Language-based access control approach for component-based software applications

Language-based access control approach for component-based software applications

R. Su, J.J. Lukkien and M.R.V. Chaudron

Abstract: Security in component-based software applications is studied by looking at information leakage from one component to another through operation calls. Components and security specifications about confidentiality as regular languages are modelled. Then a systematic way is provided to synthesise an access control mechanism, which not only guarantees all specifications to be obeyed, but also allows each user to attain maximum permissive behaviours.

1 Introduction

A component-based application consists of a collection of components, which are prefabricated as off-the-shelf products. One of the main problems in component-based software engineering (CBSE) is how to guarantee a system that is assembled from third-party components complies with its specifications. As far as the access control is concerned, a commonly used specification is about the unattainability of some information in one component to other unauthorised component. To comply with that specification, an access control mechanism is needed. In this paper we adopt the component-based framework introduced in the Robocop [1] and Space4U [2] projects.

In computer security, 'access control' is the ability to permit or deny the use of an object (a passive entity, such as a system or file) by a subject (an active entity, such as an individual or process). Access control systems provide the essential services of identification and authentication, authorisation and accountability, where identification and authentication determine the true identity of a subject that requests access authorisation determines what an authenticated subject can do and accountability identifies what a user or a process did. In this paper, we consider only the authorisation issue and leave identification/authentification and accountability to techniques in the literature, for example, we can use a password, a personal identification number (PIN) or even more extreme ways such as fingerprint, voice, retina or iris characteristics to do identification and authentication, and use audit records to handle accountability. Authorisation defines a user's rights and permissions on a system. Authorisation techniques are usually categorised into the following classes: (1) discretionary access control (DAC), including techniques such as access control lists [3, 4] and type-based access control [5–7], where the owner of a resource decides who is allowed access to the resource and what privileges they have; (2) mandatory access control (MAC), including techniques such as rule-based access control [8, 9] and lattice-based access control [10, 11], where it is the system, not the owner, who decides rights and permissions; and (3) role-based access control (RBAC) [12, 13], where a user may be assigned different rights and permissions attached to a specific role.

In a component-based framework, each component may be bought from a third party, thus, in general we have no knowledge about how each component behaves, except for operation calls in and out of a component via specified interfaces. In this paper, we consider only information leakage through predefined operation calls. We believe that those mentioned techniques in the literature have the following drawbacks. First, the assignment of rights and access privileges (ARP) to users is purely heuristic and there is no formal way to tell which ARP is better, if there exists more than one ARP. Secondly, the concept of information flow depends on an existing ARP. If the information flow does not satisfy all specifications, then the user needs to pick another ARP and repeats the same verification process. Although the process terminates eventually, the duration may be very long because in the worst case it is likely that all possible ARPs are used before the right one is found. In this paper, we define an information flow as one possible sequence of operation calls that can take place in the system. Thus, whenever the system is given, all possible information flows in the system are also fixed. Therefore, the designer's job is to block some flows that may violate specifications. There is a unique way to do that when we impose an optimality criterion, saying that the system under the access control should attain the maximum permissiveness. This criterion guides us to decide which flow should be blocked and how. The approach described in this paper is language based, where a 'language' refers to a free monoid over an alphabet under string concatenation. This makes it different from some other language-based approaches discursed in [14–16], 'languages' in these papers either refer to programming languages tailored specifically for an application domain or a logic representation system that allows descriptions of security specifications and effective processing of them. The language representation discussed in this paper is similar to those in papers about formal models of access control, [17–19], where security automata or transition diagrams for usage control are used to describe security specifications and policies. From access control point of view, security policies are essentially control policies used in this paper, which must be obeyed by the access control

doi:10.1049/iet-sen:20070026

Paper first received 7th March and in revised form 23rd June 2007

The authors are with System Architecture and Networking Group (SAN), Department of Mathematics and Computer Science, Eindhoven University of Technology, 5600 MB Eindhoven, PO Box 513, The Netherlands

E-mail: r.su@tue.nl

mechanism. But, in those papers, it is assumed that the policies are given a priori, thus their main problem is how to implement the policies or represent them with extensive expressivity. In this paper we focus on how to systematically generate those policies. Therefore, the main objective of this paper is fundamentally different from that in papers mentioned above, making the corresponding analysis approaches different as well. In short, we believe that the problem and the corresponding approach proposed in this paper can serve as a complementary part for those existing techniques in the sense that, using our approach, we can compute security policies for a component-based software application, then utilise the existing techniques to implement those policies. We can see this later in the paper.

This paper is organised as follows. Section 2 describes language-based dynamic models and their composition. Then, a language-based approach is provided in section 3 to formulate specifications. Section 4 presents a formal way to synthesise an access control mechanism with maximum permissiveness. Section 5 gives a way of implementing the control mechanism. Conclusions are drawn in Section 6.

2 Language-based dynamic models

In the Robocop [1] and Space4U [2] framework, a component, c, is a collection of services, S_c, where each service, $s \in S_c$, consists of a family of interfaces, I_s. An interface $i \in I_s$ in our framework consists of a list of operations, O_i. A 'requires' interface needs operations from other service instances, and a 'provides' interface gives operations to other service instance. A service s_1 binds s_2 on an interface i, when i is a requires interface in s_1 and a provides interface in s_2. Each operation $o \in O_i$ consists of (1) a set $o.P$ of input variables (or input arguments), where for each $p \in o.P$, its domain is denoted by $p.D$ (thus implicitly the input type is also defined); (2) a set $o.R$ of return variables (or return arguments), where each $r \in o.R$ has a domain $r.D$; (3) a behavior model describing in which order the operation o calls operations provided by other interfaces (not necessarily in the same service). The behaviour model can be a finite-state automaton (FSA), a sequence diagram or a process algebra. In a runtime environment, each service may instantiate multiple service instances, which binds with other instances to fulfil a task. In this paper, we consider only service instances, unless specified otherwise. A system consists of a collection of bounded service instances. As an illustration, a service instance specification may look as follows:

Example 1

1. service instance s
2. requires i_1
3. operation a
4. $a.P = \{a_1\}$
5. $a_1.D = \{1, 2, \ldots, 5\}$
6. $a.R = \varnothing$
7. requires i_2
8. operation b
9. $b.P = \{b_1\}$
10. $b_1.D = \{1, 2, \ldots, 10\}$
11. $b.R = \varnothing$
12. provides $i_3\{$
13. operation o
14. $o.P = \{p_1, p_2\}$
15. $p_1.D = \{1, 2, \ldots, 10\}$
16. $p_2.D = \{x | 0 \leq x \leq 10\}$
17. $o.R = \{r_1\}$
18. $r_1.D = \{\text{true, false}\}$
19. behavior:
20. $i_1.a$;
21. $i_2.b\}$

From the above specification we can see that the service instance s has three interfaces: two requires interfaces i_1, i_2, and one provides interface i_3. Interface i_3 has one operation o, which has two input parameters p_1 and p_2, where the domain of p_1 is a discrete-value set and the domain of p_2 is a continuous-value set. The operation o returns a Boolean value. The behaviour model says that within the execution of o, the operation a of i_1 is called first followed by the operation b of i_2. Similarly, the descriptions of two requires interfaces can be interpreted. We can see that operations $i_1.a$ and $i_2.b$ have no return values. Since $i_3.o$ has no interest about how $i_1.a$ and $i_2.b$ are executed, except for their return values, there is no behaviour model in the descriptions of $i_1.a$ and $i_2.b$. The specification says that whenever the operation $i_1.a$ is called within s, a value is assigned to the input variable a_1 first, which is then fed to $i_1.a$. In other words, $a_1.D$ stands for all values in s than may be assigned as an input value for the operation call $i_1.a$. A similar situation applies to $i_2.b$.

Each service can have many different instances. Let $\mathcal{S}$ be the collection of all service instances in a target system. From now on we focus only on service instances. For a slight abuse of notations, we also use s to denote a service instance, which is associated with a instance specification derived from the corresponding service specification. For each instance $s \in \mathcal{S}$, we can derive a collection of operations from the instance specification

$$O_s := \bigcup_{i \in I_s} O_i$$

and a collection of variables

$$V_s := \bigcup_{o \in O_s} [o.P \cup o.R]$$

We assume that two different service instances do not have any variable in common, namely

$$(\forall s, s' \in \mathcal{S}) s \neq s' \Rightarrow V_s \cap V_{s'} = \varnothing$$

Let

$$V := \bigcup_{s \in \mathcal{S}} V_s \quad \text{and} \quad O := \bigcup_{s \in \mathcal{S}} O_s$$

be the collection of all variables and the collection of all operations, respectively, in the system.

