An Approach to Automated Agent Deployment in Service-based Systems

Stephen S. Yau, Luping Zhu, Dazhi Huang, and Haishan Gong
Arizona State University, Tempe, AZ 85287-8809, USA
{yau, luping.zhu, dazhi.huang, haishan.gong}@asu.edu

Abstract

In service-based systems, services from various providers can be integrated following specific workflows to achieve users' goals. These workflows are often executed and coordinated by software agents, which invoke appropriate services based on situation changes. These agents need to be deployed on underlying platforms with respect to various requirements, such as access permission of agents, real-time requirements of workflows, and reliability of the overall system. Deploying these agents manually is often error-prone and time-consuming. Furthermore, agents need to migrate from hosts to hosts at runtime to satisfy deployment requirements. Hence, an automated agent deployment mechanism is needed. In this paper, an approach to automated agent deployment in service-base systems is presented. In this approach, the deployment requirements are represented as deployment policies, and techniques are developed for generating agent deployment plans by solving the constraints specified in deployment policies, and for generating executable code for runtime agent deployment and migration.

1. Introduction

In service-based systems (SBS), services from various providers can be utilized by distributed software agents, which invoke proper services following specific workflows to achieve users' goals. These agents need to be deployed on the hosts with respect to various requirements, such as access permissions of agents, real-time requirements of workflows, and reliability of the overall systems. Such requirements may restrict the choices of hosts for agents and are considered as *deployment requirements*.

For SBS with hundreds of agents, manually deploying all agents is time-consuming and error-prone. Furthermore, in order to satisfy deployment requirements, an agent may need to move from one host to another when the environment changes. Many agent platforms, such as Aglets [1], Ajanta [2], and Jade [3], support agent migration. However, no existing techniques are available for selecting suitable hosts for migration based on situation changes. Hence, it is desirable that the decisions of deploying and migrating agents to various hosts can be automatically made based on the deployment requirements and system information provided by users and developers.

In this paper, we will present an approach to automated agent deployment with the requirements on communication bandwidth and permissions of agents to be deployed in certain domains or on certain hosts. In our approach, deployment requirements are represented as deployment policies, which specify the constraints on agent deployment. We have developed techniques to solve such constraints to generate *agent deployment plans* (ADPs), which map agents to hosts to determine the hosts on which agents will be deployed under various situations. Based on the generated ADPs, executable code for runtime agent deployment and migration will be synthesized.

2. Current State of the Art

Substantial research has been done in the areas of component deployment, device placement, and dynamic agent allocation. In component deployment, OMG defined deployment as the process between acquisition and execution of software, and proposed a five-phase component deployment process [4]. Ben-Shaul, et al, [5] presented a set of protocols for users to dynamically deploy components in remote sites and reconfigure the deployment scheme through reflective stubs. These approaches do not consider the mobility of the deployed components and cannot account for dynamic environment changes in SBS.

Device placement usually has an optimization goal, such as coverage and battery life, for the deployment plan of some devices. Howard, et al, [6] presented an incremental self-deployment algorithm for mobile sensor networks to maximize network coverage. Bulusu, et al, [7] presented an incremental adaptive beacon placement technique based on the localization errors collected by a GPS-equipped mobile robot. Hu, et al, [8] formulated the problem of minimizing energy usage in a hybrid sensor network as an integer linear programming problem. Their approaches either assume uniformity of devices to be deployed, or only consider devices that can be classified in a few categories. These approaches are not suitable for agent deployment, where agents are heterogeneous.

Jang and Agha [9] introduced two dynamic agent allocation mechanisms: one for co-allocating agents according to their communication patterns to minimize communication cost, and the other for moving agents from overloaded nodes to under-loaded nodes. However, they did not consider the permission of agents to be deployed on certain domains. Loitta, et al, [10] presented an active distributed monitoring system based on mobile agents. Agents in the system can dynamically monitor network behavior, but all the agents in the system are the same. Braubach, et al, [11] presented a platform independent deployment reference model and developed specialized agents for launching and managing other agents, but user intervention is required for agent deployment.

3. Background

We have developed an approach to developing Adaptable Situation-aware Secure Service-based (AS^3) systems [12]. In AS^3 systems, Situation Awareness (SAW) agents collect context data, analyze situations, trigger actions under various situations, and provide situational information to other agents for situation analysis, service coordination, and security policy enforcement. Security agents enforce security policies in a distributed manner based on the current situation. Workflow agents coordinate the execution of workflows based on situational information.

We have presented the specifications of SAW requirements, security policies, and workflows in an AS^3 system based on a modal logic [12-16]. SAW specifications describe which SAW agents monitor what situations, and what actions are taken when situation changes. Security policies specify who have the permissions to invoke the services under various situations. Workflow specifications describe the workflows to be executed by the workflow agents, and how the workflow agents communicate with SAW agents. These specifications constitute the system configuration used for solving the agent deployment problem.

