J. Parallel Distrib. Comput. 87 (2016) 121–144

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Towards the modelling of secure pervasive computing systems: A paradigm of Context-Aware Secure Action System

François Siewe

Software Technology Research Laboratory, De Montfort University, The Gateway, Leicester, LE1 9BH, United Kingdom

HIGHLIGHTS

- Propose CASAS, a paradigm for modelling secure pervasive computing systems.
- Present the syntax and a formal semantics for CASAS.
- Present an algorithm for checking system consistency statically (i.e. at compile time).
- Define a set of operators for building complex CASAS systems from simpler ones in a compositional manner.
- Demonstrate the pragmatics of the proposed formalism using a number of real-world case studies.

ARTICLE INFO

Article history:
Received 21 June 2014
Received in revised form
25 March 2015
Accepted 22 September 2015
Available online 27 October 2015

Keywords:
Access control
Context-aware
Pervasive systems
Secure action system
Compositional verification

ABSTRACT

The design of security-critical pervasive systems is challenging due to that security constraints are often highly dependent on dynamically changing contexts. To increase the trustworthiness of pervasive systems, a dependable approach to system development must be followed, which enables seamless integration of the functional, security and context-awareness requirements. This paper proposes a paradigm which enables the specification of the functional, security and context-awareness requirements of a system in a single formalism, called *Context-Aware Secure Action System (CASAS)*. Its syntax, formal semantics and pragmatics are presented, as well as algorithms and techniques for analysing the behaviour of a pervasive computing system.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

Pervasive computing [37,38] envisions a new generation of distributed systems where computers in their multitude are hidden in the background of the user and interact calmly to provide the user with relevant information and services anytime and anywhere. These computers continuously gather and share any data about the users and their context that are relevant for decision-making and autonomously adapt themselves to changing situations, keeping disruption to the user at the minimum. These data may contain highly sensitive personal information such as the user's location, personal health information, and credit cards or bank details. Hence, to foster widespread use of pervasive computing systems, it is paramount to ensure that data are exchanged and stored securely in these systems. For example, thanks to the advances in technology, handheld devices such as mobile phones and tablet computers are getting smarter and smarter every single day; store more and more user information such as the user preferences, bank details and address book; can calculate the user location at anytime; and can communicate with the rest of the world via the Internet.

Despite the progress made in the security of traditional distributed systems, securing pervasive computing systems poses new challenges due to the dynamic nature of such systems, e.g. the topology is not fixed, the network is wireless, and computation is context-aware. Indeed, security requirements are often highly dependent on dynamically changing contexts such as user activity, location, nearby people and available resources. In order to increase the trustworthiness of pervasive computing systems, a dependable approach to system development must be followed, which enables seamless integration of the functional, the security and the context-awareness requirements of the system.

E-mail address: fsiewe@dmu.ac.uk.
URL: http://www.cse.dmu.ac.uk/~fsiewe/.

http://dx.doi.org/10.1016/j.jpdc.2015.09.008
0743-7315/© 2015 Elsevier Inc. All rights reserved.

This paper proposes the *Context-Aware Secure Action System (CASAS)*, a programming paradigm which enables the specification of all the three types of system requirements (the functional, security and context-awareness requirements) using a single formalism. In CASAS, a system is modelled as a collection of *agents* and *actions*; whereby *subject* agents can perform actions on *object* agents depending on the environment *context* and the *security policy*. The main contributions of this work can be summarised as follows:

- We propose a syntax for CASAS (Section 3); its innovative features include constructs for enabling actions to be aware of the environment context, and notations to specify security policy rules as constraints upon subject attributes, object attributes and the environment attributes. Access control condition is checked at the beginning and during the execution of any action; if the access right is revoked during execution (perhaps due to changes in the environment context), the action is stopped immediately (likewise the notion of *continuity of control* in UCON [32]). A notion of *system consistency* is defined which ensures that a system is well-formed and secure; where a secure system is defined as a system in which any action is executed only when the subject agents of that action possess the required access rights upon the object agents.
- We give a formal semantics of CASAS in Linear Temporal Logic (LTL) (Section 4); this provides a formal basis for reasoning about the behaviour of CASAS systems. Thus, safety and liveness properties of pervasive computing systems w.r.t. function, security and context-awareness can be analysed and verified in a uniform manner.
- We present an algorithm for checking *system consistency* statically (i.e. at compile time) (Section 7). This algorithm can be used in a parser to check that a system is well-formed and secure.
- We define a set of operators for building complex CASAS systems from simpler ones in a compositional manner (Section 9). We formally prove that these operators are closed under system consistency, i.e. *a system built from well-formed and secure components is well-formed and secure*.
- We also prove the algebraic properties of these operators (Section 10). Algebraic properties are customarily used to simplify the specification of a system into its canonical form.
- We propose a set of compositional proof rules (Section 11). This provides a syntactical approach to system verification; more accessible to programmers and system designers as it hides the complicated formal semantics. We prove the soundness of these proof rules w.r.t. the formal semantics of CASAS.
- Finally, we demonstrate the pragmatics of the proposed formalism using a variety of examples: a context-aware smart phone (Section 2), a Linux-like kernel integrity protection model (Section 6) and a ubiquitous learning system (Section 8).

2. Motivation and examples

The action system paradigm was first proposed by [5,6] as a paradigm for describing reactive systems. We have extended the paradigm to cater for the specification of secure and context-aware pervasive computing systems. In CASAS, the state of the system is partitioned between entities called *agent*. These entities can participate in the execution of operations called *actions*. The execution of an action can change the state of participating agents. However, the execution of an action may depend on the environment *context* and the *security policy*. In this section, we use simple examples to illustrate how a pervasive computing system can be specified in CASAS.

Consider a smart phone which autonomously reconfigures itself to adapt to the user's current activity. It does not ring when the user is in a meeting, but can vibrate; it diverts all incoming calls when the user is driving; and reinstates the default setting when the user is not involved in any of these two activities. We assume that in the default setting, the smart phone can ring and vibrate on incoming calls, but does not divert calls. This smart phone system can be modelled in CASAS as follows. We consider an agent named *phone* that has three local variables *ring*, *vib* and *div* ranged over {on, off} and initialised to the default setting, viz:

agent *phone* : **var** *ring*, *vib*, *div*; *ring*, *vib*, *div* := on, on, off.

The notation *ring*, *vib*, *div* := on, on, off represents the assignment of the values on, on and off to the variables *ring*, *vib*, and *div*, respectively.

For each relevant user activity, the smart phone must be able to detect whether the activity is taking place or not and fire the appropriate adaptation action if necessary. To do this, the smart phone needs to have a sensor that senses the user current activity. For the sake of simplicity, we consider three types of activities: *meeting* (e.g. in a meeting, conference or library), *driving* (e.g. driving a car or cycling on the road) and *idle* (i.e. not meeting nor driving). The activity sensor can also be modelled as an agent named *actSensor* and having a single local variable *actName* representing the user current activity, viz:

agent *actSensor* : **var** *actName*; *actName* := idle.

We also use the notation *p.x* to refer to the local variable *x* of an agent *p*, where appropriate. We can now specify the adaption actions that the smart phone must perform once the user activity changes.

In CASAS, an action has a name and is performed **on** an agent (or group of agents) called the *object agent* **by** another agent (or group of agents) called the *subject agent*, in a particular environment **context** represented by an agent (or group of agents) called the *context agent*. The context agent acts as a sensor, whether a physical sensor (e.g. GPS receiver, temperature sensor, accelerometer, or clock) that measures low level context (e.g. location, current temperature, acceleration, or time), or a logical sensor that calculates high-level context as a function of low and/or high level context (e.g. the user is driving, the user is at a meeting, or city of location). It provides the action with necessary context information. An action also has a guard g and a body S, represented as $g \rightarrow S$; the body is executed only if the guard evaluates to true.

For example, the agent *actSensor* can be thought of as a logical sensor that calculates the current user activity using inputs from two other sensors: a GPS receiver and a calendar; each modelled as an agent. For simplicity, we assume that in addition to latitude and longitude coordinates the GPS receiver also provides the speed of the user. The calendar agent indicates whether the user is currently in a meeting

or not. The behaviour of the logical sensor *actSensor* is modelled by the action *senseActivity* defined as follows:

action *senseActivity* **on** *actSensor* **by** *actSensor* **context** *gps*, *calendar* :
`true` → `if` *gps.speed* >= 10 `then` *actSensor.actName* := `driving`
`else if` (*calendar.curTask* = `meeting` ∧ *gps.speed* < 10) `then` *actSensor.actName* := `meeting`
`else` *actSensor.actName* := `idle`
`fi`
`fi`.

where the agents *gps* and *calendar* are defined by:

agent *gps* : **var** *lat*, *long*, *speed*; *speed* := 0.

agent *calendar* : **var** *curTask*.

Therefore, if the user is moving at a speed greater than or equal to 10 miles per hour, it is assumed that she is driving; otherwise she is either in a meeting (if her calendar says so) or her activity is deemed to be idle (i.e. not driving and not meeting). In a similar way the adaptation actions for the smart phone example can be specified as follows:

action *setMeeting* **on** *phone* **by** *phone* **context** *actSensor* :
(*actName* = `meeting` ∧ (*ring* = `on` ∨ *div* = `on`)) → *ring*, *vib*, *div* := `off`, `on`, `off`.

action *setDriving* **on** *phone* **by** *phone* **context** *actSensor* :
(*actName* = `driving` ∧ *div* = `off`) → *ring*, *vib*, *div* := `off`, `off`, `on`.

action *setIdle* **on** *phone* **by** *phone* **context** *actSensor* :
(*actName* = `idle` ∧ *ring* = `off`) → *ring*, *vib*, *div* := `on`, `on`, `off`.

These actions are aware of the activity of the user through the context agent *actSensor*. In addition to context-awareness, security constraints can be placed on actions using access control annotations as in the following example. Consider the security requirement that only the owner of a phone can modify the user preferences of that phone; and can only do so while idle (i.e. not driving nor in a meeting). Assume that the agent *phone* stores the user preferences and the owner's id in a local variable *userPref* and *ownerId*, respectively. The action *setPref* for updating the user preferences of a phone can be specified as follows:

action *setPref* <`write`> **on** *phone* **by** *user* **context** *prefObj*, *actSensor* :
(*phone.userPref* ≠ *prefObj.val*) → *phone.userPref* := *prefObj.val*.

where the agent *user* models the phone user and the agent *prefObj* contains the new user preferences. These agents are defined in the same manner as the agents above. The access control annotation <`write`> means that the subject agent (here *user*) must have the `write` access right upon the object agent (here *phone*) for this action to be executed. If during execution the subject agent loses this access right, the execution terminates at once. In CASAS, access rights are checked continuously during the execution of actions (similarly to the notion of *continuity of control* in UCON [32]). The access rights of agents are defined in a security policy using the predicate **allow**(*s*, *o*, *r*) which denotes a Boolean expression over the attributes (aka local variables) of the subject agent *s*, of the object agent *o* and of the environment represented by the remaining agents in the system. An agent *s* is granted the access right *r* upon an agent *o* in a state where this predicate evaluates to true.

The security policy for this example is then specified by the rule **allow**(*s*, *o*, `write`) = (*s.id* = *o.ownerId* ∧ *actSensor.name* = `idle`), where *s* (respectively, *o*) is a place holder for any subject agent (respectively, object agent) of any action carrying the access control annotation <`write`>. This rule says that a subject agent *s* is granted the access right `write` upon the object agent *o* if *s* represents the owner of *o* and the current activity of the agent *s* is sensed to be idle. Note that the authorisation decision is not based on the identities of the subject and object agents, but solely on their attributes and those of the environmental context. This type of access control policy (aka *attributes-based access control policy* [31,19]) is flexible and suitable for open systems like pervasive computing systems because: (i) one need not know the identities of the subjects and objects at the time the policy is written, but just the relevant security attributes; (ii) the security policy can be modified independently without the need to make changes to any other system components (i.e. agents and actions).

In this section we have illustrated the main features of CASAS using simple examples. The next section will elaborate further on its syntax and (informal) semantics.

3. Syntax of CASAS

The syntax of CASAS is given below, where symbols in **bold** are key-words and optional expressions are surrounded with the symbols '[' and ']'.

A secure context-aware action system is a tuple

CASAS $\widehat{=}$ ($\mathcal{P}$, $\mathcal{A}$, $\mathcal{R}$, *Agents*, *Actions*, *Policy*), where

- $\mathcal{P}$ is a nonempty set of agent names,
- $\mathcal{A}$ is a nonempty set of action names,
- $\mathcal{R}$ is a set of access rights, and must contain at least the two access rights: `read` and `write`.
- *Agents* is a set of agent definitions,
- *Actions* is a set of action definitions, and
- *Policy* is a set of authorisation rules.