As an illustration, consider a simplified poker game, where there are two poker players, P1 and P2, who can check their respective individual scores by calling an operation Check–Score of the Game Manager (GM). Those scores are stored separately at a memory location Data Storage (DS) which can be accessed by GM through an operation call Data-Retrieval. The system is depicted in Fig. 1, we have the following service instance specification for GM:

1. requires interface DM
2. operation Data-Retrieval
3. Data-Retrieval.P = {PlayerID}
4. PlayerID.D = {P1, P2}
5. Data-Retrieval.R = {PlayerScore}
6. PlayerScore.D = {0, 1, ..., 100}

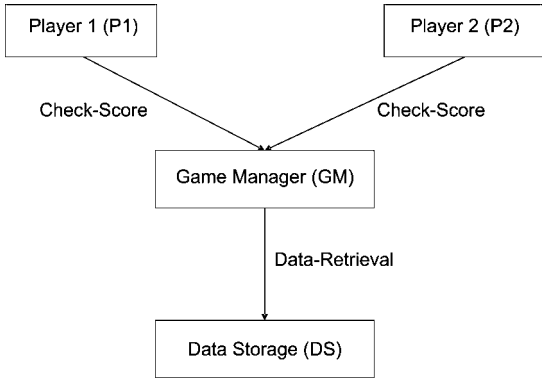

Fig. 1 *Simple poker game system*

7. provides interface IPlay
8. operation Check-Score
9. Check-Score.P = {PlayerID}
10. PlayerID.D = {P1, P2}
11. Check-Score.R = {PlayerScore}
12. PlayerScore.D = {0, 1, . . . , 100}
13. behaviour:
14. DM.Data-Retrieval

Note that GM has no interest about how Data-Retrieval is executed. Therefore the behaviour model of Data-Retrieval is not provided in the specification of GM. The similar situation occurs in the specifications for P1 and P2, described as follows. It is worth to emphasise that unprovided behaviour is different from empty behaviour because the latter says that there is no internal calls.

The service instance specification for P1 is:

1. requires interface IPlay
2. operation Check-Score
3. Check-Score.P = {P1ID}
4. P1ID.D = {P1, P2}
5. Check-Score.R = {P1Score}
6. P1Score.D = {0, 1, . . . , 100}

The service instance specification for P2 is

1. requires interface IPlay
2. operation Check-Score
3. Check-Score.P = {P2ID}
4. P2ID.D = {P1, P2}
5. Check-Score.R = {P2Score}
6. P2Score.D = {0, 1, . . . , 100}

The service instance specification for DS is

1. provides interface DM
2. operation Data-Retrieval
3. Data-Retrieval.P = {PID}
4. PID.D = {P1, P2}
5. Data-Retrieval.R = {PlData, P2Data}
6. P1Data.D = {0, 1, . . . , 100}
7. P2Data.D = {0, 1, . . . , 100}
8. behavior: empty

From those instance specifications we get the following:

V_{GM} = {PlayerID, PlayerScore}
V_{P1} = {P1ID, P1Score}
V_{P2} = {P2ID, P2Score}
V_{DS} = {PID, P1Data, P2Data}
$V = V_{GM} \cup V_{P1} \cup V_{P2} \cup V_{DS}$
O = {Check-Score, Data-Retrieval}

The information flows in the poker game are depicted in Fig. 2, where flows with the same type of arrow-headed lines belong to one operation call. From Fig. 2 we can see that Player 1 (P1) has (at least) two blocks of data: one is associated with the variable P1ID and the other with the variable P1Score. A value of P1ID can be assigned to the variable PlayerID in GM through the operation call Check-Score. The value of the return argument PlayerScore of Check-Score in GM is assigned to the variable P1Score in P1, which completes the operation call Check-Score. The information flow between P2 and GM is interpreted in the same way. The flow between GM and DS is a little bit complicated in the sense that the value of the return argument of the operation call Data-Retrieval in DS conditionally depends on the value of the input argument PID of Data-Retrieval in DS. If PID = P1, then the value of the argument P1Data in DS is assigned to PlayerScore in GM; otherwise, the value of P2Data in DS is assigned to PlayerScore in GM. The diagram suggests that, if there is no access control mechanism, then it is possible for Player 1 to obtain scores of Player 2 by simply assigning the value P2 from P1ID to PlayerScore. Apparently, this kind of flow violates the requirement of confidentiality, thus, should not be allowed. In the following part of this paper, we will propose a systematic way to find all such 'bad' flows and to develop an access control mechanism to block them. To that end, we first formalise the concept of assignments.

Definition 1: An assignment is a three-tuple [$v.x$, o, v'], where v, $v' \in V$, $x \subseteq v.D$ and $o \in O$.

In the above definition, the three-tuple [$v.x$, o, v'] denotes the assignment of any value of $x$to the variable v' through the operation call o. If x is a singleton, say {a}, then we simply use $v.a$ to denote $v.\{a\}$. For example, we have the following assignments between P1 and GM:

[P1ID.P1, Check-Score, PlayerID] and
[P1ID.P2, Check-Score, PlayerID]

which says that the value P1 (or P2) of the variable P1ID (or P2ID) is assigned to the variable PlayerID which is the input argument of the operation call Check-Score. Between GM

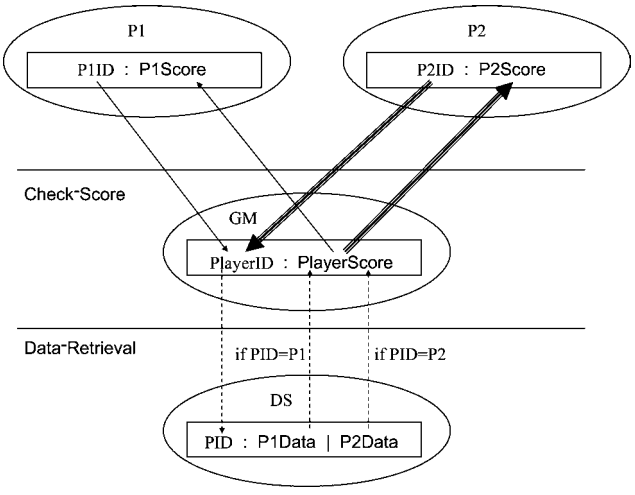

Fig. 2 *Information flows in the poker game*

and DS we have

[PlayerID.P1, Data-Retrieval, PID] and
[P1Data.{1, ..., 30}, Data-Retrieval, PlayerScore]

which says that the value P1 of the variable PlayerID is assigned to the variable PID through the operation call Data-Retrieval, and any value among {1, ..., 30} of the variable P1Data (i.e. the score of P1) can be assigned to the variable PlayerScore through (the return of) the operation call Data-Retrieval. For notation brevity we will use CS to denote Check-Score and DR for Data-Retrieval. In the term $v.x$, if $x = D$ then we simply use $v.\sharp$ instead of using $v.D$, which is only for denoting the domain of v. For any two assignments $[v.x, o, v']$ and $[v.x', o', v'']$, we assume that either $x = x'$ or $x \cap x' = \varnothing$. This assumption will be used in access control to make sure that the disablement of one assignment will not affect executions of other assignments. The concept of disablement will be explained in the following sections.

Let $i_{ass,s}$ be the set of all possible assignments associated with the service instance s, and $A^*_{ass,s}$ the Kleene star (or Kleene closure) of $A_{ass,s}$, that is $A^*_{ass,s}$ is the set of all finite strings, consisting of assignments from $A_{ass,s}$. Each finite string is also called a path. Let $A_{ass} := \cup_{s \in S} A_{ass,s}$ be the overall set of assignments. We assume that A_{ass} is finite.

Definition 2: An atomic action in a service instance s is a finite sequence in $A^*_{ass,s}$.

An atomic action denotes a sequence of assignments that must be finished completely whenever the first assignment is executed. Therefore if some relevant assignment of this sequence fails (owing to errors) or is disallowed by a control mechanism that monitors and manages the execution of the service instance s, then the entire sequence should be abandoned. For example, assigning input arguments of an operation call can be modelled as an atomic action because we can never leave any input argument unassigned when we make the call. If we make a mistake on one assignment, then we need to abandon the current call (i.e. discard all previous assignments) and make a new call. In the above poker game example, each atomic action is simply an assignment. Let $A_{act,s} \subseteq A^*_{ass,s}$ be the set of all atomic actions that can actually happen in s. We assume that $A_{act,s}$ is finite. Let $A_{act} := \cup_{s \in S} A_{act,s}$ be the set of all atomic actions for the system.

