Automated Agent Synthesis for Situation-Aware Service Coordination in Service-based Systems

[1]S. S. Yau, [2]S. Mukhopadhyay, [1]D. Huang, [1]H. Gong, [1]H. Davulcu and [1]L. Zhu

[1]*Arizona State University, Tempe, AZ 85287-8809, USA*
[1]*{yau, dazhi.huang, haishan.gong, hasan.davulcu, luping.zhu}@asu.edu*

[2]*West Virginia University, Morgantown, WV, USA*
[2]*supratik.mukhophadyay@mail.wvu.edu*

Abstract

Service-based systems have many applications, including collaborative research and development, e-business, health care, environmental control, military applications, and homeland security. Situation-aware service coordination is required for these systems to coordinate distributed activities under changing environment and workload. In this paper, an automated agent synthesis approach for situation-aware service coordination in service-based systems is presented. This approach is based on AS^3 logic and calculus, and our declarative model for SAW, which are also presented in this paper. This approach consists of three major steps: (1) analyzing situation-awareness (SAW) requirements using our declarative model for SAW and mapping the model representations to logical specifications in AS^3 logic, (2) automated synthesis of AS^3 calculus terms that define SAW agents for situation-aware service coordination, and (3) compilation of AS^3 calculus terms to execution components on an agent platform. An example is presented to illustrate our approach.

Keywords: Automated agent synthesis, situation-awareness agents, AS^3 logic, AS^3 calculus, situation-aware service coordination, service-based systems.

1. Introduction

Service-based systems, which are based on Service-Oriented Architecture [1], have the major advantage of enabling rapid composition of distributed applications, regardless of the programming languages and platforms used in developing and running the applications. Service-Oriented Architecture has been adopted in many distributed systems, such as Grid and Global Information Grid (GIG) [2], in various application domains, including collaborative research and development, e-business, health care, environmental control, military applications and homeland security. In these systems, various capabilities are provided by different organizations as services, which are software/hardware entities with well-defined interfaces to provide certain capability over wired or wireless networks using standard protocols, such as HTTP and SOAP (Simple Object Access Protocol).

Users of service-based systems can locate suitable services in the systems and invoke these services to achieve their goals. It is quite often that achieving a user's goal requires the invocation of multiple services following a specific *workflow*, which is a series of cooperating and coordinated activities. In our MURI project, "Adaptable Situation-Aware Secure Service-based (AS^3) Systems", we are developing a declarative unifying logic-based approach to developing service-based systems with situation-awareness, distributed security policy management and enforcement, and adaptive workflow management while preserving overall correctness and consistency of the systems [MURI_Part_I].

In this paper, we will focus our discussion on situation-aware service coordination in service-based systems to provide more detail information about the logical foundation of our approach and situation-awareness in service-based systems. Security and other important issues for AS^3 systems will be discussed in other papers of ours. *Service coordination* is required to ensure the correctness of workflow execution, and is a process of locating participant services, monitoring their status, invoking proper services, and propagating necessary information among them to ensure the correct results obtained from the coordinated participant services. In order to provide robustness and adaptability, service coordination needs to be situation-aware because services may be unavailable or cannot provide desirable QoS due to distributed denial-of-service attacks, system failures or system overload, and because workflows may need to be adapted when the situation changes in order to satisfy the requirements. We consider *situation-awareness (SAW)* as the capability of being aware of situations and adapting the system's behavior accordingly [3, 4]. A *situation* is a set of contexts in an application over a period of time that affects future system behavior [3, 4]. A *context* is any instantaneous, detectable, and relevant property of the environment, the system, or users, such as location, available bandwidth and a user's schedule [3, 4]. In service-based systems, autonomous software agents are often used for service coordination. These agents can monitor the status of services, communicate among themselves to exchange service requests and results, and invoke appropriate services in a coordinated manner.

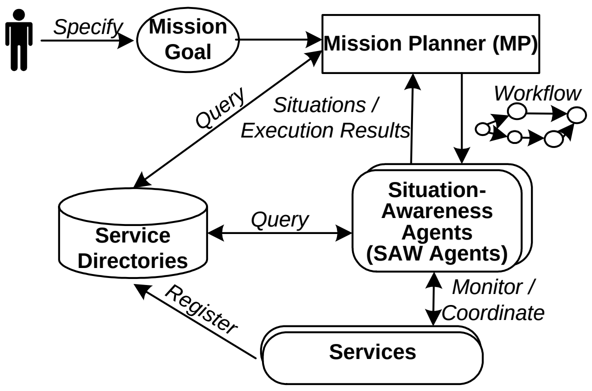


Figure 1. Situation-aware service coordination

In AS^3 systems, situation-aware service coordination is supported by a mission planner (MP) and situation-awareness (SAW) agents. As shown in Figure 1, a mission planner (MP) accepts a mission goal specification from a user, which includes both functional and SAW requirements for the goal, and finds a workflow for the specified mission goal based on available services in the service directories and current situation. SAW agents are automatically synthesized from the workflow, and each agent will monitor the execution of the workflow, assess the situation changes, and coordinate the service invocations accordingly to ensure that the specified mission goal is achieved.

In this paper, we will first introduce AS^3 logic and calculus developed in our MURI project [5], which are the logical foundations of our approach. We will also present a declarative model for SAW and a mapping between the model representations and AS^3 logic specifications. Our declarative model will be used by developers to analyze SAW requirements, including situations of interest to a user's mission goal, services to be invoked under certain situations, and requirements for workflow adaptation due to possible situation changes. Then we will present an automated agent synthesis approach to generating SAW agents for situation-aware service coordination in service-based systems, and use an example to illustrate our approach. Our approach consists of the following three major steps:

(1) Using our declarative model for SAW to analyze SAW requirements, and mapping the model representations to AS^3 logic specifications.

(2) Automated synthesis of AS^3 calculus terms that define SAW agents for situation-aware service coordination. The MP shown in Figure 1 performs workflow planning by finding a proof for the specified mission goal using the proof system of AS^3 logic. AS^3 calculus terms defining SAW agents are directly synthesized from the proof,

(3) Compilation of AS^3 calculus terms to execution components on an agent platform.

2. Current State of the Art

Situation-awareness has been studied in artificial intelligence, human-computer interactions and data fusion community. Existing work on situation-awareness may be divided into two categories. One focuses on modeling and reasoning SAW [6-11], and the other focuses on providing toolkit, framework or middleware for development and runtime support for SAW [3, 4, 12-15]. In the first category, Situation Calculus and its extensions [6-9] were developed for describing and reasoning how actions and other events affecting the world. A situation is considered as a complete state of the world, which cannot be fully described and leads to the well-known frame problem and ramification problem [7]. A core SAW ontology [10, 11] refers a situation as a collection of situation objects, including objects, relations and other situations. However, it does not address how to verify the specification and perform situation analysis. In the second category, Context Toolkit [12] provides architectural support for context-aware applications, but it does not provide analysis of complex situations. GAIA [13, 14] provides context service, space repository, security service and other QoS for managing and interacting with active spaces. MobiPADS [15] is a reflective middleware designed to support dynamic adaptation of context-aware services based on which application's runtime reconfiguration is achieved. Reconfigurable Context-Sensitive Middleware [3, 4] provides the capabilities of context data acquisition, situation analysis and situation-aware communication management, and a middleware-based situation-aware application software development framework. However, they do not support situation-aware service coordination with formally proven properties due to lack of formal framework for service-based systems in their approaches.

Substantial work has been done on service coordination [16-20]. Industrial standards, such as WS-Coordination [16] and WS-CF [17], aim at providing standard and extensible coordination frameworks to support coordinated workflows on web services, but do not address SAW in service coordination. In [18], a formal specification framework [14] was developed for modeling dynamically changing contexts and rules in reactive systems to coordinate distributed systems. MARS [19] promotes context dependent coordination by incorporating programmable coordination media in distributed systems. EgoSpaces [20] introduced a coordination model and a middleware for specifying and managing agent-centered contexts to facilitate easy application development in mobile ad hoc environments. These approaches only use current contexts in service coordination and do not consider variations of contexts over a period of time, which are important information for service coordination.

3. AS^3 Calculus and Logic

AS^3 calculus and logic are developed for rapid development of Adaptable Situation-Aware Secure Service-based (AS^3) systems with formally proven properties. AS^3 calculus provides a formal programming model for AS^3 systems. The AS^3 calculus can model timeouts and failures, and has well-defined operational semantics that involve interactions of external actions and internal computations. The external actions include communication between processes, leaving and joining groups/domains. The internal computations are method calls of services. The AS^3 calculus allows modeling various QoS requirements and dynamic adaptation at runtime by processes. The AS^3 calculus also has a well-defined equational theory that allows modeling redundancy for fault-tolerance as well as formal reasoning using a simulation relation.

AS^3 logic is a hybrid normal modal logic (in the sense of [21]) for specifying AS^3 systems. The logic has both temporal and spatial modalities for expressing situation information as well as modalities for expressing communication, service invocation, joining and leaving groups/domains. It provides atomic formulas for expressing relations among variables and nominals for identifying agents. The vocabulary of the logic does not include function symbols, but has nominals that identify agents and constant symbols that are interpreted over a domain. Models for the logic are (annotated) processes in the AS^3 calculus. The AS^3 logic allows declarative specification of QoS requirements, such as security, situation-awareness and real-time requirements. A novel proof system of AS^3 logic allows the synthesis of AS^3 calculus terms from declarative specifications in AS^3 logic as well as allowing consistency checking of specifications. The model checking problem for AS^3 logic is decidable for image-finite processes, and hence it allows verification of application-independent properties, such as deadlock freedom.

3.1 AS^3 Calculus

AS^3 calculus is based on classical process calculus [22]. The syntax of AS^3 calculus is shown in **Table 1**. A system can be a recursive or non-recursive recursive process, another system lying in a named domain (e.g., a firewall) or the parallel composition of two (sub) systems (as in classical process calculus [22]). A (recursive) process can be a name restriction, the inactive process, parallel composition of two processes (as in classical process calculus [22]), a nominal identifying a process, a process performing an external action or an internal computation, a timed out process, the failed process, or a service exporting methods (non-services are processes that export zero methods). A service can export some methods for its own use only (private methods) while others can be invoked by any process whatsoever. The methods are described by the preconditions describing the constraints on the inputs accepted by them and post-conditions describing the constraints on the outputs provided by them. Methods are called asynchronously and the process invoking the service does not wait for its return unless the returned value is used. External actions involve synchronous communication as well as joining and leaving groups called domains (as in the ambient calculus [23]). Internal computation involves beta reduction, conditional evaluation for logic control, invocation of (public) methods exported by a named service or private methods exported by the process itself and reconfiguration by method replacement.

