Automated Agent Synthesis for Situation Awareness in 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 have many applications, such as collaborative research and development, e-business, health care, military applications, and homeland security. In dynamic service-oriented computing environment, situation awareness (SAW) is needed for system status monitoring, adaptive service coordination and flexible security policy enforcement. Furthermore, various application software systems in such environments often need to reuse situational information for providing better quality of service. Hence, to greatly reduce the effort of situation-aware application software development in service-based systems as well as supporting runtime system adaptation, it is necessary to automate the development of reusable and autonomous software components, referred to as SAW agents, for context acquisition, situation analysis and reactive behaviors of the systems. In this paper, an automated agent synthesis approach for SAW in service-based systems is presented. This approach is based on AS^3 calculus and logic, and our declarative model for SAW.

Keywords: Agent synthesis, situation awareness agents, AS^3 logic, AS^3 calculus, service-based systems.

1. Introduction

Service-based systems have been applied in many areas, such as collaborative research and development, e-business, health care, environmental control, military applications and homeland security, due to the capability of enabling rapid composition of distributed applications, regardless of the programming languages and platforms used in developing and running the applications [1]. In these systems, situation awareness (SAW), which is the capability of being aware of situations and adapting the system's behavior based on situation change [2, 3], is often needed for system status monitoring, adaptive service coordination and flexible security policy enforcement [4].

Consider a service-based system, 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 illustrates the importance of SAW in a service-based system, and is used throughout the paper to illustrate our approach:
(0) The rescue center (*rc*) receives an SOS message from a ship (*bs*) indicating that *bs* has an accident and some passengers are seriously injured.
(1) Upon detecting such a situation, *rc* is responsible for locating proper services to rescue the injured passengers.
(2) Based on the *bs*'s location and the status of various rescue services, *rc* notifies a helicopter H_A to pick up the injured passengers and send them to a nearby hospital.
(3) However, before H_A arrives at *bs*, the latest weather report indicates that the wind in the surrounding area is too strong for H_A to fly safely. Hence, H_A returns to its base, and *rc* notifies a nearby medical ship to go to *bs* to provide emergency medical treatment for passengers.

In this example, situational information, such as SOS messages, weather and status of rescue services, is required for *rc* to coordinate the rescue operation. Furthermore, most situational information in this example can be shared with or reused in other applications of this system, such as searching for missing ships or persons.

The sharing and reusing of situational information is a common requirement of many service-based systems, such as Global Information Grid, which require dynamic coordination and adaptation. Hence, it is necessary to provide reusable SAW capability in service-based systems. To greatly reduce the effort of situation-aware (SA) application software development in service-based systems as well as supporting runtime system adaptation, it is necessary to automate the development of reusable and autonomous software components, referred to as SAW agents, for context acquisition, situation analysis and reactive behaviors of the systems.

In this paper, we will present an approach to automated agent synthesis for SAW in service-based systems. Our approach is based on our declarative SAW model [5], and AS^3 calculus and logic for rapid development of Adaptable Situation-Aware Secure Service-Based (AS^3) systems [4]. SAW requirements are analyzed and graphically specified using our SAW model, and automatically translated into declarative AS^3 logic specifications, from which AS^3

calculus terms defining SAW agents are synthesized. We will first briefly introduce our SAW model, and AS^3 calculus and logic. Then, we will present our approach, including service and SAW specifications in AS^3 logic, and the algorithms for synthesizing SAW agents.

2. Current State of the Art

Substantial work has been done on SAW in artificial intelligence, human-computer interactions and data fusion community. The existing approaches may be divided into two categories: One focuses on modeling and reasoning SAW [6-12], and the other focuses on providing toolkit, framework or middleware for development and runtime support for SAW [2, 3, 13-16].

In the first category, Situation Calculus [6] and its variants [8-9] represent dynamical domains. However, the definitions of "situation" used in Situation Calculus and its variants are quite different. McCarthy [6] considers a situation as a complete state of the world, while Reiter *et al.* [9] 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 [8] has been developed for developing applications in the dynamic domains. However, it cannot handle sensing actions and no development support is provided either. Some research has been done on developing conceptual models for SAW, such as a core SAW ontology [10, 11] and CoBrA Ontology [12]. They only focus on how to represent SAW.

