal situation and not used to define any temporal situation. If s is an atomic situation, then num_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 num_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 , num_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 $s \triangleright TS$, i.e. s is not a temporal situation, but is used to define a temporal situation. In this case, 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 .

To achieve efficient distributed situation analysis, the final selection of *sink point* for situation s should be the host that requires the minimum communication cost among all possible hosts.

$$H = \{h_k\}$$

$$n > 1, s \notin TS, s \times TS$$

$$n > 1, s \notin TS, s \times TS$$

$$n > 1, s \in TS$$

6.1.2. 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 reduce the effort of re-synthesizing SAW agents.

6.2. Partitioning algorithm

Partitioning the set of specified situations is conducted in the following three steps: (1) find related hosts for each situation, (2) determine the *sink point* for each situation, and (3) further partition a set of situations with the same *sink point* to multiple subsets based on their situation composition trees. Situation composition trees can be easily constructed based on situation definitions. Our **Partitioning** algorithm uses **FindRelatedHosts** algorithm and **FindPartitioning** algorithm. These algorithms are shown as follows:

Partitioning 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 $host_{sys}$

```

1:  $L = \{\}$ 
2: for each situation  $s_i$  in  $sList$  do
3:    $relatedHostList_i = FindRelatedHosts(s_i)$ 
4:    $costResultList_i = \{\}$ 
5:   for each host  $h_k$  in  $relatedHostList_i$  do
6:      $c_i = cost(s_i, h_k)$ , and insert  $c_k$  in  $costResultList_i$ 
7:   end for
8:   Get host  $h_k$  for  $s_i$ , where  $c_i = cost(s_i, h_k)$  and
 $c_i = get\_minimum(costResultList_i)$ . If there are multi-
ple such hosts, choose the one that has minimum
number of situations assigned currently
9:   Insert  $sinkPoint(s_i, h_k)$  in  $L$ 
10: end for
11:  $agentList = FindPartition(sList)$ 

```

FindRelatedHosts(s_i) algorithm:

Require: a situation s_i in $sList$, a list L recording the latest analysis results of sink points, SAW specifications, and network topology specifications

```

1:  $hostList_i = \{\}$ 
2: if  $s_i$  is an atomic situation then
3:   for each service invocation  $v_i$  in  $s_i$  do
4:     Get host  $h_p$  which provides  $v_i$  or provides input
parameters to  $v_i$ 
5:     Insert  $h_p$  in  $hostList_i$  if  $h_p \notin hostList_i$ , and record
the number of  $h_p$  in  $hostList_i$ 
6:   end for
7: else if  $s_i$  is a logical composite situation then
8:   for each situation  $s_x$  in  $s_i$ 's definition do
9:     Get  $sinkPoint(s_x, h_x)$  from  $L$ 
10:    Insert  $h_x$  in  $hostList_i$  if  $h_x \notin hostList_i$ , and record
the number of  $h_x$  in  $hostList_i$ 
11:   end for
12: else if  $s_i$  is a temporal situation defined by a situation
 $s_y$  then
13:   Get  $sinkPoint(s_y, h_y)$  from  $L$ 
14:   Insert  $h_y$  in  $hostList_i$  if  $h_y \notin hostList_i$ , and record
the number of  $h_y$  in  $hostList_i$ 
15: end if
16: Find all  $trigger(s_i, a)$  and  $host(a, h_k)$  from SAW and
network topology specifications
17: Insert  $h_k$  in  $hostList_i$ , if  $h_k \notin hostList_i$ , and record the
number of  $h_k$  in  $hostList_i$ 

```

FindPartition($sList$) algorithm:

Require: $sList$, $hList$, $treeList$, L

```

1:  $agentList = \{\}$ 
2: for each host  $h_i$  in  $hList$  do //  $1 \leq i \leq n$ ,  $n$  is the count of
 $hList$ 
3:    $agentListOnHost_i = \{\}$ 
4:   for each  $tree_k$  in  $treeList$ 
5:      $agentList_{ik} = \{\}$ 
6:     Insert  $agentList_{ik}$  in  $agentListOnHost_i$ 
7:   end for
8: end for