Definition 3: A dynamic model of a service instance s is a subset of $A^*_{act,s}$.

A dynamic model describes all possible sequential behaviours of a specific local service instance. For the application purpose, a dynamic model is usually considered as a regular sublanguage of $A^*_{act,s}$. As an illustration, the dynamic model of P1 is

$$L_{P1} := (([\text{P1ID.P1}, \text{CS}, \text{PlayerID}] + [\text{P1ID.P2}, \text{CS}, \text{PlayerID}]) [\text{PlayerScore}.\sharp, \text{CS}, \text{P1Score}])^*$$

which says that P1 repetitively calls the operation CS, in the sense that it passes an ID (either the value P1ID.P1 or P1ID.P2) to the input argument PlayerID of CS, then waits for the return value PlayerScore of CS (to be assigned to P1Score). Similarly, the dynamic model of P2 is described as follows

$$L_{P2} := (([\text{P2ID.P1}, \text{CS}, \text{PlayerID}] + [\text{P2ID.P2}, \text{CS}, \text{PlayerID}]) [\text{PlayerScore}.\sharp, \text{CS}, \text{P2Score}])^*$$

The dynamic model of GM is depicted in Fig. 3, where state with the symbol $\leftrightarrow$ denotes that it is not only the initial state but also a final state. Each path that starts with the initial state and ends at a final state is called recognisable by the automaton. In Fig. 3, there is only one final state. The model says that GM repetitively waits for the operation call CS from either P1 or P2, then makes the call DR to obtain a score from DS and returns the score to the original caller through CS.

In this paper, we focus on centralised access control synthesis. For that sake we need a centralised system model, which can be obtained from composition of dynamic models of local service instances by using synchronous product, which is introduced as follows. Let Σ be an alphabet and $\Sigma' \subseteq \Sigma$. We define the natural projection $P: \Sigma^* \to \Sigma'^*$ as follows

$$P(\epsilon) = \epsilon \tag{1}$$

$$(\forall \sigma \in \Sigma)\, P(\sigma) = \begin{cases} \sigma & \text{if } \sigma \in \Sigma' \\ \epsilon & \text{if } \sigma \in \Sigma' \end{cases} \tag{2}$$

$$(\forall t \in \Sigma^*, \sigma \in \Sigma)\, P(t\sigma) = P(t)P(\sigma) \tag{3}$$

If $B \subseteq \Sigma^*$, then $P(B) := \{P(t) | t \in B\}$. We use $2^{\Sigma'^*}$ to denote the power set of Σ'^*, that is, the collection of all subsets of Σ'^*. The inverse image function of P is P^{-1}: $2^{\Sigma'^*} \to 2^{\Sigma^*}$, defined by

$$(\forall W \in 2^{\Sigma'^*})\, P^{-1}(W) := \{t \in \Sigma^* | P(t) \in W\}$$

In case $W = \{t'\}$, a singleton, we write $P^{-1}(t')$ for $P^{-1}(\{t'\})$.

Let Σ_1, Σ_2 be two alphabets and $\Sigma := \Sigma_1 \cup \Sigma_2$, and $P_1: \Sigma^* \to \Sigma_1^*$ and $P_2: \Sigma^* \to \Sigma_2^*$ be the natural projections. Then for a pair of languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, the synchronous product of L_1 and L_2 is $L_1 || L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$. In other words

$$L_1 \| L_2 = \{t \in \Sigma^* | P_1(t) \in L_1 \ \& \ P_2(t) \in L_2\}$$

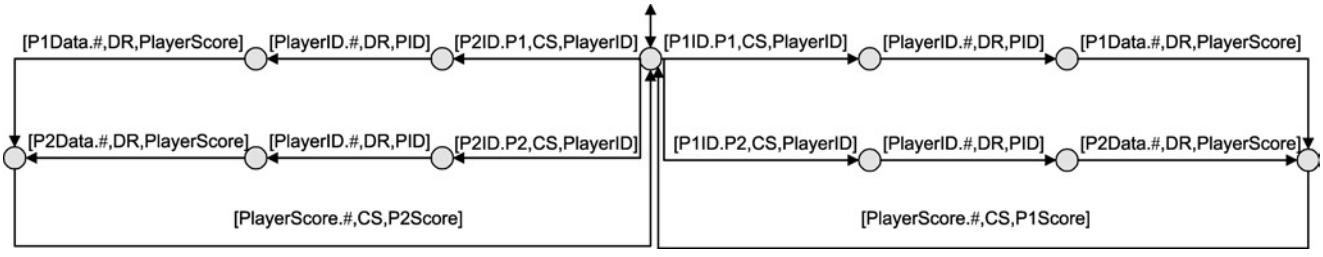

Fig. 3 *Dynamic model of the game manager*

It has been shown that $\|$ is commutative and associative [20]. Therefore, for a family of alphabets $\{\Sigma_i | i \in I\}$ and a set of languages $\{L_i \subseteq \Sigma^* | i \in I\}$, where I is an index set, the I-fold synchronous product $\|_{i\in I} L_i$ is well defined.

In the poker game example, $A_{\text{act},P1}$, $A_{\text{act},P2}$ and $A_{\text{act,GM}}$ are alphabets. The synchronous product $L = L_{P1}\|L_{P2}\|L_{\text{GM}} = L_{\text{GM}}$. Note that each atomic action is a finite sequence of assignments. Synchronous product of two atomic actions is essentially a composition of underlying atomic assignments. Given two languages $L_1 \subseteq A^*_{act,1} \subseteq A^*_{ass,1}$ and $L_2 \subseteq A^*_{act,2} \subseteq A^*_{\text{ass},2}$, if we do not impose any restriction on $A_{\text{act},1}$ and $A_{\text{act},2}$, then it is likely that composition of atomic actions may not be consistent with what really happens on composition of atomic assignments. For example, suppose we have two atomic actions

$$a_1 = [v_1, o_1, v_1'][v_2, o_2, v_2'] \in A_{\text{act},1} \subseteq A^*_{\text{ass},1}$$

$$a_2 = [v_1, o_1, v_1'][v_3, o_3, v_3'] \in A_{\text{act},2} \subseteq A^*_{\text{ass},2}$$

where $A_{\text{ass},1} \cap A_{\text{ass},2} = \{[v_1, o_1, v_1{}']\}$. Because $a_1 \neq a_2$, by the definition of synchronous product over $A_{\text{act},1}$ and $A_{\text{act},2}$, we have the following result

$$\begin{aligned}\{a_1\}\|\{a_2\} = \{&a_1a_2 = [v_1, o_1, v_1'][v_2, o_2, v_2'][v_1, o_1, v_1'] \\ &[v_3, o_3, v_3'], a_2a_1 = [v_1, o_1, v_1'][v_3, o_3, v_3'] \\ &[v_1, o_1, v_1'][v_2, o_2, v_2']\}\end{aligned}$$

Unfortunately, this result does not correctly reflect what happens in the system because the assignment $[v_1, o_1, v_1']$ must be executed simultaneously in both a_1 and a_2. On the other hand, if we compute synchronous product of a_1 and a_2 over $A_{\text{ass},1}$ and $A_{\text{ass},2}$, then we end up with two finite strings $[v_1, o_1, v_1'][v_2, o_2, v_2'][v_3, o_3, v_3']$ and $[v_1, o_1, v_1'][v_3, o_3, v_3'][v_2, o_2, v_2']$, and none of them is an atomic action. Thus, we have encountered inconsistency between composition of atomic actions and assignments. To avoid this inconsistency, we make the following assumptions. Given an atomic action $a \in A_{\text{act}} \subseteq A^*_{\text{ass}}$, we use $\Sigma(a) \subseteq A_{\text{ass}}$ to denote all assignments that appear in a. The assumption says that for any two atomic actions $a \in A_{\text{act},s} \subseteq A^*_{\text{ass},s}$ and $a' \in A_{\text{act},s'} \subseteq A^*_{\text{ass},s'}$, either $a = a'$ or $\Sigma(a) \cap \Sigma(a') = \varnothing$. In other words, two atomic actions are either the same or share no assignments. It turns out that this assumption is only a mild one in the component-based framework. The reason is as follows. Considering that each atomic action is about assignments of either input arguments or return arguments of an operation call, if two atomic actions share a few assignments, then they must share the rest of assignments of (input or return) arguments. Thus, in general, this assumption is rather easy to be satisfied for most (if not all) component-based software applications.