Each of these last three components is described below.

Table 1
Syntax of statements.

$S ::= \alpha \mid \alpha; S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{fi}$
$\alpha ::= y := exp \mid \texttt{skip}$
$exp ::= Aexp \mid b \mid Sexp \mid exp\ ,\ exp$
$Aexp ::= x \mid d \mid -\ Aexp \mid Aexp + Aexp \mid Aexp - Aexp \mid Aexp * Aexp \mid Aexp / Aexp$
$Sexp ::= \emptyset \mid \{exp, \ldots, exp\} \mid Sexp \cap Sexp \mid Sexp \cup Sexp \mid Sexp \setminus Sexp$
$b ::= true \mid false \mid Aexp\ rop\ Aexp \mid exp \in Sexp \mid \neg b \mid b \wedge b \mid b \vee b$
$rop ::= = \mid \neq \mid > \mid >= \mid < \mid <=$

3.1. Agent definition

An agent definition has the following form:

$$\textbf{agent}\ p\ [:\ \textbf{var}\ y\ [;\ S]]. \tag{1}$$

where $p \in \mathcal{P}$ is the agent name, y is a nonempty set of local variables and the statement S stands for the initialisation of the local variables in y. The set y is also known as the *state* of the agent p. The initialisation statement S has the form $y' := c$, where c is a list of constants and y' is a non-empty subset of y. It is assumed that a variable which is not explicitly initialised is assigned an arbitrary value (in its range).

In the sequel, we let $Var(p)$ and $Init(p)$ denote the set y of local variables and the initialisation statement S of the agent p, respectively. If $x \in Var(p)$, we use sometimes the notation $p.x$ to refer to the variable x where appropriate.

Definition 3.1 (*Agent Consistency*)**.** An agent definition, with initialisation statement $y := c$, is consistent if no variable occurs more than once in the list y of variables, all variables in y are local and the size of y matches that of the list c.

Example 3.2. Suppose a climate control system is composed of:

- a temperature sensor; modelled as an agent named 'thermo' carrying a single local variable 'temp' initialised to 10 °C:

  agent *thermo* : **var** *temp*; *temp* := 10.

- an air conditioner; modelled as an agent named 'aircon' with a single local variable 'status':

  agent *aircon* : **var** *status*.

- a thermostat; modelled as a stateless agent:

  agent *thermostat*.

3.2. Action definition

An action definition has the following form:

$$\textbf{action}\ a[<R_a>+]\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ [\textbf{context}\ C_a]:\ g_a \rightarrow S_a \tag{2}$$

where $a \in \mathcal{A}$ is the action name; $R_a \subseteq \mathcal{R}$ is a set of access rights; O_a, U_a and C_a are sets of agents; the Boolean condition g_a is the *functional guard* of the action; and the statement S_a is the *body* of the action.

The agents in the set O_a are called *target agents* or *object agents*; those in U_a are called *subject agents*; and the agents in C_a are called *context agents* for this action. The intuition is that the subject agents perform that action on the target agents using the context agents to sense the environment context. The subject agents and the target agents must synchronise (i.e. not involved in the execution of any other action) during the execution of the action; therefore we call an agent in the set $O_a \cup U_a$ a *synchronisation agent* of the action a. Two actions which share a common synchronisation agent cannot be executed in parallel. On the contrary, the agents in the set C_a need not be synchronised during the execution of the action. That is, they can be involved in the execution of another action while the action a is being executed. However, the state of an agent in C_a cannot be modified by the body of the action a, but can only be read.

The set R_a contains the access rights that each subject agent must possess upon each target agent at the beginning and during the execution of the action a. If any subject agent's access right is revoked during the execution of the action (e.g. due to changes in the environment context), the execution of that action stops immediately. The security policy specifies conditions under which agents are granted access rights. We denote by $\mathcal{P}_a \ \widehat{=}\ O_a \cup U_a \cup C_a$, the set of all agents that participate in action a. We also denote by $Var(a) \ \widehat{=}\ \cup_{p \in \mathcal{P}_a} Var(p)$, the set of local variables of all the agents participating in the action a. The functional guard and the body of an action a may only refer to the local variables in $Var(a)$. For each agent $p \in \mathcal{P}$, we denote by $\mathcal{A}_p$ the set of actions the agent p participates in as a synchronisation agent.

The syntax of a statements S is given in Table 1, where y is a (list of) variable(s), *exp* denotes an (list of) expression(s) (i.e. arithmetic, set, or Boolean expressions), x is a variable symbol, d a constant symbol and b stands for a Boolean expression. In an assignment statement $y := exp$, the list *exp* of expressions must match the list y of recipient variables. The `skip` statement does nothing and terminates in one time unit. The symbol ';' denotes the sequential composition of programs. The conditional statement `if` keeps its usual meaning as in imperative programming languages. It follows that a statement is a sequential process of any granularity provided it is deterministic and terminates in finite time. However, an action definition must satisfy additional syntactic constraints as stated in Definition 3.5.

Definition 3.3 (*Dependency*)**.** In a statement S, we say that a variable y_1 depends on a variable y_2 if the final value of y_1 depends on the initial value of y_2.

For example, in the statement of Eq. (3), the final value of the variable $p_2.x$ depends on the initial value of the variable $p_1.x$ while the final value of the variable $p_2.y$ is the constant value 5. According to Definition 3.3, it follows that the variable $p_2.x$ depends on the variable $p_1.x$, but the variable $p_2.y$ does not depend on the variable $p_3.y$.

$$p_2.x, p_2.y := 2 * p_1.x, p_3.y + 1;\ p_2.y := 5. \tag{3}$$

Definition 3.4 (*Direct Information Flow*)**.** Given an action 'a', we say that information can flow from a participating agent p_1 to a participating agent p_2 during the execution of that action if there exists a local variable of p_1 which depends on a local variable of p_2 in the body of that action.

For instance, consider the action in Eq. (4) whose body is defined as in Eq. (3). By Definition 3.4, it follows that information can flow from the agent p_1 to the agent p_2 during the execution of that action; but not from p_3 to p_2.

$$\textbf{action}\ a\ \textbf{on}\ p_1\ \textbf{by}\ p_2\ \textbf{context}\ p_3 : true\ \rightarrow\ p_2.x, p_2.y := 2 * p_1.x, p_3.y + 1;\ p_2.y := 5. \tag{4}$$

Definition 3.5 (*Action Consistency*)**.** The definition of an action 'a' is consistent if it fulfils the following requirements:

- *Requirement* 1: Any variable that occurs in the functional guard or the body of the action belongs to the state of some participating agent; in an assignment statement $y := exp$, no variable may occur more than once in the list y of variables and the size of y must match that of exp.
- *Requirement* 2: The body of the action 'a' must not contain statements that change the state of an agent in C_a.
- *Requirement* 3: In the body of the action 'a', information can flow only from the agents in the set $U_a \cup C_a$ to the agents in the set O_a, or from the agents in the set O_a to the agents in the set U_a; no information flow is allowed between distinct agents in the set $O_a \setminus U_a$ nor between distinct agents in the set $U_a \setminus O_a$.
- *Requirement* 4: If the body of the action 'a' implies information would flow from a subject agent (in U_a) to a distinct object agent (in O_a) then the set R_a must contain the access right `write`. Reciprocally, if the body of the action 'a' implies information would flow from an object agent to a distinct subject agent then the set R_a must contain the access right `read`.

We propose in Section 7 a series of algorithms to check statically that an action definition meets these requirements.

Definition 3.6 (*System Consistency*)**.** A $CASAS$ is consistent if all its action definitions and agent definitions are consistent and no action nor agent is defined more than once.

Definition 3.7 (*Joint action/Private action*)**.** An action that involves more that one distinct participating agent is called a *joint action*, otherwise it is called a *private action*.

Example 3.8. Following up Example 3.2, the thermostat controls the air conditioner using two actions: the action 'switchOff' to switch the air conditioner off when the temperature is below 5 °C and the action 'switchOn' to switch the air conditioner on when the temperature is above 25 °C:

$$\textbf{action}\ switchOff < \texttt{write} > \textbf{on}\ airCon\ \textbf{by}\ thermostat\ \textbf{context}\ thermo : (temp < 5 \wedge status = \texttt{on}) \rightarrow status := \texttt{off}.$$

$$\textbf{action}\ switchOn < \texttt{write} > \textbf{on}\ airCon\ \textbf{by}\ thermostat\ \textbf{context}\ thermo : (temp >= 25 \wedge status = \texttt{off}) \rightarrow status := \texttt{on}.$$

These two actions require the thermostat agent to have `write` access right upon the aircon agent in order to be able to switch it on or off. The context agent thermo senses the environment and provides the current value of the temperature through its local variable temp.

Example 3.9. Assume the temperature decreases by 0.1 °C per time unit when the air conditioner is on and increases by 0.02 °C per unit of time when the air conditioner is off. This is modelled by the following two actions: tempDn and tempUp, respectively.

$$\textbf{action}\ tempDn\ \textbf{on}\ thermo\ \textbf{by}\ thermo\ \textbf{context}\ aircon :\ (status = \texttt{on}) \rightarrow temp := temp - 0.1.$$
$$\textbf{action}\ tempUp\ \textbf{on}\ thermo\ \textbf{by}\ thermo\ \textbf{context}\ aircon :\ (status = \texttt{off}) \rightarrow temp := temp + 0.02.$$

In this example, the thermo agent is at the same time the subject and the target agent. The aircon agent is used here as a context agent that senses the state of the air conditioner (on or off). We assume that no access right is required for the thermo agent to access its own state.

Example 3.10. A digital clock that ticks every time unit can be modelled as follows:

$$\textbf{agent}\ clock :\ \textbf{var}\ time;\ time := 0.$$
$$\textbf{action}\ tick\ \textbf{on}\ clock\ \textbf{by}\ clock : true \rightarrow time := time + 1.$$

Initially, the clock is set to 0. The action 'tick' has no context agent, and rightly so because time is independent of any context. This action increases the clock's time by one unit at each execution step.

3.3. Security policy

The security policy *Policy* is a set of authorisation rules of the form:

$$\textbf{allow}(s, o, r) = b$$

where b is a Boolean expression over the attributes of the subject agent s, the attributes of the object agent o and those of the environmental context represented by the remaining agents in the system. Such a policy rule means that the agent s is granted the access right r upon the agent o if the Boolean expression b holds. There are two security requirements for the execution of an action a. At any time during the execution of the action,

i. Each subject agent must possess all the rights in the set R_a upon each object agent, i.e. the following predicate h_a^1 must hold, where '$\widehat{=}$' means 'defined by':

$$h_a^1 \mathrel{\widehat{=}} \begin{cases} true & \text{if } R_a = \emptyset \\ \bigwedge\limits_{s \in U_a} \bigwedge\limits_{o \in O_a} \bigwedge\limits_{r \in R_a} \textbf{allow}(s, o, r) & \text{otherwise.} \end{cases} \tag{5}$$

ii. Each agent in the set $O_a \cup U_a$ must possess `read` access right upon each context agent, i.e. the following predicate h_a^2 must hold:

$$h_a^2 \mathrel{\widehat{=}} \begin{cases} true & \text{if } C_a = \emptyset \\ \bigwedge\limits_{s \in (O_a \cup U_a)} \bigwedge\limits_{o \in C_a} \textbf{allow}(s, o, \texttt{read}) & \text{otherwise.} \end{cases} \tag{6}$$

Definition 3.11 (*Security Guard*)**.** If $a \in \mathcal{A}$ is an action defined as in Eq. (2), we call the *security guard* of that action the predicate h_a defined by:

$$h_a \mathrel{\widehat{=}} h_a^1 \wedge h_a^2.$$

Definition 3.12 (*Ready Agent*)**.** We say that an agent is *ready* for the execution of an action if it is not a synchronisation agent for that action or it is not currently involved in the execution of another action for which that agent is a synchronisation agent.

Definition 3.13 (*Enabled Action*)**.** We say that an action is *enabled* if both its functional guard and its security guard evaluate to *true* and all its synchronisation agents are ready.

Example 3.14. Suppose a transfer of a large amount of money say £10,000 in a bank requires the cooperation of two employees who have each the authority to perform such a transaction. The two employees must agree for the transaction to take place. Indeed, this security measure is usually taken in financial institutions to limit frauds from insiders.