For a term T, let T[u/x] denote the term obtained from T by replacing every occurrence of the variable x in T with the term u, in which the variable x does not occur. Structural congruence between processes (canonically for systems) is defined as the smallest congruence relation ≡ that satisfies the following:

- A process is congruent to its alpha-renamed variant..
- For processes P, Q and R, and names n and m, $(P \| Q) \| R \equiv P \| (Q \| R)$, $P\|Q \equiv Q\|P$, $P \| 0 \equiv P$, $P+Q \equiv Q+P$, $(P+Q)+R \equiv P+(Q+R)$ and $(\text{new } n)\ (\text{new } m)\ P \equiv (\text{new } m)\ (\text{new } n)\ P$.
- For two processes P and Q, if $P \equiv Q$ then

1. $C.P \equiv C.Q$	2. $E.P \equiv E.Q$
3. $P\|R \equiv Q\|R$	4. $R\|P \equiv R\|Q$
5. $N[P] \equiv N[Q]$	6. $(\text{new } n)\ P \equiv (\text{new } n)\ Q$

7. Q ≡ P	8. P+R ≡ Q+R

Table 1. Syntax of AS^3 Calculus

(System) S::= fix I=P (recursive process) N[S] (system in a named domain) S||S (parallel composition of two systems)	N ::= X (name variable) n (name)
(Processes) P::= (new n) P (name restriction) 0 (inactive process) P||P (parallel composition) I (identifier) E.P (external action) C.P (internal computation) P_1+P_2 (nondeterministic choice) Fail(I) (failure) catch(I).P (failure handler) time t.P (timeout) $P\{l_1(x_1),\ldots l_k(x_k);\ldots l_n(x_n)\}$ (method export) $l_1,\ldots l_k$ are private methods that can be invoked by P itself only while $l_{k+1},\ldots l_n$ are public methods that can be invoked by other processes.	**(External actions)** E::= M (Domain actions) K (Communication actions) K::= (x) (input) <M> (output) M::= *in* N (enter a domain) *out* N (exit a domain) *open* N (open a domain) M.M' (concatenation of actions) ε (no action)
(Internal computations) C::= let x=C instantiate P (beta reduction) if C(x) then P else P' (conditional evaluation) $I{:}l_i(y)$ (method invocation) $I{:}l_i \leftarrow I'{:}l_j$ (method replacement) P (constraint evaluation) C.C (concatenation) ε (no-computation) tt (constant true) ff (constant false) $\perp$ (failed computation) $I{:}l_i$= pre::post(x_i) (method definition)	

The following restrictions are imposed on processes:
1. Recursive processes are guarded
2. Parallel composition through recursion is not allowed (similar to Pi-calculus [24])

A simple type system can be designed to check for well-formedness of processes. These restrictions ensure that every process is image finite, which means that a closed process term (a process term without free variables) can only evolve (in zero or more steps) into finitely many non-congruent process terms using the reduction rules.

In this paper, we do not provide the full operational semantics of AS^3 calculus. Instead, we illustrate the service invocation in its operational semantics. The operational semantics of the service invocation is given by the following rules.

$$\frac{I{:}l_i = \text{pre::post}(x_i)}{I{:}l_i(y) \rightarrow \text{pre::post}(x_i)[y/x_i]} \qquad \text{(service invocation 1)}$$

$$\frac{\text{pre}[y/x_i]\to \text{tt},\quad x_i \text{ free in pre}}{\text{pre::post}(x_i)[y/x_i]\to \text{post}[y/x_i]}$$ (service invocation 2)

$$\frac{\text{pre}[y/x_i]\to \text{ff},\quad x_i \text{ free in pre}}{\text{pre::post}(x_i)[y/x_i]\to \perp}$$ (service invocation fail)

In the above for an expression φ, we write φ(x) to denote that the variable x is free in φ The basic idea behind the above rules is that when a method of a service is invoked with input y, the pre and post conditions are beta-reduced (service invocation 1). If the pre-condition beta-reduces to true, the result of the method invocation is the beta reduct of the post-condition (service invocation 2) once the call returns. Otherwise, the method invocation fails (service invocation fail).

3.2 AS^3 Logic

AS^3 logic is a normal hybrid modal logic [21]. Assume that we are supplied with a countable set of names ***N***. Let m, n range over ***N***, and $\{T_1,\ldots, T_k, \text{IN}, \boldsymbol{N}\}$ be a set of types, where IN denotes the set of natural numbers. Let *Sig* = $\{T_{i1} \times\ldots\times T_{il} \mid \forall j{:}T_{ij} \in \{T_1,\ldots,T_k, \text{IN}, \boldsymbol{N}\}\}$ denote the set of all product types. Assume that there exists a countable set of variables. Let X denote a set of variables, and x, y range over ***X***. A function *type*: $\boldsymbol{X} \to \{T_1,\ldots,T_k, \text{IN}, \boldsymbol{N}\}$ assigns each variable a type. Let V be a vocabulary consisting of a finite set of constant symbols and a finite set of predicate symbols. Let p, q range over the predicate symbols in V. A function *arity* assigns each predicate symbol p ∈ V a natural number, called the arity of p. Each predicate symbol p ∈V is equipped with a signature from *Sig*. Let $\{U_1,\ldots,U_m\}$ be a set of nominals or identifiers. The syntax of AS^3 logic is given by the following BNF in **Table 2**.

Table 2. Syntax of AS^3 logic

η	a name or a variable with type *N*
φ1, φ2 ::=	formula
T	true
0	inactivity
U	nominal
n	name
pred($x_1,\ldots,x_n$)	atomic formula
x ~ c	atomic constraint
// ~::=> | <| ≤| ≥, c is a natural number	
φ1 ∨ φ2	disjunction
┐φ	negation
E(φ1 ***U*** φ2)	until
E(φ1 ***S*** φ2)	since
Θ φ	somewhere
φ1|| φ2	parallel composition
η[φ]	firewall match
K(u; φ)	knowledge of u
serv(u; φ)	service invocation returning u
∃m φ	existential quantification over names
∃ U φ	existential quantification over nominals
∃t φ	existential quantification over time
in(n) φ	behavior on entering firewall
out(n) φ	behavior on leaving firewall
<u> φ	behavior after sending message

In the above table, we assume that the type(x) = IN and the types of the variables $x_1,\ldots,x_n$ conform to the signature of p in pred($x_1,\ldots,x_n$). The term u stands for constants and c ∈ IN. Here, we only give an intuitive explanation of the most important formulas. In the sequel, we will use the term process and agent interchangeably. Intuitively, the nominals act as identifiers to processes. The name n refers to a firewall named n. The knowledge formula intuitively says that a process learns the individual referred to by the constant u from another process and then satisfies φ. The modality serv(u; φ) indicates that a process receives the individual referred to by the constant u as the result of a service invocation and then satisfies φ. The formula in(n)φ (resp. out(n)φ) describes the behavior of the process after entering (resp. exiting) a firewall named n. The formula <u>φ describes the behavior of a process after sending out the individual referred to by the constant named u. The AS^3 logic is a hybrid modal logic in the sense that nominals and names that refer to processes and firewalls respectively form primitive formulas. The following connectives and modalities can be defined in terms of the primitive connectives and modalities defined above:

$\Diamond\varphi := E(\mathbf{T}\ \boldsymbol{U}\ \varphi)$	eventually
$\Box\varphi := \neg\Diamond\neg\varphi$	globally
$\boldsymbol{P}\varphi := E(\varphi\ \boldsymbol{S}\ \mathbf{T})$	past
$\forall t\ \varphi := \neg\exists t\neg\varphi$	universal quantification over time
$\forall m\ \varphi := \neg\exists m\neg\varphi$	universal quantification over name
$\forall U\ \varphi := \neg\exists U\neg\varphi$	universal quantification over nominals
$\boldsymbol{\alpha}\ \varphi := \neg\Theta\neg\varphi$	everywhere
$E\ next(\varphi) := \neg(\exists U, \Diamond U \wedge U \rightarrow \Diamond\varphi) \wedge \Diamond\varphi$	next action

We do not describe the full semantics of the AS^3 logic in this paper, but illustrate the salient features below. AS^3 logic formulas are interpreted over systems or processes decorated with atomic formulas:

S ⊨ I if S is the system fix I=P
S ⊨ <u> φ if S:=fix I=P and there exists Q, R,S,T P≡<u>Q, R≡ (x).S,T≡ P||R and Q ⊨ φ
S ⊨ pred(u1,…,un) if S is decorated with pred(u1,…,un)
S ⊨ in(n) φ if S:=fix I=P and there exists Q, n, R, T, P≡ in n.Q, Q ⊨ φ @n, T ≡ P || n[R]
P ⊨ K(u;φ) if P ≡(x).Q and Q[u/x] ⊨ φ
P ⊨ serv(u;φ) if P≡ let x=I:l_i(y) instantiate Q for some method l_i exported by some process identified by I and Q[u/x] ⊨ φ

Using AS^3 logic, we can specify services in AS^3 systems using the following template:

method(***variables***; ***serviceName***; ***agent***)
∧ ***isService***(***serviceName***)
→ ◊serv(***results***; ***agent***)

In this template, all the bold-faced italicized terms should be replaced by user-defined terms in the actual service specifications. ***method*** is the name of a method provided by a service identified by ***serviceName***. ***variables*** is a list of variables as inputs to the method. ***agent*** is the agent that will monitor and coordinate this service. ***results*** is a list of outputs ***agent*** recorded after calling ***method***. ***isService*** is a user-defined atomic formula that checks whether a service belongs to a particular service type. ***isService*** is necessary since multiple services providing the same functionality may exist in a service-based system. To simplify the service specification, we consider these services having the same service type, and specify the service type instead of each individual services. For example, multiple ambulance services can be specified as follows:

/* Defining methods of ambulance service */
load_injury(***loc, people***; ***AMB***; ***AMBAgent***)
∧ ***isAmbulance***(***AMB***)
→ ◊serv(**"injuryLoaded"**; ***AMBAgent***)
//A *load_injury* method of the ambulance service is defined.
//This specification says that if AMB is an ambulance service, and AMBAgent calls *load_injury* method with two
//parameters ***loc*** (location) and ***people*** (the injured person to be loaded onto the ambulance, AMBAgent will eventually
//record "injuryLoaded" as the result of calling load_injury.

… …

/* Defining isAmbulance */
isAmbulance(ambCar) //ambCar is an ambulance service
isAmbulance(ambHelicopter) //ambHelicopter is an ambulance service
… …

4. Modeling and Specifying SAW Requirements for Service Coordination

As shown in Section 3.2, AS^3 logic is a very expressive hybrid modal logic, which can be used to specify SAW requirements. However, the syntax and semantics of the AS^3 logic are very complicated, and hence it is difficult for developers to write specifications directly in the AS^3 logic. To address this issue, we have developed a declarative model for SAW, and the mappings between our model representations and AS^3 logic specifications. Thus, developers can first generate the model representations for SAW requirements for service coordination in service-based systems and a tool that is currently under development in our project on AS^3 systems will automatically map the model representations to AS^3 logic specifications. In this subsection, we will introduce our declarative model for SAW, and the mappings from our model representations to AS^3 logic specifications.

❖ **Our declarative model for SAW requirements for service coordination**

We consider a *service* as a process, which can accept inputs from other processes and produce outputs. Hence, a *service-based system* can be considered as a collection of parallel processes, each of which can send/retrieve data to/from other processes. Consequently, the service coordination in such a system becomes the coordination of these parallel processes. Our model for SAW requirements for service coordination includes two modeling aspects: (1) situations, and (2) the relations between situations and processes.