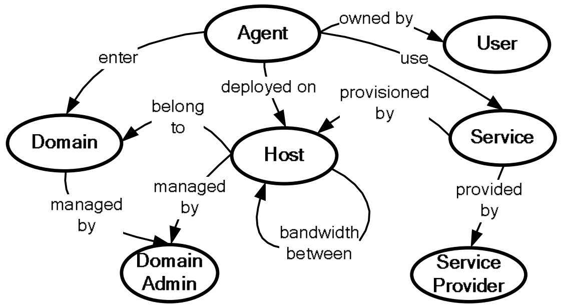


Figure 1. An ontology for various entities in SBS

The specifications of system configuration are part of the inputs of our approach to automated agent deployment, which generates a set of concurrent processes to control the deployment and migration of agents. We use AS^3 calculus [15], which is based on classical process calculus [17], to describe these processes, and have developed an AS^3 calculus to Java compiler to compile the processes to executable Java code. AS^3 calculus can model timeouts, failures, service invocations, communications [14], and process migrations. Process migration is modeled as "*mv to n*", which denotes the action "*move to ambient n*".

4. Problem Formulation

In this section, we will formulate the agent deployment problem in SBS. Fig. 1 depicts an ontology for various entities in SBS, in which the ellipses represent the entities in the system, including *agent*, *service*, *host*, *domain*, *user*, *service provider*, and *domain administrator*. Each arrow represents the relation between two entities. An *agent* is a software component owned by a *user* and acting on behalf of the user to use various services. A *service* is the capability provided by a *service provider* and is provisioned by a host. A *host* is a computer in the network, where agents and services can be deployed, and belongs to a specific domain. A *domain* is a group of hosts administrated as a unit with common rules. Domains and hosts are managed by *domain administrators*.

To illustrate an agent deployment problem, consider the emergency response system connecting two domains shown in Fig. 2: *Police Department (PD)* and *Hospital (HP)*. *host$_1$* and *host$_2$* belong to *DP*, while *host$_3$* belongs to *HP*. The bandwidths between *host$_1$* and *host$_2$*, *host$_1$* and *host$_3$* are 10 and 4 respectively. Services *search* (search injury passengers) and *setPerimeter* (setup perimeter) are provisioned by *host$_1$* and *host$_2$* respectively. Services *getStatus* (get status of injury passengers), *ambToH* (send passengers to hospital by ambulance), and *heliToH* (send

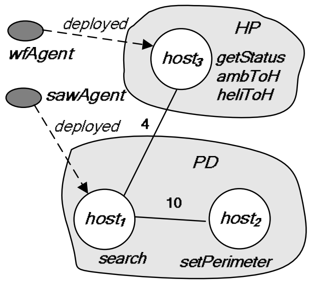


Figure 2. The emergency response system example

passenger to hospital by helicopter) are provisioned by *host*$_3$. When a car accident occurs, *sawAgent* and *wfAgent* need to be deployed to execute a specific workflow shown in Fig. 3 to rescue the injured passengers. *sawAgent* analyzes situations *accDetected*, *readySearch*, and *readyGetStatus*. *wfAgent* invokes services *setPerimeter*, *search*, and *getStatus* sequentially. When the status is critical, *wfAgent* uses *heliToH* to send passengers to hospital; otherwise, it uses *ambToH*.

Our approach requires the following three inputs: 1) a set of agents to be deployed; 2) specifications of system configuration, including SAW specifications, security policy specifications, workflow specifications, and network topology specifications; and 3) deployment policies. Our approach will generate agent deployment plans satisfying the deployment policies, and the corresponding controllers for runtime deployment and migration of agents.

To formulate the agent deployment problem, we need the following definitions. Let U denote the set of users, S the set of services, AS the set of agents to be deployed, H the set of the hosts, D the set of domains, and Θ the set of situations.

Def. 1: A *spatial relation* r between an agent a and an entity e, which can be a service, a host, a domain, or an agent, is a condition on a's location relative to the location of e.

We consider two types of deployment requirements: i) *communication bandwidth requirements*, which describe that bandwidth between a and e must be larger than a certain value under certain situation; and ii) *permission requirements*, which specify whether a is or is not co-located with e under certain situation. According to Def. 1, "bandwidth between a and e is larger than a certain value" is a spatial relation and is denoted by *bandwidth*. Whether "a is co-located with e" is also a spatial relation and is denoted by *together* for "is" and *restrict* for "is not". Therefore, there are three types of spatial relations *together*, *restrict* and *bandwidth* used in our approach.

An SBS can have the following six types of deployment policies:

Type 1) *agent_situ policy* is a 4-tuple $<A, e, C, r>$ indicating that A and e need to satisfy r under C, where A is either an agent or all the agents in the system; C is a specific situation or is unspecified, which means that the policy should be satisfied under any situation.

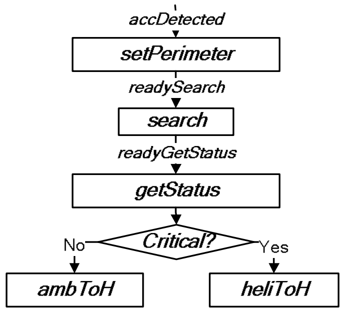


Figure 3. The workflow for the example

Type 2) *user_situ policy* is a 4-tuple $<u, e, C, r>$ indicating that e and the agents owned by a user u need to satisfy r under C.

Type 3) *user_service policy* is a 4-tuple $<u, e, s, r>$ indicating that e and the agents owned by u need to satisfy r when agents invoke service s.