In the second category, Context Toolkit [13] provides a set of ready-to-use context processing components for developing context-aware applications. GAIA [14, 15] is a distributed middleware infrastructure aims at providing development and execution support for context-aware applications in ubiquitous computing environment. MobiPADS [16] is a reflective middleware designed to support dynamic adaptation of context-aware services based on which application's runtime reconfiguration is achieved. RCSM [2, 3] provides the capabilities of context acquisition, situation analysis and SA communication management, and a middleware-based SA application software development framework. However, there are no existing approaches to automated synthesis of software components for providing runtime execution support for SAW in service-oriented computing environment.

3. Background

In [4], we presented a method to developing AS^3 systems, which are collections of services, users, processes and resources acting in response to certain situations without violating their security policies. **Fig. 1** shows the architecture of an AS^3 system. In an AS^3 system,

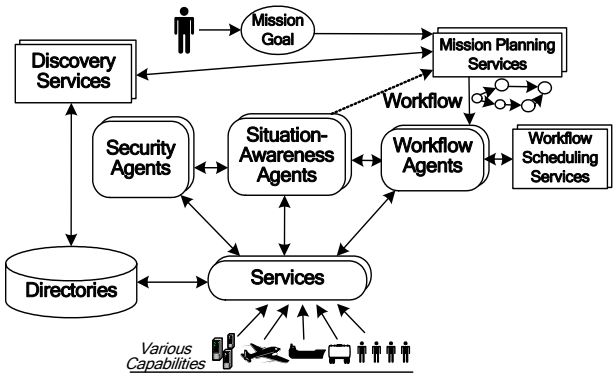

Fig. 1. Architecture of an AS^3 System

organizations publish various 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 fulfilling users' goals based on security policies, situations and available resources. *Workflow Agents* coordinate the execution of workflows based on situational information.

This paper will only focus on discussing our approach to automated agent synthesis for SAW in service-based systems. Our approach is based on a declarative model for SAW, and AS^3 calculus and logic, which are developed for the rapid development of AS^3 systems. Our declarative model [5] formally defines the constructs for SAW, and has well-defined graphical representations for SAW requirements. Hence, it facilitates the developers to rapidly analyze and model SAW requirements. AS^3 calculus provides a formal programming model for AS^3 systems. AS^3 logic is a hybrid normal modal logic [17] for specifying AS^3 systems.

3.1 A Model for SAW

In our SAW model, we have formally defined the essential constructs like contexts, situations and relations among situations and actions [5]. Our SAW model is language-neutral and can be translated to specifications of various formal languages, such as Transaction F-Logic and AS^3 Logic. To facilitate rapid modeling of SAW requirements, we have defined graphical representations for the constructs in our SAW model and developed a supporting GUI tool for developers to model SAW requirements. **Fig. 2** illustrates the graphical representation

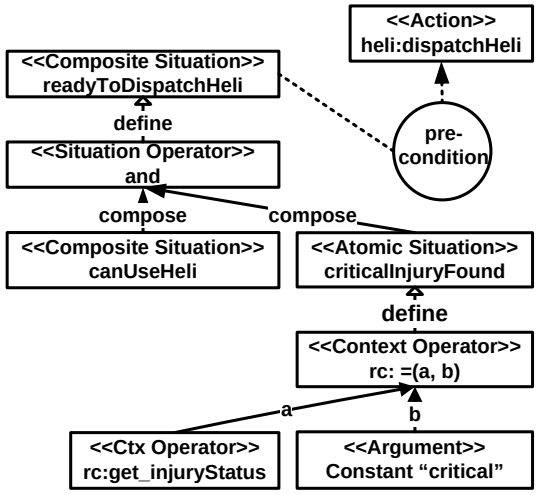

Fig. 2. Partial graphical representations of the example

of situation "ready to dispatch helicopter to accident location" in our "sea rescue" example. Boxes represent the entities in the model. The type of an entity is quoted by "<<" and ">>". Solid lines with arrows connect entities for composition or definition. Circles represent relations among situations and actions. In addition, the entities are associated with various attributes, such as context types and termination conditions of situation analysis. These attributes are required for synthesizing SAW agents.