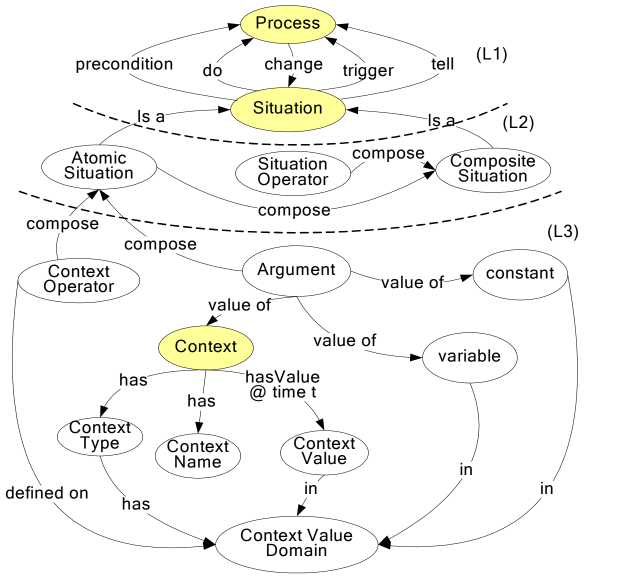


Figure 2. A conceptual view of our declarative model for SAW

Figure 2 shows a conceptual view of our model, which contains three layers. Layer L1 captures four basic relations (precondition, do, trigger, tell) between a process and a situation. It also reflects that execution of a process may change the situation. Layer L2 shows that a situation could either be an atomic situation or a composite situation, which is composed of atomic situations and situation operators. Layer L3 depicts how an atomic situation is constructed by context information and context operators.

Since context acquisition and operations on contexts are highly domain-specific and often involve low-level system processes, our model will not include the ways contexts are collected and the semantics of operations on contexts. Instead, we assume that each context is collected periodically by invoking at least one service in a service-based system, and a service that can collect a context also implement operations for preprocessing this context.

Def. 1: A *context* is a measurable property of the environment, the system or users:

- c_i is the unique name of a context
- $\tau_i = contextType(c_i)$, the context type of c_i.
- $D_i = domainOf(\tau_i)$, the value domain of τ_i.
- Each context has a specific value $v_x \in D_i$ at a particular time t_x in the past, which is represented as $v_x = hasValue(c_i, t_x)$. For simplicity, we use $hasValue(c_i)$ to represent the current value of context c_i.

According to our assumptions, $hasValue(c_i, t_x)$ corresponds to a service call for context acquisition in the real system being modeled. Hence, when we model a real system, we will use the actual service call for context acquisition to represent $hasValue(c_i, t_x)$.

Def. 2: An *argument arg* is one of the following:

- a constant value in a context value domain D_i
- a variable ranging over a context value domain D_i
- $hasValue(c_i, t)$, in which c_i is a context, and t is a time variable.

An *arg* is *bounded* if it is a constant, a variable with a value v_x ($\in D_i$) assigned to it, or $hasValue(c_i, t)$ at a given time t_x.

Def. 3: Given a set of arguments, $\{arg_1, \ldots, arg_n\} \subseteq D_i$, two types of *context operators* are defined as follows:

- Boolean operators: $op_i(arg_1, \ldots, arg_n) = true \mid false$
- Value operators: $op_j(arg_1, \ldots, arg_n) = v \in D_i$

Def. 4: A *term* is either an application of a context operator, $op(arg_1, \ldots, arg_n)$, or a nested application of context operators, $op(term_1 \mid arg_1, \ldots, term_n \mid arg_n)$.

Def. 5: An *atomic situation*, *aSi,* is a term which returns a boolean value and contains only one boolean operator. It can be expressed as follows: *aSi (x1, ..., xm) ≡ op (arg1, ..., argn) | op (term1 | arg1, ..., termn | argn)*, where *op* is a boolean operator, $x_1, ..., x_m$ are all unbounded arguments of *op*, and $term_1, ... term_n$ do not contain boolean operators.

Def. 6: A *composite situation*, cS_i, is composed of other situations and situation operators. Six situation operators: $\neg$, $\wedge$, $\vee$, *P*, *H*, and *Know* are defined . The syntax and semantics of these operators, and how to use them to define composite situations are shown as follows:
1) $cS_i \equiv aS_x$: An atomic situation is a composite situation.
2) $cS_i \equiv \neg cS_x \mid cS_x \wedge cS_y \mid cS_x \vee cS_y$: Negation of a situation and conjunction/disjunction of two situations are composite situations.
3) $cS_i \equiv P(cS_x, \omega, \varepsilon)$: cS_x was true sometime within $[now\text{-}\omega, now\text{-}\omega+\varepsilon]$
4) $cS_i \equiv H(cS_x, \omega, \varepsilon)$: cS_x was always true within $[now\text{-}\omega, now\text{-}\omega+\varepsilon]$
5) $cS_i \equiv Know(cS_x, agent)$: An *agent* knows that cS_x is true
6) All composite situations are defined by recursively applying (1) – (5).

Def. 7: Let s_0 and s_1 be situations, σ and φ processes, and a_0 an action. The following four relations among situations, processes and actions, which are necessary for expressing requirements of situation-aware service coordination in service-based systems, are defined as follows:
1) $precondition(\varphi, s_0)$: s_0 must be true when φ can be executed.
2) $do(\varphi, s_0, s_1)$: The execution of φ makes s_1 true when s_0 is true.
3) $trigger(\sigma, a_0, s_0)$: When σ knows that s_0 is true, σ triggers a_0. This relation models the *reactive behavior* of processes.
4) $tell(\sigma, \varphi, s_0)$: σ sends φ the information about s_0. This relation models the *knowledge sharing* between processes.

Def. 8: A model M for SAW in a service-based system is a tuple (C, T, S, Φ, R, CH), where C is the set of contexts in the system, T is the set of timestamps appeared in the system since the system started to run, S is the set of definitions of the situations in the system, Φ is the set of processes in the system, R is the set of relations between situations and processes defined in the system, and CH is the context history, which consists of values of the contexts in the system over T.

Our model for SAW in service-based systems has strong expressive power for the following reasons:

- Based on Def. 6, our model can express temporal relations among instances of contexts.
- Based on Def. 7, our model allows service providers and developers to define the situations that trigger, allow or prohibit the execution of processes in the service-based system.
- Using the four relations in Def. 7, our model allows modeling control structures, which are commonly used in service coordination.
- Our model can be used to express the situation that timestamped common knowledge [25] is attained among distributed processes. Timestamped common knowledge is very important for the coordination of distributed process.

Based on our model for SAW, developers can perform the following process to analyze the SAW requirements for service coordination in service-based systems, and generate the model representations for SAW requirements:

(i) Among the available services, identify which service is relevant, and what context and context operators are provided by the service, as shown in Layer L3 of Figure 2.
(ii) A service is considered as a process in our model. Identify situations and the relations between situations and processes as shown in Layer L1 of Figure 2.
(iii) If the identified situations contain situation operators, decompose them to atomic situations, as shown in Layer L2 of Figure 2. The atomic situations are constructed based on the identified contexts and context operators in step (i).

To make the usage of our model for SAW easier, we define the following graphical representations of SAW requirements as shown in Figure 3 based on our model for SAW. Figures 3(a), (b) and (c) show the graphical representations for atomic situations, composite situations and relations among situations, processes and actions, respectively.

As shown in Figure 3, we use boxes to represent the various entities in our model. The type of an entity is quoted by "<<" and ">>", and shown at the top of the box representing the entity. We use circles to represent the various relations (Def. 7) in our model. The text in each circle indicates the type of the relation represented by the circle.

A solid arrow is used to connect an argument with a context operator, a context operator with another context operator, a situation (either atomic situation or composite situation) with a situation operator, a process with a situation operator, or a situation operator with another situation operator. A solid arrow indicates that the data returned from the entity at the arrow's starting point is an input to the (context or situation) operator at the arrow's ending point. When an operator takes inputs from several entities, the texts on the solid arrows indicate which input is from which entity.

An arrow with blank ending is used to connect a context operator with an atomic situation, or a situation operator with a composite situation. An arrow like this indicates that the situation at its ending point is defined by the operator at its starting point.

Dotted lines with solid arrow endings are used to connect different entities involved in the four relations in Def. 7.