Type 4) *user_agent policy* is a 4-tuple $<u, e, A, r>$ indicating that e and the agents owned by u need to satisfy r when the agents communicate with A.

Type 5) *agent_service policy* is a 4-tuple $<A, e, s, r>$ indicating that A and e need to satisfy r when A invokes service s.

Type 6) *agent_agent policy* is a 4-tuple $<A, e, A', r>$ indicating that A and e need to satisfy r when A communicates with other agents A'.

A special case of *agent_situ policy* $<a, e, situ, r>$ is called a *primitive policy*, where $a \in AS$, and $situ \in \Theta$.

A network topology specification includes:

1) *dMember(h, d)*: Host h belongs to domain d.

2) *serviceHost(s, h)*: Service s is deployed on h.

3) *cBandwidth*(h_1, h_2, b): The bandwidth between host h_1 and host h_2 is b. When $h_1 = h_2$, then $b = \infty$.

The specification of a system configuration is denoted by follows:

$$ConfSpec = \{SAWSpec\} \cup \{SecSpec\} \cup \{WfSpec\} \cup \{NetSpec\},$$

where *SAWSpec* is the set of SAW specifications; *SecSpec* is the set of security policy specifications; *WfSpec* is the set of workflow specifications; *NetSpec* is the set of network topology specifications. The detailed information for *SAWSpec*, *SecSpec*, and *WfSpec* can be found in [12-16].

Deployment policies from various parties are usually independent of a specific system configuration. In order to enforce deployment policies, deployment constraints are generated from the deployment policies with respect to a system configuration. We define a deployment constraint as follows:

Def. 2: A *deployment constraint* under *situ* is a spatial relation r between an agent a and an entity e, which the deployment of a must satisfy r under *situ*.

Deployment constraints are categorized to two classes: *absolute constraints*, in which each constraint specifies the spatial relation between an agent and a non-agent entity that must be satisfied under a certain situation; and *relative constraints*, in which each constraint specifies the spatial relation between two agents that must be satisfied under a certain situation.

Def. 3: A *host assignment* for an agent a_i under a certain situation $situ_j$, denoted by $host(situ_j, h_k)$, is said to be valid if no deployment constraint involving a_i is violated when a_j is deployed on h_k under $situ_j$.

Def. 4: A *candidate deployment plan* for an agent a_i, denoted by $cPlan(a_i)$, is $\{cHosts(situ_1, hostSet_1), ...,$

cHosts($situ_n$, $hostSet_n$)}, which indicates that a_i can be deployed on a host in $hostSet_j$ under situation $situ_j$.

Def. 5: An *agent deployment plan* (ADP) for an agent a_i, denoted by *dPlan*(a_i), is {*host*($situ_1$, h_1), ..., *host*($situ_n$, h_n), *host*(*true*, h_{true})}, which identifies that the *host*($situ_i$, h_i) under $situ_i$, i=1, 2,…, n, are valid.

Based on the above definitions, the automated agent deployment problem can be represented as a 9-tuple

<*U, S, H, D, Θ, AS, PS, ConfSpec, PLAN*>,

where *U* is a set of users, *S* is a set of services, *H* is a set of hosts, *D* is a set of domains, *Θ* is a set of situations, *AS* is a set of deploying agents, *PS* is a set of deployment policies, and *ConfSpec* is the set of system configuration specifications, and *PLAN* is a set of ADPs for each agent in *AS*.

5. Our Overall Approach

Our approach to automated agent deployment consists of the following steps:

S-1) Collect the deployment policies from various parties, and send them to Policy Manager Service (see Sec. 6).

S-2) The Policy Manager Service invokes Constraint Generation Service to generate the corresponding constraints (see Sec. 7) and store them in a constraint repository.

S-3) Each agent a_i invokes Plan Generation Service to generate a *cPlan*(a_i) by solving absolute constraints (see Sec. 8). If the invocation is successful, send the *cPlan*(a_i) to Candidate Plan Service, and go to *S-4)*. If the invocation fails, go to *S-6)*.

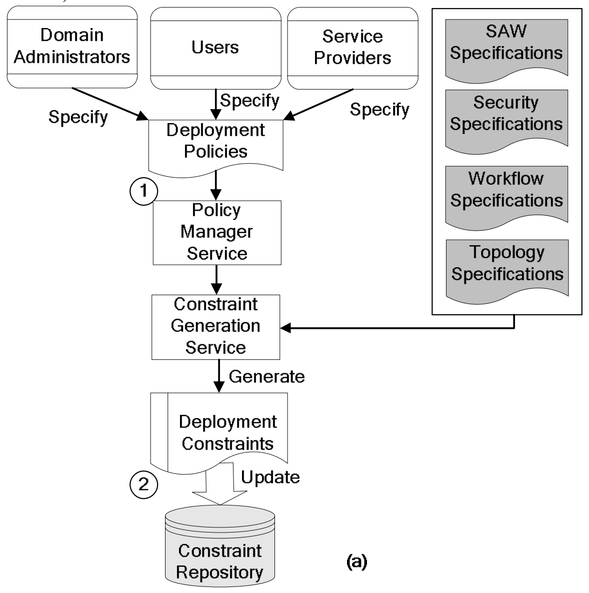
S-4) After Candidate Plan Service collects the *cPlan*(a_i) for all the agents, it invokes Plan Composition Service (*PCS*) to compose all the candidate plans into ADPs by solving the relative constraints (see Sec. 8), and go to *S-5)*. If *PCS* cannot solve all the relative constraints, go to *S-6).*