3 Security specifications

Given a system dynamic model, a security specification describes what behaviours are allowed or disallowed in the system. In this paper, such behaviours are modelled as languages.

Definition 4: A language-based security specification is a subset of A^*_{act}.

Moreover, we focus on specifications about disallowed behaviours, namely strings that should be prevented from happening in the system. For that purpose, we introduce the following concept to model 'bad' information flows in the proposed language-based framework.

Definition 5: Given two variables $v, v' \in V$ and $x \subseteq v.D$, we say $v.x$ is retrievable by v' within a model $L \subseteq A^*_{\text{act}}$ if there exists a path $t = [v_1.x_1, o_1, v_1'] \cdots [v_n.x_n, o_n, v_n'] \in L$ such that

1. $(\exists r_1, r_2, \ldots, r_m{:}1 \leq r_1 < r_2 < \cdots < r_m \leq n) v_{r_1} = v \wedge x \subseteq x_{r_1} \wedge v'_{r_m} = v'$
2. $(\forall k{:}\ 1 \leq k \leq m-1) v'_{r_k} = v_{r_{k+1}} \wedge x_{r_k} \subseteq x_{r_{k+1}}$
3. $(\forall k{:}\ 1 \leq k \leq m-1)(\forall i{:}\ r_k < i < r_{k+1})[v_i.x_i, o_i, v_i'] = [v_{r_k}.x_{r_k}, o_{r_k}, v'_{r_k}] \vee v_i' \neq v'_{r_k}$

The path t is called a path of threat from $v.x$ to v'.

The first condition says that along the path t there exists a subsequence of assignments which starts with $v.x$ and ends at v'. The second condition says that, along that sequence of assignments any value of x can be passed to v' through assignments. The last condition says that between every two consecutive assignments in that sequence, say $[v_{r_k}.x_{r_k}, o_{r_k}, v'_{r_k}]$ and $[v_{r_{k+1}}.x_{r_{k+1}}, o_{r_{k+1}}, v'_{r_{k+1}}]$, there is no other assignment along t that can change the value of v_{r_k} before it is passed to $v_{r_{k+1}}$ (otherwise, values of x will get lost before they reach v').

As an illustration, let the synchronous product $L := L_{P1}\|L_{P2}\|L_{\text{GM}}$ be the model of the poker game. One possible security specification in the poker game is that there is no 'peeking' between two players, that is

1. L contains no path of threat from P1Data.♯ to P2Score.
2. L contains no path of threat from P2Data.♯ to P1Score.

It is not difficult to see that there is one path of threat from P1Data.♯ to P2Score, which is

$$\begin{aligned}t_1 = &[\text{P2ID.P1, CS, PlayerID}][\text{PlayerID.}\sharp\text{, DR, PID}] \\ &[\text{P1Data.}\sharp\text{, DR, PlayerScore}] \\ &[\text{PlayerScore.}\sharp\text{, CS, P2Score}]\end{aligned}$$

and one path of threat from P2Data.♯ to P1Score, which is

$$\begin{aligned}t_2 = &[\text{P1ID.P2, CS, PlayerID}][\text{PlayerID.}\sharp\text{, DR, PID}] \\ &[\text{P2Data.}\sharp\text{, DR, PlayerScore}] \\ &[\text{PlayerScore.}\sharp\text{, CS, P1Score}]\end{aligned}$$

We can further show that the set of all paths of threat in the poker game is $L(t_1 + t_2)L$.

Given the collections $\{A_{\text{ass},s} | s \in S\}$, $\{A_{\text{act},s} \subseteq A^*_{\text{ass},s} | s \in \mathcal{S}\}$ and $\{L_s \subseteq A^*_{\text{act},s} | s \in \mathcal{S}\}$, it may be convenient for a user to specify only a two-tuple $(v.x, v')$, saying that $v.x$ is not retrievable by v'. Then, we need an automatic procedure to compute a collection L_E $(v.x, v')$ of all paths of threat in the system $L := \|_{s\in S} L_s$. For that sake, we provide the following algorithm:

1. Let $A_{\text{ass}} := \cup_{s\in S} A_{\text{ass},s}$.
2. Construct all paths $\mu = [v_1.x_1, o_1, v_1'] \cdots [v_m.x_n, o_m, v_m'] \in A^*_{\text{ass}}$ satisfying the following
   - $v_1 = v \wedge x \subseteq x_1 \wedge v_n = v'$
   - $(\forall k{:}\ 1 \leq k \leq m-1) v_k' = v_{k+1} \wedge x_k \subseteq x_{k+1}$

   Let S be the collection of those paths.
3. S obtained above can be proved to be regular, thus, recognisable by a finite-state automaton, say $\boldsymbol{S} = (Y, \Sigma, \xi, y_0, Y_m)$, where Y is the state set, Σ the alphabet (i.e. the collection of all assignments in S), ξ a (partial) transition map, y_0 the initial state and $Y_m \subseteq Y$ the marker

(or final) states. Perform the following revisions on $\boldsymbol{S}$. At each state $y \in Y$, let

$$\phi(y) := \{\sigma \in \Sigma | \xi(y, \sigma) \text{ is defined}\}$$
$$\psi(y) := \{\sigma \in \Sigma | (\exists y' \in Y)\xi(y', \sigma) = y\}$$
$$\theta(y) := \{[\hat{v}.z, \hat{o}, \hat{v}'] \in A_{\text{ass}} - \psi(y) | (\exists [\tilde{v}.w, \tilde{o}, \tilde{v}'] \in \phi(y))\hat{v}' = \tilde{v}\}$$

Here $\phi(y)$ denotes all actions exiting from y, $\psi(y)$ for actions entering y, and $\theta(y)$ for actions that may alter values of some variables associated with actions in $\phi(y)$ (thus, violates condition (3) in Definition 5). For each element $\sigma \in \theta(y)$, we add a new transition $\xi(y, \sigma) = y_0$. Then, we selfloop all elements of $A_{\text{ass}} - \phi(y) - \theta(y)$ at y. Let the resulting finite-state automaton be $\boldsymbol{S}'$, which generates the language W.

Proposition 1: In Step (2), the set $S$is computable within a finite number of steps.

Proof: Let

$$H := \{(v, x)\} \cup \{(\mu', x') | \mu' \in V \wedge x' \subseteq \mu'.D \wedge (\exists o' \in O, \mu'' \in V)[\mu'.x', o', \mu''] \in A_{\text{ass}}\}$$

Since, by assumption, A_{ass} is finite, the set H is also finite. We construct a directed graph Gr = (Ver,Edg), where Ver $\subseteq H$ denotes the vertex set of Gr and Edg $\subseteq$ Ver $\times$ Ver the edge set of Gr, such that the following condition holds:

(a) The root node of Gr is (v, x).
(b) $([\mu', x'], [\mu'', x'']) \in$ Edg iff there is an assignment $[\mu'.\hat{x}', o', \mu''] \in A_{\text{ass}}$ with $x' \subseteq \hat{x}' \subseteq x''$.
(c) Gr is the largest graph satisfying (a) and (b).

Since Ver is finite, the edge set Edg is also finite. Thus, the directed graph Gr must be a finite graph, which can be constructed within a finite number of steps. Let

$$g\text{: Edg} \to 2^{A_{\text{ass}}}\text{: } ([\mu', x'], [\mu'', x'']) \mapsto g([\mu', x'], [\mu'', x'']) := \{[\mu'.\hat{x}', o', \mu''] \in A_{\text{ass}} | x' \subseteq \hat{x}' \subseteq x''\}$$

be a mapping which labels each edge of Gr with a collection of assignments that satisfy condition (b). Given a directed path $[v_1, x_1][v_2, x_2] \cdots [v_m, x_m]$ in Gr let

$$l([v_1, x_1] \cdots [v_m, x_m]) := g([v_1, x_1], [v_2, x_2])g([v_2, x_2], [v_3, x_3]) \cdots g([v_{m-1}, x_{m-1}], [v_m, x_m])$$

denote all sequences of atomic assignments that associate with the directed path, where

$$g([v_1, x_1], [v_2, x_2])g([v_2, x_2], [v_3, x_3]) \cdots g([v_{m-1}, x_{m-1}], [v_m, x_m])$$

denotes concatenation of sets $g([v_1, x_1], [v_2, x_2])$, $g([v_2, x_2], [v_3, x_3]), \ldots, g([v_{m-1}, x_{m-1}], [v_m, x_m])$. Let ϕ be the set of all directed paths in Gr, which start with (v, x) and end at (v', x') for some $x' \subseteq v'.D$. We can see that ϕ can be encoded as a subgraph Gr$' = (\text{Ver}', \text{Edg}')$ of Gr, which is finite. Attach to each edge of Gr$'$ the corresponding label. Then, we can see that $S = \cup_{p \in \Phi} l(p)$. Thus, the proposition follows. □

From the proof of Proposition 1 we can see that the resulting directed subgraph Gr$'$ associated with labels on its edges is a finite state machine, whose state set is simply the vertex set Ver$'$ of Gr$'$, its alphabet is $\cup_{e \in \text{Edg}} g(e)$ and its (partial) transitions are the following three-tuples

$$\bigcup_{([\mu', x'], [\mu'', x''']) \in \text{Edg}'} \{[\mu', x']\} \times g([\mu', x'], [\mu'', x'']) \times \{[\mu'', x'']\}$$

Thus, S is regular. We now have the following result.