```

```

9: for each situation  $s_x$  in  $sList$ , where  $sinkPoint(s_x, h_i) \in$ 
 $L$  do
10:  if  $s_x \in tree_1 \cap \dots \cap tree_k \dots \cap tree_m$ , where  $tree_k \in tre-$ 
 $eList$ ,  $1 \leq k \leq m$  then
11:    Find  $agentList_{ik}$ , where  $agentList_{ik} \in agentLis-$ 
 $tOnHost_i$ , and  $agentList_{ik}$  has the minimum number
of situations currently
12:    Insert  $s_x$  in  $agentList_{ik}$  if  $s_x \notin agentList_{ik}$ 
13:  end if
14: end for
15:  $agentList = agentListOnHost_1 \cup \dots \cup agentListOnHost_i$ 
 $\dots \cup agentListOnHost_n$ , where  $agentListOnHost_i \neq \{\}$ 

```

In our **Partitioning** algorithm, finding related hosts for a situation is done using **FindRelatedHosts** algorithm in Line 3, and determining the *sink points* for the situations is done in Lines 4–8. After analyzing all the specified situations, partitioning the set of specified situations is done using **FindPartition** algorithm in Line 11.

Let us use the “sea rescue” example to illustrate this algorithm. Suppose that services *rc* and *heli* are provided by host $host_{rc}$, and $host_{heli}$, respectively. The bandwidth between $host_{rc}$ and $host_{heli}$, denoted by $bw(host_{rc}, host_{heli})$, is assumed to be 30Mbps.

First, we initialize an empty list L (Line 1 of **Partitioning** algorithm). For atomic situation *lowWindVelocity* (AS1 in Table 4), initialize an empty host list $hostList_{lww}$ (Line 1 of **FindRelatedHosts** algorithm). From the specifications in Table 4, we know that situation *lowWindVelocity* is determined by comparing the context value of *Vel* and a constant 18. The value of *Vel* is returned by method *get_windVelocity* of service *rc* on host $host_{rc}$. Hence, we insert $host_{rc}$ into $hostList_{lww}$, and initialize the count of $host_{rc}$ in $hostList_{lww}$ to be 1 (Lines 2–6 of **FindRelatedHosts** algorithm). Because the context value of *ALoc* used by method *get_windVelocity* is provided by service *rc* on $host_{rc}$, we increase the count of $host_{rc}$ in $hostList_{lww}$ to 2 (Lines 2–6 of **FindRelatedHosts** algorithm). No service invocation should be triggered under situation *lowWindVelocity*. Line 16 of **FindRelatedHosts** algorithm. The *sink point* of situation *lowWindVelocity* is $host_{rc}$ because $hostList_{lww}$ only contains $hostList_{rc}$. We insert $sinkPoint(lowWindVelocity, host_{rc})$ in L (Lines 8–9 of **Partitioning** algorithm). Similarly, the *sink point* for atomic situation *accidentDetected* (AS3 in Table 4) is $host_{rc}$, the *sink point* for situation *withinRange* is $host_{heli}$. For temporal situation *lowWindForAWhile* (CS1 in Table 4), we initialize an empty list $hostList_{faw}$. We get $sinkPoint(lowWindVelocity, host_{rc})$ from L and insert $host_{rc}$ into $hostList_{faw}$ (Lines 12–14 of **FindRelatedHosts** algorithm). 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 Table 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

$host_{heli}$ also with count of 1 (Lines 7–11 and Lines 16–17 of **FindRelatedHosts** algorithm). The communication cost for choosing $host_r$ as the *sink point* for situation *canUseHeli* and the communication cost for choosing $host_{heli}$ as the *sink point* for situation *canUseHeli* are calculated as follows:

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

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

Because $host_{rc}$ has more situations than $host_{heli}$, the *sink point* for situation *canUseHeli* is $host_{heli}$ (Line 8 of **Partitioning** algorithm). We insert *sinkPoint*(*canUseHeli*, $host_{heli}$) in L (Line 9 of **Partitioning** algorithm). Choosing *sink points* for other situations can be done in the same way. Then, we partition the set of situations with the same *sink point* based on their situation composition trees using **FindPartition** algorithm (Line 11 of **Partitioning** algorithm). In this example, *agentList* is first initialized as an empty list (Line 1 of **FindPartition** algorithm). For each host, a list of lists, each of which corresponds to a situation composition tree, are initialized and inserted into *agentList* (Lines 2–7 of **FindPartition** algorithm). We have already known that situations *accidentDetected*, *lowWindVelocity* and *lowWindForAWhile* have the same *sink point*, which is $host_{rc}$. Based on their definitions, the situations *lowWindVelocity* and *lowWindForAWhile* belong to the same situation composition tree. Hence these two situations are grouped together and inserted into the same list (Lines 9–14 of **FindPartition** algorithm), which contains a sub-set of situations that all have $host_{rc}$ as the *sink point* and will be analyzed by the same SAW agent. Similarly, situation *accidentDetected* will be inserted into a different list and be analyzed by a different SAW agent. Finally, each element of *agentList* is a list of situations having the same sink point in the same situation composition tree (Line 15 of **FindPartition** algorithm).

6.3. Complexity analysis of the Partitioning algorithm

To analyze the complexity of our **Partitioning** algorithm, we first give the following two definitions: The *length of an atomic situation (LAS)* is the number of service invocations used to collect context values 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.

Theorem 1 (complexity of the Partitioning algorithm). *Given s services, h hosts, and p situations. The complexity of the Partitioning algorithm is $O(3p^2 + 2s \times p + p \times h^2 + p \times h \times k^2)$, where k is the number of situation composition trees in the system.*

Proof. Assume that p situations include x atomic situations, y logical composite situations and z temporal situations. In addition, assume that the maximum *LAS* is l_{as} , the maxi-

imum *LLCS* is l_{lcs} , each situation has at most f related hosts, the maximum number of trigger relations for a situation is r and the maximum number of input parameters for a situation is e . In the network topology specifications, we know that there should be s *serviceHost* formulas describing the deployment of s services, and $h \times (h - 1)/2$ *bw* formulas describing bandwidths among h hosts. \square

- (1) The complexity of **FindRelatedHosts** algorithm is analyzed as follows: for each service invocation in an atomic situation, it takes at most s steps to find the service name from service specifications and at most s steps to find the host providing the service from network topology specifications. Hence, for an atomic situation with at most l_{as} service invocations, it takes at most $(l_{as} + e) \times s$ steps to find related service names and $(l_{as} + e) \times s$ steps to find related hosts. For a logical composite situation, it takes at most $l_{lcs} \times (s + p)$ steps. For a temporal situation, it takes at most p steps because a temporal situation is defined by one situation and at most p steps are needed to find the related host from L . Since at most r service invocations can be triggered under the situation, it takes at most $r \times s \times p$ steps to find related hosts. Hence, for x atomic situations, y logical composite situations and z temporal situations, the complexity of finding related hosts is $O(x \times (l_{as} \times (s + s)) + y \times (l_{lcs} \times (s + p)) + z \times p + r \times s \times p)$.
- (2) The complexity of **FindPartition** algorithm is analyzed as follows: Initialization (Lines 2–8) takes $h \times k$ steps. For one situation, it takes at most p steps to find its *sink point* from L , at most p steps to find which situation composition trees it belongs to, and at most $k \times h \times k$ steps to iterate $h \times k$ lists in *agentList* to identify the situation composition tree that has the least number of elements since the situation can belong to at most k trees. Hence, the complexity of **FindPartition** algorithm is $O(h \times k + p \times (p + p + k \times h \times k))$.
- (3) The complexity of **Partitioning** algorithm is analyzed as follows: For p situations, it takes at most $x \times (2(l_{as} + e) \times s) + y \times (l_{lcs} \times (s + p)) + z \times p + r \times s \times p$ steps to find related hosts, and at most $p \times h \times (h - 1)/2$ steps to find the bandwidths from network topology specifications and at most $p \times f$ steps to calculate the communication costs between each two different related hosts, and at most $p \times f$ steps to choose the *sink point* of the situation based on cost calculation results. To partition p situations, it takes at most $h \times k + p \times (p + p + k \times h \times k)$ steps. Hence, the complexity of **Partitioning** algorithm is $O(x \times (2(l_{as} + e) \times s) + y \times (l_{lcs} \times (s + p)) + z \times p + r \times s \times p + p \times h \times (h - 1)/2 + 2p \times f + h \times k + p \times (p + p + k \times h \times k))$. Because l_{as} , l_{lcs} , f and r are usually small numbers, the final complexity of **Partitioning** algorithm is $O(3p^2 + 2s \times p + p \times h^2 + p \times h \times k^2)$.