S-5) Base on the ADPs, *PCS* synthesizes deployment controllers (see Sec. 8), which control the deployment and migration of agents. When a situation, under which some agents need to migrate to other hosts, is detected by controllers via communicating with SAW agents, controllers will deploy and migrate these agents to their destination hosts by invoking the APIs provided by underlying agent platform, such as Aglets [1], Ajanta [2], and Jade [3]. This completes the agent deployment process.

S-6) There are two types of failures: 1) in the candidate deployment plan generation process in *S-3)*, which is handled by the Generation Failure Handling Service (*GFHS*), and 2) in the ADPs composition process in *S-4),* which is handled by the Composition Failure Handling Service (*CFHS*). Both *GFHS* and *CFHS* handle these failures by identifying the deployment policies which cause the failures, and notifying the owners of all these identified deployment policies to change their deployment policies (see Sec. 9). Then, go to *S-1)*.

Fig. 4(a) describes the generation of deployment policies and deployment constraints, including *S-1)* and *S-2),* while Fig. 4(b) depicts the generation of ADPs and controllers, including *S-3)* to *S-7).*

6. Generation of Deployment Policies



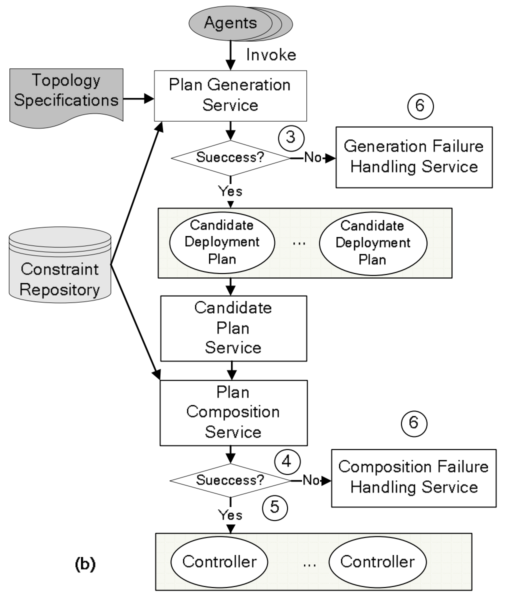


Figure 4. Our approach to automated agent deployment, (a) generation of deployment policies and deployment constraints, and (b) generation of ADPs and controllers

In this section, we will present the generation of deployment policies from deployment requirements [see *S-1)* in Sec. 5]. To do so, we will first identify the following four elements:

- *Who:* an agent to be deployed, all agents in the system, or a set of agents owned by a user.
- *Whom:* an entity, which can be a service, a host, a domain, or an agent
- *When:* under a particular situation, invoking a service, or communicating with other agents
- *What:* a spatial relation that should be satisfied

For instance, the previous example has the following four deployment requirements:

Req 1: For any agent invoking service *getStatus*, the communication bandwidth between that agent and *getStatus* must be larger than 5.

Req 2: *sawAgent* is not allowed to access *HP*.

Req 3: For any agent invoking service *search*, the communication bandwidth between that agent and *search* must be larger than 5.

Req 4: *wfAgent* and *sawAgent* need to be deployed on the same host under *accDetected*.

For Req 1, we find that *Who* is all agents; *Whom* is *getStatus* service; *What* is relation "communication bandwidth must be larger than 5", and *When* is invoking service *getStatus*. Similarly, we can find these four items for each of Req 2, Req 3 and Req 4.

After the above requirement analysis, we can easily generate deployment policies by identifying <*who, whom, when, what*> in deployment requirements. In the previous example, the following four deployment policies are generated corresponding to the Reqs. 1-4:

P-1: <$\forall$*agent, getStatus, getStatus, 5*>

P-2: <*sawAgent, HP,* $\forall$*situation, restrict*>

P-3: <$\forall$*agent, search, search, 5*>

P-4: <*wfAgent, sawAgent, accDetected, together*>

7. Generation of Deployment Constraints

As we discussed in Sec. 4, deployment policies are usually independent of a specific system configuration. In order to enforce the deployment policies for a specific system configuration, deployment constraints need to be generated from the deployment policies with respect to the system configuration [see *S-2)* in Sec. 5]. These constraints are stored in the constraint repository and will be used by Plan Generation Service for *cPlan*(a_i) generation and by *PCS* for *dPlan*(a_i) composition.

The deployment constraints from deployment policies can be generated by the constraint generation service as follows:

C-1) Each deployment policy is converted to a set of primitive policies by policy conversion algorithm.

C-2) Each primitive policy <a_i, *e*, $situ_j$, *r*> is directly mapped to a deployment constraint according to the values of *e* and *r*.

C-3) The all types of deployment constraints are converted to a subset of constraint types by reduction rules in order to simplify the generation of ADPs and controllers.