Proposition 2: Let $(v.x, v')$, L and W be the same as above. Then, $L_E(v.x, v') = L \cap W$.

Proof: On the basis of the description of the algorithm and Proposition 1 we can see that $L_E(v.x, v') \subseteq L \cap W$. Thus, we only need to show that $L \cap W \subseteq L_E(v.x, v')$. Let

$$t = [v_1.x_1, o_1, v_1'] \cdots [v_n.x_m, o_m, v_m'] \in L \cap W$$

Since $t \in W$ and W is recognisable by $\boldsymbol{S}'$, by the definition of $\boldsymbol{S}'$ (which is derived from $\boldsymbol{S}$, i.e. from Gr$'$ in Proposition 1), we get that

1. $(\exists r_1, r_2, \ldots, r_m: 1 \le r_1 < r_2 < \cdots < r_m \le n) v_{r_1} = v \wedge x \subseteq x_{r_1} \wedge v'_{r_m} = v'$
2. $(\forall k: 1 \le k \le m-1) v'_{r_k} = v_{r_{k+1}} \wedge x_{r_k} \subseteq x_{r_{k+1}}$
3. $(\forall k: 1 \le k \le m-1)\ (\forall i: r_k < i < r_{k+1})[v_i.x_i, o_i, v_i'] = [v_{r_k}.x_{r_k}, o_{r_k}, v'_{r_k}] \vee v_i' \ne v'_{r_k}$

By Definition, 5 we know that $t \in L_E(v.x, v')$. Thus, $L \cap W \subseteq L_E(v.x, v')$, as required. □

If the user has more than one specification, say $\{(v_i.x_i, v_i') | i \in I\}$ for some finite index set I, then the set $L_E := \cup_{i \in I} L_E(v_i.x_i, v_i')$ contains all paths of threat. As an illustration, in the poker game a user may provide the following specifications

[P1Data.♯, P2Score] and [P2Data.♯, P1Score]

In Step (1), we construct the set A_{ass} which is

$$A_{\text{ass}} = \{[\text{P1ID.P1, CS, PlayerID}], [\text{P1ID.P2, CS, PlayerID}], [\text{P2ID.P1, CS, PlayerID}], [\text{P2ID.P2, CS, PlayerID}], [\text{PlayerID.♯, DR, PID}], [\text{P1Data.♯, DR, PlayerScore}], [\text{P2Data.♯, DR, PlayerScore}], [\text{PlayerScore.♯, DR, P1Score}], [\text{PlayerScore.♯, DR, P2Score}]\}$$

InStep (2), we construct the set S, which turns out to be

$$S = \{[\text{P1Data.♯, DR, PlayerScore}] [\text{PlayerScore.♯, CS, P2Score}], [\text{P2Data.♯, DR, PlayerScore}] [\text{PlayerScore.♯, CS, P1Score}]\}$$

We can check that S is recognised by the following finite state automaton depicted in Fig. 4. Notice that in this automaton the initial state is the one with an incoming arrow-headed line '→' without any edge label, that is, y_0 is the initial state. A final state is the one with an outgoing arrow-headed line '→' without any edge label. The following

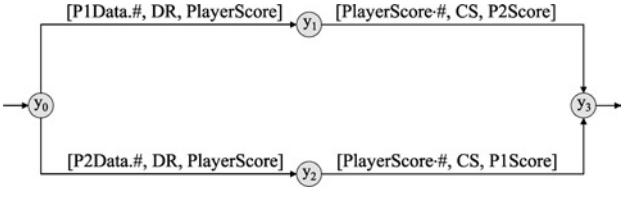

Fig. 4 *FSA* $\boldsymbol{S}$ *that recognises the language S*

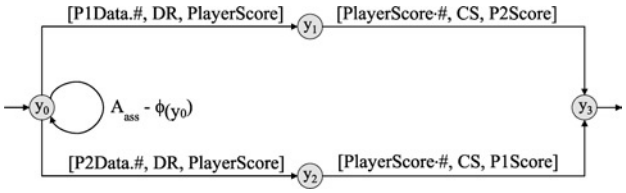

Fig. 5 *Modify transitions at y_0 of $\boldsymbol{S}$*

automaton models obey the same notation rule. In Fig. 4, y_3 is the only final state. In Step (3) we modify $\boldsymbol{S}$ as follows:

- For state y_0, we have

$$\begin{aligned}\phi(y_0) :=& \{[\text{P1Data.}\sharp, \text{DR}, \text{PlayerScore}],\\ & [\text{P2Data.}\sharp, \text{DR}, \text{PlayerScore}]\}\\ \psi(y_0) :=& \varnothing\\ \theta(y_0) :=& \varnothing\end{aligned}$$

Thus, at state y_0, we only selfloop all elements of $A_{\text{ass}} - \phi(y_0) - \psi(y_0) - \theta(y_0) = A_{\text{ass}} - \phi(y_0)$. The resulting FSA is depicted in Fig. 5.

- For state y_1 we have

$$\begin{aligned}\phi(y_1) :=& \{[\text{PlayerScore.}\sharp, \text{CS}, \text{P2Score}]\}\\ \psi(y_1) :=& \{[\text{P1Data.}\sharp, \text{DR}, \text{PlayerScore}]\}\\ \theta(y_1) :=& \{[\text{P2Data.}\sharp, \text{DR}, \text{PlayerScore}]\}\end{aligned}$$

Thus, at state y_1 we selfloop all elements of $A_{\text{ass}} - \phi(y_1) - \theta(y_1)$ and add one more transition $\xi(y_1, [\text{P2Data.}\sharp, \text{DR}, \text{PlayerScore}]) = y_0$. The resulting FSA is depicted in Fig. 6.

- For state y_2 we use the similar treatment as y_1. The resulting FSA is depicted in Fig. 7.
- Finally, for state y_3 we have

$$\begin{aligned}\phi(y_3) :=& \varnothing\\ \psi(y_3) :=& \{[\text{PlayerScore.}\sharp, \text{CS}, \text{P1Score}],\\ & [\text{PlayerScore.}\sharp, \text{CS}, \text{P2Score}]\}\\ \theta(y_3) :=& \varnothing\end{aligned}$$

Thus, at state y_3 we selfloop all elements of A_{ass}. The resulting FSA, which is named as $\boldsymbol{S}'$, is depicted in Fig. 8. The language W generated by $\boldsymbol{S}'$ is all strings that start with the initial state y_0 and end at the marker (or final) state y_3. We can show that $L_E = L \cap W$ is $L(t_1 + t_2)L$ (as claimed before), where $L = L_{P1} \| L_{P2} \| L_{GM}$ and

$$\begin{aligned}t_1 =& [\text{P2ID.P1}, \text{CS}, \text{PlayerID}][\text{PlayerID.}\sharp, \text{DR}, \text{PID}]\\ & [\text{P1Data.}\sharp, \text{DR}, \text{PlayerScore}]\\ & [\text{PlayerScore.}\sharp, \text{CS}, \text{P2Score}]\\ t_2 =& [\text{P1ID.P2}, \text{CS}, \text{PlayerID}][\text{PlayerID.}\sharp, \text{DR}, \text{PID}]\\ & [\text{P2Data.}\sharp, \text{DR}, \text{PlayerScore}]\\ & [\text{PlayerScore.}\sharp, \text{CS}, \text{P1Score}]\end{aligned}$$