3.2 AS^3 Calculus and Logic

AS^3 calculus [18] is based on classical process calculus [19]. AS^3 calculus can model timeouts, failures, service invocations, and communications. Part of the syntax of AS^3 calculus used in this paper is shown in **Table 1**. A (recursive) process can be the inactive process, parallel composition of two processes, a nominal identifying a process, or a process performing an external action or an internal computation. Continuation passing [20] is used to provide semantics of asynchronous service invocations. In **Table 1**, $I:l_i(y)\hat{}cont$ denotes the invocation of the method l_i of a service I with parameter y and continuation *cont*. External actions involve input and output actions (as in the ambient calculus [21]) on named channels. Internal computation involves beta reduction, conditional evaluation for logic control, and service invocations.

AS^3 logic has both temporal and spatial modalities for expressing situation information, as well as modalities for expressing communication and service invocation. It provides atomic formulas for expressing relations among variables and nominals for identifying agents. AS^3 logic allows declarative specification of QoS requirements, such as security, SAW and real-time requirements. Models for the logic are processes in the AS^3 calculus. Part of the syntax of AS^3 logic to be used in this paper is shown in **Table 2**. In addition, we can use primitive connectives and modalities in **Table 2** to define the following useful connectives and modalities, which are used to specify services and SAW requirements:

- Eventually: $\mathrm{diam}(\varphi) := E(T\ U\ \varphi)$
- Universal quantification on time: $\forall t\ \varphi := \neg\exists t \neg\varphi$

4. Our Approach

Our approach to automated agent synthesis for SAW in service-based systems consists of the following four steps:

Step (1) Use our SAW model to analyze and model SAW requirements. The process of analyzing and modeling SAW requirements will be described in Sec. 4.1.

Step (2) Translate model representations to AS^3 logic specifications.

Step (3) Synthesize SAW agents in AS^3 calculus terms

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)		
E.P	(external action)	C::=	**//Internal computations**
C.P	(internal computation)	let *x*=D instantiate P	(beta reduction)
P_1 *plus* P_2	(nondeterministic choice)	if *exp* then P else P'	(conditional evaluation)
time t.P	(timeout)		
		D::= $I:l_i(y)\hat{}cont$	(method invocation)

Table 2. Part of the syntax of AS^3 logic

φ_1, φ_2 ::=	formula	$E(\varphi 1\ U\ \varphi 2)$	until
T	true	$E(\varphi 1\ S\ \varphi 2)$	since
U	nominal	$k(u; \varphi)$	knowledge of u
$\mathrm{pred}(x_1,\ldots,x_n)$	atomic formula	$\mathrm{serv}(x;u;\sigma;\varphi)$	invocation of service σ using
$x \sim c$	atomic constraint		input x by φ and returning u
// ~::= > | <| ≤| ≥ | =, c is a natural number		$\exists t\ \varphi$	existential quantification on time
$\varphi 1 \vee \varphi 2$	disjunction	<u> φ	behavior after sending message
$\neg\varphi$	negation	$\varphi 1 \wedge \varphi 2$	conjunction

automatically from AS^3 logic specifications.

Step (4) Compile AS^3 calculus terms into executable Java codes. A compiler for this purpose has been developed to generate executable SAW agents on SINS (Secure Infrastructure for Networked Systems) platform [22].

To facilitate evaluation of situations requiring historical data, four system services have been developed:

a) *appendHistory(SituName, SituData, Timestamp, AgentName)* service stores situational information and removes outdated data. Data related to a situation is co-located with the SAW agent monitoring the situation.

b) *chkSituP(SituName, ω, ε, AgentName)* checks whether the situation was true sometime within [*now*-ω, *now*-ω+ε].

c) *chkSituH(SituName, ω, ε, AgentName)* checks whether the situation was always true within [*now*-ω, *now*-ω+ε].

d) *retrieveRelatedData(SituName, ω, ε, AgentName, Type)* retrieves relevant context data of the situation.

Due to space limitation, the design and implementation of these services are omitted here.