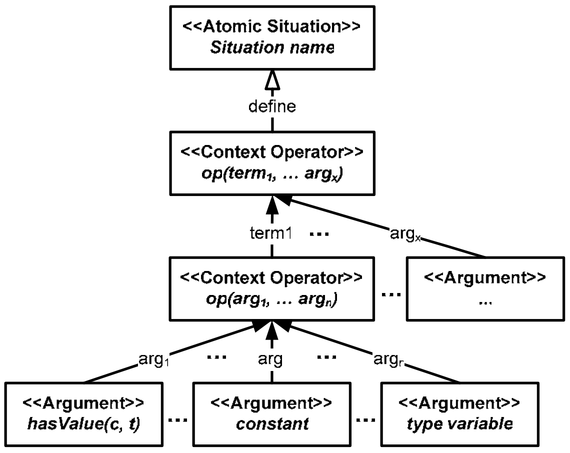


Figure 3(a). Graphical representations for atomic situations

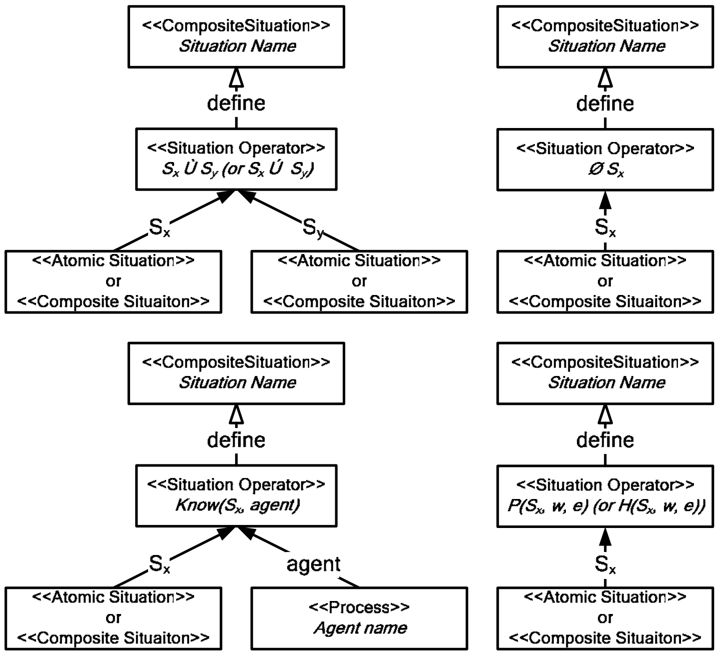


Figure 3(b). Graphical representations for composite situations

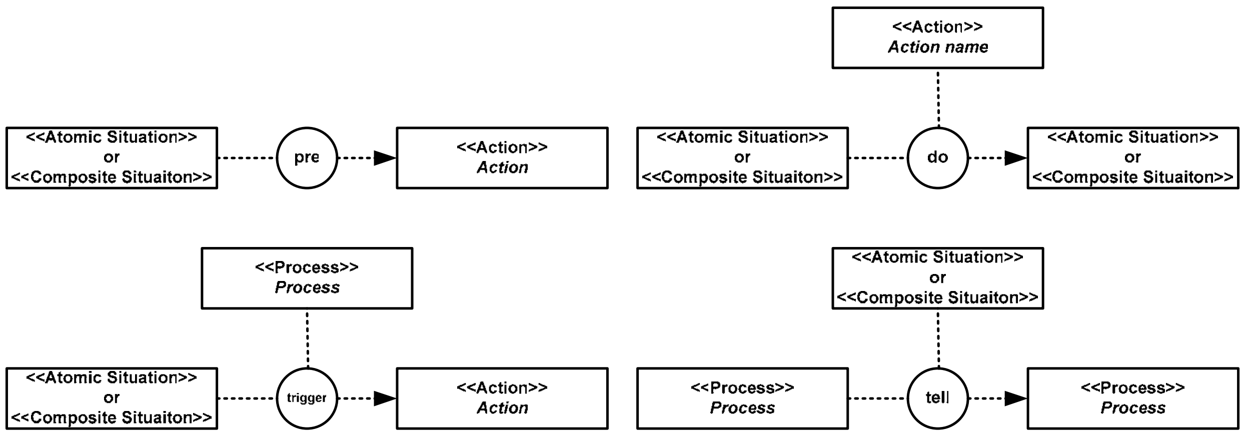


Figure 3(c). Graphical representations for relations among situations, processes and actions

❖ **Mapping the model representations to AS^3 logic specifications**

The model representations generated from Steps (i) – (iii) need to be mapped to AS^3 logic specifications for generating a proof for the mission goal and synthesizing SAW agents. As shown in Tables 3(a) – (d), we have developed the following mapping between our model representations and AS^3 logic specifications. Based on this mapping, a tool is currently under development to provide a GUI for developers to model SAW requirements and automatically map the model representations to AS^3 logic specifications.

Table 3(a). Mapping from our model representations for context & contexts operators to AS^3 Logic specifications

Model Representations	AS^3 Logic Specifications	Explanations
$v_k = op(a_1, \ldots, a_m)$	**method** $(a_1, \ldots, a_m, v_k;$ **service**; **agent** $)$ $\wedge$ **isService(service)** $\rightarrow \Diamond$**serv**$(a_1, \ldots, a_m, v_k;$ **agent**$)$	• v with a subscript (e.g. v_k) stands for result of a context operation • op stands for a context operator (see Def. 4) • a with a subscript (e.g. a_1, a_m) stands for a parameter • As stated previously, context operators (op) in our model are mapped to service invocation in AS^3 systems. Hence, the specification of context operators are the same as service specifications discussed in Section 3.2.

Table 3(b). Mapping from our model representations for atomic situations to AS^3 logic specifications

Model Representations	AS^3 Logic Specifications	Explanations
$aS_i(a_1, \ldots, a_m) \equiv$ (a) $op(arg_1, \ldots, arg_n)$, or (b) $op(term_1 \mid arg_1, \ldots, term_n \mid arg_n)$	(a) **serv**$(a_1, \ldots, a_m, v_k;$ **agent**$)$ $\rightarrow \Diamond K(a_1, \ldots, a_m, v_k,$ **True**, "@situation", aSi_name; **agent**$)$ or (b) **serv**$(a_1, \ldots, a_m, v_k;$ **agent**$)$ $\wedge$ **atomic constraints** $\rightarrow \Diamond K(a_1, \ldots, a_m, v_k,$ **True**, "@situation", aSi_name; **agent**$)$	• aS with a subscript (e.g. aS_i) stands for an atomic situation (see Def. 6) • arg with a subscript stands for a argument (see Def. 3) • term with a subscript stands for a term (see Def. 5) • "@situation" is a string used to represent what the agent eventually knows is about a situation • mapping rule (a) means that if the **agent** records the result v_k of a context operation, the **agent** eventually knows that the situation aS_i is **True** • mapping rule (b) means that if the **agent** records the result v_k of a context operation, and v_k satisfies the **atomic constraints** (see Section 3.2), the **agent** eventually knows that the situation aS_i is **True**

Table 3(c). Mapping from our model representations for composite situations to AS^3 logic specifications

Model Representations	AS^3 Logic Specifications	Explanations
1) $cS_i \equiv aS_x$	1) serv($a_1, \ldots, a_m, v_k$; **agent**) $\rightarrow \Diamond K(a_1, \ldots, a_m, v_k$, **True**, "@situation", aSi_name; **agent**) or serv($a_1, \ldots, a_m, v_k$; **agent**) $\wedge$ **atomic constraints** $\rightarrow \Diamond K(a_1, \ldots, a_m, v_k$, **True**, "@situation", aSi_name; **agent**)	• cS with a subscript (e.g. cS_i) stands for a composite situation (see Def. 7) • $\neg$, $\wedge$, $\vee$, P, H, and Know are situation operators • ω and ε are two constants used to represent a period of time in the past. • t stands for a time variable, t_{now} stands for current time
2) $cS_i \equiv \neg cS_x \mid cS_x \wedge cS_y \mid cS_x \vee cS_y$	2) $\neg cS_x \mid cS_x \wedge cS_y \mid cS_x \vee cS_y$ $\rightarrow \Diamond K(a_1, \ldots, a_m$, **True**, "@situation", cSi_name; **agent**)	• mapping rule 1) means that a composite situation could be defined by a atomic situation
3) $cS_i \equiv P(cS_x, \omega, \varepsilon)$	3) $\exists t$, ($t_{now}-\omega < t < t_{now}-\omega+\varepsilon$) $\wedge$ $cS_x(t)$ $\rightarrow \Diamond K(a_1, \ldots, a_m$, **True**, "@situation", cSi_name; **agent**)	• mapping rule 2) means that if result of logical operations on multiple situations is true, the **agent** eventually knows that composite situation cS_i is **True**
4) $cS_i \equiv H(cS_x, \omega, \varepsilon)$	4) $\forall t$, ($t_{now}-\omega < t < t_{now}-\omega+\varepsilon$) $\wedge$ $cS_x(t)$ $\rightarrow \Diamond K(a_1, \ldots, a_m$, **True**, "@situation", cSi_name; **agent**)	• mapping rule 3) means that if a situation cS_x was ever true within $[t_{now}-\omega, t_{now}-\omega+\varepsilon]$, the **agent** eventually knows that composite situation cS_i is **True**
5)$cS_i \equiv$ Know(cS_x, agent)	5) $<cS_x>$**T** || **agent** $\rightarrow \Diamond K(a_1, \ldots, a_m$, **True**, "@situation", cSi_name; **agent**)	• mapping rule 4) means that if a situation cS_x was always true within $[t_{now}-\omega, t_{now}-\omega+\varepsilon]$, the **agent** eventually knows that composite situation cS_i is True • mapping rule 5) means that if some process outputs cS_x is **True** and behaves like **T** and runs in parallel with **agent**, it is implied that the **agent** eventually knows that composite situation cS_i is **True**

Table 3(d). Mapping from our model representations for relations among situations, actions and processes to AS^3 logic specifications

Model Representations	AS^3 Logic Specifications	Explanations
i) precondition(service$_x$, s$_0$) ii) do(φ, s$_0$, s$_1$) iii) trigger(σ, act$_0$, s$_0$) iv) tell(σ, φ, s$_0$)	i) $\neg s_0 \wedge$ **isService(service$_x$)** $\rightarrow \neg$ method ($a_1, \ldots, a_m, v_k$; **service$_x$**; **T**) ii) $s_0 \rightarrow \Box(\varphi \rightarrow \Diamond s_1)$ iii) $K(s_0; \sigma) \rightarrow$ E next(act$_0$) iv) $K(s_0; \sigma) \rightarrow \Diamond \exists n[<s_0>\sigma \parallel \varphi]$	• s with a subscript (e.g. s$_0$) stands for a situation • service with a subscript (e.g. service$_x$) stands for a service • act with a subscript (e.g. act$_0$) stands for a action • mapping rule (i) means that even a service instance **service$_x$** belongs to a desirable service type, but if situation s$_0$ is not True, a method of **service$_x$** still cannot be invoked. • mapping rule (ii) means that if the situation s$_0$ is true, it is implied that it is always true that the execution of φ will eventually make the situation s1 become true. In other words, s1 is the effect of executing φ in the situation s$_0$. For execution of a process may have different effects in different situations. • mapping rule (iii) means that if a process knows the situation s$_0$ is true and behaves like σ, it is implied that the process currently satisfies act$_0$, i.e. the process starts to execute action act$_0$ when it knows the situation s$_0$ is true. • mapping rule (iv) means if a process knows the situation s$_0$ is true and behaves like σ, it is implied that there exists a domain n, where process σ eventually moves into, and outputs result of s$_0$, and runs in parallel with φ. σ runs in parallel with φ only in a domain means whatever φ receive is from σ, and vice versa.

5. Automated Synthesis of AS^3 Calculus Terms Defining SAW Agents

SAW agents are distributed autonomous software entities, which should have the following capabilities to support situation analysis and service coordination:

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

(C2) **Communication among agents**, including communication with other agents to exchange context and situation information, service status, and requests and responses for service invocation.

(C3) **Context acquisition and situation analysis**, including collecting contexts from its participant services and analyzing situations continuously based on its configuration.

To enable the automated synthesis of SAW agents, we use AS^3 calculus as the programming model for SAW agents, and develop a deductive technique to synthesize AS^3 calculus terms defining SAW agents from AS^3 logic specification automatically. In this subsection, we will first show how the various capabilities of SAW agents can be modeled using AS^3 calculus, and then we will highlight our deductive technique for automated agent synthesis.

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

The various capabilities (C1 – C3) of SAW agents can be modeled using AS^3 calculus as follows:

(C1) Management of participant services by SAW agents is modeled by internal computations and external communications in AS^3 calculus. In service-based systems, collecting information on the status of participant services is done by invoking certain methods provided by participant services. Such service invocation is modeled by the method invocation in AS^3 calculus (see Section 3.1). Similarly, invoking appropriate participant services is also modeled by the method invocation in AS^3 calculus. The conditions that determine when a service should be invoked is modeled by the conditional evaluation and constraint evaluation in AS^3 calculus. For example, suppose we want to model a TicketBooking agent, which will check the Ticketing service of an airline to find out the availability of tickets on the flight XYZ for a particular date, and reserve one ticket if there are still tickets left. Such an agent is modeled using AS^3 calculus as follows:

```
Fix TicketBooking := let x = TicketingService:checkAvailability(flightNo, date) instantiate
            //collecting information from the participant service, "TicketingService"
            if x > 0                    //conditional evaluation
              then TicketingService:reserveTicket(flightNo, date, 1)          //service invocation
              else <flightNo, date, "This flight is booked full." >
```

(C2) Communication among agents can be modeled by the input and output actions in AS^3 calculus. One restriction imposed on communication between two agents is that the agents must be running in parallel, i.e. if **Agent$_1$ = (u).T, and Agent$_2$ = <u>.T**, **Agent$_1$** can receive u only when the calculus term **Agent$_1$** || **Agent$_2$** is satisfied in the system. This restriction is used to enforce access control in the system. For agents not running in parallel, e.g. **Agent$_1$ || n [Agent$_2$]**, they must move into the same domain, e.g. n[**Agent$_1$** || **Agent$_2$**] before they can communicate with each other. Such movement is modeled by the domain actions in AS^3 calculus. For example, assuming Agent$_1$ || n [Agent$_2$] initially. If Agent$_1$ = (u).T and Agent$_2$ = <u>.T, Agent$_1$ and Agent$_2$ cannot communicate. However, if Agent$_1$ = (u).T and Agent$_2$ = out n.<u>.T, Agent$_2$ will first move out of the domain n, and then send out u. And Agent$_1$ will eventually receive u.