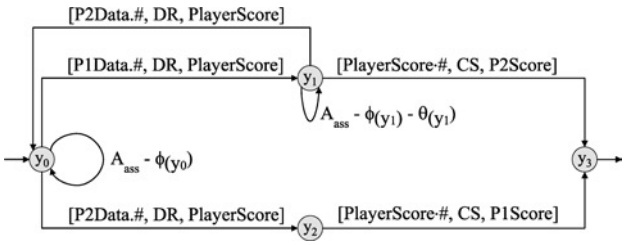

Fig. 6 *Modify transitions at y_1 of $\boldsymbol{S}$*

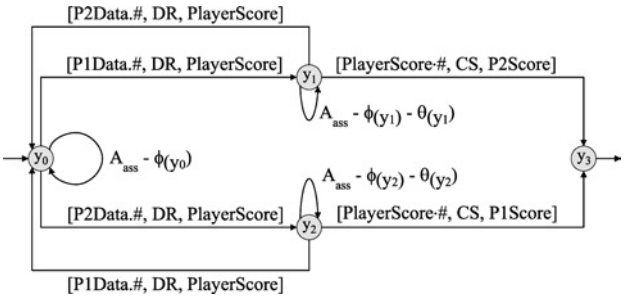

Fig. 7 *Modify transitions at y_2 of $\boldsymbol{S}$*

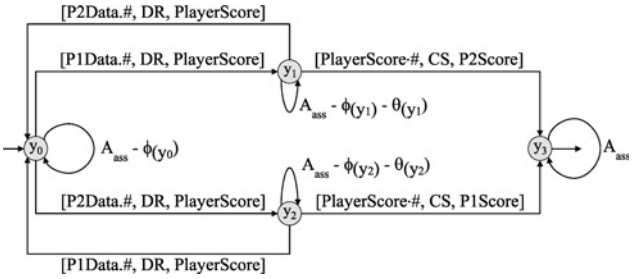

Fig. 8 *Modify transitions at y_3 of $\boldsymbol{S}$*

4 Supremal controllable behaviour satisfying security specifications

Let L_E be the collection of all possible paths of threat with respect to those given two-tuple specifications. The language $L - L_E$ is the collection of all sequences of assignments that will not result in information leakage. From an application point of view, if there is a path $t \in L - L_E$ and a path $t\sigma \in L_E$, then it is required for an access control unit to be able to disable (or forbid) the execution of the atomic action $\sigma \in A_{\text{act}}$ following t. Sometimes this is not possible. For example, in the poker game, the return of CS may not be externally blocked after the assignments of input arguments. If P1 uses the ID of P2 to call CS (i.e. [P1ID.P2, CS, PlayerID]), and if the access control unit allows such a assignment, then P1 will get the score of P2 in the end. Thus, to capture such a phenomenon we introduce the concept of controllability.

We partition A_{ass} into two disjoint sets $A_{\text{ass,c}}$ and $A_{\text{ass,uc}}$, where each assignment in $A_{\text{ass,c}}$ is called a controllable assignment, denoting that a user has means to forbid its execution. and each element in $A_{\text{ass,uc}}$ is called an uncontrollable assignment. It can be the architect of the system who decides which assignments are controllable and which are not. An atomic action in A_{act} is controllable if it consists of only controllable assignments; otherwise it is

uncontrollable. Let $A_{act,c} \subseteq A_{act}$ be the collection of all controllable atomic actions in A_{act}, and $A_{act,uc} := A_{act} - A_{act,c}$ the collection of all uncontrollable atomic actions. Given two sequences $t, t' \in A_{act}^*$, we say $t$is a prefix substring of t', denoted as $t \leq t'$, if

$$(\exists \mu \in A_{act}^*)\, t\mu = t'$$

Given a sublanguage $L' \subseteq A_{act}^*$, we use $\overline{L'} := \{t \in A_{act}^* | (\exists t' \in L')t \leq t'\}$ to denote the prefix closure of L'.

Definition 6: Given a language $L \subseteq A_{act}^*$, a sublanguage $L' \subseteq L$ is controllable with respect to L and $A_{act,uc}$ if $\overline{L'}A_{act,uc} \cap \bar{L} \subseteq \overline{L'}$.

Defition 6 says that L' is controllable with respect to L and $A_{act,uc}$ if there exists no sequence $t \in \overline{L'}$ that can be extended to a sequence $t' \in \bar{L}$, which is outside $\overline{L'}$, by concatenating uncontrollable atomic actions to t. Thus, whenever there is an atomic action σ that makes $t\sigma$ out of $\overline{L'}$, we can disable (or forbid) the execution of σ, because it must be a controllable atomic action. Given a sublanguage $E \subseteq L$, let

$$C_{L,E} := \{K \subseteq E | K \text{ is controllable with respect to} L \text{ and } A_{act,uc}\}$$

be the collection of all controllable sublanguages of E. For any two controllable sublanguages $K_1, K_2 \in \mathcal{C}_{L,E}$, we can derive that

$$\begin{aligned}
\overline{K_1 \cup K_2}A_{act,uc} \cap \bar{L} &= (\overline{K_1} \cup \overline{K_2})A_{act,uc} \cap \bar{L} \\
&\quad \text{by the property of prefix closure} \\
&= (\overline{K_1}A_{act,uc} \cap \bar{L}) \cup (\overline{K_2}A_{act,uc} \cap \bar{L}) \\
&\quad \text{by the property of concatenation} \\
&\subseteq \overline{K_1} \cup \overline{K_2} \text{ because } K_1 \text{ and } K_2 \\
&\quad \text{are controllable sublanguages} \\
&= \overline{K_1 \cup K_2} \\
&\quad \text{by the property of prefix closure}
\end{aligned}$$

which means $L_1 \cup L_2$ is also a controllable sublanguage of E. In fact, it is shown that, the union of a countable or uncountable number of controllable sublanguages of E is still a controllable language of E [20]. Thus, $\mathcal{C}_{L,E}$ is a join-semi-lattice under the partial order of set inclusion. The largest controllable sublanguage of E exists, which is denoted as $\text{Sup}\mathcal{C}_{L,E}$. We aim to compute this largest element, which can be obtained by using techniques developed in the supervisory control theory (SCT) [20]. It has been shown in [21] that the time complexity of computing $\text{Sup}\mathcal{C}_{L,E}$ is polynomial with respect to the size of L (i.e. the size of the state set of the minimum automaton that recognizes L) and the size of E. Nevertheless, we want to point out that, in the worst case the size of L is exponential with respect to the sizes of the constituent components, and so is the size of E. Therefore the centralised controller synthesis approach proposed here is only suitable for dealing with a small or medium size problem. The main purpose of this paper is to introduce this new type of access controller synthesis. For large-scale applications, we may need to use more advanced supervisor synthesis techniques, for example, decentralised, distributed, hierarchical or modular approaches, which will be addressed in our future papers. As an illustration, in the poker game the controllable atomic actions are

$$A_{act,c} = \{[\text{P1ID.P1, CS, PlayerID}], [\text{P1ID.P2,CS,PlayerID}], [\text{P2ID.P1, CS, PlayerID}], [\text{P2ID.P2, CS, PlayerID}]\}$$

The legal behaviour $L - L_E = L - L(t_1 + t_2)L$ is depicted in Fig. 9. It turns out that $L - L_E$ is controllable with respect to L and $A_{act,uc} = A_{act} - A_{act,c}$. Thus, $\text{Sup}\mathcal{C}_{L,L-L_E} = L - L_E$. At the initial state as shown in Fig. 9 neither [P1ID.P2, CS, PlayerID] nor [P2ID.P1, CS, PlayerID] is allowed, that is, a player cannot pretend to be the other player when calling CS. Practically, to make sure P1ID.P2 and P2ID.P1 are detectable, each player can be assigned a unique password allowing P1ID or P2ID to be determined by an access control unit, which belongs to the issue of identification and authentication.

The access controller synthesis (ACS) approach proposed above is similar to the work of centralised glue code synthesis (GCS) [22–24]. In GCS, a centralised system dynamic model is constructed from components' dynamic models. Then a centralised adaptor is constructed to make sure that the dynamic behaviour of the coordinated system, that is, the original system and the adaptor, satisfies the given specifications, for example, deadlock free. Since the ACS framework proposed here and GCS in the literature are more or less instantiations of the SCT in different areas, it is not surprising for us to see their similarity at the conceptual (or general) level, although their system models may be different, owing to different problems that they each deal with, which result in different computational procedures. For example, in GCS the objective of synthesis is usually to remove 'bad' states, for example, deadlock or livelock states, but in this paper we focus on removing paths, which need not necessarily result in state removals.