4.1 Specifying SAW Requirements

❖ **Analyzing SAW Requirements**

Based on the ontology of our SAW model [5], developers can analyze the SAW requirements as follows:

(i) Identify the relevant services, and what contexts and context operators are provided by the services.
(ii) Identify situations and the relations among situations and actions (methods provided by services).
(iii) If the identified situations contain situation operators, decompose them to atomic situations.
(iv) The atomic situations are constructed using the identified contexts and context operators.

❖ **Specifying SAW Requirements**

After requirement analysis, developers, without any knowledge about AS^3 logic, can construct the graphical representations of these SAW requirements (see Sec. 3.1), and generate AS^3 logic specifications from the graphical representations using our SAW modeling tool.

The generation of AS^3 logic specifications for SAW requirements is straightforward. The specifications usually contain three parts: service specifications, situation specifications and reactive behavior specifications.

(1) Service specifications: Each method of a service σ is described by the method signature *m(a;b;σ;φ)* and a modality *serv(x;u;σ;φ)* in AS^3 logic as follows:

$$m(a;b;\sigma;\varphi) \rightarrow serv(x;u;\sigma;\varphi)$$

The method signature *m(a;b;σ;φ)* describes a method *m* of service σ with input *a* and output *b*. The modality *serv(x;u;σ;φ)* describes an event indicating that the agent φ invokes service σ with input *x* and returns *u* as output. This formula provides a mapping between service implementations and context operators. For example, the following specification describes a method of service *rc* for collecting the context "injStat":

get_injuryStatus(int(ALoc); str(Stat); rc; saw_rcAgent) $\rightarrow$ *serv(loc(ALoc); injStat(Stat); rc; saw_rcAgent)*

In this specification, the variables (*ALoc* and *Stat*) used in the modality *serv* are typed using their context types (*loc* and *injStat*), whereas the same variables used in the method signature of *get_injurySatus* are typed using the data types (*int* and *str*). This allows developers to map the context types, which are platform-independent and only used for high-level reasoning on SAW, to the actual data types supported by the low-level execution platform.

(2) Situation specifications: For a situation *S*, which is represented by a node N_s in the graphical representations, a formula with the following form will be generated in the AS^3 logic specifications:

$$Definition \rightarrow diam(k([x_1, \ldots, x_n, S], AGENT)), \quad \textbf{(Eq. 1)}$$

where $x_1, \ldots, x_n$ are the related contexts; *AGENT* is the name of the SAW agent monitoring *S*; *k([x_1, ..., x_n, S], AGENT)* denotes *S* and will be used anywhere this situation is referred to; and *Definition* contains a set of operations for collecting related contexts and analyzing *S*. The entire formula means that *AGENT* will eventually have the knowledge about *S* if *AGENT* performs all the operations specified in *Definition*.

Definition of the situation *S* is generated from the sub-graph G_S linked to node N_s through a "definition" arc. By traversing G_S, necessary operations for analyzing *S* are translated to the corresponding AS^3 logic formulas:

(i) For the situation operators in our model [5], $\neg$, $\wedge$ and $\vee$ are provided in AS^3 logic; *P* (sometimes) and *H* (always) are defined using the existential and universal quantifications on time in AS^3 logic; and *Know* is represented using *k(u; φ)*.
(ii) The context operators are translated to AS^3 logic formulas using *serv(x;u;σ; φ)* and atomic constraints.

Attributes defined on situations (see Sec. 3.1), including the frequency of checking situations and the conditions for terminating the execution of an SAW agent, are also described in the specifications. For example, suppose a situation "a seriously injured passenger has been found" is analyzed by *saw_rcAgent* every 10 time units until the situation *rescueSuccess* becomes true. AS^3 logic specification for this situation is as follows:

serv(loc(ALoc);injStat(Stat);rc;saw_rcAgent) $\wedge$ *Stat = 'critical'* $\rightarrow$ *diam(k([loc(ALoc), injStat(Stat), criticalInjuryFound], monitor_until(10, rescueSuccess), saw_rcAgent))*

(3) Reactive behavior specifications: The "trigger" relation in our SAW model describes reactive behaviors of the system. Specification of a trigger relation in AS^3 logic is a simple formula in the following format:

trigger(S, M),

where a method M is triggered when the situation S is true. **Fig. 3** shows partial SAW specifications in the "sea rescue" example.