Let acc_1 and acc_2 be two agents modelling the source and destination bank accounts; the agents emp_1 and emp_2 being the two employees involved in the transaction. The state of a bank account comprises a security level sL and balance bal; while each employee has a security clearance sC and a consent variable ok. The amount of money to be transferred is provided by the context agent sum.

agent acc_1 : **var** sL, bal; $sL, bal := 0, 0$.
agent acc_2 : **var** sL, bal; $sL, bal := 0, 0$.
agent sum : **var** val; $val := 10000$.
agent emp_1 : **var** sC, ok; $sC, ok := 0, \texttt{false}$.
agent emp_2 : **var** sC, ok; $sC, ok := 3, \texttt{false}$.
action $transfer$<`write`> **on** acc_1, acc_2 **by** emp_1, emp_2 **context** sum :
$(acc_1.bal >= sum.val \ \wedge \ emp_1.ok \wedge emp_2.ok) \rightarrow$
$emp_1.ok, emp_2.ok, acc_1.bal, acc_2.bal \ := \ \texttt{false}, \texttt{false}, acc_1.bal - sum.val, acc_2.bal + sum.val$.
allow$(s, o, \texttt{write}) = (s.sC >= o.sL)$.
allow$(s, sum, \texttt{read}) = \texttt{true}$.

In Example 3.14, the security policy says that an employee is authorised to perform a money transfer from or to a bank account if his/her security clearance is not less than the security level of the bank account. Furthermore, a transfer can take place only under the agreement of two authorised employees. The policy is enforced by implementing the action *transfer* as a joint action that synchronises the bank account source, the bank account destination and the two employees. Each of the employees expresses his/her agreement for a transfer by setting the local variable *ok* to true.

3.4. Concurrent execution model of CASAS

The concurrent execution of a CASAS proceeds as follows. The initialisation statements of the agents are executed to create an initial state of the system. We assume that a variable which is not explicitly initialised is assigned an arbitrary value (in its range). Starting from the initial state, the actions are repeatedly executed as long as there are actions that are enabled. The initialisation statements and the bodies of the actions are assumed to be deterministic. We also assume that an initialisation statement always terminates, and the body of an action terminates when the guard is true.

At any time, an agent is involved in the execution of at most one action for which it is a synchronisation agent. When many actions involving a common synchronisation agent are enabled, one of them is chosen non-deterministically yet fairly for execution. We consider two fairness requirements: (i) *action fairness* that states that if an action is enabled infinitely often then that action is also executed infinitely

Table 2
Syntax of LTL.

$\psi ::= q \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \exists x\, \psi \mid \bigcirc \psi \mid \psi_1 \mathbf{U} \psi_2$

Table 3
Semantics of LTL.

σ, i	$\models$	q iff $I_{\sigma_i}(q) = tt$.
σ, i	$\models$	$\neg\psi$ iff $\sigma, i \not\models \psi$.
σ, i	$\models$	$\psi_1 \vee \psi_2$ iff $\sigma, i \models \psi_1$ or $\sigma, i \models \psi_2$.
σ, i	$\models$	$\exists x\, \psi$ iff there exists σ' such that $\sigma \sim_x \sigma'$ and $\sigma', i \models \psi$
σ, i	$\models$	$\bigcirc\psi$ iff $\sigma, i+1 \models \psi$.
σ, i	$\models$	$\psi_1 \mathbf{U} \psi_2$ iff there exists $k \geq i$ such that $\sigma, k \models \psi_2$ and for all j such that $i \leq j < k$; $\sigma, j \models \psi_1$.

often; (ii) *agent fairness* asserts that no agent may wait indefinitely while some of the joint actions it participates in are infinitely often enabled. As long as this restriction is obeyed, any number of actions may be executed in parallel. Access control is continuously enforced at the beginning and during the execution of actions; and those actions for which the authorisation conditions cease to hold terminate immediately.

As customarily in action systems, joint action is the only mechanism to model synchronisation and communication among agents. No explicit synchronisation or communication primitives are needed in the language. Synchronisation is achieved through the functional and security guards (that must evaluate to true for the action to be executed) and the readiness of the participating agents. Communication may take place in the body of the action by updating an agent's local state using information from other agents. The formal semantics of CASAS is presented in the following section.

4. Formal semantics of CASAS

In this section we give the formal semantics of a CASAS in Linear Temporal Logic (LTL) [33,28]. First an overview of LTL is presented and then the behaviour of a CASAS is specified as an LTL formula. This provides a basis for reasoning about properties of a CASAS.

4.1. Overview of LTL

The syntax of LTL is given in Table 2, where ψ stands for a formula, q a predicate and x a variable symbol.

Let $\mathcal{V}$ be the set of all variable symbols and $\mathcal{D}$ the range of these variables. A model of an LTL formula is an infinite sequence $\sigma = \sigma_0\sigma_1\cdots$ of states, where a state σ_i is an assignment of values to variables, $i \in \mathbb{N}$; where $\mathbb{N}$ is the set of non-negative integers. Let $\Sigma = \mathcal{V} \to \mathcal{D}$ be the set of all possible states, and Σ^ω be the set of all infinite sequences of states. We write $\sigma, i \models \psi$ to mean that the formula ψ is true in the state σ_i of the model σ. This notion is used to define the semantics of LTL formulae in Table 3, where $I_s(q) \in \{tt, ff\}$ is the interpretation of the predicate q in the state s, and tt and ff represent the truth values 'true' and 'false', respectively. The notation $\sigma \sim_x \sigma'$ means that the two infinite sequence of states σ and σ' are identical with the possible exception of their mappings for the variable x.

Definition 4.1 (*Satisfiability*)**.** A formula ψ is *satisfiable* if there exists a model $\sigma \in \Sigma^\omega$ and $i \in \mathbb{N}$ such that $\sigma, i \models \psi$.

Definition 4.2 (*Validity*)**.** A formula ψ is *valid* and we denote by $\models \psi$ if it is satisfied by all model.

Additional logical operators and temporal operators can be derived as depicted in Table 4. We also give in Table 5 a graphical representation of the semantics of the temporal operators used in this paper, where '•' denotes a state, '⟶' represents a single transition and '⇢' one or many transitions. Following are examples of LTL formulae:

- ψ_1 holds permanently or until ψ_2 holds: $\psi_1 \mathbf{W} \psi_2$;
- eventually ψ holds permanently: $\diamond\Box\psi$;
- ψ holds infinitely often: $\Box \diamond \psi$;
- the variable x keeps its value to the next state, we also say x is stable and write $stable(x)$:

$$stable(x) \;\widehat{=}\; \exists v\, ((x = v) \wedge \bigcirc(x = v)). \tag{7}$$

4.2. Formal semantics of CASAS in LTL

In order to reason about properties of a CASAS, we present in this section its formal semantics in LTL. Let $(\mathcal{P}, \mathcal{A}, \mathcal{R}, Agents, Actions, Policy)$ be a CASAS. First we define the function $\mathcal{M}_V[\![\alpha]\!]$, where α denotes a basic statement as in the grammar in Table 1 and V is a set of variables (including all the variables modified by α), as follows:

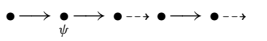
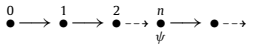
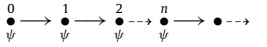
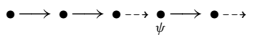
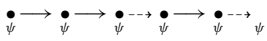
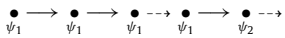
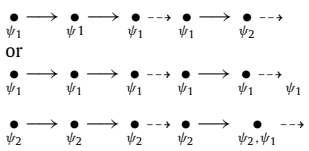
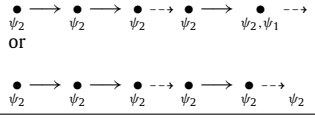
- $\mathcal{M}_V[\![x := exp]\!] \;\widehat{=}\; (\exists v\, (v = exp) \wedge \bigcirc(x = v)) \;\wedge\; \wedge_{u \in (V \setminus \{x\})}\, stable(u)$.
- $\mathcal{M}_V[\![\texttt{skip}]\!] \;\widehat{=}\; \wedge_{u \in V}\, stable(u)$.

This function maps an assignment statement $x := exp$ to a formula that sets the value of the variable x in the next state to the value of the expression exp in the current state, and keeps any other variable in V unchanged. This formula captures the traditional semantics of the

Table 4
Derived operators.

$true$	$\equiv$	$q \vee \neg q$	Satisfied by all model
$false$	$\equiv$	$\neg true$	Satisfied by no model
$\psi_1 \Rightarrow \psi_2$	$\equiv$	$\neg\psi_1 \vee \psi_2$	Logical implication
$\diamond\psi$	$\equiv$	$true\ \mathbf{U}\ \psi$	Sometimes
$\Box\psi$	$\equiv$	$\neg \diamond \neg\psi$	Always
$\psi_1\ \mathbf{R}\ \psi_2$	$\equiv$	$\neg(\neg\psi_1\ \mathbf{U}\ \neg\psi_2)$	Release, dual of until
$\psi_1\ \mathbf{W}\ \psi_2$	$\equiv$	$(\psi_1\ \mathbf{U}\ \psi_2) \vee \Box\psi_1$	Weak until
$\bigcirc^0 \psi$	$\equiv$	ψ	Initially
$\bigcirc^{n+1} \psi$	$\equiv$	$\bigcirc^n \bigcirc\psi$	At time $n+1$
$\bigcirc^{\leq n} \psi$	$\equiv$	$\wedge_{i=0}^{n} \bigcirc^i \psi$	For a prefix of length n
$\bigcirc^{\leq} \psi$	$\equiv$	$\vee_{n\geq 0} \bigcirc^{\leq n} \psi$	For some prefix

Table 5
Graphical illustration of temporal operators.

Formula	Graphical semantics
$\bigcirc\psi$	
$\bigcirc^n \psi$	
$\bigcirc^{\leq n} \psi$	
$\diamond\psi$	
$\Box\psi$	
$\psi_1\ \mathbf{U}\ \psi_2$	
$\psi_1\ \mathbf{W}\ \psi_2$	
$\psi_1\ \mathbf{R}\ \psi_2$	

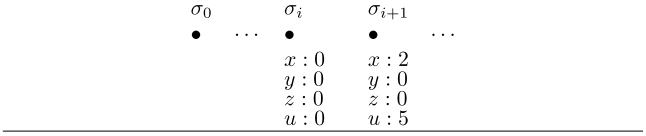


Fig. 1. Illustration of $\mathcal{M}_{\{x,y,z\}}[\![x := 2]\!]$.

assignment statement in programming languages. Likewise, $\mathcal{M}_V[\![\texttt{skip}]\!]$ says that all the variables in the set V keep their values unchanged in the next state, with respect to Eq. (7). This is to mean that the statement `skip` does no computation apart from letting the system transition to the next state. However, in CASAS the semantics of these basic statements is constrained to enable continuous enforcement of access control; i.e. a security guard h must hold in the current state before any statement is executed, otherwise a termination statement T is performed instead to end the executing action. The termination statement T is an assignment statement that updates some system variables as we will see in Eq. (13).

Note that the formula $\mathcal{M}_V[\![\alpha]\!]$ controls only the behaviours of the variables in the set V. Therefore, any variable not in V can change independently during the execution of the statement α. For example, the formula $\mathcal{M}_V[\![x := exp]\!]$ for an assignment statement, at the current state σ_i, denotes the set of all infinite sequence of states $\sigma \widehat{=} \sigma_0\sigma_1 \cdots \sigma_i\sigma_{i+1} \cdots$ for which the value of x in the next state σ_{i+1} is equal to the value of the expression exp in the current state σ_i and the other variables in V remain unchanged. Fig. 1 illustrates the behaviour of $x := 2$ relatively to the set $\{x, y, z\}$ which is a subset of the system state $\{x, y, z, u\}$. The value of x is equal to 2 in the next state and y and z are kept unchanged. However, the variable u is controlled by the environment and therefore can take any value in the next state.

We can now give the semantics of a statement S (described by the grammar in Table 1) relatively to a security guard h, a termination statement T and a set V of variables (including all the variables modified by S or T), as an LTL formula $\mathcal{M}_V^{h,T}[\![S]\!]$ defined inductively on the structure of statements as follows:

$$\mathcal{M}_V^{h,T}[\![x := exp]\!] \widehat{=} (h \wedge \mathcal{M}_V[\![x := exp]\!]) \vee (\neg h \wedge \mathcal{M}_V[\![T]\!]). \tag{8}$$

$$\mathcal{M}_V^{h,T}[\![\texttt{skip}]\!] \widehat{=} (h \wedge \mathcal{M}_V[\![\texttt{skip}]\!]) \vee (\neg h \wedge \mathcal{M}_V[\![T]\!]). \tag{9}$$

$$\mathcal{M}_V^{h,T}[\![\alpha; S]\!] \widehat{=} (h \wedge \mathcal{M}_V[\![\alpha]\!] \wedge \bigcirc\mathcal{M}_V^{h,T}[\![S]\!]) \vee (\neg h \wedge \mathcal{M}_V[\![T]\!]). \tag{10}$$

$$\mathcal{M}_V^{h,T}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}]\!] \widehat{=} (b \wedge \mathcal{M}_V^{h,T}[\![S_1]\!]) \vee (\neg b \wedge \mathcal{M}_V^{h,T}[\![S_2]\!]). \tag{11}$$

It follows that an assignment statement (Eq. (8)) or the statement `skip` (Eq. (9)) is executed only if the security guard h holds in the current state; otherwise the termination statement T is executed. A sequential composition $\alpha; S$ is executed as follows: if the security guard holds in the current state then the basic statement α is executed followed by the execution of S in the next state with respect to the security guard h and the termination statement T; else (i.e. the security guard does not hold in the current state) the termination statement is performed and the execution ends (Eq. (10)). As for the conditional statement (Eq. (11)), if the condition b holds in the current state then S_1 is executed else S_2 is executed, with respect to the security guard h and the termination statement T.