(C3) Similar to (C1), service invocation for context acquisition and processing context information by SAW agents is modeled by the method invocation in AS^3 calculus. Situation analysis process in SAW agents is modeled by conditional evaluation, input and output actions and timed out process in AS^3 calculus (see Section 3.1). For example, suppose a SafeGuard agent will periodically send out a message "Secure" if everything is fine. A MonitorAgent, which runs in parallel with the SafeGuard agent and will trigger the alarm service if the situation "No message has been received from SafeGuard agent for more than 300 time units" is true, is modeled as follows:

```
fix MonitorAgent =
    (message).MonitorAgent + time 300.Alarm:alert("noMessageFromSafeGuard").MonitorAgent
```

Here, (message) is the input action for receiving messages from SafeGuard agent, "time 300" is a timed-out process with the timeout set to 300 units of time, and "+" is the non-deterministic choice between the two sub-processes "(message).MonitorAgent" and "time 300.Alarm:alert("noMessageFromSafeGuard").MonitorAgent".

In addition, the recursive process, concatenation and parallel composition of processes, timed out process, conditional evaluation and fail computations in AS^3 calculus can be used to model complex control structures, such as loops, exception handling and conditional branches, for more complex behaviors of SAW agents.

❖ **Automated synthesis of AS^3 calculus terms from AS^3 logic specifications by deduction**

Our deductive technique for synthesizing AS^3 calculus terms from AS^3 logic specifications consists of the following steps:

1. Service specifications in AS^3 logic along with proof rules of the logic form a theory of AS^3 systems. Service specifications are expressed by a set of axioms in AS^3 logic that describe the preconditions for invoking a service as well as the postconditions that constrain the outputs resulting from the invocation. This set of axioms is called the AS^3 theory.
2. Functional requirements of the mission goal along with SAW requirements are specified in AS^3 logic.
3. Synthesis amounts to a proof of the requirements using the AS^3 theory.
4. Calculus terms are directly synthesized from the proof.

For Steps 1 and 2, the specifications for services and SAW requirements can be generated using our approach as shown in Section 4. For Step 3, we have developed a static proof theory for AS^3 logic. The inference system consists of a set of axioms along with the following rules:

- Modus Ponens (MP): $\vdash \varphi \wedge (\varphi \rightarrow \psi) \rightarrow \psi$
- Substitution: There are two substitution rules:

Substitution A: If φ is a valid formula and ψ is a subformula of φ, ψ is an atomic formula (see Section 3.2), and τ is a formula in the AS^3 Logic, then infer $\varphi[\tau/\psi]$

Substitution B: If φ and τ are valid formulas and ψ is a subformula of φ, and $\neg\psi$ is not a formula of φ, then infer $\varphi[\tau/\psi]$

- Generalization: There are two types of generalization rules for modalities and quantifiers respectively.

Generalization A: $\vdash\varphi \rightarrow \Box\varphi$ $\quad\vdash\varphi \rightarrow \Theta\varphi$ $\quad\vdash\varphi \rightarrow K(u, \varphi)$ for any u, $\quad\vdash\varphi \rightarrow$ serv(u, φ) for any u

Generalization B: $\vdash\varphi \rightarrow \forall x\ \varphi$, $\vdash\varphi \rightarrow \forall n\ \varphi$, $\vdash\varphi \rightarrow \forall U\ \varphi$

The set of axioms includes all axioms of propositional normal modal hybrid logic (including the Barcan formula) along with the following axioms:

T1: $\Theta(\sigma \parallel n[\varphi]) \leftrightarrow$ next_hierarchy(σ,φ) (σ and φ are satisfied in adjacent domains)
T2: next_hierarchy(φ,σ)$\rightarrow\Theta\sigma$
T3: $\Theta\Diamond\varphi\rightarrow\Diamond\Theta\varphi$
T4: $\varphi\rightarrow\Theta\varphi$
T5: $\Theta\Theta\varphi\leftrightarrow\varphi$
D1: serv(u, σ) $\rightarrow \neg$serv(u, $\neg\sigma$)
D2: K(u, σ) $\rightarrow \neg$K(u, $\neg\sigma$)

Rule T1 is a shorthand to state that σ and φ are satisfied in adjacent domains. Rule T2 states that if σ and φ are satisfied in adjacent domains then somewhere σ must be satisfied. Rules T3, T4 and T5 are taken from the proof system for the ambient logic in [26]. D1 and D2 are self-duality axioms.

Besides we have the same axioms for domains as in [26]

$(n[]\ \neg 0)$	$n[\sigma] \rightarrow \neg 0$
$(n[] \rightarrow)$	$(\sigma \rightarrow \varphi) \rightarrow (n[\sigma] \rightarrow n[\varphi])$
$(n[] \wedge)$	$n[\sigma] \wedge n[\varphi] \rightarrow n[\sigma\wedge\varphi]$
$(n[] \vee)$	$n[\sigma] \vee n[\varphi] \rightarrow n[\sigma\vee\varphi]$

and the same axioms for quantifier $\forall$ as in [26]

$(\forall \eta \in \Lambda, \sigma\{x \leftarrow \eta\} \rightarrow \varphi) \rightarrow (\forall x.\sigma \rightarrow \varphi)$
$(\sigma\rightarrow\varphi) \rightarrow (\sigma \rightarrow\forall x.\varphi)$

The mission goal specified (as an AS^3 logic formula) by a user or a developer can be achieved if a proof for the specified formula can be found using our proof system. Then, AS^3 calculus terms defining the agents for achieving the mission goal are directly synthesized based on canonical transformation from the proof. The synthesized AS^3 calculus terms can be verified using a model and type checker, which are currently under development, to ensure that a set of application independent properties, such as deadlock free, are satisfied.

❖ **Workflow adaptation**

A workflow may need to be adapted during its execution due to the following reasons:

R1. New constraints for the workflow are identified and submitted by users/developers. In practice, it is likely that users/developers may identify certain missing constraints for the workflow after SAW agents have been synthesized to execute the workflow. Hence, it is desirable that our SAW agents can be rapidly re-synthesized to accommodate the new requirements.

R2. Certain resources or services required for the workflow execution fail. Since it is almost impossible to identify all possible failures and corresponding correction steps before executing the workflow, it is desirable that our system can have the capability to adapt the workflow at run-time to deal with resource or service failures.

R3. It is possible that some steps of the workflow are non-deterministic because they depend on information (situation) which is not available only until the workflow is executing. A solution to this problem provided by traditional workflow planning without requiring runtime adaptation is to plan a workflow with conditional branches, i.e. generate a subworkflow for each possible condition and execute one of them based on information available in runtime. However, this kind of planning is very slow if there are many possible branches, and will waste substantial amount of computation power since a large portion of the generated workflow will not be executed. It will be beneficial if we can postpone the synthesis of these branches until the time that the information determining which branch should be taken is available.

In our MURI project, we are developing a dynamic proof system for AS^3 logic to support workflow adaptation caused by R1-R3. To deal with workflow adaptation caused by R1, the following steps need to be taken:

(1) Model and specify new requirements using our approach in Section 4.
(2) Find a new proof for the mission goal and re-synthesize SAW agents using the dynamic proof system of AS^3 logic. Instead of finding a completely new proof, the dynamic proof system will try to find the proof based on the current situation and the partial result obtained from the executed portion of the workflow, i.e. the dynamic proof system will try to find an alternative way that can reuse the partial results from prior workflow execution to achieve the mission goal.

To deal with workflow adaptation caused by R2, we are incorporating domain-specific dynamic reconfiguration constraints into our approach, which will enforce customizable failure semantics in workflow execution. In our approach, dynamic reconfiguration constraints are modeled as situations under which the workflow execution will produce correct result. The synthesized SAW agents will capture any violation of the dynamic reconfiguration constraints in runtime, and try to adapt the workflow to achieve the mission goal based on the situation at that time by performing partial re-planning using the dynamic proof system of AS^3 logic.

To deal with workflow adaptation caused by R3, we are incorporating a new feature, online planning with workflow template into the dynamic proof system of AS^3 logic, which will first synthesize a workflow that is abstractly defined using certain workflow template, and will fabricate more detailed workflow structure on demand.

6. Compilation of AS^3 Calculus Terms to Execution Components on an Agent Platform

As shown in Section 5, SAW agents that coordinate the services to achieve a user's mission goal are defined by the synthesized AS^3 calculus terms, which encode a workflow (generated by deduction using the AS^3 logic proof system) to fulfill the mission goal. To deploy and execute the workflow, the synthesized AS^3 calculus terms need to be compiled to execution components on an agent platform, such as Secure Infrastructure for Networked Systems (SINS) [27], Aglets [28] and Ajanta [29]. In our approach, the AS^3 calculus terms are compiled to agents running on the SINS platform developed at the US Naval Research Laboratory because of the following reasons:

(1) SINS platform comprises SINS Virtual Machines (SVM) that provide various support for agents running on SVMs, such as instantiating agents and secure and reliable group communication among agents.

(2) Agents running on SINS are specified using Secure Operation Language (SOL) [30] and can be verified and compiled to other programming languages (e.g. Java). SOL is a platform-independent high level synchronous programming language, which has the ability to express a wide class of enforceable safety and security policies and a set of design and analysis tools. In particular, a compiler from SOL to Java has been developed and used to generate the actual codes of the generated agents.

Hence, compiling AS^3 calculus terms to SOL programs is easier without the need of handling any platform dependent low-level details. And the generated programs can be migrated to different systems that support SINS. Currently, we are developing a compiler from AS^3 calculus to SOL.

7. An Example

In this section, we will use an "*Accident Response*" example to illustrate our research. To save space, we use a simplified version of the example in [5], in which Police Department (*PD*), Fire Stations (*FS*) and Ambulance Services (*AMS)* provide various capabilities as services. A service-based system connects and coordinates these services to handle emergency situations. A Mission Planner (*MP*) using our approach described in Section 5 is used to automatically synthesize workflows using these services to fulfill various mission goals.

A *911 call center* gets a report specifying that an accident has happened at location *L* during rush hour. In response to such a situation, the following workflow is automatically generated by the *MP* to coordinate field rescue operations. The goal of the workflow is to rescue injured passengers (if any) and send them to the hospital.