As mentioned in Sec. 4, there are six types of deployment policies. Here, we will only show how to convert Type 3) a *user_service policy* to a set of primitive policies, since other types can be converted in a similar way: For a *user_service policy* <*u, e, s, r*>, add a deployment policy <a_i, *e, s, r*> to a list *pList* for each a_i owned by *u*, remove the original *user_service* policy <*u, e, s, r*>. For each deployment policy <a_i, *e, s, r*> in *pList,* select all situations under which a_i will invoke service *s* based on the *WfSpec* into a set *situ_list;* add a primitive policy <a_i, *e*, $situ_j$, *r*> for each $situ_j \in situ\_list$.

C-2) maps each primitive policy to a deployment constraint. A primitive policy <a_i, *e*, $situ_j$, *r*> indicates that "the deployment of agent a_i needs to satisfy the spatial relation *r* between a_i and *e*", which is a deployment constraint. Therefore, <a_i, *e*, $situ_j$, *r*> is directly mapped to a deployment constraint.

Table 1. All the deployment constraint types in our approach

No.	Constraint type	Description
C1	*hostTogether(a, h, situ)*	Agent *a* is on (or is migrating to) host *h* when situation *situ* is true
C2	*serviceTogether(a, s, situ)*	Agent *a* is on the same host with service *s* when situation *situ* is true
C3	*domainTogether(a, d, situ)*	Agent *a* is on a host of domain *d* when situation *situ* is true
C4	*hostRestrict(a, h, situ)*	Agent *a* is not on host *h* when situation situ is true
C5	*serviceRestrict(a, s, situ)*	Agent *a* is not on the same host with service *s* when situation *situ* is true
C6	*domainRestrict(a, d, situ)*	Agent *a* is not on a host of domain *d* when situation *situ* is true
C7	*hostBandwidth(a, h, b, situ)*	Agent *a* is on the host h_1 such that the communication bandwidth between h_1 and *h* is larger than *b* when situation *situ* is true
C8	*serviceBandwidth(a, s, b, situ)*	Agent *a* is on the host *h* such that the communication bandwidth between *h* and the host of service *s* is larger than *b* when situation *situ* is true
C9	*agentTogether*(a_1, a_2, *situ*)	Agents a_1 and a_2 are on the same host when situation *situ* is true
C10	*agentRestrict*(a_1, a_2, *situ*)	Agents a_1 and a_2 are not on the same host when situation *situ* is true
C11	*agentBandwidth*(a_1, a_2, *b, situ*)	The communication bandwidth between the host of agent a_1 and the host of agent a_2 is larger than *b* when situation *situ* is true

Table 2. Constraint reduction rules

No.	Rule
R-1	*serviceTogether(a, s, situ)* $\wedge$ *serviceHost(a, h)* $\Rightarrow$ *hostTogether(a, h, situ)*
R-2	*domainTogether(a, d, situ)* $\wedge$ $\forall h$ *(¬ dMember(h, d))* $\Rightarrow$ *¬hostTogether(a, h, situ)*
R-3	*serviceBandwitdh(a, s, b, situ)* $\wedge$ *serviceHost(s, h)* $\Rightarrow$ *hostBandwidth(a, h, b, situ)*
R-4	*serviceRestrict(a, s, situ)* $\wedge$ *serviceHost(a, h)* $\Rightarrow$ *¬hostTogether(a, h, situ)*
R-5	*domainRestrict(a, d, situ)* $\wedge$ *dMember(h, d)* $\Rightarrow$ *¬ hostTogether(a, h, situ)*
R-6	*agentRestrict*$(a_1, a_2,$ *situ*$)$ $\Rightarrow$ ¬*agentTogether*$(a_1, a_2,$ *situ*$)$
R-7	*hostRestrict(a, h, situ)* $\Rightarrow$ *¬ hostTogether(a, h, situ)*

For a primitive policy $<a_i, e, situ_j, r>$, there are four choices for *e*: service, agent, host, or domain, and there are three choices for *r*: *together*, *restrict*, or *bandwidth*. Hence, there are 12 choices for a primitive policy. Each of the 12 choices, except the communication bandwidth between an agent and a domain, needs to be considered as a type of deployment constraints. Hence, 11 types of deployment constraints are enforced in our approach and they are all listed in Table 1. It is noted that C1 to C8 are absolute constraints, while C9 to C11 are relative constraints.

C-3) converts the 11 types of constraints to a subset of constraint types by applying negation (¬) and conjunction ($\wedge$) operations on them. We have identified a set of constraint reduction rules listed in Table 2 and a subset of constraint types *subCTypeSet* = {*hostTogether*, *agentTogether*, *hostBandwidth*, *agentBandwidth*} for converting the constraints of the 11 types to the constraints of *subCTypeSet* so that our approach only needs to solve the constraints of *subCTypeSet*. In Table 2, R-1 to R-5 define that constraints in left-hand side can be converted to the constraints in right-hand side under the topology specifications on the left-hand side. R-6 and R-7 define that constraints in the left-hand side can be converted to the constraints in the right-hand side by applying negation (¬) operation on them.