We are now ready to define the semantics of an action. Let the set Var of variables (see Section 3.1) be extended to include (i) a Boolean variable $ready_p$, for each agent $p \in \mathcal{P}$, indicating whether the agent is ready or currently involved in the execution of some action; (ii) a Boolean variable $exec_a$, for each action $a \in \mathcal{A}$, indicating whether the action a is being executed or not. Remember that the body of an action $a \in \mathcal{A}$ defined as in Eq. (2), cannot contain statements that change the state of an agent in the set C_a. Only variables in the states of the synchronisation agents of an action can be modified by the execution of that action. Let $V_a \mathrel{\widehat{=}} \{exec_a\} \cup \{ready_p \mid p \in O_a \cup U_a\} \cup \cup_{p \in O_a \cup U_a} Var(p)$ be the set of variables that are allowed to be modified by the execution of the action a.

An action a is enabled if both its functional guard g_a and its security guard h_a hold and all its synchronisation agents are ready (see Definition 3.13). This is formulated in LTL by the formula

$$enabled_a \mathrel{\widehat{=}} g_a \wedge h_a \wedge \bigwedge_{p \in O_a \cup U_a} ready_p.$$

Let $ready^a$ be the list of all the variables $ready_p$, $p \in O_a \cup U_a$ and $false^a$ (resp. $true^a$) a list of constants $false$ (resp. $true$) that matches the list $ready^a$. The execution of each action a begins by performing an initialisation statement Eq. (12) to set all its synchronisation agents busy and inform the environment that the action is being executed; and ends by performing a termination statement Eq. (13) to release its synchronisation agents and signal to the environment that its execution is terminated.

$$\alpha_a^{init} \mathrel{\widehat{=}} ready^a, exec_a := false^a, true \tag{12}$$

$$\alpha_a^{term} \mathrel{\widehat{=}} ready^a, exec_a := true^a, false. \tag{13}$$

The semantics of an action a is then given by the formula

$$\psi_a \mathrel{\widehat{=}} enabled_a \wedge \mathcal{M}_{V_a}^{h_a, \alpha_a^{term}} [\![\alpha_a^{init}; S_a; \alpha_a^{term}]\!], \tag{14}$$

Note that the variables in V_a which are not modified in the body S_a of the action a are kept unchanged during the execution of S_a; for instance $ready^a$ and $exec_a$.

The initial state of the system is described by the state formula

$$initial \mathrel{\widehat{=}} (\bigwedge_{p \in \mathcal{P}} ready_p) \wedge (\bigwedge_{a \in \mathcal{A}} \neg exec_a) \wedge (\bigwedge_{p \in \mathcal{P}'} (y_p = c_p)), \tag{15}$$

where $\mathcal{P}'$ is the set of agents $p \in \mathcal{P}$ such that $Init(p) \mathrel{\widehat{=}} y_p := c_p$. This means that initially all the agents are ready for participating in the execution of actions, no action is being executed as yet, and the initialisation statements of agents have been performed.

The final state of the system is reached when all agents are ready and no guard is true. This is expressed by the state formula

$$final \mathrel{\widehat{=}} (\bigwedge_{p \in \mathcal{P}} ready_p) \wedge (\bigwedge_{a \in \mathcal{A}} \neg(g_a \wedge h_a)). \tag{16}$$

Moreover, during the execution of a system, the following requirements must be met:

Stability. The state of an agent can only change as an effect of the execution of some action in which that agent participates, i.e.

$$stab \mathrel{\widehat{=}} (\bigwedge_{p \in \mathcal{P}} ((\bigwedge_{a \in \mathcal{A}_p} \neg exec_a) \Rightarrow \bigwedge_{x \in Var(p)} stable(x))) \mathbf{W}\, final \tag{17}$$

where $\mathcal{A}_p$ denotes the set of all actions for which p is a synchronisation agent; $Var(p)$ represents the set of the local variables of p; and $stable(x)$ means that the variable x keeps its current value to the next state as defined in Eq. (7).

Progress. Unless the final state is reached, at any time some action must be scheduled for execution; this is modelled by the LTL formula *progress* as follows:

$$progress \mathrel{\widehat{=}} (\bigvee_{a \in \mathcal{A}} (\psi_a \vee exec_a)) \mathbf{W}\, final. \tag{18}$$

Competition. When several actions are simultaneously enabled and compete for overlapping sets of synchronisation agents, only one of them must be chosen for execution; this is expressed by the LTL formula *compete* as follows:

$$compete \mathrel{\widehat{=}} (\bigwedge_{a \in \mathcal{A}} (exec_a \Rightarrow \bigwedge_{b \in B_a \setminus \{a\}} \neg exec_b)) \mathbf{W}\, final \tag{19}$$

where B_a is the set of all actions that shared some synchronisation agent with the action a, including the action a itself; i.e. $B_a \mathrel{\widehat{=}} \{b \in \mathcal{A} \mid (O_a \cup U_a) \cap (O_b \cup U_b) \neq \emptyset\}$.

Concurrency. At any time, all enabled actions are executed in parallel subject to the competition requirement; this is specified by the LTL formula *conc* as follows:

$$conc \mathrel{\widehat{=}} \left(\bigwedge_{a \in \mathcal{A}} \left(enabled_a \Rightarrow \bigcirc \left(\bigvee_{b \in B_a} exec_b\right)\right)\right) \mathbf{W}\, final. \tag{20}$$

Action fairness. Remember that if two or more actions sharing a common synchronisation agent are enabled at the same time, only one of them is chosen (non-deterministically) for execution. In order to ensure that computations are *fair*, we use the notion of *action fairness*

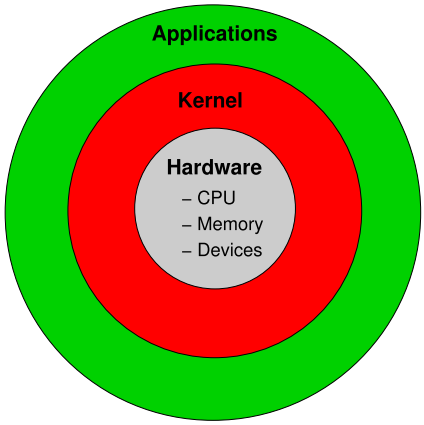


Fig. 2. The kernel connects user applications to the computer hardware.

which states that if an action is enabled infinitely often then that action is also executed infinitely often. The action fairness requirement is specified by the following formula:

$$actFair \;\widehat{=}\; \underset{a\in\mathcal{A}}{\wedge}(\Box\diamond enabled_a \Rightarrow \Box\diamond \psi_a). \tag{21}$$

Agent fairness. It is also important that computations are *agent fair*. The notion of *agent fairness* requires that no agent may be indefinitely held up while some of the joint actions it participates in as a synchronisation agent are infinitely often enabled. This requirement is formalised as

$$agtFair \;\widehat{=}\; \underset{p\in\mathcal{P}}{\wedge}((\Box\diamond \underset{a\in\mathcal{A}_p}{\vee} enabled_a) \Rightarrow \Box\diamond \neg ready_p). \tag{22}$$

The semantics of the concurrent execution of CASAS is then defined by the formula

$$\mathcal{M}[\![\texttt{CASAS}]\!] \;\widehat{=}\; \bigcirc(initial \wedge stab \wedge progress \wedge compete \wedge conc \wedge actFair \wedge agtFair). \tag{23}$$

The formula $\mathcal{M}[\![\texttt{CASAS}]\!]$ states that the execution of an action system starts with an initialisation step, and repeatedly all the enabled actions are executed in parallel. However, at any time an agent is participating in the execution of at most one action. Access control is enforced at the beginning and during the execution of any action; if the access right is revoked during execution (e.g. due to changes in the environment context), that action ends immediately by performing a termination statement to release the participating agents. The execution of the action system terminates if a state is reached where all the agents are ready and all the action guards are false. If the computation is infinite then both the *action fairness* and the *agent fairness* requirements are enforced. In the following section, we demonstrate how the proposed paradigm can be used to model a practical real-world pervasive computing system.

5. Threat model

A pervasive computing system is complex and comprises a variety of subjects and objects interacting continuously through wireless networks; these include users, their mobile devices, sensors, actuators, disk devices, and services. These components collect various data about the users and their context and use them in decision-making to control the system behaviour. For example, sensible user information such are preferences, location information and financial data may be stored on their mobile devices to be readily available to the system for quick decision-making in providing high quality services anytime anywhere. We assume that a complex system like this can be compromised. By exploiting known or hidden vulnerabilities to gain privileged permissions, determined attackers can access users' personal information, e.g. to steal them or to modify them; tamper with sensed data that are later used in decision-making within the system; or disrupt the normal functioning of individual services. We assume, however, that the attackers lack tools and expertise to perform hardware-based attacks, such as sniffing on the system bus or timing analysis.

The security enforcement mechanism embedded in the semantics of CASAS (see Section 4) ensures that access control is done before the access and continuously during the access, taking into account the context of usage. Hence, an attack may start and fail to complete because detected and stopped during access. For example, an attacker may steal the credentials of a legitimate user to launch an attack, but lack knowledge of the context requirements for the usage of services.

6. Example: a Linux-like kernel integrity protection model

In this section we consider the problem of protecting the integrity of the kernel of a Linux-like operating system [39,26]. This example demonstrates how access control to sensitive kernel objects can be modelled in CASAS. As depicted in Fig. 2, protecting the integrity of an OS kernel is important as all user applications rely upon the kernel to function correctly. Therefore any attack to the kernel can compromise the security of the whole system. The critical components of the kernel vulnerable to attack include for instance the kernel text, the system call table, and the interrupt descriptor table. For example, the system call table contains the addresses of the functions that implement each system call; while the interrupt descriptor table is used by the processor to determine the correct response to interrupts and exceptions. These components, if not protected, can be modified by active processes and loadable kernel modules. We model an active process or a loadable kernel module p as an agent with an attribute *text* corresponding to its program text.

agent p : **var** *text*.

Table 6
Attribute grammar.

(a) Attribute grammar for Requirement 1

$S ::= \; y := exp\ [S.single := (|y| = |exp|) \wedge \neg(\exists i, j : i \neq j \wedge y_i \in \{y_j\})]$
$\mid \texttt{skip}\ [S.single := true]$
$\mid S_1; S_2\ [S.single := S1.single \wedge S2.single]$
$\mid \texttt{if}\ b\ \texttt{then}\ S_1$
$\quad \texttt{else}\ S_2$
$\quad \texttt{fi}\ [S.single := S1.single \wedge S2.single]$

(b) Attribute grammar for Requirement 2

$S ::= \; y := exp\ [S.update := \cup_{i=0}^{|y|-1}\{y_i\}]$
$\mid \texttt{skip}\ [S.update := \emptyset]$
$\mid S_1; S_2\ [S.update := S1.update \cup S2.update]$
$\mid \texttt{if}\ b\ \texttt{then}\ S_1$
$\quad \texttt{else}\ S_2$
$\quad \texttt{fi}\ [S.update := S1.update \cup S2.update]$

(c) Attribute grammar for Requirement 3

$S ::= \; y := exp\ [S.dep := cleanUp(\{(y_i, Var(exp_i)) \mid 0 \leq i < |y|\})]$
$\mid \texttt{skip}\ [S.dep := \emptyset]$
$\mid S_1; S_2\ [S.dep := merge(S1.dep, S2.dep)]$
$\mid \texttt{if}\ b\ \texttt{then}\ S_1$
$\quad \texttt{else}\ S_2$
$\quad \texttt{fi}\ [S.dep := add(append(S1.dep, S2.dep), Var(b))]$

We assume a hash function H that can be used to calculate the hash-value of a process text. We also model each kernel space object (e.g. the kernel text, the system call table or the interrupt descriptor table) as an agent m defined by:

agent m : **var** *status*, *type*, *address*, *acl*.

where the attribute *status* denotes the status of the object; the attribute *type* represents the type of the object (e.g. kernel_text or sys_call_table); the attribute *address* indicates the address of the object in the kernel space; and *acl* is the set of the hash-values of the texts of the processes and loadable kernel modules that are allowed to write on m.