7. Automated synthesis of SAW agents

7.1. Representing SAW agents using AS³ 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 AS³ calculus terms, which define SAW agents. The main advantage of using AS³ calculus is to provide platform-independent models of the agents, which capture the essential processes of context acquisition, situation analysis and action triggering. These models can later be used to verify the synthesized agents by a model checker. Platform-specific compilers can be developed to compile AS³ calculus terms to executable code on different platforms. Here, we will focus on the synthesis algorithms of SAW agents in AS³ calculus terms.

Before presenting our SAW agent synthesis algorithms, we first need to examine how SAW agents are defined using AS³ 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 Table 4), *canUseHeli* (CS1 in Table 4), and *readyToDispatchHeli* (CS2 in Table 4). The main process of *saw_heliAgent* is defined by L16–L18 in Table 4. 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 *withinRange_Agent*. L18 recursively executes the *saw_heliAgent*.

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 AS³ calculus:

- The input and output actions in AS³ 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 same names of situations.
- The parallel composition and non-deterministic choice (see Table 1) in AS³ 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.
- The method invocation and atomic constraint evaluation in AS³ calculus are used to represent operations on contexts.
- The timeout and recursive processes in AS³ calculus are used to represent periodical context acquisition and situation analysis.

7.2. The SAW agent synthesis algorithms

Given a set of SAW specifications, the SAW agent can be synthesized by the following process:

- 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.

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

L16	fix <i>saw_heliAgent</i> =
L17	{ ch <i>accidentDetected</i> (integer ALoc, bool S0) . <i>withinRange_Agent</i> (integer ALoc) } par
	<i>canUseHeli_Agent</i> par <i>readyToDispatchHeli</i>
L18	<i>saw_heliAgent</i>

Fig. 5. An example SAW agent in AS³ calculus.

- (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

$Def_i \rightarrow k([x_0, \dots, x_n] aS_i, \text{monitor_until}(f_i, \text{cond}_i))$

- 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 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 $ch aS_i(x_0, \dots, x_n, \text{true})$ is in the *then* branch, an output action $ch aS_i(x_0, \dots, x_n, \text{false})$ 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 $(\text{time } f_i, aS_i_agent(req_i))$ 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, \dots, x_n and aS_i ’s value
- 16: **end if**
- 17: Append $plus\ ch\ cond_i(\text{bool } Status). zero$ to the end of aL_i

SynComp algorithm:

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

$Def_i \rightarrow k([x_0, \dots, x_n], cS_i, \text{monitor_until}(f_i, \text{cond}_i))$

- 1: **for** each formula $k([c_0, \dots, c_j], S_j)$ in Def_i **do**
- 2: Generate an input action $ch S_j(x_0, \dots, x_n, S_j_result)$ 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 $(S_j_result = \text{true})$

- 5: **else if** S_j is in the form $\text{not}(S_j')$, where S_j' is the name of a situation **then**
- 6: Generate a condition expression in the format of $(S_j_result = \text{false})$
- 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

$\forall T(\text{or}\exists T), \text{CurrentTime} - \varpi < T < \text{CurrentTime} - \varpi$
 $+ \varepsilon, k([c_0, \dots, c_j], S_j) \rightarrow k([x_0, \dots, x_n],$
 $tS_i, \text{monitor_until}(f_i, \text{cond}_i))$

- 1: Generate statement for invoking service $chkSituP(S_j, \omega, \varepsilon)$ or $chkSituH(S_j, \omega, \varepsilon)$
- 2: Generate statement for invoking service $retrieveRelatedData(S_j, \omega, \varepsilon)$
- 3: Generate *if-then-else* statements with the generated conditional evaluations, and placed them after all the input actions as Line 8 in *SynAtom*
- 4: Output actions for sending the situation analysis result and actions to be triggered are added on proper branches as Line 9 in *SynAtom*
- 5: Generate statement $(\text{time } f_i, tS_i_agent(req_i) \text{ plus } ch\ cond_i(\text{bool } Status). zero)$

SynMain algorithm:

Require a list of situations L for agent $agent_i$

- 1: **for** each situation s in L
- 2: **if** s needs input perimeters p_1, \dots, p_n for instantiating its corresponding sub-process **then**
- 3: Find a set of situations $S = \{s_k, \dots, s_j\}$ from situation specifications, such that they provide $\{p_1, \dots, p_n\}$ as outputs
- 4: **for** each s' in S
- 5: Generate an output action $ch s'(\text{contextType } p_i, \dots, \text{contextType } p_m, \text{bool } S')$

```

6:   end for
7:   Concatenate output actions using “par”
8:   Generate a statement of.  $s\_agent(\text{contextType}$ 
    $p_1, \dots, \text{contextType } p_n)$ 
9:   else
10:    Generate  $s\_agent$ 
11:   end if
12: end for
13: Concatenate statements using  $par$ 
14: Generate a statement of.  $agent_i$  for recursion

```

We will again use the “sea rescue” example to illustrate the above process. Based on the partition results, saw_heli_Agent monitors three situations $withinRange$ (AS2 in Table 4), $canUseHeli$ (CS1 in Table 4), and $readyToDispatchHeli$ (CS2 in Table 4). Hence, sub-process $withinRange_Agent$ for analyzing situation $withinRange$ 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 Lines 2–10 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:

$$let boolResult = heli$$

$$: withinFlightRange(integerALoc) instantiate P,$$

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 Table 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 s services, the complexity of agent synthesis is $O((p + 2s) \times p)$.*

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_{cs} , 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 $O(x \times (l_{as} + g) \times s)$ steps. For synthesizing sub-processes for y composite situations, it takes $O(y \times (l_{cs} + g \times s))$ steps. For synthesizing sub-processes for z temporal situations, it takes $O(z \times g \times s)$ steps. To synthesize the

main processes, it takes $p \times (e \times p + g \times s)$ steps. Since l_{as} , l_{ics} , g , e are usually small numbers, the total complexity is $O(x \times (l_{as} + g) \times s) + y \times (l_{ics} + g \times s) + z \times g \times s + p \times (e \times p + g \times s) = O((p + 2s) \times p)$. \square

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 5.

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 partitioning and SAW agent synthesis algorithms

Our partitioning and SAW agent synthesis algorithms were implemented using Prolog. A test generation tool,

which randomly generates AS³ logic specifications for SAW, was developed using Java. The inputs of the tool are the numbers of situations, services, methods, contexts and relations. The outputs are specifications of services, situations and relations in AS³ logic. These randomly generated specifications are for evaluation purpose only and without real meanings. Programs were run on a desktop with Pentium D CPU 3.00 GHz and 2 G RAM.

Fig. 6 shows the time comparison of partitioning and synthesizing SAW agents for 10–1000 situations ($LAS = LLCS = 3, 6$ and 10 , respectively) with 20% logical composite situations and 80% atomic and temporal situations. It takes about 1.56 and 2.47 and 4.2 s to partition and synthesize 100 situations when LAS and $LLCS$ are set to 3, 6 and 10, respectively. It takes about 14.3, 24.9 and 38.8 min to partition and synthesize 1000 situations when LAS and $LLCS$ are set to 3, 6 and 10, respectively. The time needed for partitioning and synthesizing situations with bigger numbers of LAS and $LLCS$ can be estimated following the trend captured in our experiments.

Fig. 7 shows the partitioning and agent synthesis time for 80 situations containing 1/3 logical composite situations with $LLCS = 3$, and 2/3 temporal situations and atomic situations with $LAS = [1, 15]$. It takes about 2.5 s to partition and synthesize 80 situations with 1/3 situations being logical composite situations and $LAS = 15$. Fig. 8 shows the partitioning and agent synthesis time for 80 situations containing 1/3 logical composite situations with $LLCS = [2, 15]$, and 2/3 temporal situations and atomic situations with $LAS = 2$. It takes about 1.5 s to partition and synthesize 80 situations with 1/3 situations being logical composite situations and $LLCS = 15$.