The proposed synthesis approach is also similar to approaches in the model/module checking (MMC) [25–28]. But, the goals of ACS and MMC are fundamentally different. The goal of MMC is to verify weather the given system satisfies all specifications. The outcome of such verification is usually a binary decision: either 'yes' or 'no', with a few counter examples when no is

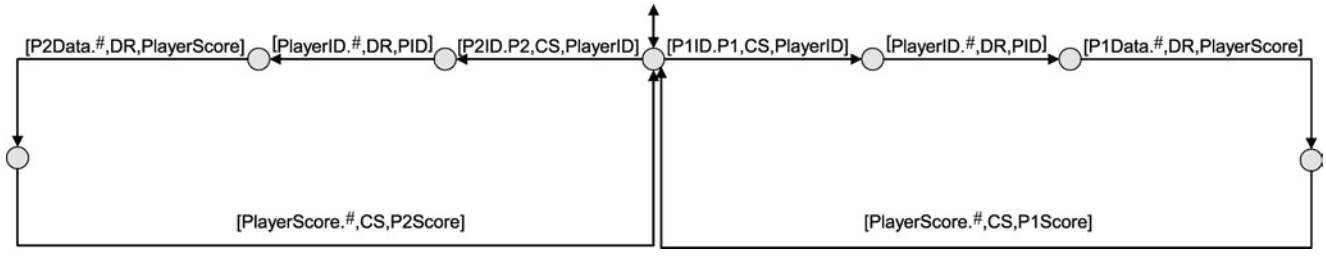

Fig. 9 *Legal behaviour* $L - L_E$

encountered. In contrast, the goal of ACS is to constrain the system behaviour so that any possible undesired behaviour will not occur. Thus, ACS is far beyond simply providing a binary decision as MMC does. Nevertheless, many computational techniques for MMC can be used in ACS, for example, we may use binary decision diagrams (BDDs) to encode those finite-state automata to make computation more efficient in terms of space and time complexity.

5 One way of implementing the proposed access control mechanism

The language-based access control mechanism essentially contains three types of information: (1) all possible paths denoting evolution behaviour of the system; (2) for each path the set of atomic actions that are subsequently allowed; (3) for each path the set of atomic actions that are subsequently disallowed. If a language is generated by an automaton, then we simply replace the term 'path' with the term 'state'. The idea of such a control mechanism is similar to history-based access control, [29, 30], although technical details are different. Regarding how to block a flow from one value to a variable, we can adopt the concept of privilege levels to fulfill the task of enabling and disabling specific transitions. More explicitly, by assigning to each value or variable a specific set of privilege levels, a flow is allowed if and only if it is from a value with low privilege to a variable with higher privilege (i.e. read low, write high), or vice versa (read high, write low). At this point we can see that the access controller synthesis proposed in this paper can play a role complement to those existing access control techniques in the sense that, the proposed approach computes an access control mechanism that can be implemented by existing techniques such as assigning the privilege levels used in type-based or lattice-based access control. Moreover, we will show that the computed access control mechanism can provide a guidance on how to implement it efficiently, for example, to assign static privilege levels in a systematical way as described in the remaining of this section.

Given a system model $L \subseteq A_{act}^*$ and the controllable sublanguage $L' := \mathrm{Sup}\mathcal{C}_{L,L-L_E}$, where L_E consists of all paths of threat based on security specifications. Let $\Sigma_{L'} \subseteq A_{ass}$ be the collection of all assignments appearing in L', and $\Sigma_{L,L',c} \subseteq A_{ass}$ the collection of all controllable assignments that need to be disabled in L in order to obtain L'. We assume that $\Sigma_{L'} \cap \Sigma_{L,L',c} = \varnothing$, which says that disabling assignments in $\Sigma_{L,L',c}$ will not cause any assignment in $\Sigma_{L'}$ to be accidentally disabled. If this assumption does not hold, then the static privilege-assignment approach cannot be used to realise the proposed control mechanism because it is impossible to assign static privilege levels to two entities such that in one circumstance their privilege levels disallow information flow between them, but in another circumstance the opposite happens. Suppose the assumption holds. Then we introduce the following concept.

Definition 7: Let V be the set of variables appearing in $\Sigma_{L'} \cup \Sigma_{L,L',c}$, and

$$Q := V \cup \{v.x | v \in V \wedge (\exists [v.x', o, v'] \in A_{ass})x = x'\}$$

Let $\mathbb{N}$ be the set of all natural numbers. A function

$$f: Q \to \mathbb{N}$$

is a security mapping with respect to L and L' if

1. $(\forall v \in V)(\forall v.x \in Q)f(v) \leq f(v.x)$
2. $(\forall [v.x, o, v'] \in \Sigma_{L,L',c})f(v.x) > f(v')$
3. $(\forall [v.x, o, v'] \in \Sigma_{L'})f(v.x) \leq f(v')$

The first condition in Definition 7 says that the security level of a variable is always at most as high as any value that it may take. The second and the third conditions say that an atomic assignment is allowed only when the security level of the value is not higher than that of the receiving variable. We can see that conditions 2 and 3 implement the principle of read-low and write-high. As an illustration, in the poker game we have

$$\Sigma_{L,L',c} = \{[\mathrm{P1ID.P2}, \mathrm{CS}, \mathrm{PlayerID}], [\mathrm{P2ID.P1}, \mathrm{CS}, \mathrm{PlayerID}]\}$$

and $\Sigma_{L'} = A_{ass} - \Sigma_{L,L',c}$. We can choose the partial order $\leq$ as the ordinary total order associated with the real numbers. The security mapping f can be defined as follows

$$(\forall w \in \mathcal{R})f(w) := \begin{cases} 0 & \text{if } w \in \{\mathrm{P1ID.P2},\ \mathrm{P2ID.P1},\ \mathrm{P1ID},\ \mathrm{P2ID}\} \\ 1 & \text{otherwise} \end{cases}$$

Whenever a service instance requests to execute an assignment $[v.x,\ o,\ v'] \in A_{ass}$, the access control unit will first check whether such an assignment exists. To that end, the access unit holds information about $A_{ass,s}$ for each service instance s. If $[v.x, o, v']$ is indeed a pre-specified assignment, then the access control unit will compare $f(v.x)$ and $f(v')$, and decide, based on the read-low write-high principle, whether the request for execution can be granted. We have the following result.

Proposition 3: Given two regular languages $L \in A_{act}^*$ and $L' \subseteq L$, let Q be the same as above. Then the existence of a security mapping $f: Q \to \mathbb{N}$ with respect to L and L' is decidable.

Proof: Since Q is finite, and each pair of elements $a,b \in Q$ has only two possibilities: either $f(a) \leq f(b)$ or $f(a) > f(b)$. Thus, we only need $|Q|$ distinct values, say $R := \{1, 2, \ldots, |Q|\}$, where $|Q|$ denotes the cardinality of Q. If there exists a security mapping $\hat{f}: Q \to R$, then there must exist a security mapping $f: Q \to \mathbb{N}$. On the other hand, if there exists a security mapping $f: Q \to \mathbb{N}$, then we can define a new function

$$g: \mathbb{N} \to R$$

such that

$$(\forall a, b \in Q)f(a) \leq f(b) \Longleftrightarrow g(f(a)) \leq g(f(b))$$

The existence of g is obvious because we can arrange values of $f(Q)$ in an ascending order, and for each element $a \in Q$, the location of the value of $f(a)$ in that order can be defined as the value of $g(f(a))$. We now have a security mapping $\hat{f}: Q \to R$, where $\hat{f} := g \circ f$. Therefore, there exists a security mapping $f: Q \to \mathbb{N}$ if and only if there exists a security mapping $\hat{f}: Q \to R$. Since both Q and R are finite, the existence of a security mapping $f: Q \to R$ can be decided in a finite number of steps. Thus, the proposition is true. □

In fact, we can use the following procedure to decide the existence of a security mapping $f: Q \to \mathbb{N}$. Suppose $\Sigma_{L'} \cup \Sigma_{L,L',c} = \{[v_1.x_1,\ o_1, v_2], \ldots, [v_n.x_n, o_n, v_{n+1}]\}$, where $v_i.x_i \subseteq v_i.D$. The finiteness of $\Sigma_{L'} \cup \Sigma_{L,L',c}$

comes from the assumption that A_{ass} is chosen to be finite. We construct a directed graph Gr = (Ver, Edg) as follows.