4.2 Automated Synthesis of SAW Agents

Instead of directly synthesizing SAW agents in platform-dependent programming languages, such as C++, Java and C#, our automated agent synthesis method first synthesizes the AS^3 calculus terms, which define SAW agents. The main advantage of using AS^3 calculus is that AS^3 calculus provides us 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 in the verification of the synthesized agents. Platform-specific compilers can be developed to compile AS^3 calculus terms into executables on different execution platforms. So far, we have developed a compiler to compile AS^3 calculus terms into agents on SINS platform [22]. Due to space limitation, we will not present the compiler here, but focus on the synthesis algorithms of SAW agents in AS^3 calculus terms.

❖ **Defining SAW agents using AS^3 calculus**

Before presenting the synthesis algorithms, let us first examine how SAW agents are defined using AS^3 calculus. **Fig. 3** shows the specifications of an SAW agent, *saw_heliAgent*, in our "sea rescue" example. **Fig. 4** depicts the AS^3 calculus terms of *saw_heliAgent*. The main

```
        /* service specifications */
SERV1)  get_windVelocity([int(ALoc), int(Time)]; int(Vel); rc; saw_rcAgent)
        → serv([loc(ALoc), int(Time)]; windVel(Vel); rc; saw_rcAgent)
SERV2)  withinRange(int(ALoc); bool(Result); heli; saw_heliAgent)
        → serv(loc(ALoc); bool(Result); heli; saw_heliAgent)
SERV3)  backToBase([ ];[ ];heli; SAW_heliAgent)
        → serv([ ];[ ]; heli; saw_heliAgent)
        /* atomic situation specifications */
 AS1)   serv([int(ALoc), int(Time)];windVel(Vel);rc;saw_rcAgent) ∧ Vel < 10
        → diam(k([int(Time), windVel(Vel), lowWindVelocity], monitor_until(10, rescueSuccess), saw_rcAgent))
 AS2)   serv(loc(Aloc); bool(Result); heli; saw_heliAgent) ∧ Result = true
        → diam(k([withinRange], monitor_until(50, rescueSuccess), saw_heliAgent))
        /* composite situation specifications*/
 CS1)   ∀Time  CurrentTime-30 < Time < CurrentTime
        ∧ k([int(Time), windVel(Vel), lowWindVelocity], monitor_until(10, rescueSuccess), saw_rcAgent)
        → diam(k([lowWindForAWhile], monitor_until(-1, rescueSuccess), saw_heliAgent))
 CS2)   k([loc(ALoc), accidentDetected], saw_rcAgent)
        ∧ k([lowWindForAWhile], saw_rcAgent)
        ∧ k([withinRange], saw_heliAgent))
        → diam(k([loc(ALoc), canUseHeli], monitor_until(-1, rescueSuccess), saw_heliAgent))
        /* reactive behaviorspecifications */
 RB1)   trigger(k([loc(Aloc), windVel(Vel), not(canUseHeli)], saw_heliAgent), serv([ ];[ ]; heli; saw_heliAgent))
```

Fig. 3. Part of the SAW specifications in the example

```
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 accidentDetected(integer ALoc, bool S0) . 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     {canUseHeli_Agent(integer ALoc, bool S0)
L14      plus ch rescueSuccess(string Status) . zero }
       ||
L15  fix saw_heliAgent =
L16    { ch accidentDetected(integer ALoc, bool S0) . withinRange_Agent(integer ALoc) } par canUseHeli_Agent) .
L17    saw_heliAgent
```

Fig. 4. An example SAW agent in "sea rescue"

process of *saw_heliAgent* is defined by L15-L17. L16 instantiates two sub-processes (*withinRange_Agent* and *canUseHeli_Agent*) in parallel to analyze two situations (AS2 and CS2 in **Fig. 3**). An input action for collecting the information of "accidentDetected" situation is performed in L16 before instantiating *winthinRange_Agent*. L17 recursively execute the *saw_heliAgent*.

The sub-process *canUseHeli_Agent* is defined by L8-L14. *canUseHeli_Agent* first collects information on the situations "accidentDetected" (S0), "lowWindForAWhile" (S1), and "withinRange" (S2) in L9. Then the result of analyzing the situation "canUseHeli" is sent based on the truth value of S0, S1 and S2 (L10-L12). In addition, a method "backToBase" is triggered in L12.