(1) A patrol car provided by *PD* (*PDCar*), a Fire Engine (*FE*) provided by *FD*, and an Ambulance (*AMB*) provided by *AMS* are identified and sent to *L*.

(2) After the *PDCar* arrives at *L*, the police on PDCar setup a perimeter to secure the accident site.

(3) Once the police setup the perimeter, the fire fighters on *FE* start searching and rescuing passengers trapped in the damaged vehicles.

(4) After the passengers are rescued, the paramedics load the injured passengers onto the *AMB* and send them to a hospital.

Table 4 shows the services and methods provided by these services. In our system, five agents, MA, FEAgent, PDAgent, AMBAgent and HELIAgent, will be generated to monitor and coordinate these services respectively. **Figure 4** depicts the AS^3 logic specifications of some services in **Table 4**.

Table 4. Services and their methods in the example

Services	**Methods**	**Functionalities**
911Service	get_status	Return an accident report including the time and location of the accident.
FE	move	Move the FE to a location specified by a parameter of the method and return the location that the FE arrives at.
	search_passenger	Return a list of passengers trapped in the damaged vehicles.
	rescue	Get passengers out of the damage vehicles.
	isStatusOf	Return a boolean value for whether a found passenger is in a certain status
	sameLocation	Return a boolean value for whether a FE is at a location where a status is reported in a accident report
PD	move	Move the PDCar to a location specified by a parameter of the method and return the location that the PD arrives at.
	setup_perimeter	Setup a perimeter at a location specified by a parameter of the method.
	isStatusOf	Return a boolean value for whether a perimeter at a location is in a certain status
AMB	move	Move the AMB to a location specified by a parameter of the method and return the location that the AMB arrives at.
	load_injury	Load the injured passenger onto the AMB.
	send_hospital	Send the injured passenger to the hospital.
	isStatusOf	Return a boolean value for whether a loaded injury is in a certain status
Helicopter (H)	move	Move the H to a location specified by a parameter of the method and return the location that the H arrives at.
	load_injury	Load the injured passenger onto the H.
	send_hospital	Send the injured passenger to the hospital.

```
/* Service specifications */
get_status(aStatus, aLoc, aTime; 911Service; MA)        //MA invokes get_status method of 911Service
∧ is911(911Service)                                     //Service type checking
→ ◊serv(aStatus, aLoc, aTime; MA)                       //MA gets an accident report (aStatus, aLoc, aTime)


setup_perimeter(aLoc; PDCar; PDAgent)                   //PDAgent invokes setup_perimeter method of PDCar
∧ isPoliceCar(FE)                                       //Service type checking
→ ◊serv(aLoc, "perimeterSetup", PDCar; PDAgent)         //PDAgent gets the result of setting up a perimeter at aLoc
```

Figure 4. Part of the service specifications for the example

We will first show how to model the SAW requirements in this example. Then the model representations will be mapped to AS^3 logic specifications based on the mapping table presented in Section 4. Due to the limited space, only parts of model representations and AS^3 logic specification are shown here. We will also show the process of finding a proof for the goal in this example, and the AS^3 calculus terms synthesized from the proof and the corresponding SOL programs. Finally, we will briefly explain how the workflow is adapted in runtime when an additional constraint and a new method in AMB service are added.

❖ **Modeling and specifying SAW requirements in the example**

To illustrate the process of modeling and specifying SAW requirement in the example, we show the generation of the model representations of the SAW requirements for *FE:search_passenger*, *FE:rescue*, *AMB:assess_injury* and *AMB:load_injury* based on the process described in Section 4 as follows:

(1) Identify all situations related to *FE:search_passenger*, *FE:rescue*, and *AMB:load_injury* as shown in Table 5.

Table 5. Situations related to *FE:search_passenger*, *FE:rescue*, *AMB:assess_injury* and *AMB:load_injury*

	Related situations	**Relation among situations, processes and actions**
FE:search_passenger	s_0: FE has already arrived at the accident site, and a police perimeter has been setup at the accident site.	precondition(FE:search_ passenger, s_0) *// s_0 is the precondition of FE:search_passenger*
	s_1: A passenger is trapped in the damaged vehicle.	do(FE:search_ passenger, s_0, $s_1 \otimes s_2 \otimes s_3$) // "$\otimes$" is a shorthand operator that can be // defined as follows: // $a \otimes b \equiv (a \wedge \neg b) \vee (\neg a \wedge b)$ *// $s_1 \otimes s_2 \otimes s_3$ is the effect of FE:search_passenger*
	s_2: A passenger is outside the damaged vehicle.	
	s_3: No passenger is found at the accident site.	
FE:rescue	s_1	trigger(FEAgent, FE:rescue, s_1) *// FE:rescue should be triggered if FEAgent knows that* *// there are passengers trapped in the damaged vehicle*
	S_2	do(FE:rescue, s_1, s_2) *// s_2 is the effect of FE:rescue* tell(FEAgent, AMBAgent, s_2) *// FEAgent tells AMBAgent that a passenger is outside* *// the damaged vehicle*
AMB:load_injury	s_4: AMBAgent knows that a passenger is outside the damaged vehicle.	precondition(AMB:load_injury, s_4) *// s_4 is the precondition of AMB:load_injury*
	s_5: An injured passenger has been loaded onto the ambulance	do(AMB:load_injury, s_4, s_5)

(2) There are two composite situations s_0 and s_4 in (1). The decomposition of these two situations is shown as follows:

$s_0 \equiv s_6 \wedge s_7$ //s_6: FE has arrived at the accident site.
//s_7: A police perimeter has been setup at the accident site.

$s_4 \equiv K(s_2, \text{AMBAgent})$

(3) Identify the relevant contexts and the context operators needed for recognizing these situations:

/* Contexts in situations s_0 to s_7 */

Accident report (aReport): {(aStatus, aLoc, aTime) | aStatus ∊ {"no accident", "car accident"},
aTime is the number of seconds elapsed from Janury 1st, 1900 to the time of the accident,
aLoc is longitude/latitude coordinates of the accident site }

Location of FE (fLoc): longitude/latitude coordinates of FE

Location of AMB (mLoc): longitude/latitude coordinates of AMB

Status of the police perimeter (periStatus): {"perimeterSetup", "perimeterSetupFail"}

Passengers found (pFound): {(id, position) | id is an identifier for a passenger assigned by fire fighters, position ∈ {“trapped”, “outside”, “notFound”} }

Injury loaded (iLoad): {“loaded”}

/* Context operators in situations s_0 to s_7 */
911Service:get_status(*l, t*) returns value of context aReport, which is the status of a location *l* at time *t*
FE:search_passenger(*l*) returns value of context pFound, which are passengers found at the location *l*
FE:isStatusOf(*p, status*) returns a boolean value for whether a found passenger, *p,* is in a certain status
FE:sameLocation(*l, r, status*) return a boolean value for whether a location l, is the same location where in a accident report a status is detected
PD:setup_perimeter(*l*) returns value of context periStatus, which is the status for setting up a police perimeter at the location *l*
PD:isStatusOf(*l*) returns a boolean value for whether perimeter at a location, *l*, is in a certain status
AMB:isStatusOf(*i, status*) returns a boolean value for whether injury loaded, *i*, is in a certain status

Based on the results of steps (1) – (3), the model representations of SAW requirements for *FE:search_passenger*, *FE:rescue*, and *AMB:load_injury* are shown in **Figure 5**. The graphical model representations of of SAW requirements for *FE:search_passenger* is shown in **Figure 6**. The generated model representations are automatically mapped to AS^3 logic specifications using the mapping tables in Section 4. Some of the AS^3 logic specifications for SAW requirements in the example are depicted in **Figure 7**.

/* atomic situations */
s_1 ≡ FE:isStatusOf(FE:search_passenger(FE:get_location()), “trapped”)

s_2 ≡ FE:isStatusOf(FE:search_passenger(FE:get_location()), “outside”)

s_3 ≡ FE:isStatusOf(FE:search_passenger(FE:get_location()), “notFound”)

s_5 ≡ AMB:isStatusOf(AMB:load_injury(*p’*), “loaded”) // *p’* is the passenger with minor injury

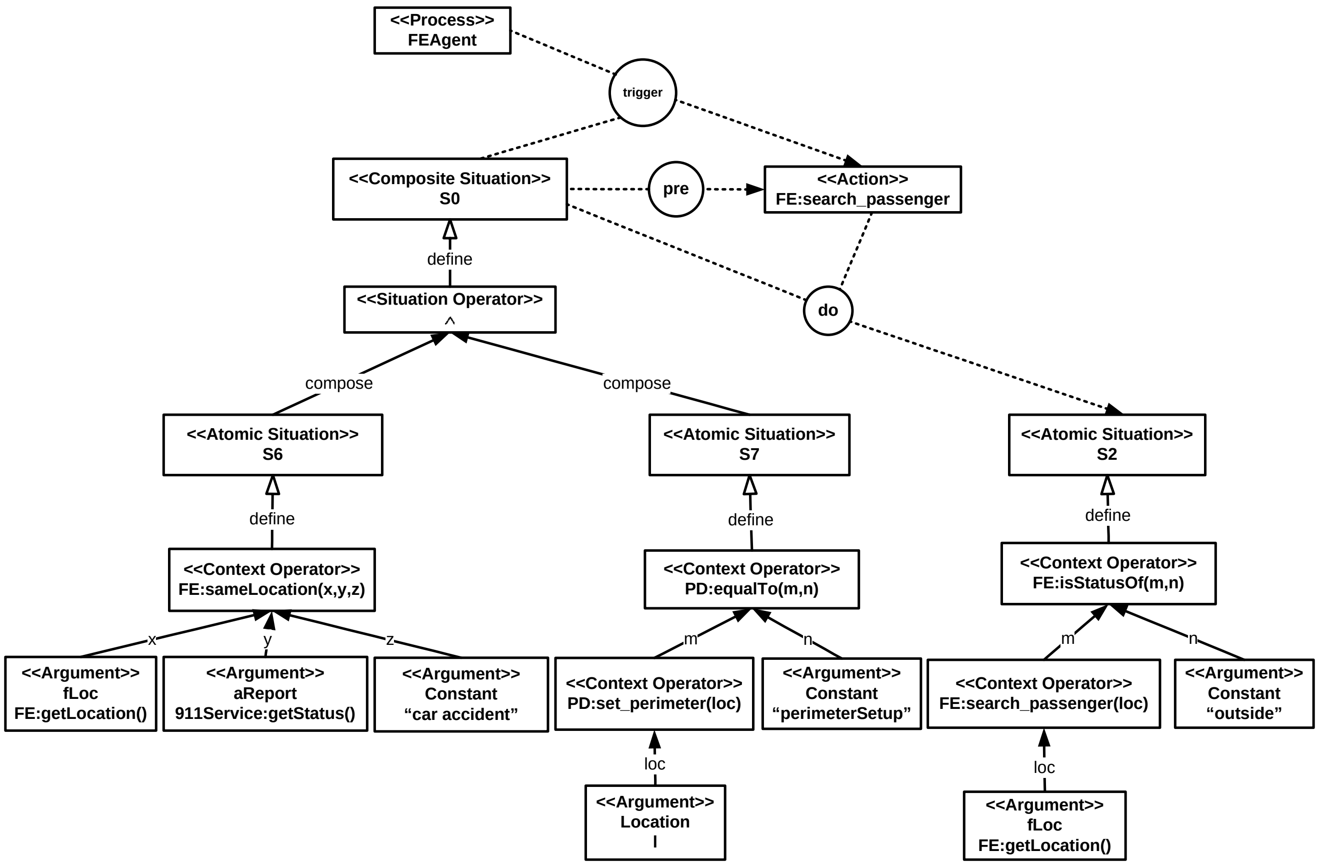
s_6 ≡ FE:sameLocation(FE:getLocation(), 911Service:getStatus(*l*, *t*), “car accident”) // *l* is a location, *t* is a timestamp

s_7 ≡ PD:isStatusOf(PD:setup_perimeter(*l*), “perimeterSetup”) // *l* is the parameter specifying the location of accident

/* composite situations */
$s_0 \equiv s_6 \wedge s_7$
$s_4 \equiv K(s_2, \text{AMBAgent})$

/* relation between situations and actions */
See **Table 5**.