An active process or loadable kernel module p may try to modify an object m in the kernel space by performing an action of the following form, for some statements S_1 and S_2:

action *update*<`write`> **on** m **by** p, *valObj* :

$$(m.type = valObj.type) \;\rightarrow\; S_1; m.address[valObj.index] := valObj.val; S_2 \tag{24}$$

where *valObj* is an agent that specifies the type of the kernel space object (e.g. sys_call_table), the index of the entry of that object to be modified and the new value, viz.

agent *valObj* : **var** *type*, *index*, *val*.

For simplicity, we also use the notation '$m.address[valObj.index]$' as a short hand for the memory location '$m.address + valObj.index * \texttt{word_size}$' in Eq. (24). For a process p to perform the action *update* on a kernel space object m, it is required that p has the `write` access right upon m. The access rights are assigned to agents according to the following authorisation rules:

allow$(valObj, m, \texttt{write}) = m.type \notin \{kernel_text, sys_call_table, interrupt_desc_table\}$
allow$(p, m, \texttt{write}) = H(p.text) \in m.acl \wedge m.type \notin \{kernel_text, sys_call_table, interrupt_desc_table\}$

It follows that a process or loadable kernel module p that has been tampered with cannot write in the kernel space. Moreover, no process is allowed to modify the kernel text, the system call table and the interrupt descriptor table. The continuity of access control means that these rules are checked before the action *update* is started and continuously during its execution. Therefore, if the executing process p is compromised (i.e. its program text is tampered with) before or during execution of this action then the execution of the action will be denied or terminated immediately.

7. Checking consistency of actions

This section proposes algorithms for checking the consistency of action definitions with respect to Definition 3.5. These algorithms apply the technique of *attribute grammar* [24] to the grammar of statement given in Table 1. Thus the non-terminal S of that grammar is assigned attributes and each production is annotated with an evaluation rule which determines the values of those attributes when this production is used during parsing.

For Requirement 1 of Definition 3.5, we assign the non-terminal symbol S an attribute *single* which is set to true if no variable occurs more than once in the left hand side of the same assignment statement and false otherwise. The attribute grammar that shows how the value of the attribute *single* is calculated is depicted in Table 6(a), where annotations are enclosed in a pair of square brackets. Recall that in an assignment statement $y := exp$, y is a non-empty list of variables y_i and exp is a non-empty list of expressions exp_i; the value of each expression exp_i is assigned to the corresponding variable symbol y_i; $0 \leq i < |y| = |exp|$. The evaluation rule for the assignment statement sets the attribute *single* to true if the y_i, $0 \leq i < |y|$, are all distinct and false otherwise. For the `skip` statement, the evaluation rule sets the attribute *single* simply to true as there is no variable involved in this statement. For a sequential composition or a conditional statement, the value of the attribute *single* is simply the conjunction of the values of the attribute *single* of the statements in argument.

Algorithm 1: Checking Consistency Requirement 1

Input: S_a: root of the AST for the body of action a,
g_a: the functional guard of the action a,
O_a: set of object agents of the action a,
U_a: set of subject agents of the action a,
C_a: set of context agents of the action a.
Output: whether action a meets Requirement 1.
$X := Var(g_a) \cup Var(S_a)$;
$Y := \cup_{p \in O_a \cup U_a \cup C_a} Var(p)$;
if $X \subseteq Y$ **then return** $S_a.single$ **else return** false

Algorithm 2: Checking Consistency Requirement 2

Input: S_a: root of the AST for the body of action a,
C_a: set of context agents of the action a.
Output: whether action a meets Requirement 2.
$X := \cup_{p \in C_a} Var(p)$;
if $S_a.update \cap X = \emptyset$ **then return** true **else return** false

For illustration, Fig. 3 depicts, using an Abstract Syntax Tree (AST), how the values of attributes are synthesised by a parser. Fig. 3(a) shows a AST for the statement:

$$b, d := 2 * b + c/d, 5; \qquad b, c, e := e + 5, b + c, 3.$$

The AST of Fig. 3(b) corresponds to the statement:

$$\texttt{if } d = 0 \texttt{ then skipelse } b, \quad d := 2 * b + c/d, 5; \qquad b, c, b := e + 5, b + c, 3 \texttt{ fi}.$$

These examples cover all syntactic categories of the attribute grammar in Table 6; yet the value of the attribute *single* is true for the former and *false* for the latter. It follows that an action definition satisfies Requirement 1 if that action refers only to local variables and the value of the attribute *single* is synthesised to true for the body of that action, as specified in Algorithm 1 where for the sake of simplicity, $Var(C)$ denotes the set of all variable symbols that occur in the syntactic construct C.

As for Requirement 2, S is assigned an additional attribute *update* whose value is the set of all variables that are likely to be modified during the execution of S. The respective evaluation rules are given by the attribute grammar in Table 6(b). The intuitive meaning of these rules is that the synthesised value of the attribute *update* is the set of all the variables that occur in the left hand side of an assignment statement. The example of Fig. 3 shows how the value of the attribute *update* is synthesised. Therefore Requirement 2 is simply checked by Algorithm 2.

Requirement 3 requires to compute the dependency graph upon the variables occurring in a program derived from S. To this end, S is assigned an attribute *dep* which is a set of tuple (u, T) such that u is a variable that occurs in S and T is the set of variables the variable u depends upon in S, with respect to Definition 3.3. We call such a set a *dependency graph*. Here are some examples of how the attribute *dep* is calculated:

- If S denotes the program '$b, c := b + 2 * c/d + 3, 5$' then $S.dep = \{(b, \{b, c, d\}), (c, \emptyset)\}$. In this example, the initial values of the variables b, c and d are used to calculate the final value of the variable b; but no variable is used in the calculation of the final value of the variable c and the variable d is not modified.
- Yet if S denotes the program '$\texttt{if } d > 0 \texttt{ then } b, c := b + 2 * c/d + 3, 5 \texttt{ else skip fi}$' then $S.dep = \{(b, \{b, c, d\}), (c, \{d\})\}$ because the value of the variable d is used through the condition $d > 0$ to calculate the final values of the variables b and c.
- Yet again, if S denotes the program '$b, c := b + 2 * c/d + 3, 5;\ b, c := 3, b + 1$' then $S.dep = \{(b, \emptyset), (c, \{b, c, d\})\}$ because the final value of the variable b is a constant, while that of the variable c depends on the initial values of the variables b, c and d.

The evaluation rules for synthesising the value of the attribute *dep* are formally defined in Table 6(c). If V is a dependency graph, we let $V^1 \widehat{=} \{u \mid \exists T : (u, T) \in V\}$ and $V(u) \widehat{=} T$ such that $(u, T) \in V$. The function *merge* combines two dependency graphs into one as follows:

$$merge(V, W) \ \widehat{=}\ \{(x, T) \in V \mid x \in V^1 \setminus W^1\} \cup \left\{(x, (T \setminus V^1) \cup T') \mid (x, T) \in W \ \wedge T' = \bigcup_{z \in (T \cap V^1)} V(z)\right\}.$$

Besides, the functions *cleanUp*, *append* and *add* are defined as follows, where V and W are dependency graphs and T a set of variable symbols:

- $cleanUp(V) \ \widehat{=}\ \{(x, T) \mid x \in V^1 \wedge T = \cup_{(x,T') \in V} T'\}$
- $append(V, W) \ \widehat{=}\ \{(x, T) \in V \cup W \mid x \notin V^1 \cap W^1\} \cup \{(x, T \cup T') \mid (x, T) \in V \wedge (x, T') \in W\}$
- $add(V, T) \ \widehat{=}\ \{(x, T' \cup T) \mid (x, T') \in V\}$.

The example in Fig. 3 shows how the dependency graph *dep* is calculated for a statement derived from the axiom S. Algorithm 3 checks whether an action definition meets Requirement 3.

The dependency graph is also used to check Requirement 4 as depicted in Algorithm 4. If a local variable $p.x$ of an agent p depends on a local variable $q.y$ of another agent q in the body of an action then information may flow from agent q to agent p during the execution of that action (see Definition 3.4). Such a flow of information must be permitted by the access rights of that action as specified in Algorithm 4.

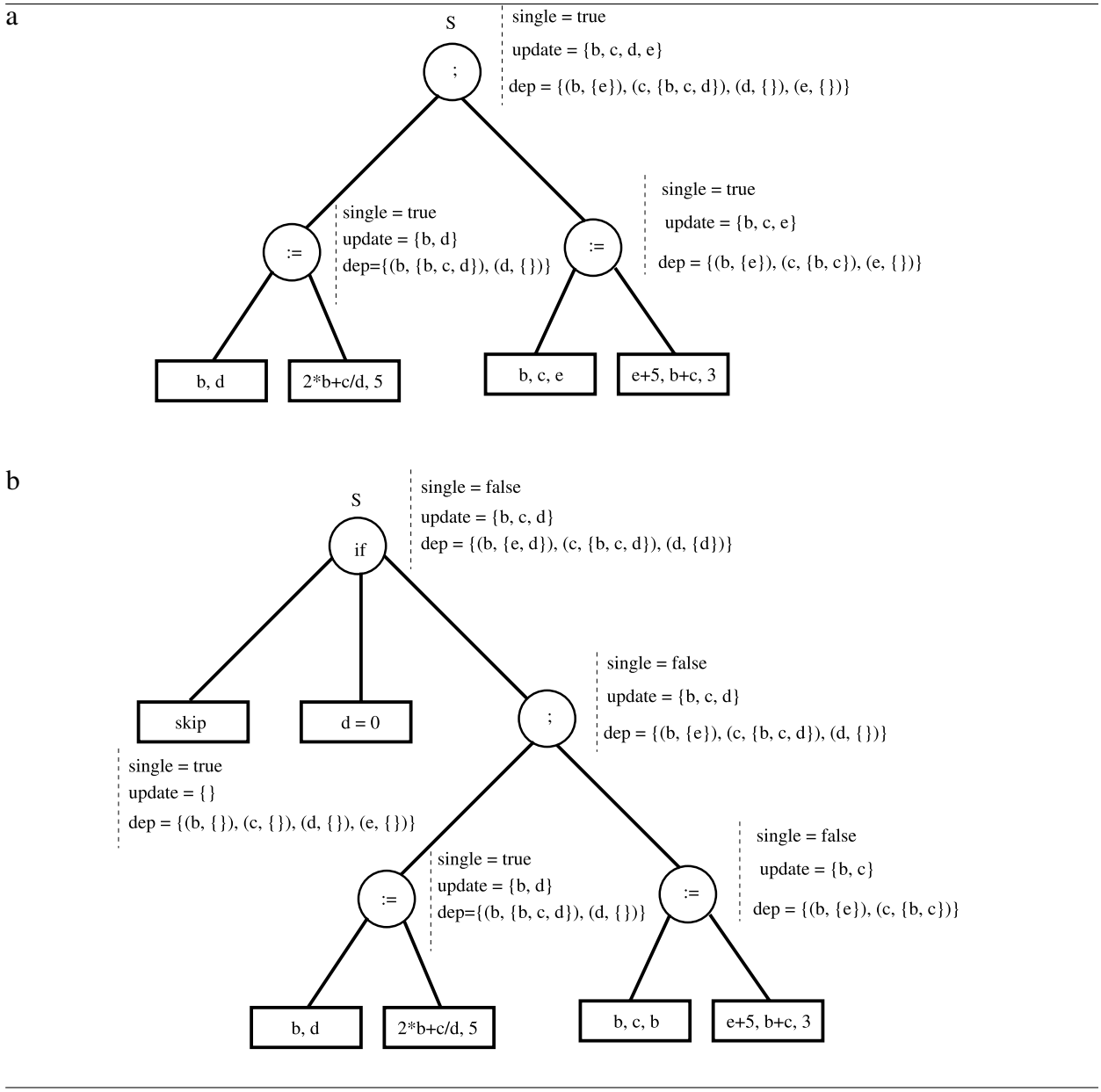


Fig. 3. Examples of AST of attribute grammar in Table 6.

Algorithm 3: Checking Consistency Requirement 3

Input: S_a: root of the AST for the body of action a,
O_a: set of the object agents of the action a,
U_a: set of the subject agents of the action a,
R_a: set of the access rights of the action a.
Output: whether action a meets Requirement 3.
for ***each*** p ***in*** $O_a \setminus U_a$ **do**
  for ***each*** $q \neq p$ ***in*** $O_a \setminus U_a$ **do**
    for ***each*** x ***in*** $Var(p)$ **do**
      if $S_a.dep(x) \cap Var(q) \neq \emptyset$ **then return** false
for ***each*** p ***in*** $U_a \setminus O_a$ **do**
  for ***each*** $q \neq p$ ***in*** $U_a \setminus O_a$ **do**
    for ***each*** x ***in*** $Var(p)$ **do**
      if $S_a.dep(x) \cap Var(q) \neq \emptyset$ **then return** false
return true

Example 7.1. We now demonstrate how these algorithms work using an example. Let p_1, p_2 and p_3 be three agents and a_1, a_2 and a_3 three actions defined as follows.