This example illustrates the following important points on defining SAW agents using AS^3 calculus:

(a) The input and output actions in AS^3 calculus are used to model communications among SAW agents. When an SAW agent determines the truth value of a situation named *S*, it sends all the related contexts and the truth 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 easily get situational information based on the names of situations.

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

(c) The method invocation and conditional evaluation in AS^3 calculus is used to model operations on contexts.

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

❖ **Agent Synthesis Algorithms**

Given a set of SAW specifications, our SAW agent synthesis process consists of the following three steps:

(P1) For each atomic situation, synthesize a sub-process to analyze the situation using ***SynAtom*** algorithm.

(P2) For each composite situation, generate a sub-process to analyze the situation using ***SynComp*** algorithm.

(P3) For each SAW agent, synthesize its main process to initialize the sub-processes for analyzing situations monitored by the agent using ***SynMain*** algorithm.

We will use the example shown in **Fig. 4** to explain the algorithms. The corresponding specifications are in **Fig. 3**.

As specified in **Fig. 3**, *saw_heliAgent* monitors two situations: "withinRange" (AS2) and "canUseHeli" (CS2). Hence, in **(P1)**, a sub-process, *withinRange_Agent*, for analyzing "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; saw_heliAgent)* 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 required and acquired variable lists of (AS2) are also updated. Now, $reqL_2$ = [*loc(Aloc)*], $acqL_2$ = [*bool(Result)*], aL_2 = [*withinRange(int(ALoc); bool(Result); heli; saw_heliAgent)*].

***SynAtom* algorithm:**

0 **For** each atomic situation specification aS_i:
$Def_i \rightarrow k([x_0, \ldots, x_n, aS_i], monitor\_until(f_i, cond_i), A_0)$

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

3 **switch**(T_j)

4 **case** T_j is $serv(I_j;O_j;S_j;A_0)$, 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 **case** T_j is $K(O_j;SM_j;A_0)$, where SM_j is a service name concatenated with a method name, add a an input action to aL_i, and append O_j to $acqL_i$

6 **case** T_j is an atomic constraint, generate an *if-then-else* statement for T_j and append it to aL_i:

6a Generate the constraint evaluation for T_j,

6b Add the output action $ch\ aS_i(x_0, \ldots, x_n,$ ***true***) in the "*then*" branch, and add the output action $ch\ aS_i(x_0, \ldots, x_n,$ ***false***) in the "*else*" branch

6c 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

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

8 Append the following statement for recursion and termination to the end of aL_i:
$(\ time\ f_i.aS_i(req_i)\ plus\ ch\ cond_i(bool\ Status)\ .\ zero)$

Since the second atomic formula *Result = true* in (AS2) matches the case in line 6 of ***SynAtom***, an *if_then_else* statement is generated following (6a-6c). Now, aL_2 = [*withinRange(int(ALoc); bool (Result); heli; saw_heliAgent), 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 6 ends. Since $reqL_2$ contains a 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; saw_heliAgent)* is a beta reduction in AS^3 calculus:

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

where *P* denotes a process of subsequent operations.

In this example, the subsequent operations is the "if-then-else" statement in aL_2, since the variable *Result* used in the "if-then-else" statement is the output from the method *withinRange*. 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 line 8 of ***SynAtom***.

In **(P2)**, a sub-process for analyzing "canUseHeli" (CS2) is generated using ***SynComp*** algorithm.

By scanning (CS2), the following formulas are found:

- *k([loc(ALoc), accidentDetected], saw_rcAgent)*
- *k([lowWindForAWhile], saw_rcAgent)*
- *k([withinRange], saw_heliAgent)*

Hence, the corresponding input actions and condition expressions are generated following (3-4) of ***SynComp***:

Input Actions	Condition Exp.
ch accidentDetected(integer ALoc, bool S0)	S0 = true
ch lowWindForAWhile(bool S1)	S1 = true
ch withinRange(bool S2)	S2 = true

Following (5a-5b) of ***SynComp***, the input actions are concatenated using "**par**", and the subsequent condition evaluation is generated (L9-L12 in **Fig. 4**). Finally, L13-L14 in **Fig. 4** are generated since *monitor_until(-1, rescueSuccess)* is specified in (CS2).