Table 3 shows the results of our deployment constraint generation process from all the deployment policies in the previous example. Because of limited space, we will only show how to generate constraints from P-2. In *C-1)*, P-2 is converted to three primitive policies: *<sawAgent, HP, accDetected, restrict>*, *<sawAgent, HP, readySearch, restrict>*, and *<sawAgent, HP, readyGetStatus, restrict>* because *accDetected*, *readySearch, readyGetStatus* are three situations in the system. In *C-2)*, *<wfAgent, getStatus, 5, readyGetStatus>* is mapped to *serviceBandwidth(wfAgent, getStatus, 5, readyGetStatus)* because *getStatus* is a service and 5 is a *bandwidth* relation. In *C-3)*, *serviceBandwidth(wfAgent, getStatus, 5, readyGetStatus)* is reduced to *hostBandwidth(wfAgent,* $host_3$*, 5, readyGetStatus)* using rule R-3 because *getStatus* is provisioned on $host_3$.

8. Plans and Controllers Generation

In this section, we will present the generation of ADPs and controllers. As discussed in *S-3)*, we first generate all *cPlan*(a_i) by the *CPlanGeneration* algorithm, which is given as follows:

G-1) Select all the absolute constraints related to the invoking agent a_i from constraint repository.

G-2) Initialize *cPlan*(a_i) = {*cHosts(true, H)*}, which indicates that a_i can be deployed on any host in *H* under any situation in the system.

G-3) Remove the hosts, which are not satisfied with any deployment constraints of the types *hostTogether* and *hostBandwidth* in *subCTypeSet* from *cHosts*.

G-3.1) For each *hostTogether*$(a_i, h_j, situ_k)$, find the host set $hostSet_m$ for a_i under $situ_k$ from *cPlan*(a_i). If $h_j \in hostSet_m$, let $hostSet_m = [h_j]$, and return to *G-3)*. Otherwise invoke *GFHS* to handle the failure.

G-3.2) For each ¬*hostTogether*$(a_i, h_j, situ_k)$, find $hostSet_m$ for a_i under $situ_k$ from *cPlan*(a_i). If $h_j \in hostSet_m$, remove h_j from $hostSet_m$. If $hostSet_m=\varnothing$, invoke *GFHS* to handle the failure. If not empty, return to *G-3)*.

G-3.3) For each *hostBandwidth*$(a_i, h_j, d, situ_k)$, find $hostSet_m$ for a_i under $situ_k$ from *cPlan*(a_i). Remove each host, whose communication bandwidth with h_j is

Table 3. The results of deployment constraint generation process for deployment policies in example

	Deployment policy	Primitive policy	Constraint	Reduced constraint	
P-1	*<∀agent, getStatus, getStatus, 5>*	*<wfAgent, getStatus, readyGetStatus, 5>*	*serviceBandwidth(wfAgent, getStatus, 5, readyGetStatus)*	*hostBandwidth(wfAgent,* $host_3$*, 5, readyGetStatus)*	RC-1
P-2	*<sawAgent, HP, ∀situation, restrict>*	*<sawAgent, HP, accDetected, restrict>*	*domainRestrict(sawAgent, HP, accDetected)*	*¬hostTogether(sawAgent,* $host_3$*, accDetected)*	RC-2
		<sawAgent, HP, readySearch, restrict>	*domainRestrict(sawAgent, HP, readySearch)*	*¬hostTogether(sawAgent,* $host_3$*, readySearch)*	RC-3
		<sawAgent, HP, readyGetStatus, restrict>	*domainRestrict(sawAgent, HP, readyGetStatus)*	*¬hostTogether(sawAgent,* $host_3$*, readyGetStatus)*	RC-4
P-3	*<∀agent, search, search, 5>*	*<wfAgent, search, readySearch, 5>*	*serviceBandwidth(wfAgent, search, 5, readySearch)*	*hostBandwidth(wfAgent,* $host_1$*, 5, readySearch)*	RC-5
P-4	*<wfAgent, sawAgent, accDetected, together>*	*<wfAgent, sawAgent, accDetected, together>*	*agentTogether(wfAgent, sawAgent, accDetected)*	*agentTogether(wfAgent, sawAgent, accDetected)*	RC-6

less than d, from $hostSet_m$. If $hostSet_m=\varnothing$, invoke *GFHS* to handle the failure. If not empty, return to *G-3)*.

After all $cPlan(a_i)$ are generated, we will generate the ADP and controller of each agent [see *S-4)* and *S-5)* in Sec. 5] using the $dPlan(a_i)$ composition process, which is given as follows:

D-1) Select all the relative constraints for all agents from constraint repository.

D-2) Compose $cPlan(a_i)$ to $dPlan(a_i)$ by solving all the relative constraints using a satisfiability (SAT) constraint solver, such as Disolver [18] or Koalog [19]. If all the relative constraints cannot be solved together, *CFHS* is invoked to handle failure.

D-3) For each agent, generate a controller in AS^3 calculus based on $dPlan(a_i)$.

D-3.1) According to Def. 5, each element in $dPlan(a_i)$ of agent a_i has a host h_j and a situation $situ_k$. Add all these hosts in a host set $hostSet_i$. Go to *D-3.2)*.