Algorithm 4: Checking Consistency Requirement 4

Input: S_a: root of the AST for the body of action a,
O_a: set of the object agents of the action a,
U_a: set of the subject agents of the action a,
R_a: set of the access rights of the action a.
Output: whether action a meets Requirement 4.
$ans := \emptyset$;
for ***each*** p ***in*** U_a **do**
  for ***each*** $q \neq p$ ***in*** O_a **do**
    for ***each*** x ***in*** $Var(p)$ **do**
      if $S_a.dep(x) \cap Var(q) \neq \emptyset$ **then** $ans := ans \cup \{\texttt{read}\}$
for ***each*** p ***in*** O_a **do**
  for ***each*** $q \neq p$ ***in*** U_a **do**
    for ***each*** x ***in*** $Var(p)$ **do**
      if $S_a.dep(x) \cap Var(q) \neq \emptyset$ **then** $ans := ans \cup \{\texttt{write}\}$
if $ans \subseteq R_a$ **then return** true **else return** false

Table 7
Checking consistency of actions.

Algorithm	1	2	3	4	Consistent?
a_1	True	True	True	True	Yes
a_2	True	False	True	False	No
a_3	False	True	False	True	No

agent p_1 : **var** b, c; $b, c := 0, 0$.
agent p_2 : **var** e, f; $e, f := 0, 0$.
agent p_3 : **var** d; $d := 5$.
action $a_1 <$ `write` $>$ **on** p_1 **by** p_3 **context** p_2:
$(f >= b + e) \rightarrow b, d := 2 * b + c/d, 5;\ b, c, e := e + 5, b + c, 3$.
action $a_2 <$ `read` $>$ **on** p_1, **by** p_2 **context** p_3:
$(d > 0 \wedge f < b) \rightarrow b, d := 2 * b + c/d, 5;\ b, c, e := e + 5, b + c, 3$.
action $a_3 <$ `read, write` $>$ **on** p_1, p_2 **by** p_3:
$true \rightarrow$ `if` $d = 0$ `then skip else` $b, d := 2*b+c/d, 5;\ b, c, b := e+5, b+c,$ `3 fi`.

Note that the bodies of the actions a_1 and a_2 are represented by the AST in Fig. 3-a while that of action a_3 corresponds to the AST in Fig. 3-b. The outputs of the above algorithms applied to each of these actions are recapitulated in Table 7. It follows that the action a_1 is consistent whereas a_2 and a_3 are not. The action a_2 fails to meet Requirement 2 and Requirement 4 because the variable d of the context agent p_3 is modified in the body of that action (i.e. $d \in S.update$ in Fig. 3-a), and information can flow from p_2 and p_1 (because $e \in S.dep(b)$) which is not consistent with the access rights of the action a_2. As for the action a_3, Requirement 1 is not met because $S.single$ is false (see Fig. 3-b); neither Requirement 3 is met due to information flow between two distinct object agents p_1 and p_2 (e.g. $e \in S.dep(b)$).

8. Example: a ubiquitous learning system

We consider a ubiquitous learning (aka *u-learning*) system that enables mobile users to access course materials using their smart phones anytime and anywhere. The system provides three main u-learning services: the lecture service, the tutorial service and the test service. These services can deliver contents in three different formats: `text`, `audio` and `video`; accessible depending on the condition of use. Access control requirements can be defined to ensure the safety and the security of the users. We consider the following requirements of the u-learning system and show how they can be specified in `CASAS`.

Req-1: Services are available in text format only when the network connection is weak.
Req-2: When driving, the user can only access services in the audio format.
Req-3: To start a test, the user's smart device must have at least 5 megabytes of free memory and 20% available battery power.

The user can specify own access requirements to meet personal circumstances and protect own security and privacy, for example:

Req-4: A u-learning service must not have access to the user's contacts on her mobile device.
Req-5: When the user is in a meeting, services are available only in the text format.
Req-6: When taking a test, all incoming calls must be diverted to a voice messaging system.
Req-7: A u-learning service can only access the user smart phone's bluetooth interface when the user is in a private place such as meeting room, home or office.

We will extend the smart phone system specified in Section 2 to cater for the requirements of the *u*-learning system. Hence we consider four additional attributes for the agent *phone*: the attribute *bluetooth* ranging over {`on`, `off`} that models a bluetooth interface; the attribute *freeMem* that stores the amount of free memory on the phone; the attribute *battery* that tells how much battery power is left on the phone; and the attribute *network* that indicates whether the network bandwidth is `low`, `medium` or `high`. Thus the agent *phone* is defined as follows:

agent *phone* : **var** *ring*, *vib*, *div*, *bluetooth*, *freeMem*, *battery*, *network*; *ring*, *vib*, *div*, *bluetooth* := `on`, `on`, `off`, `off`.

Table 8
Specification of actions.

Lecture service
action *startLectureText* < `write` > **on** *lectureText* **by** *user* **context** *phone* : (*action* = `start` ∧ *type* = `lecture` ∧ *format* = `text` ∧ *status* = `off`) → *status* := `on`.
action *endLectureText* < `write` > **on** *lectureText* **by** *user* **context** *phone* : (*action* = `end` ∧ *type* = `lecture` ∧ *format* = `text` ∧ *status* = `on`) → *status* := `off`.
action *stopLectureText* < `write` > **on** *lectureText* **by** *phone* **context** *user* : (*status* = `on`) → *status* := `off`.
Tutorial service
action *startTutorialText* < `write` > **on** *tutorialText* **by** *user* **context** *phone* : (*action* = `start` ∧ *type* = `tutorial` ∧ *format* = `text` ∧ *status* = `off`) → *status* := `on`.
action *endTutorialText* < `write` > **on** *tutorialText* **by** *user* **context** *phone* : (*action* = `end` ∧ *type* = `tutorial` ∧ *format* = `text` ∧ *status* = `on`) → *status* := `off`.
action *stopTutorialText* < `write` > **on** *tutorialText* **by** *phone* **context** *user* : (*status* = `on`) → *status* := `off`.
Test service
action *startTestText* < `write` > **on** *testText* **by** *user* **context** *phone* : (*action* = `start` ∧ *type* = `test` ∧ *format* = `text` ∧ *status* = `off` ∧ *freeMem* ≥ 5 ∧ *battery* ≥ 20) → *status* := `on`.
action *endTestText* < `write` > **on** *testText* **by** *user* **context** *phone* : (*action* = `end` ∧ *type* = `test` ∧ *format* = `text` ∧ *status* = `on`) → *status* := `off`.
action *stopTestText* < `write` > **on** *testText* **by** *phone* **context** *user* : (*status* = `on`) → *status* := `off`.
Bluetooth access
action *bluetoothOn* < `write` > **on** *phone* **by** *service* **context** *actSensor* : (*phone.bluetooth* = `off`) → *phone.bluetooth* := `on`.
action *bluetoothOff* < `write` > **on** *phone* **by** *service* **context** *actSensor* : (*phone.bluetooth* = `on`) → *phone.bluetooth* := `off`.

The contacts file on the smart phone can be modelled as an agent:

agent *contacts*.

A user of the phone can also be modelled as an agent comprising three attributes that represent the user request: *action* ranging over {`start`, `end`} representing the nature of the request; *type* which is the type of service requested; and *format*, the format of the requested service. Thus a triple (*action* = `start`, *type* = `lecture`, *format* = `text`) is a request to start a lecture service in the text format, while (*action* = `end`, *type* = `lecture`, *format* = `text`) is a request to end such a service. The agent *user* is defined as follows:

agent *user* : **var** *action*, *type*, *format*.

A u-learning service is modelled as an agent of the form *xxxYYY*, where *xxx* is the service type and *YYY* the service format. For example, the agent *lectureText* represents a lecture service in the text format. Each service agent has one attribute *status* which is set to on when the service is being used and to `off` otherwise.

agent *lectureText* : **var** *status*; *status* = `off`.

A user's request to access a *u*-learning service is handled by an action. We distinguish three types of actions: *startxxxYYY* (aka a *start* action) to start a service *xxxYYY* to honour a user request; *endxxxYYY* (aka an *end* action) to end the usage of a service *xxxYYY* when the user requests to do so; and *stopxxxYYY* (aka a *stop* action) to stop the usage of a service due to security violation. Table 8 depicts the specification of these actions for the text format. The actions for the audio and video formats are defined in a similar way. Note that while a start or end action is initiated by the user (see the **by** clauses in the definitions of these actions), a stop action is initiated by the smart phone to stop the usage of a service when the access control condition is not satisfiable. The `write` access right is required for the execution of these action because they modify the state of the object agent. This access right is granted according to the security policy depicted in Table 9, where the parameter *service* represents any service of the u-learning system and the parameter *s* is a place holder for any user or service. The system requirements each policy rule represents are indicated in the column on the right. Note that the requirement **Req-3** is enforced through the functional guard of the action *startTestText* (and the audio and video versions respectively).

For a stop action (i.e. an action of the form *stopxxxYYY*) to be performed on a service, the subject agent *phone* must have the `write` access right upon that service. The policy rule for this action is defined in Table 9 as '**allow**(*phone*, *service*, `write`) = ¬**allow**(*user*, *service*, `write`)', meaning that the agent *phone* has the right to perform such action on a service once the user access right to that service is revoked, e.g. due to changes in the environment context. The execution of this action ends the service usage by setting the attribute *status* to `off`. Hence, access control is enforced continuously at the beginning and during service usage. The last two policy rules in Table 9 are provided to allow information to flow from the context agents *phone* and *actSensor* to the user and service agents during the execution of the actions specified in Table 8, w.r.t. Eq. (6).

9. Composition of CASAS

In this section, we define operators for composing CASASs. This is useful for building complex systems from simpler ones in a compositional manner. In the sequel we consider two systems:

$$\texttt{CASAS}_i \mathrel{\widehat{=}} (\mathcal{P}_i, \mathcal{A}_i, \mathcal{R}_i, \mathit{Agents}_i, \mathit{Actions}_i, \mathit{Policy}_i), \quad i = 1, 2.$$

We assume that any action $a \in \mathcal{A}_i \cap \mathcal{A}_j$, $i \neq j$, has the same definition in $\mathit{Actions}_i$ and $\mathit{Actions}_j$. Similarly, any agent $p \in \mathcal{P}_i \cap \mathcal{P}_j$, $i \neq j$, has the same definition in Agents_i and Agents_j. Let $\mathit{final}_i \mathrel{\widehat{=}} (\wedge_{p \in \mathcal{P}_i} \mathit{ready}_p) \wedge \wedge_{a \in \mathcal{A}_i} \neg(g_a \wedge h_a)$ be a formula that holds in a state if all the agents

Table 9
Security policy.

Lecture service			
allow(*user*, *lectureText*, `write`)	=	*actSensor.actName* ≠ `driving`	**Req-2**
allow(*user*, *lectureAudio*, `write`)	=	*actSensor.actName* ≠ `meeting` ∧ *network* ≠ `low`	**Req-1** & **5**
allow(*user*, *lectureVideo*, `write`)	=	*actSensor.actName* ≠ `driving` ∧ *network* ≠ `low` ∧ *actSensor.actName* ≠ `meeting`	**Req-1**, **2** & **5**
Tutorial service			
allow(*user*, *tutorialText*, `write`)	=	*actSensor.actName* ≠ `driving`	**Req-2**
allow(*user*, *tutorialAudio*, `write`)	=	*actSensor.actName* ≠ `meeting` ∧ *network* ≠ `low`	**Req-1** & **5**
allow(*user*, *tutorialVideo*, `write`)	=	*actSensor.actName* ≠ `driving` ∧ *network* ≠ `low` ∧ *actSensor.actName* ≠ `meeting`	**Req-1**, **2** & **5**
Test service			
allow(*user*, *testText*, `write`)	=	*actSensor.actName* ≠ `driving` ∧ *phone.div* = `on`	**Req-2** & **6**
allow(*user*, *testAudio*, `write`)	=	*actSensor.actName* ≠ `meeting` ∧ *network* ≠ `low` ∧ *phone.div* = `on`	**Req-1**, **5** & **6**
allow(*user*, *testVideo*, `write`)	=	*actSensor.actName* ≠ `driving` ∧ *network* ≠ `low` ∧ *actSensor.actName* ≠ `meeting` ∧ *phone.div* = `on`	**Req-1**, **2**, **5** & **6**
Other rules			
allow(*phone*, *service*, `write`)	=	¬**allow**(*user*, *service*, `write`)	**Req-1**, **2**, **5** & **6**
allow(*service*, *phone*, `write`)	=	*actSensor.actName* = `meeting`	**Req-7**
allow(*service*, *contact*, `write/read`)	=	`false`	**Req-4**
allow(*s*, *phone*, `read`)	=	`true`	Eq. (6)
allow(*s*, *actSensor*, `read`)	=	`true`	Eq. (6)

are ready and no action is enabled in that state in system $\texttt{CASAS}_i$, for $i = 1, 2$. Recall that g_a and h_a stand for the functional guard and the security guard of action a, respectively. Similarly, let $Init_i(p)$ denote the initialisation statements of an agent p in the system $\texttt{CASAS}_i$, for $i = 1, 2$.