Figure 5. Part of the model representations for the example



<<Process>> FEAgent
trigger
<<Composite Situation>> S0
pre
<<Action>> FE:search_passenger
define
<<Situation Operator>> ^
do
compose
compose
<<Atomic Situation>> S6
<<Atomic Situation>> S7
<<Atomic Situation>> S2
define
define
define
<<Context Operator>> FE:sameLocation(x,y,z)
<<Context Operator>> PD:equalTo(m,n)
<<Context Operator>> FE:isStatusOf(m,n)
x
y
z
m
n
m
n
<<Argument>> fLoc FE:getLocation()
<<Argument>> aReport 911Service:getStatus()
<<Argument>> Constant "car accident"
<<Context Operator>> PD:set_perimeter(loc)
<<Argument>> Constant "perimeterSetup"
<<Context Operator>> FE:search_passenger(loc)
<<Argument>> Constant "outside"
loc
loc
<<Argument>> Location l
<<Argument>> fLoc FE:getLocation()

Figure 6. Part of the graphical model representation for the example

❖ **Automated synthesis of SAW agents**

As discussed in Section 5, the process of synthesizing SAW agents to achieve the mission goal is done by the AS^3 logic proof system. In this example, the mission goal is to send passengers injured in the accident to the hospital. This goal is formally specified in AS^3 logic as follows:

get_status(aStatus, aLoc, aTime; 911Service; MA)
$\wedge$ is911(911Service)
$\rightarrow$ Θ◊K(aLoc, True, "@situation", "rescueSuccess"; MA)

We need to find a proof for this formula based on the service specifications, SAW requirement specifications, and the proof theory of AS^3 logic. We will not show the entire proof process here due to its length, but only show the first step in the proof to illustrate the idea. The proof is based on forward chaining [31].

1. Starting from **get_status(aStatus, aLoc, aTime; 911Service; MA) $\wedge$ is911(911Service)**

    //Matched service specification
    get_status(aStatus, aLoc, aTime; 911Service; MA) $\wedge$ is911(911Service)
    $\rightarrow$ ◊serv(aStatus, aLoc, aTime; MA)

    //A fact stating that 911Service is an instance of 911 call center.
    is911(911Service)

    ◊serv(aStatus, aLoc, aTime; MA)

2. Continue to find a proof from **◊serv(aStatus, aLoc, aTime; MA)** to **Θ◊K(aLoc, True, "@situation", "rescueSuccess"; MA)** using forward-chaining.

```
/* Atomic Situations */

s_2: A passenger is outside the damaged vehicle.
serv(aLoc, id, position, FE; FEAgent) ∧ position = "outside"          //The atomic constraint, position = "outside" is satisfied by
                                                                      //the result of invoking search_passenger
→ ◊K(aLoc, id, position, True, "@situation", "passengerNotTrapped"; FEAgent)  //FEAgent knows the situation that a passenger is
                                                                              //trapped in a damaged vehicle at aLoc
/* s_1 and s_3 can be defined similar to s_2. */
… …

/* Composite Situations */
s_4: AMBAgent knows that a passenger is outside the damaged vehicle.
<aLoc, id, position, True, "@situation", "passengerNotTrapped"> T || AMBAgent
        //A process in parallel with AMBAgent outputs the information on situation s_2
→ ◊ K(aLoc, id, position, True, "@situation", "passengerNotTrapped"; AMBAgent)
        //AMBAgent knows the situation that a passenger is outside the damaged vehicle
… …

/* Relations between situations, processes and actions */
trigger(FEAgent, FE:rescue, s_1): FE:rescue should be triggered if FEAgent knows that there are passengers trapped in the
damaged vehicle
K(aLoc, id, position, True, "@situation", "passengerTrapped"; FEAgent)
→ E next(rescue(aLoc, id; FE; FEAgent)

tell(FEAgent, AMBAgent, s_2): FEAgent tells AMBAgent that a passenger is outside the damaged vehicle
K(aLoc, id, position, True, "@situation", "passengerNotTrapped"; FEAgent)
→ ∃n ◊n[<aLoc, id, position, True, "@situation", "passengerNotTrapped"> FEAgent || AMBAgent]
… …
```

Figure 7. Some AS^3 Logic specifications for SAW requirements in the example

After the complete proof is found, the calculus terms synthesized from the proof are shown in **Figure 8**. These calculus terms define four SAW agents. **Figure 9**depicts how the execution of these SAW agents cooperatively and coordinately carry out the workflow for rescuing any injured passengers when an accident occurred.

```
/* MonitorAgent */
fix MA =
        let (aStatus, aLoc, aTime) = 911Service:get_status() instantiate
        if aStatus = "car accident"
          then <aLoc, aTime, True, "@situation", "accidentDetected">.MA
          else MA

/* PDAgent */
fix PDAgent =
        (l, t, r, "@situation", "accidentDetected").
        if r = True
          then car1:move(l). car1:setup_perimeter(l). <l, True, "@situation", "perimeterSetup">. PDAgent
          else PDAgent

/* FEAgent */
fix FEAgent =
        (l, t, r, "@situation", "accidentDetected").
        if r = True
          then    fe1:move(l).
                  (l1, r1, "@situation", "perimeterSetup").
                  let (id, position)  = fe1:search_passenger(l1) instantiate
                          if position = "passengerTrapped"
                            then    fe1:rescue(l, id).
                                    <l, id, True, "@situation", "passengerNotTrapped">.FEAgent
                            else    if position = "passengerNotTrapped"
                                      then    <l, id, True, "@situation", "passengerNotTrapped">.FEAgent
                                      else    <l, id, True, "@situation", "passengerNotFound">.FEAgent
          else    FEAgent

/* AMBAgent */
fix AMBAgent =
        (l, t, r, "@situation", "accidentDetected").
        if r = True
          then    amb1:move(l).
                  (l, i, s, "@situation", "passengerNotTrapped").
                  if s = True
                    then    amb1:load_injury(l, i).
                            amb1:send_hospital(l, i).
                            <l, True, "@situation", "rescueSuccess">. AMBAgent
                    else AMBAgent
          else AMBAgent
```
Figure 8. Automated synthesized AS^3 Calculus terms of example

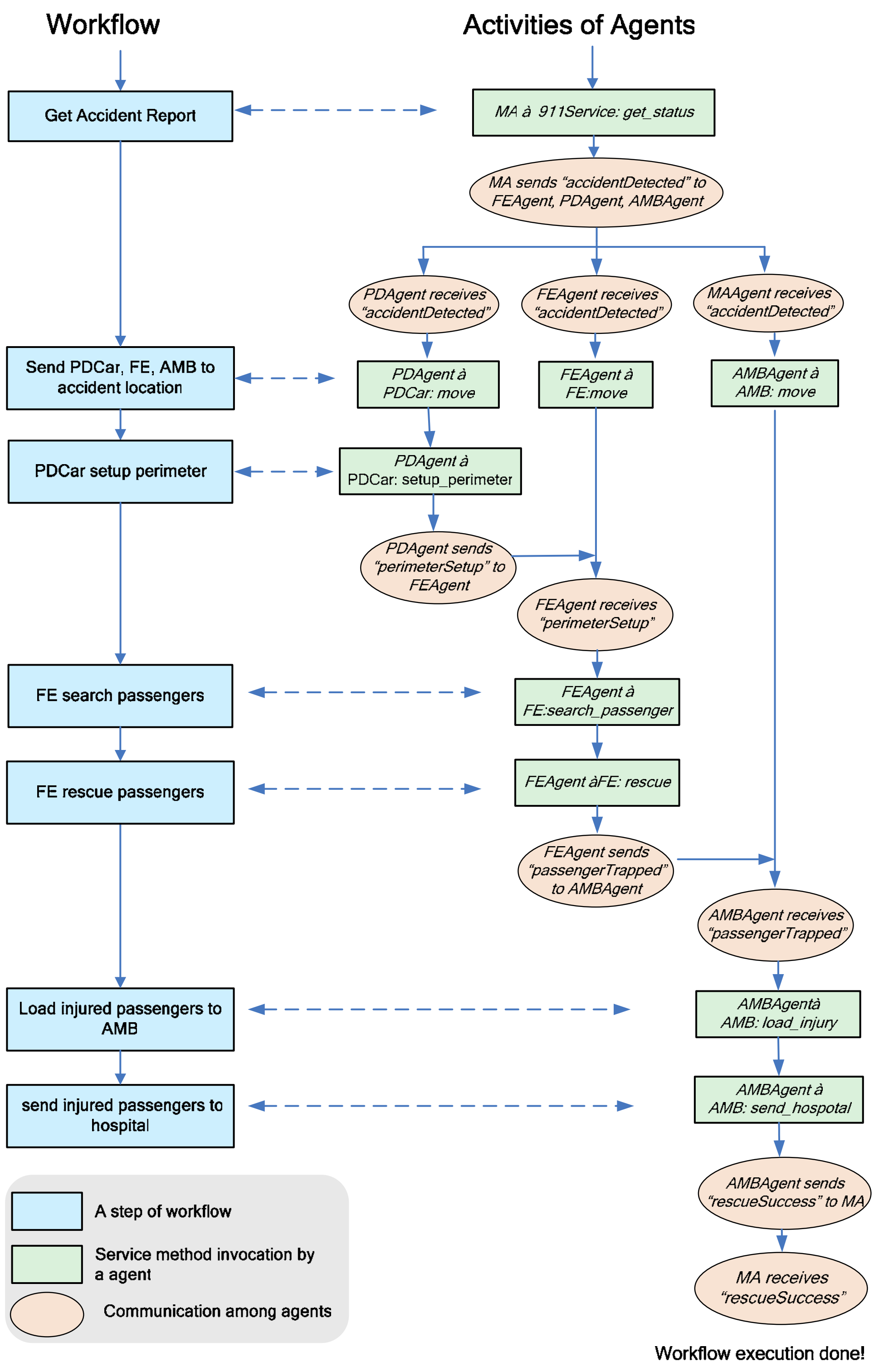

Workflow
Activities of Agents
Get Accident Report
MA à 911Service: get_status
MA sends "accidentDetected" to FEAgent, PDAgent, AMBAgent
PDAgent receives "accidentDetected"
FEAgent receives "accidentDetected"
MAAgent receives "accidentDetected"
Send PDCar, FE, AMB to accident location
PDAgent à PDCar: move
FEAgent à FE:move
AMBAgent à AMB: move
PDCar setup perimeter
PDAgent à PDCar: setup_perimeter
PDAgent sends "perimeterSetup" to FEAgent
FEAgent receives "perimeterSetup"
FE search passengers
FEAgent à FE:search_passenger
FE rescue passengers
FEAgent àFE: rescue
FEAgent sends "passengerTrapped" to AMBAgent
AMBAgent receives "passengerTrapped"
Load injured passengers to AMB
AMBAgentà AMB: load_injury
send injured passengers to hospital
AMBAgent à AMB: send_hospotal
AMBAgent sends "rescueSuccess" to MA
MA receives "rescueSuccess"
A step of workflow
Service method invocation by a agent
Communication among agents
Workflow execution done!