D-3.2) For each host h_m in $hostSet_j$, find all situations, under which a_i will be deployed on h_m, add these situations in a situation set $situSet_m$, and form a pair (h_m, $situSet_m$). Go to *D-3.3)*.

D-3.3) For each pair (h_m, $situSet_m$), where $situSet_m$ = [$situ_1$,…$situ_n$] formed in *D-3.2)*, generate AS^3 calculus in the format: "if s == $situ_1$ | … | s == $situ_n$ then *mv to* h_m. a_i". Go to *D-3.4)*.

D-3.4) Concatenate all the calculi generated in *D-3.3)* to a calculus. Add "fix controller_a_i = String situChannel(String s)." at the beginning of the calculus and "controller_a_i." at the end of the calculus.

In *D-2)*, a constraint solver is needed. We have developed an SAT constraint solver using constraint handling rules in XSB [20]. The constraint solver solves constraints by finding a host assignment for each agent under various situations from $cPlan(a_i)$ so that all the host assignments are valid.

Here, we will discuss how to generate $cPlan(a_i)$ and compose $dPlan(a_i)$ with the reduced constraints (RC-1 to RC-6) in example discussed in Sec. 7.

In *G-1)*, RC-1 to RC-5 are selected. In *G-2)*, *cPlan(wfAgent)* = {*cHosts(true, [*$host_1$, $host_2$, $host_3$*])*} and *cPlan(sawAgent)* = {*cHosts(true, [*$host_1$, $host_2$, $host_3$*])*} are initialized. In *G-3)*, when solving RC-1, *G-3.3)* is executed. [$host_1$, $host_2$, $host_3$] is selected from *cPlan(sawAgent)*. [$host_3$] is obtained by removing $host_1$ and $host_2$ from [$host_1$, $host_2$, $host_3$] because the communication bandwidth between $host_1$ or $host_2$ and $host_3$ is less than 5. After RC-1 to RC-5 are solved, *cPlan(wfAgent)* = {*cHosts(accDetected, [*$host_1$, $host_2$, $host_3$*]), cHosts(readyGetStatus, [*$host_3$*]), cHosts(readySearch, [*$host_1$*])*}, and *cPlan(sawAgent)*={*cHosts(accDetected, [*$host_1$, $host_2$*]), cHosts(readSearch, [*$host_1$, $host_2$*]), cHosts(readyGetStatus, [*$host_1$, $host_2$*])*} are obtained.

Table 4. A controller in AS^3 calculus

```
fix controller_wfAgent =
  String situChannel(String s).
  if s == "readyGetStatus" then
    mv to host3. wfAgent.
  if s == "accDetected" | s == "readySearch" then
    mv to host1. wfAgent
controller_wfAgent.
```

In *D-1)*, RC-6 is selected. In *D-2)*, the constraint solver solves RC-6, and generates *dPlan(wfAgent)* = {*host(accDetected,* $host_1$*), host(readySearch,* $host_1$*), host(readyGetStatus,* $host_3$*)*} and *dPlan(sawAgent)* = {*host(accDetected,* $host_1$*), host(readySearch,* $host_1$*), host(readyGetStatus,* $host_1$*)*}. *dPlan(wfAgent)* indicates that *wfAgent* will be deployed on $host_1$ under *accDetected* and *readySearch*, on $host_3$ under *readyGetStatus*. *dPlan(sawAgent)* indicates that *sawAgent* will be deployed on $host_1$. In *D-3)*, controller for *wfAgent* in AS^3 calculus is generated as Table 4 and will be compiled and executed in the underlying agent platform.

9. Failure Handling

In this section, we will discuss how GFHS handles the failures in the $cPlan(a_i)$ generation process and how *CFHS* handles the failures in the $dPlan(a_i)$ composition process.

GFHS handles the failures in the $cPlan(a_i)$ generation process as follows:

F-1) From the processed absolute constraints for agent a_i, trace each failure to the deployment policies which generate these absolute constraints

F-2) Remove all the constraints generated by these deployment policies from the constraint repository.

F-3) Report these deployment policies to the policy owners. After policy owners update their deployment policies, constraint generation service will be invoked to generate deployment constraints for the updated deployment policies, and store them to the constraint repository. Go to *S-3)* in Sec. 5.

CFHS handles the failures in the $dPlan(a_i)$ composition process as follows:

H-1) Trace each failure to the deployment policies causing the failure: Let δ be a set {a_i} of agents. Find all the agents having relative constraints with any agent in δ, and add these agents in δ. Repeat this process until no more agents can be added. Find the deployment policies for all the agents in δ.

H-2) Remove all the constraints generated from these deployment policies identified in *H-1)*.

H-3) Report the deployment policies identified in *H-1)* to the policy owners. After policy owners update their deployment policies, generate all the deployment constraints from the updated deployment policies as described in Sec. 7.

H-4) Generate all the *cPlan(a_i)* for the agents in δ as described in Sec. 8. Go to *S-4)* in Sec. 5.