9.1. Sequential composition

A sequential composition $\texttt{CASAS}_1; \texttt{CASAS}_2$ denotes a system that behaves like $\texttt{CASAS}_1$ and if $\texttt{CASAS}_1$ terminates it then behaves like $\texttt{CASAS}_2$. Hence this system must be able to detect when $\texttt{CASAS}_1$ reaches its final state and to start dynamically $\texttt{CASAS}_2$ from that state. Let $watchdog \notin \mathcal{P}_1 \cup \mathcal{P}_2$ be a new agent that sets a local variable *flag* to true when it detects the final state of $\texttt{CASAS}_1$. Initially *flag* is set to false. The agent *watchdog* participates in a new action $switch \notin \mathcal{A}_1 \cup \mathcal{A}_2$ which sets the variable *flag* to true and executes the initialisation statement of $\texttt{CASAS}_2$ when $\texttt{CASAS}_1$ reaches its final state. To do this, the agent *watchdog* senses the execution state of the system $\texttt{CASAS}_1$ through its agents in $\mathcal{P}_1$. Both are defined as follows:

$$\textbf{agent}\ watchdog : \textbf{var}\ flag;\ flag := false. \tag{25}$$

$$\textbf{action}\ switch\ \textbf{on}\ \mathcal{P}_2\ \textbf{by}\ watchdog\ \textbf{context}\ \mathcal{P}_1 \setminus \mathcal{P}_2 :$$
$$(final_1 \wedge \neg watchdog.flag) \rightarrow watchdog.flag, p_1.y_{p_1}, \ldots, p_n.y_{p_n} := true, c_{p_1}, \ldots, c_{p_n} \tag{26}$$

where $p_i \in \mathcal{P}_2$ and there exists an initialisation statement $Init_2(p_i) = (y_{p_i} := c_{p_i})$, for $i = 1, \ldots, n$. The notation $p_i.y_{p_i}$ represents the local state of the agent p_i, prefixed with the agent's name to distinguish between local variables of different agents, $i = 1, \ldots, n$.

We now define a sequential composition as follows:

$$\texttt{CASAS}_1; \texttt{CASAS}_2 \ \widehat{=}\ (\mathcal{P}, \mathcal{A}, \mathcal{R}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \ \widehat{=}\ \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{watchdog\}$;
- $\mathcal{A} \ \widehat{=}\ \mathcal{A}_1 \cup \mathcal{A}_2 \cup \{switch\}$;
- $\mathcal{R} \ \widehat{=}\ \mathcal{R}_1 \cup \mathcal{R}_2$;
- $Agents \ \widehat{=}\ Agents_1 \cup Agents_2 \cup \{(25)\}$;
- $Actions \ \widehat{=}\ Actions'_1 \cup Actions'_2 \cup (Actions_1 \cap Actions_2) \cup \{(26)\}$, where $Actions'_1$ is identical to $Actions_1 \setminus Actions_2$ with each action definition Eq. (27) in $Actions_1 \setminus Actions_2$ rewritten as Eq. (28) in $Actions'_1$.

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a : g_a \rightarrow S_a. \tag{27}$$

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a \cup \{watchdog\} : (g_a \wedge \neg watchdog.flag) \rightarrow S_a. \tag{28}$$

- $Actions'_2$ is identical to $Actions_2 \setminus Actions_1$ with each action definition Eq. (29) in $Actions_2 \setminus Actions_1$ rewritten as Eq. (30) in $Actions'_2$.

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a : g_a \rightarrow S_a. \tag{29}$$

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\textbf{context}\ C_a \cup \{watchdog\} : (g_a \wedge watchdog.flag) \rightarrow S_a. \tag{30}$$

- *Policy* is obtained by merging the policies $Policy_1$ and $Policy_2$ as follows:
  (i) **allow**(*s*, *watchdog*, `read`) = *true*.
  (ii) **allow**(*s*, *o*, `read`) = *true*, for $s \in \mathcal{P}_2 \cup \{watchdog\}$ and $o \in \mathcal{P}_1 \setminus \mathcal{P}_2$.
  (iii) **allow**$(s, o, r) = (\neg watchdog.flag \wedge \textbf{allow}_1(s, o, r)) \vee (watchdog.flag \wedge \textbf{allow}_2(s, o, r))$.

Note that the actions in $\mathcal{A}_1 \setminus \mathcal{A}_2$ are not enabled if *flag* evaluates to true and those in $\mathcal{A}_2 \setminus \mathcal{A}_1$ are not enabled if *flag* evaluates to false; however, the actions in $\mathcal{A}_1 \cap \mathcal{A}_2$ are not affected. Initially *flag* is set to false and it is kept so until $\texttt{CASAS}_1$ terminates. At the termination of $\texttt{CASAS}_1$ the action *switch* is enabled because *flag* is still false. Since it is the only enabled action in that state, it is selected for execution. Its execution sets *flag* to true, performs all the initialisation statements of $\texttt{CASAS}_2$ and preserves the post-conditions of $\texttt{CASAS}_1$ that are not affected by these initialisation statements (see the formal semantics of CASAS in Section 4, page 9). From there on, only actions in $\texttt{CASAS}_2$ can be enabled because *flag* is true.

The security policies $Policy_1$ and $Policy_2$ are merged in such a way that $Policy_1$ applies during the execution of $\texttt{CASAS}_1$ and $Policy_2$ during the execution of $\texttt{CASAS}_2$ (see the rule (iii)). The security requirement in Eq. (6) for the execution of the action *switch* is given by (ii). Because the agent *watchdog* is added in the context of all actions in $(\mathcal{A}_1 \cup \mathcal{A}_2) \setminus (\mathcal{A}_1 \cap \mathcal{A}_2)$, the synchronisation agents of these actions must have `read` access right upon the agent watchdog; so the rule (i). Theorem 9.1 states that the sequential composition is closed under system consistency (proof in the Appendix).

Theorem 9.1. *If* $\texttt{CASAS}_1$ *and* $\texttt{CASAS}_2$ *are consistent then* $\texttt{CASAS}_1$; $\texttt{CASAS}_2$ *is consistent.*

9.2. Parallel composition

A parallel composition $\texttt{CASAS}_1 \parallel \texttt{CASAS}_2$ denotes the superposition of two systems $\texttt{CASAS}_1$ and $\texttt{CASAS}_2$ and is defined as follows:

$$\texttt{CASAS}_1 \parallel \texttt{CASAS}_2 \mathrel{\widehat{=}} (\mathcal{P}, \mathcal{A}, \mathcal{R}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \mathrel{\widehat{=}} \mathcal{P}_1 \cup \mathcal{P}_2$;
- $\mathcal{A} \mathrel{\widehat{=}} \mathcal{A}_1 \cup \mathcal{A}_2$;
- $\mathcal{R} \mathrel{\widehat{=}} \mathcal{R}_1 \cup \mathcal{R}_2$;
- $Agents \mathrel{\widehat{=}} Agents_1 \cup Agents_2$;
- $Actions \mathrel{\widehat{=}} Actions_1 \cup Actions_2$;
- *Policy* is obtained by merging the policies $Policy_1$ and $Policy_2$ as follows:
  - (i) $\mathbf{allow}(s, o, r) = \mathbf{allow}_1(s, o, r)$, for $s, o \in \mathcal{P}_1 \setminus \mathcal{P}_2$.
  - (ii) $\mathbf{allow}(s, o, r) = \mathbf{allow}_2(s, o, r)$, for $s, o \in \mathcal{P}_2 \setminus \mathcal{P}_1$.
  - (iii) $\mathbf{allow}(s, o, r) = (\mathbf{allow}_1(s, o, r) \wedge \mathbf{allow}_2(s, o, r))$, for $s, o \in \mathcal{P}_1 \cap \mathcal{P}_2$.

In a parallel $\texttt{CASAS}_1 \parallel \texttt{CASAS}_2$, an access right is granted to an agent common to $\texttt{CASAS}_1$ and $\texttt{CASAS}_2$ if that agent is granted that access right in both subsystems (rule (iii)). The agents that are not common to the two subsystems keep the access rights they are granted in their respective subsystems (rules (i) and (ii)). The parallel composition is closed under system consistency (proof in the Appendix).

Theorem 9.2. *If* $\texttt{CASAS}_1$ *and* $\texttt{CASAS}_2$ *are consistent then* $\texttt{CASAS}_1 \parallel \texttt{CASAS}_2$ *is consistent.*

9.3. Conditional

The conditional $b?\texttt{CASAS}_1 : \texttt{CASAS}_2$ behaves like $\texttt{CASAS}_1$ if initially the Boolean expression b holds and like $\texttt{CASAS}_2$ otherwise. Any variable symbol that occurs in the Boolean expression b must be part of the state of $\texttt{CASAS}_2$ or the state of $\texttt{CASAS}_2$. The definition of a conditional uses a new agent $choice \notin \mathcal{P}_1 \cup \mathcal{P}_2$ and a new action $select \notin \mathcal{A}_1 \cup \mathcal{A}_2$ defined in Eq. (31) and Eq. (32), respectively. The Boolean variable *cond* in the agent *choice* stores the value of the Boolean expression b in the initial state. The action *select* is executed first (because initially $lock = 0$) to assign *cond* the value of the expression b. The variables *cond* and *lock* are used to control the execution of the actions in $\texttt{CASAS}_1$ and $\texttt{CASAS}_2$. Only actions in $\texttt{CASAS}_1$ are executed when *cond* is true, otherwise only those in $\texttt{CASAS}_2$ are executed.

$$\textbf{agent } choice : \textbf{ var } cond, lock;\ cond, lock := false, 0. \tag{31}$$

$$\textbf{action } select \textbf{ on } choice \textbf{ by } choice \textbf{ context } \mathcal{P}_1 \cup \mathcal{P}_2 : (choice.lock = 0) \rightarrow choice.cond, choice.lock := b, 1. \tag{32}$$

So a conditional is defined as follows:

$$b?\texttt{CASAS}_1 : \texttt{CASAS}_2 \mathrel{\widehat{=}} (\mathcal{P}, \mathcal{A}, \mathcal{R}, Agents, Actions, Policy)$$

where

- $\mathcal{P} \mathrel{\widehat{=}} \mathcal{P}_1 \cup \mathcal{P}_2 \cup \{choice\}$;
- $\mathcal{A} \mathrel{\widehat{=}} \mathcal{A}_1 \cup \mathcal{A}_2 \cup \{select\}$;
- $\mathcal{R} \mathrel{\widehat{=}} \mathcal{R}_1 \cup \mathcal{R}_2$;
- $Agents \mathrel{\widehat{=}} Agents_1 \cup Agents_2 \cup \{(31)\}$;
- $Actions \mathrel{\widehat{=}} Actions'_1 \cup Actions'_2 \cup ComAct \cup \{(32)\}$, where
  - *ComAct* is identical to $Actions_1 \cap Actions_2$ with each action definition Eq. (33) in $Actions_1 \cap Actions_2$ rewritten as Eq. (34) in *ComAct*.

$$\textbf{action } a{<}R_a{>} \textbf{ on } O_a \textbf{ by } U_a \textbf{ context } C_a : g_a \rightarrow S_a. \tag{33}$$

$$\textbf{action } a{<}R_a{>} \textbf{ on } O_a \textbf{ by } U_a \textbf{ context } C_a \cup \{choice\} : (g_a \wedge choice.lock = 1) \rightarrow S_a. \tag{34}$$

  - $Actions'_1$ is identical to $Actions_1 \setminus Actions_2$ with each action definition Eq. (35) in $Actions_1 \setminus Actions_2$ rewritten as Eq. (36) in $Actions'_1$.

$$\textbf{action } a{<}R_a{>} \textbf{ on } O_a \textbf{ by } U_a \textbf{ context } C_a : g_a \rightarrow S_a. \tag{35}$$

$$\textbf{action } a{<}R_a{>} \textbf{ on } O_a \textbf{ by } U_a \textbf{ context } C_a \cup \{choice\} : (g_a \wedge choice.lock = 1 \wedge choice.cond) \rightarrow S_a. \tag{36}$$

– $Actions'_2$ is identical to $Actions_2 \setminus Actions_1$ with each action definition Eq. (37) in $Actions_2 \setminus Actions_1$ rewritten as Eq. (38) in $Actions'_2$.