Table 5
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

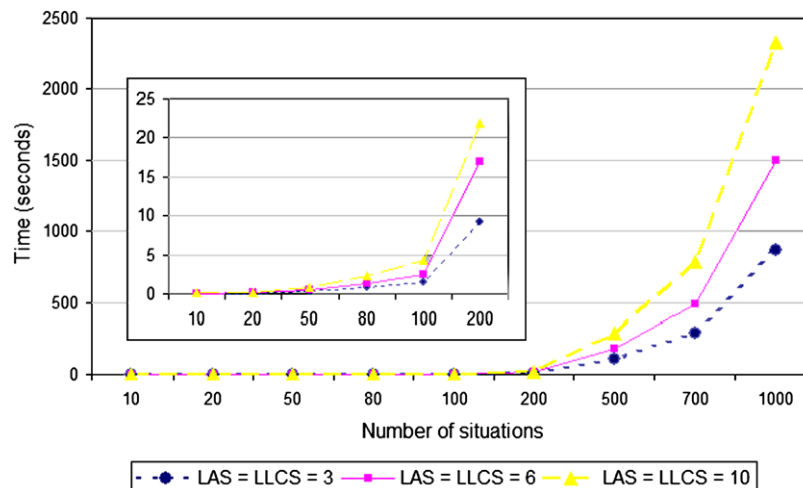


Fig. 6. Partitioning and agent synthesis time for 10–1000 situations.

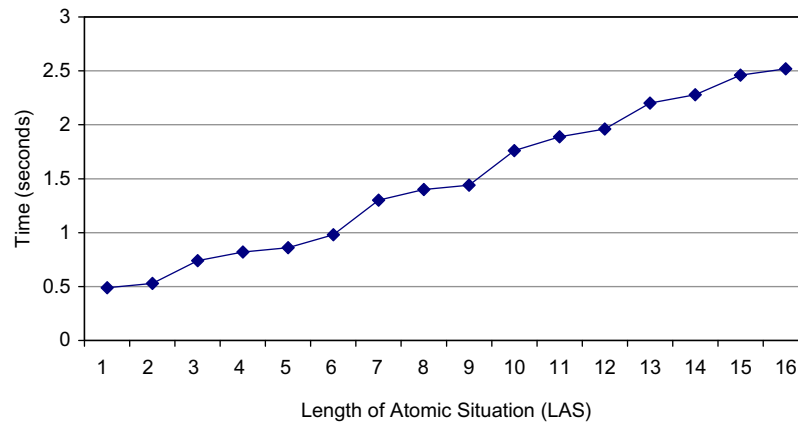


Fig. 7. Partitioning and agent synthesis time for 80 situations with different LAS.

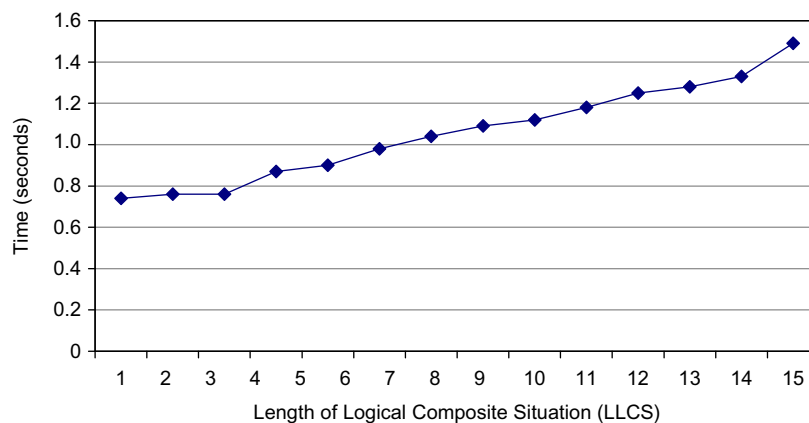


Fig. 8. Partitioning and agent synthesis time for 80 situations with different LLCS.

The above evaluation results show that our partitioning 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 AS³ 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 AS³ logic specifications. An algorithm for partitioning the set of specified situations to decompose the situation analysis tasks to distributed SAW agents 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 partitioning and agent synthesis algorithms are efficient. However, so far, the SAW agents are only capable of analyzing truth-value based situations. Preliminary algorithms for automated consistency and redundancy checking of SAW specifications have been developed and are under evaluation. The special system service is centralized and subjects to single-point failure. Future work includes consistency and redundancy checking on SAW specifications, development of decentralized system services for analyzing temporal situations, finding an optimized approach to partitioning the set of specified situations, extensions for handling fuzzy situations, semantic-based context discovery, and privacy protection in SAW.