1. Let Ver := $\mathcal{Q}$ be the vertex set of Gr.
2. For each $v \in V$ and $v.x \in \mathcal{Q}$, draw a directed edge from v to $v.x$. Thus, $(v,v.x) \in$ Edg.
3. For each assignment $[v.x, o.v'] \in \Sigma_{L'}$, draw a directed edge from $v.x$ to v'.
4. For each assignment $[v.x, o.v'] \in \Sigma_{L,L',c}$, draw a directed edge from v' to $v.x$.
5. The edge set Edg only contains edges described in Steps 2–4.

If there exists a directed loop in Gr such that one of the relevant edge $(v.x,v')$ is associated with an assignment $[v.x, o, v'] \in \Sigma_{L,L',c}$, then we know that the security mapping f does not exist. The reason is simple: the directed loop requires that all relevant nodes in the loop must have the same value under f, but on the other hand, $[v.x, o, v'] \in \Sigma_{L,L',c}$ requires that $f(v.x) \neq f(v')$, contradiction. If there is no such a directed loop in Gr, then we can construct a security mapping f as follows. First, define an equivalence relation $\equiv$ on Ver such that,

$$(\forall a, b \in \text{Ver}) a \equiv b \Longleftrightarrow \text{there is a directed loop in Gr containing } a \text{ and } b$$

Define a quotient graph Gr/$\equiv$, which must be acyclic. Then there exists a value assignment f such that two nodes in different equivalence classes, that is, they are in the quotient graph, have different values, and nodes in the same equivalence class have the same value. It is not difficult to see that such a value assignment is actually a security mapping f: $\mathcal{Q} \to \mathbb{N}$.

The above description suggests that a user can systematically assign privilege levels to relevant entities in a component-based framework, instead of somehow 'guessing' those privilege assignments, as commonly used in those mentioned approaches in the literature.

6 Conclusions

In this paper, we have proposed a language-based access control mechanism. The dynamic of each service instance is modelled by a regular language. Specification for confidentiality are also regular. Then, by solving a control problem we can construct a transition diagram that tells which operation call is allowed and which is not. By this means, every information flow in the system that may lead to a security breach will be blocked. Meanwhile, the controlled system attains its maximum permissiveness.

The current approach is applicable to a system that has only one processor. If multiple processors are used, then the system has concurrent behaviour, that is, more than one assignment can happen at the same instant. To handle that, we need vectors of atomic assignments to capture concurrency. Furthermore, if the system is very large and there are many specifications, then we may need to use more advanced synthesis techniques, for example, decentralised, distributed or modular controller synthesis, to obtain an access control mechanism, These advanced techniques may also allow a target component-based application to be dynamically reconfigurable in the sense that the number of constituent components can be increased or decreased in a runtime environment and only part of the controller related to those reconfigured components need to be updated, which cannot be achieved in the centralised synthesis approach proposed in this paper. All these are our ongoing research topics.

7 References

1 Robocop: robust open component-based software architecture: URL http:\\www.hitech-projects.com\euprojects\robocop\deliverables.htm
2 Public deliverables of the Space4U project: URL http:\\www.hitech-projects.com\euprojects\space4u\deliverables.htm
3 Lampson, B.W.: 'Protection', *ACM SIGOPS Operating Syst. Rev.*, 1974, **8**, (1), pp. 18–24
4 Lampson, B., Abadi, M., Burrows, M., and Wobber, E.: 'Authentication in distributed systems: theory and practice', *ACM Trans. Comput. Syst. (TOCS)*, 1992, **10**, (4), pp. 265–310
5 Bugliesi, M., Colazzo, D., and Crafa, S.: 'Type based discretionary access control'. Fifteenth Int. Conf. Concurrency Theory (CONCUR 2004), London, England, 31 August–3 September, 2004
6 Gordon, A.D., and Jeffrey, A.: 'Types and effects for asymmetric cryptographic protocols'. IEEE Computer Security Foundations Workshop (CSFW), June 2002
7 Myers, A.C., and Liskov, B.: 'Protecting privacy using the decentralized label model', *ACM Trans. Softw. Eng. Method.*, 2000, **9**, (4), pp. 410–442
8 Didriksen, T.: 'Rule based database access control—a practical approach'. Proc. 2nd ACM Workshop on Role-based access control, Fairfax, Virginia, US, 1997, pp. 143–151
9 Li, H., Zhang, X., Wu, H., and Qu, Y.: 'Design and application of rule based access control policies'. Semantic Web and Policy Workshop, 4th Int. Semantic Web Conf., Galway, Ireland, 7 November 2005
10 Denning, D.E.: 'A lattice model of secure information flow', *Comm. ACM*, 1976, **19**, (5), pp. 236–243
11 Sandhu, R.S.: 'Lattice-based access control models', *IEEE Comput.*, 1993, **26**, (11), pp. 9–19
12 Sandhu, R.S., Coyne, E.J., Feinstein, H.L., and Youman, C.E.: 'Role-based access control models', *IEEE Comput.*, 1996, **29**, (2), pp. 38–47
13 Ferraiolo, D.F., Kuhn, D.R., and Chandramouli, R.: 'Role based access control' (Artech House, 2003)
14 Spinellis, D., and Gritzalis, D.: 'Panoptis: intrusion detection using a domain-specific language', *J. Comput. Security*, 2002, **10**, (1–2), pp. 159–176
15 Eckmann, S.T., Vigna, G., and Kemmerer, R.A.: 'STATL: an attack language for state-based intrusion detection', *J. Comput. Security*, 2002, **10**, (1–2), pp. 71–103
16 Ahn, G.J., and Sandhu, R.: 'Role-based authorization constraints specification', *ACM Trans. Inf. Syst. Secur.*, 2000, **3**, (4), pp. 207–226
17 Schneider, F.B.: 'Enforced security policies', *ACM Trans. Inf. Syst. Security*, 2000, **3**, (1), pp. 30–50
18 Inverardi, P., and Mostarda, L.: 'A distributed intrusion detection approach for security software architecture'. Lecture Notes in Computer Science 3527' (Springer, 2005), pp. 168–184
19 Zhang, X., Parisi-Presicce, F., Sandhu, R., and Park, J.: 'Formal model and policy specification of usage control', *ACM Trans. Inf. Syst. Secur.*, 2005, **8**, (4), pp. 351–387
20 Wonham, W.M.: 'Supervisory Control of Discrete-Event Systems'. Systems Control Group, Dept. of ECE, University of Toronto, http://www.control.toronto.edu/cgi-bin/dldes.cgi, 2004
21 Wonham, W.M., and Ramadge, P.J.: 'On the supremal controllable sublanguage and a given language', *SIAM J Control Optim.*, 1987, **25**, (3), pp. 637–659
22 Yellin, D., and Strom, R.: 'Protocol specifications and component adaptors', *ACM Trans. Program. Languages Syst.*, 1997, **19**, (2), pp. 292–333
23 Inverardi, P., and Tivoli, M.: 'Software architecture for correct components assembly', in Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture, LNCS 2804, 2003, pp. 92–121
24 Tivoli, M., and Autili, M.: 'SYNTHESIS, a tool for synthesizing correct and protocol-enhanced adaptors', *RSTI L'Objet J.*, 2006, **12**, (1), pp. 77–103
25 Clarke, E.M., Emerson, E.A., and Sistla, A.P.: 'Automatic verification of finite-state concurrent systems using temporal logic specifications', *ACM Trans. Program. Lang. Syst.*, 1986, **8**, (2), pp. 244–263
26 Kupferman, O., and Vardi, M.Y.: 'Module checking revisited'. Proc. 9th Int. Conf. Computer Aided Verification LNCS 1254, 1997, pp. 36–47
27 Clarke, E.M. Jr., Grumberg, O., and Peled, D.A.: 'Model checking' (MIT Press, Cambridge London, MA, England)
28 Mantel, H.: 'Information flow control and applications – bridging a gap', FME 2001, LNCS 2021, 2001 pp. 153–172

29 Edjlali, G., Acharya, A., and Chaudhary, V.: 'History-based access control for mobile code'. 5th ACM Conf. Comput. Communications Security, San Francisco, CA, USA, 1998, pp. 38–48

30 Banerjee, A., and Naumann, D.A.: 'History-based access control and secure information flow'. In Proc. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS), Nice, France, 8–11 March 2005, pp. 27–48