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

Due to space limitation, we only highlight the main idea of ***SynMain***. In ***SynMain***, if a situation monitored by an SAW agent depends on context data collected by other SAW agent, proper input actions will be generated, 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 external variable *ALoc*. By searching the situation specifications, situation "accidentDetected" provides the value of *ALoc*. Hence an input action (L16 in **Fig. 4**) is synthesized to collect *ALoc*. Then the sub-process for analyzing situation "withinRange"(AS2) is instantiated with an input parameter (*ALoc*) (L16 in **Fig. 4**). Similarly, we can also generate the instantiation statement for the sub-process that monitors situation "canUseHeli" (CS2).

Finally, the instantiation statements for the two sub-processes are composed using "par" (L16 in **Fig. 4**). And a recursion statement is added at the end of *saw_heliAgent*.

***SynComp* algorithm:**

0 **For** each composite situation specification cS_i: $Def_i \rightarrow k([x_0, \ldots, x_n, cS_i], monitor\_until(f_i, cond_i), A_0)$

1 **If** cS_i is defined using situation operator *P* or *H*, go to Line 6; otherwise continue to Line 2

2 **For** each formula $k([c_0, \ldots, c_j, S_j], A_j)$ in Def_i,

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

4 Generate a condition expression $condExp_j$ for S_j:

4a **If** S_j is the name of a situation, then generate $(S_j\_result = true)$

4b **If** S_j is in the form not(S_j'), where S_j' is the name of a situation, then generate $(S_j\_result = false)$

5 Concatenate all the input actions using "**par**" or "**plus**", generate the corresponding conditional evaluation as follows and then go to Line 7:

5a For a conjunction ($\wedge$) in Def_i, the corresponding input actions are concatenated using "**par**", and the condition expressions are concatenated using "**and**"

5b For a disjunction ($\vee$) in Def_i, the corresponding input actions are concatenated using "**plus**", and the condition expressions are concatenated using "**or**"

6 **If** cS_i is defined using situation operator *P* or *H*, Def_i has the following form:
$\forall T\ CurrentTime - \varpi < T < CurrentTime - \varpi + \varepsilon \wedge k([c_0, \ldots, c_j, S_j], A_j)$

6a Generate an input action to get the information of S_j

6b Get current time *ts*, generate statement for invoking service $appendHistory(S_j, value\ of\ S_j, ts, A_j)$

6c Generate statement for invoking service $chkSituP(S_j, \omega, \varepsilon, A_j)$ or $chkSituH(S_j, \omega, \varepsilon, A_j)$

6d Generate statement for invoking service $retrieveRelatedData(S_j, \omega, \varepsilon, A_j)$

7 Generate *if-then-else* statements with the generated conditional evaluations, and placed them after all the input actions. Output actions for sending the situation analysis result and actions to be triggered are added on proper branches as in ***SynAtom***

8 Generate statement for recursion and termination as in ***SynAtom***

5. Evaluations

To examine the applicability of our approach to real-world applications, we have evaluated the computational cost of each step in our approach based on analytical results and a case study. A Prolog-based implementation of our agent synthesis algorithms, and a compiler developed using GENTLE, is used in our case study. In the following descriptions, the number of situations used in the definition of a composite situation is referred to as *Composition Factor* (*CF*).

For Step (1), a case study has been done, in which a novice user and an expert user used our SAW modeling tool to model the SAW requirements of an application with 19 services, 10 atomic situations, 5 composite situations (average CF = 2.2) and 21 relations for reactive

behaviors. The times spent by the users are shown in the following table:

	Studying the tool	Modeling			
		Service	Atomic situ.	Comp. situ.	Relation
Novice	35 min.	35 min.	30 min.	8 min.	20 min.
Expert	-	10 min.	10 min.	5 min.	5 min.

Hence, we can estimate that an average user will spend about 1, 2 and 0.5 minutes in modeling a service, an atomic situation and a relation respectively with our tool. The time needed for modeling a composite situation increases as *CF* grows. Since our approach allows reusing defined situations to compose new situations, *CF* can be effectively reduced to a small number.