Acknowledgement

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. In: *Proceedings of the Ninth International*

- Workshop on Future Trends of Distributed Computing System (FTDCS 2003), pp. 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* 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 Transactions on Software Engineering* 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* 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* 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* 31 (1–3), 59–84.
- Matheus, C.J., Kokar, M.M., Baclawski, K., 2003. A core ontology for situation awareness. In: *Proceedings of the Sixth International Conference on Information Fusion*, pp. 545–552.
- May, D., Shepherd, R., 1984. The transputer implementation of occam. In: *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 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*. Ph.D. Thesis, University of Toronto, Toronto.
- Ranganathan, A., Campbell, R.H., 2003. A Middleware for Context-aware Agents in Ubiquitous Computing Environments. In: *Proceedings of ACM International Middleware Conference*, pp. 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* 1 (4), 74–83.
- Sorensen, M.H., Urzyczyn, P., 2006. *Lectures on the Curry–Howard Isomorphism*. Elsevier.
- Thuraisingham, B.M., 1996. *Data management Systems: Evolution and Interoperation*. CRC Press.
- Yau, S.S., Karim, F., Wang, Y., Wang, B., Gupta, S.K.S., 2002a. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing* 1 (3), 33–40.
- Yau, S.S., Wang, Y., Karim, F., 2002. Development of situation-aware application software for ubiquitous computing environments. In: *Proceedings of 26th IEEE International Computer Software and Application Conference*, pp. 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. In: *Proceedings of the 28th Annual International Computer Software and Application Conference (COMPSAC 2004)*, Hong Kong, pp. 452–457.
- Yau, S.S., Huang, D., Gong, H., Davulcu, H., 2005a. Situation-awareness for adaptable service coordination in service-based systems. In: *Proceedings of the 29th Annual International Computer Software and Application Conference (COMPSAC 2005)*, pp. 107–112.
- Yau, S.S., Mukhopadhyay, S., Huang, D., Gong, H., Davulcu, H., Zhu, L., 2005b. 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., 2006a. Automated agent synthesis for situation awareness in service-based systems. In: *Proceedings of the 30th Annual International Computer Software and Application Conference (COMPSAC 2006)*, pp. 503–512.
- Yau, S.S., Huang, D., Gong, H., Yao, Y., 2006b. Support for situation-awareness in trustworthy ubiquitous computing application software. *Journal of Software Practice and Engineering (JSPE)*, 893–921.
- Yau, S.S., Davulcu, H., Mukhopadhyay, D., Huang, S., Yao, Y., Gong, H., 2007a. Adaptable situation-aware secure service-based (AS³) systems. In: Wang, C., King, S., Wachter, R., et al. (Eds.), *Information Security Research: New Methods for Protecting Against Cyber Threats*. Wiley Publishing Inc., pp. 585–596.
- Yau, S.S., Mukhopadhyay, S., Davulcu, H., Huang, D., Gong, H., Singh, P., Gelgi, F., 2007b. Automated situation-aware service composition in service-oriented computing. *International Journal of Web Services Research (IJWSR)* 4 (4), 59–82.
- Yau, S.S., Yao, Y., in press. An adaptable distributed trust management framework for development of secure service-based systems. *Journal of Autonomic and Trusted Computing (JoATC)*.

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 Institute of Electrical and Electronics Engineers (IEEE) 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, and data privacy. 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 received the Ph.D. degree in computer science at Arizona State University, Tempe, Arizona, USA in 2007. Her research interests include situation-aware software development, and ubiquitous computing. She received the BS degree in computer science from Zhejiang University, China She is currently a software engineer at PayPal Inc. Contact her at hgong@paypal.com.

Dazhi Huang is a Ph.D. student in the Department of Computer Science and Engineering at Arizona State University, Tempe, Arizona, USA. His research interests include middleware, mobile and ubiquitous computing, and workflow scheduling in service-oriented computing environments. He received the B.S. degree 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, Tempe, Arizona, USA. 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. candidate in the Computer Science and Engineering Department at Arizona State University. His research interests include distributed systems, software deployment. He received his B.S. degree from Xian Jiaotong University, China, and the M.S. degree from Zhejiang University, China, both in computer science. Contact him at Luping.Zhu@asu.edu.