Figure 9. Execution of SAW agents in the example

❖ **Compilation of AS^3 calculus terms to SOL programs**

The synthesized calculus terms are then compiled to SOL programs for actual deployment on SINS platform. **Figure 10** depicts some fragments of a SOL module corresponding to the AMBAgent.

```
// Module for Ambulance Agent
Module AMBAgent {
type definitions
          tString = string;
          tBool = boolean;

monitored variables
          tString ambName;
          tInt accident_latitude;
          tInt accident_longitude;
          tBool result;
          tString situ_identifier;
          tString situ_name;

          tInt amb_latitude;
          tInt amb_longitude;

          tString people;

controlled variables
          tString cServiceInvoke;
          tString cAmbStatus;

definitions
          cServiceInvoke = initially null then
          if {
                    // Call move method when receiving new accident report
                    [] @F(ambName == PREV(ambName)) |
                      @F(accident_longitude == PREV(accident_longitude)) |
                      @F(accident_latitude == PREV(accident_latitude)) |
                      @T(result) |
                      @T(situ_name == "accidentDetected")  -> "move(" + ambName + "," + accident_longitude + "," +
                                                                          accident_latitude+")";

                    // Call load_injury method after fire fighters get the passenger out of the damaged vehicle
                    … …

                    // Call send_hospital method after the rescued passenger has been loaded to the ambulance
                    … …
          }
          cAmbStatus = initially null then
          if {
                    // Update AMB status to show whether the ambulance has arrived at the accident site
                    []amb_longitude == accident_longitude & amb_latitude == accident_latitude &
                      @T(result) & situ_identifier == "@situation" & situ_name == "detectAccident"
                              -> amb_longitude+","+amb_latitude+","+ambName +",True,@situation, arrived";

                    otherwise
                              -> amb_longitude+","+amb_latitude+","+ambName+",False,@situation, notArrived";

                    //Update AMB status to show whether the rescued passenger has been loaded to the ambulance
                    … …
          }
}
```

Figure 10. A SOL module generated from the compilation of AS^3 calculus term for AMBAgent

❖ **Workflow adaptation**

Assume that a new constraint for the workflow in this example is submitted by users and a new method in AMB is provided by developers after the SAW agents has been deployed and started to execute. The new constraint states that if an injured passenger is in critical condition, a helicopter (*H*) service should be used instead of *AMB* to send the passenger to a nearby hospital. In response to this new constraint, AMB service is updated by adding a new method assess_injury, which will return the injury status ("minor injury" or "critical condition") of a passenger found at the accident site. With this new constraint, the original workflow may need to be adapted in runtime, depending on the situation that can only be detected during the workflow execution, i.e. an injured passenger may need to be sent to the hospital by a helicopter if the result of invoking assess_injury shows that the injured passenger is in critical condition. Such an adaptation requires the following tasks to be performed:

1. The new service specification should be added to the original set of service specifications. For this example, the following formula needs to be added:

assess_injury(id; AMB; AMBAgent)	//AMBAgent invokes assess_injury method of AMB
$\wedge$ isAmbulance(AMB)	//Service type checking
$\rightarrow \Diamond$serv(id, injuryStatus, AMB; AMBAgent)	//AMBAgent gets the injury status of the passengers specified by id

The SAW requirements identified earlier in this section need to be modified and the following new SAW requirements need to be added:

//**New contexts**

Injury status	{(id, iStatus) | id is an identifier of a rescued passenger assigned by fire fighters, iStatus $\in$ {"minor injury", "critical condition"}}

AMB:assess_injury(id) returns the injury status of the passenger identified by id.

//**New situations**

s_8: A passenger has suffered minor injury.

$s_8 \equiv$ AMB:assess_injury(id) = (id, "minor injury")

s_9: A passenger is in critical condition

$s_9 \equiv$ AMB:assess_injury(id) = (id, "critical condition")

//Modify the precondition of AMB:load_injury

precondition(AMB:load_injury, s_8)

//Assess_injury should be triggered once AMBAgent knows that a passenger is outside the damaged vehicle.

trigger(AMBAgent, AMB:assess_injury, s_4)

//A helicopter should be notified that a passenger is in critical condition and needs to be sent to hospital by the helicopter

tell(AMBAgent, HELIAgent, s_9)

trigger(HELIAgent, H:move, s_9)

precondition(H:load_injury, s_9)

The original AS^3 logic specifications will be modified and new specifications will be added correspondingly.

2. A new proof needs to be found to satisfy the new constraints. The AMBAgent needs to be re-synthesized to insert the new action "assess_injury", and a new HELIAgent will be synthesized to coordinate the helicopter service, which is not used previously. The new proof will be found by partial re-planning using the dynamic proof system we are currently developing.

8. Conclusions and Future Work

In this paper we have presented an approach to automated agent synthesis for situation-aware service coordination in service-based systems. Our approach is based on our AS^3 logic and calculus developed for Adaptable Situation-aware Secure Service-based (AS^3) systems. A model for SAW requirements, the mapping from the model representations to AS^3 logic specifications, and a deductive synthesis technique of AS^3 calculus terms defining SAW agents that enable adaptive coordination in service-based systems have been presented. For our MURI project, we are developing a dynamic proof system to deal with various situations requiring workflow adaptation, and tools supporting the modeling of SAW requirements and generation of AS^3 logic specifications. We are also incorporating other QoS, such as security and real-time in the service coordination. Other related future work includes model-based diagnosis and recovery to defend systems against malicious agents, learning high-level strategies for adapting workflows, and online detection, localization and correction of faults in SAW.

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. The authors appreciate many useful discussions with Ramesh Bharadwaj and Jim Kirby from Naval Research Laboratory.

References

[1] Web Services Architecture. Available at: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
[2] U.S. Department of Defense Directive (DODD) 8100.1: "Global Information Grid (GIG) Overarching Policy," The Pentagon, Washington D.C., 2002.
[3] 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 Applications Conf.*, 2002, pp. 233-238.
[4] S. S. Yau, et al, "Reconfigurable Context-Sensitive Middleware for Pervasive Computing," *IEEE Pervasive Computing*, vol. 1(3), 2002, pp. 33-40.
[5] S. S. Yau, H. Davulcu, S. Mukhopadhyay, et al., "Overview of Adaptable Situation-Aware Secure Service-based (AS^3) Systems," MURI Book Chapter Part I???.
[6] J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *Machine Intelligence 4*, 1969, pp. 463-502.
[7] J. A. Pinto, *Temporal Reasoning in the Situation Calculus*, PhD Thesis, University of Toronto, 1994.
[8] J. McCarthy., "Situation Calculus with Concurrent Events and Narrative", http://wwwformal.stanford.edu/jmc/narrative/narrative.html, 2000.
[9] D. Plaisted "A Hierarchical Situation Calculus", *J. Computing Research Repository (CoRR)*, 2003.
[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] 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.
[13] M. Roman, et al., "A middleware infrastructure for active spaces," *IEEE Pervasive Computing*, vol. 1(4), 2002.
[14] A. Ranganathan, and R.H. Campbell, "A middleware for context-aware agents in ubiquitous computing environments", *Proc. ACM/IFIP/USENIX Int'l Middleware Conf.*, 2003, pp. 143-161.
[15] 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.
[16] Web Services Coordination (WS-Coordination), Available at: http://www-106.ibm.com/developerworks/library/ws-coor/
[17] Web Services Coordination Framework (WS-CF), http://www.oracle.com/technology/tech/webservices/htdocs/spec/WS-CF.pdf
[18] P. Braioneand G. P. Picco, "On Calculi for Context-Aware Coordination", *Proc. 6th Coordination Conf.,* 2004, pp. 38-54
[19] G. Cabri, L. Leonardi and F. Zambonelli, "Engineering Mobile Agent Applications via Context-Dependent Coordination", *IEEE Tran. On Software Engineering,* vol. 28(11), 2002, pp. 1039-1055.
[20] C. Julien and G. Roman, "Egocentric context-aware programming in ad hoc mobile environments", *Proc. 10th Int'l. Symp. On the Foundations of Software Eng.*, 2002, pp. 21-30.
[21] P. Blackburn, M. deRijke, and Y. Venema, *Modal Logic*, Cambridge University Press, 2003.
[22] A. J. R. J. Milner, *Communicating and Mobile Systems: the πä -Calculus*. Cambridge University Press, 1999.
[23] L. Cardelli and A. D. Gordon, "Mobile ambients," *Theoretical Computer Science*, vol. 240(1), 2000, pp. 177-213.
[24] M. Dam, "Model checking mobile processes," *Proc. CONCUR*, 1993.
[25] J. Halpern and Y. Moses, "Knowledge and common knowledge in a distributed enviroment," *J. ACM*, vol. 37(3), 1990, pp. 549-587.
[26] L. Cardelli and A .D. Gordon, "Anytime, anywhere: Modal logics for mobile ambients", *Proc. 27th ACM Symp. on Principles of Programming Languages*, 2000, pp. 365-377.
[27] R. Bharadwaj, "Secure Middleware for Situation-Aware Naval C^2 and combat Systems," *Proc. 9th Int'l Workshop on Future Trends of Distributed Computing System (FTDCS 2003)*, 2003, pp. 233-240.
[28] Aglets website, http://aglets.sourceforge.net/
[29] Ajanta project website, http://www.cs.umn.edu/Ajanta/
[30] R. Bharadwaj, "SOL: A Verifiable Synchronous Language for Reactive Systems," *Proc. Synchronous Languages, Applications, and Programming (SLAP' 02),* 2002. http://chacs.nrl.navy.mil/publications/ CHACS/2002/2002bharadwaj-entcs.pdf
[31] U. Kuter and D. Nau, "Forward Chaining Planning in Nondeterministic Domains," *Proc. 19th National Conf. on Artificial Intelligence (AAAI'04)*, 2004.