To illustrate the above failure handling processes using our previous example, we add the deployment policy P-5 <*sawAgent, host$_3$, accDetected, together*>, from which constraint RC-7 *hostTogether(sawAgent, host$_3$, accDetected)* is generated. A failure occurs in *S-3)* of our approach because *sawAgent* cannot be deployed on any host under *accDetected* due to P-5. In *F-1)*, *GFHS* traces the failure to P-2 and P-5. In *F-2)*, constraints RC-2, -3, -4, and -7 are removed from the constraint repository due to this failure. In *F-3)*, the deployment policy owners of P-2 and P-5 are informed of the constraints of their deployment policies and asked to change or remove their policies. Suppose that P-5 is removed, and another policy (P-6) <*wfAgent, sawAgent, readyGetStatus, together*> is added. From P-6, we generate constraint RC-8 *agentTogether(wfAgent, sawAgent, readyGetStatus)*. Constraints RC-2, -3, -4, and -8 from P-2 and P-6 are generated and added to the constraint repository. A failure occurs in *S-4)* of our approach because *wfAgent* cannot be deployed on any host under *readyGetStatus* due to P-6. In *H-1)*, δ is initially {*wfAgent*}. *sawAgent* is added in δ because a relative constraint RC-8 between *wfAgent* and *sawAgent* is found. *CFHS* traces this failure to the deployment policies (P-1, -2, -3, -4, and -6). In *H-2)*, all the constraints are removed from the constraint repository due to this failure. In *H-3)*, policy owners of these policies are informed of the constraints of their policies and asked to change or remove their policies and so forth.

10. Conclusions and Future Work

In this paper, we have presented an approach to automated agent deployment for SBS. We have considered two types of deployment requirements: communication bandwidth and permission. In SBS for critical applications, it is also important to include the deployment requirements on system reliability and performance. In addition, because deployment policies are manually generated from deployment requirements, it is desirable to have a GUI tool to expedite the generation of deployment policies from deployment requirements.

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] Aglets web site, http://aglets.sourceforge.net/.
[2] Ajanta, http://www.cs.umn.edu/Ajanta/.
[3] Jade, http://jade.tilab.com/.
[4] The Object Management Group, "Deployment and Configuration of Component-based Distributed Applications Specification", http://www.omg.org/docs/ptc/03-07-02.pdf.
[5] I. Ben-Shaul, O. Holder, and B. Lavva, "Dynamic Adaptation and Deployment of Distributed Components In Hadas", *IEEE Trans. on Software Engineering*, vol. 27(9), 2001, pp. 769-787.
[6] A. Howard, M. J Mataric, and G. S Sukhatme, "An Incremental Self-Deployment Algorithm for Mobile Sensor Networks", *Autonomous Robots,* vol. 13(2), 2002, pp. 113-126.
[7] N. Bulusu, J. Heidemann, and D. Estrin, "Adaptive Beacon Placement," *Proc. Int'l Conf. on Distributed Computing Systems*, 2001, pp. 489-498.
[8] W. Hu, et al., "Deploying Long-Lived and Cost-effective Hybrid Sensor Networks," *Elsevier Ad-Hoc Networks,* vol. 4(6), 2006, pp. 749-767.
[9] M. Jang, and G. Agha, "Dynamic Agent Allocation for Large-Scale Multi-Agent Applications", *Int'l Workshop on Massively Multi-Agent Systems*, 2004, pp. 19-33.
[10] A. Liotta, G. Pavlou, and G. Knight, "Exploiting Agent Mobility for Large Scale Network Monitoring", *IEEE Network*, vol. 16, no. 3, May 2002, pp. 7-15.
[11] L. Braubach, et al., "Deployment of Distributed Multi-agent Systems", *Proc. Int'l Workshop on Engineering Societies in the Agents World*, 2005, pp. 261-276.
[12] S. S. Yau, H. Davulcu, and S. Mukhopadhyay, "Adaptable Situation-Aware Secure Service-based Systems", *Proc. IEEE Int'l Symp. On Object-oriented Real-time distributed Computing*, 2005, pp.308-315.
[13] S. S. Yau, et al., "Situation-Awareness for Adaptable Service Coordination in Service-based Systems", *Proc. Ann. Int'l Computer Software and App. Conf.*, 2005, pp.107-112.
[14] S. S. Yau, et al., "Automated Agent Synthesis for Situation Awareness in Service-based Systems", *Proc. Ann. Int'l Computer Software and App. Conf.*, 2006, pp. 502-512.
[15] S. S. Yau, et al., "Automated Agent Synthesis for Situation-Aware Service Coordination in Service-based Systems", *Technical Report, ASU-CSE-TR-05-008*, 2005. http://dpse.eas.asu.edu/AS3/papers/ASU-CSE-TR-05-09.pdf.
[16] S. S. Yau, et al., "An Adaptable Security Framework for Service-based Systems", *Proc. IEEE Int'l Workshop on Object-oriented Real-time Dependable Systems*, 2005, pp. 28-35.
[17] R. Milner, "Communicating and Mobile Systems: the π-Calculus", *Cambridge University Press*, 1999.
[18] Disolver, http://research.microsoft.com/~youssefh/DisolverWeb/Disolver.html.
[19] Koalog, http://www.koalog.com/php/jcs.php.
[20] D. S. Warren, et al., "The XSB Programmer's Manual", version 2.5, vols. 1 and 2, 2001.