For Step (2), the complexity of the translation from model representations to AS^3 logic specifications is O(G), where G is the size of the graphical representations for SAW requirements. For Step (3), the complexity of the entire agent synthesis process is dominated by the complexity of ***SynAtom***, which is O(m*(m+n)*x), where m is the number of atomic situations, n is the number of services, and x is the maximum length of the specifications of atomic situations. This shows that our agent synthesis process is efficient (has polynomial time complexity), and has good scalability. In our case study, Steps (2), (3) and (4) together only took less than 10 seconds.

6. Conclusions and Future Work

In this paper we have presented an approach to automated agent synthesis for SAW in service-based systems. Our approach is based on our SAW model and AS^3 calculus and logic. Service and SAW specification format in AS^3 logic is presented. Algorithms for automated synthesis of SAW agents are described. SAW agents synthesized using our approach can analyze atomic situations based on results of various context operations provided in a service-based system, and the logical or temporal composition of situations. Currently, the SAW agents are only capable of truth-value based situation analysis. Future work in this direction includes consistency and redundancy checking on the specification, development of support for agent deployment and agent mobility, and extensions for handling fuzzy situations.

Acknowledgment

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

References

[1] Web Services Architecture. Available at: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[2] S. S. Yau, Y. Wang and F. Karim, "Development of Situation-Aware Application Software for Ubiquitous Computing Environments", *Proc. 26th IEEE Int'l Computer Software and App. Conf.*, 2002, pp. 233-238.

[3] S. S. Yau, et al, "Reconfigurable Context-Sensitive Middleware for Pervasive Computing," *IEEE Pervasive Computing*, vol. 1(3), 2002, pp. 33-40.

[4] S. S. Yau, et al, "Adaptable Situation-Aware Secure Service Based Systems", *Proc. 8th IEEE Int'l Symp. on Object-oriented Real-time distributed Computing*, 2005, pp. 308-315.

[5] S. S. Yau, et al, "Situation-Awareness for Adaptable Service Coordination in Service-based Systems", *Proc. 29th Annual Int'l Computer Software and App. Conf.*, 2005, pp. 107-112.

[6] J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *Machine Intelligence 4*, 1969, pp. 463-502.

[8] H. J. Levesque, et al, "GOLOG: A logic programming language for dynamic domains", *J. Logic Programming*, vol. 31(1-3), 1997, pp. 59 -84.

[9] R. Reiter, *"Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems"*, MIT Press, 2001.

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

[11] C. J. Matheus, et al, "Constructing RuleML-Based Domain Theories on top of OWL Ontologies", *Proc. 2nd Int'l Workshop on Rules and Rule Markup Languages for the Semantic We*b, 2003, pp. 81–94.

[12] H. Chen, T. Finin, and A. Joshi. "An Ontology for Context-Aware Pervasive Computing Environments". *Special Issue on Ontologies for Distributed Systems, Knowledge Engineering Review*, vol.18, Sep. 2003, pp. 197-207.

[13] A.K. Dey, and G.D. Abowd, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-aware Applications", *Human-Computer Interaction*, vol. 16(2-4), 2001, pp. 97-166.

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

[15] A. Ranganathan, and R.H. Campbell, "A Middleware for Context-aware Agents in Ubiquitous Computing Environments", *Proc. ACM Int'l Middleware Conf.*, 2003, pp. 143-161.

[16] A.T.S. Chan, and S.N. Chuang, "MobiPADS: a Reflective Middleware for Context-aware Computing," *IEEE Trans. on Software Engineering*, vol. 29(12), 2003, pp. 1072-1085.

[17] P. Blackburn, M. deRijke, and Y. Venema, *Modal Logic*, Cambridge University Press, 2003.

[18] S. S. Yau, et al., "Automated Agent Synthesis for Situation-Aware Service Coordination in Service-based Systems," *Technical Report, ASU-CSE-TR-05-008*, August, 2005. http://dpse.eas.asu.edu/AS3/papers/ASU-CSE-TR-05-009.pdf

[19] A. J. R. J. Milner, *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[20] A. Appel, "Compiling with Continuations." Cambridge University Press, 1992

[21] L. Cardelli and A. D. Gordon, "Mobile Ambients," *Theoretical Computer Science*, vol. 240(1), 2000, pp. 177-213.

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