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a : g_a \rightarrow S_a. \tag{37}$$

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a \cup \{choice\} : (g_a \wedge choice.lock = 1 \wedge \neg choice.cond) \rightarrow S_a. \tag{38}$$

- *Policy* is obtained by merging the policies $Policy_1$ and $Policy_2$ as follows:
  (i) $\textbf{allow}(choice, o, \texttt{read}) = true$.
  (ii) $\textbf{allow}(s, choice, \texttt{read}) = true$.
  (iii) $\textbf{allow}(s, o, r) = choice.lock = 1 \wedge ((choice.cond \wedge \textbf{allow}_1(s, o, r)) \vee (\neg choice.cond \wedge \textbf{allow}_2(s, o, r)))$.

The two policies $Policy_1$ and $Policy_2$ are merged in such a way that $Policy_1$ is enforced if the subsystem $\texttt{CASAS}_1$ is selected for execution; otherwise $Policy_2$ is enforced (rule (iii)). The security requirement in Eq. (6) for the execution of the action *select* is given by (i). Because the agent *choice* is added in the context of all actions in $\mathcal{A}_1 \cup \mathcal{A}_2$, the synchronisation agents of these actions must have `read` access right upon the agent *choice* (rule (ii)). The conditional is closed under system consistency (proof in the Appendix).

Theorem 9.3. *If* $\texttt{CASAS}_1$ *and* $\texttt{CASAS}_2$ *are consistent then* $b?\texttt{CASAS}_1 : \texttt{CASAS}_2$ *is consistent, for some Boolean expression b upon local variables of* $\texttt{CASAS}_1$ *and* $\texttt{CASAS}_2$.

9.4. Unless

A system $\langle b\rangle\texttt{CASAS}_1$ behaves like the system $\texttt{CASAS}_1$ and terminates if the system $\texttt{CASAS}_1$ terminates or an event b occurs during execution. The event b is a Boolean expression over the local variables of $\texttt{CASAS}_1$. We call this operator *unless*. Its definition uses a new agent *detector* $\notin \mathcal{P}$ that senses the occurrence of the event b through a new action *detect* $\notin \mathcal{A}$ and informs other agents when this event happens as specified below.

$$\textbf{agent}\ detector :\ \textbf{var}\ dummy;\ dummy := false. \tag{39}$$

$$\textbf{action}\ detect\ \textbf{on}\ detector\ \textbf{by}\ detector\ \textbf{context}\ \mathcal{P}_1 : (detector.dummy = false \wedge b) \rightarrow detector.dummy := true. \tag{40}$$

So the system $\langle b\rangle\texttt{CASAS}_1$ is defined as follows:

$$\langle b\rangle\texttt{CASAS}_1 \ \widehat{=}\ (\mathcal{P}_1 \cup \{detector\}, \mathcal{A}_1 \cup \{detect\}, \mathcal{R}_1, Agents_1 \cup \{(39)\}, Actions'_1 \cup \{(40)\}, Policy) \tag{41}$$

where

- $Actions'_1$ is identical to $Actions_1$ with each action definition Eq. (42) in $Actions_1$ rewritten as Eq. (43) in $Actions'_1$.

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a : g_a \rightarrow S_a. \tag{42}$$

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a \cup \{detector\} :\ (g_a \wedge \neg detector.dummy) \rightarrow S_a. \tag{43}$$

  The functional guard of each action is strengthened with the condition $\neg detector.dummy$ so that the action is not enabled after the condition b becomes true. The system will then terminate.
- *Policy* is obtained from $Policy_1$ as follows:
  (i) $\textbf{allow}(detector, o, \texttt{read}) = true$.
  (ii) $\textbf{allow}(s, detector, \texttt{read}) = true$.
  (iii) $\textbf{allow}(s, o, r) = \textbf{allow}_1(s, o, r)$.

The operator *unless* is also closed under system consistency (proof in the Appendix).

Theorem 9.4. *If* `CASAS` *is consistent then* $\langle b\rangle$`CASAS` *is consistent, for some Boolean expression b upon local variables of* `CASAS`.

9.5. Interrupt

An interrupt system $t : \texttt{CASAS}_1$ behaves like the system $\texttt{CASAS}_1$ for the first t time units, where t is a non-negative integer. It terminates when $\texttt{CASAS}_1$ terminates in less than t time units or t time units have elapsed since the execution of $\texttt{CASAS}_1$ started. Its definition uses a new agent *clock* $\notin \mathcal{P}_1 \cup \mathcal{P}_2$ that has a local variable *time* initialised to 0, and participates in the execution of a new action *tick* $\notin \mathcal{A}_1 \cup \mathcal{A}_2$ that increments the value of the variable *time* by 1 at each transition of the system until it reaches the value t.

$$\textbf{agent}\ clock :\ \textbf{var}\ time;\ time := 0. \tag{44}$$

$$\textbf{action}\ tick\ \textbf{on}\ clock\ \textbf{by}\ clock\ \textbf{context}\ \mathcal{P}_1 :\ (\neg final_1 \wedge clock.time < t) \rightarrow clock.time := clock.time + 1. \tag{45}$$

The system $t : \texttt{CASAS}_1$ is defined as follows:

$$t : \texttt{CASAS}_1 \ \widehat{=}\ (\mathcal{P}_1 \cup \{clock\}, \mathcal{A}_1 \cup \{tick\}, \mathcal{R}_1, Agents_1 \cup \{(44)\}, Actions'_1 \cup \{(45)\}, Policy) \tag{46}$$

where

- $Actions'_1$ is identical to $Actions_1$ with each action definition Eq. (47) in $Actions_1$ rewritten as Eq. (48) in $Actions'_1$.

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a : g_a \rightarrow S_a. \tag{47}$$

$$\textbf{action}\ a{<}R_a{>}\ \textbf{on}\ O_a\ \textbf{by}\ U_a\ \textbf{context}\ C_a \cup \{clock\} : (g_a \wedge clock.time < t) \rightarrow S_a. \tag{48}$$

- *Policy* is obtained from $Policy_1$ as follows:
  (i) $\textbf{allow}(clock, o, \texttt{read}) = true$.
  (ii) $\textbf{allow}(s, clock, \texttt{read}) = true$.
  (iii) $\textbf{allow}(s, o, r) = \textbf{allow}_1(s, o, r)$.

Table 10
Algebraic laws for CASAS.

(PAR-1) $\texttt{CASAS}_1 \| \texttt{CASAS}_2 = \texttt{CASAS}_2 \| \texttt{CASAS}_1$
(PAR-2) $\texttt{CASAS}_1 \| (\texttt{CASAS}_2 \| \texttt{CASAS}_3) = (\texttt{CASAS}_1 \| \texttt{CASAS}_2) \| \texttt{CASAS}_3$
(PAR-3) $\texttt{CASAS} \| \texttt{CASAS} = \texttt{CASAS}$
(COND-1) $b?\texttt{CASAS}_1 : \texttt{CASAS}_2 = (\neg b)?\texttt{CASAS}_2 : \texttt{CASAS}_1$
(COND-2) $(\mathit{false})?\texttt{CASAS}_1 : \texttt{CASAS}_2 = \texttt{CASAS}_2$
(UNLESS) $(\mathit{false})\texttt{CASAS} = \texttt{CASAS}$

The access rights granted in the subsystem $\texttt{CASAS}_1$ are kept (rule (iii)) and each agent must have read access right upon the new agent clock to be able to check the time (rule (ii)). Because of the security requirement Eq. (6), the agent *clock* must have `read` access right upon all the agents in $\texttt{CASAS}_1$ for the action *tick* to be enabled (rule (i)). Note that no action (including the new action *tick*) is enabled when the value of *clock.time* equals t and thereafter. Therefore the system will stop. Theorem 9.5 says that the *unless* operator is closed under system consistency (proof in the Appendix).

Theorem 9.5. *If* CASAS *is consistent then* $t : \texttt{CASAS}$ *is consistent, for some non-negative integer* t.

10. Algebraic laws for CASAS

This section investigates the algebraic laws exhibited by the composition of CASASs presented in the previous section. These laws are depicted in Table 10 as equational relationships between systems. The laws PAR-1 to PAR-3 state that the parallel composition is commutative, associative and idempotent, respectively. The law COND-1 asserts that a conditional system behaves the same if one negates its condition and swaps its then and else components. The law COND-2 says that if the condition of a conditional is not realisable (i.e. *false*), then this conditional always behaves like its second component. Finally, the law UNLESS captures the case where the event of a *unless* system never happens.

Theorem 10.1. *The algebraic laws in Table 10 are sound.*

Proof. The proofs of the soundness of these laws are straightforward from the definitions of the respective operators. Indeed, the proofs of the laws PAR-1, PAR-2 and PAR-3 follow from the commutativity, associativity and impotency of the union and the intersection of sets. The law COND-1 follows from the fact that $\neg\neg b \equiv b$ in the actions in Eq. (36) and Eq. (38). The laws COND-2 and UNLESS also follow immediately from Eqs. (36) and (38); and (40) and Eq. (43) respectively.

11. Compositional verification

This section proposes a set of compositional proof rules for verifying properties of CASASs. We assume that system properties are expressed in LTL; these may be functional, temporal, security or context-awareness properties of the system. The soundness of these proof rules is established based on the formal semantics of CASAS given in Section 4. Indeed, a CASAS satisfies a property ψ (expressed in LTL) if its semantics implies $\bigcirc\psi$, where $\bigcirc$ represents the initialisation step. However, a syntactical approach for system verification is more useful in practice as it hides the complicated formal semantics of the system to the system designers. So we define a satisfaction relation **sat** in Definition 11.1.

Definition 11.1. A secure context-aware action system CASAS satisfies a property ψ, denoted by CASAS **sat** ψ, if the semantics of CASAS implies $\bigcirc\psi$, i.e.

$$\texttt{CASAS}\ \textbf{sat}\ \psi\ \textbf{iff}\ \models \mathcal{M}[\![\texttt{CASAS}]\!] \Rightarrow \bigcirc\psi.$$

For example, if CASAS models a banking system, one may check the functional requirement that any transaction on a bank account *acc* leaves its balance (*acc.bal*) not lower than the minimum threshold (*acc.min*), i.e.

$$\texttt{CASAS}\ \textbf{sat}\ \Box(\mathit{acc.bal} \geq \mathit{acc.min}).$$

Similarly, we can prove properties about the security policy enforced by a system, using the predicate **allow**. For example, one can verify whether a trusted agent *john*, say may eventually be granted the right to read the information stored in the file *foo*, i.e.

$$\texttt{CASAS}\ \textbf{sat}\ \Diamond\ \textbf{allow}(\mathit{john}, \mathit{foo}, \mathit{read}).$$

It can also be checked if a *bad* guy *bob*, say is never granted the right to modify the file *foo*, i.e.

$$\texttt{CASAS}\ \textbf{sat}\ \Box\neg\textbf{allow}(\mathit{bob}, \mathit{foo}, \mathit{write}).$$

As for context-awareness properties, one can check if a smart phone *phone* (specified as in Section 2) switches eventually to silent mode when the user is in a meeting, i.e.

$$\texttt{CASAS}\ \textbf{sat}\ \Box(\mathit{actSensor.actName} = \mathit{meeting}\ \Rightarrow\ \Diamond\,(\mathit{phone.ring} = \mathit{off})).$$

To ease the verification of such properties, a list of compositional proof rules is given in Table 11 where the symbols ψ and ϕ stand for a LTL formula, and the symbols ω and b for a state formula (i.e. a LTL formula that contains no temporal operator). Although this list is not by any means complete, it enables compositional reasoning whereby properties of a system can be established from the properties of its immediate components without additional information about the internal behaviour of these components.

Table 11
Compositional verification rules.

(S-1) $\dfrac{\texttt{CASAS}\ \mathbf{sat}\ \psi_1 \quad \models \psi_1 \Rightarrow \psi_2}{\texttt{CASAS}\ \mathbf{sat}\ \psi_2}$	(S-2) $\dfrac{\texttt{CASAS}\ \mathbf{sat}\ \psi_1 \quad \texttt{CASAS}\ \mathbf{sat}\ \psi_2}{\texttt{CASAS}\ \mathbf{sat}\ (\psi_1\ \wedge\ \psi_2)}$
(S-3) $\dfrac{\texttt{CASAS}_1\ \mathbf{sat}\ \bigcirc^{\leq}\omega}{(\texttt{CASAS}_1; \texttt{CASAS}_2)\ \mathbf{sat}\ \bigcirc^{\leq}\omega}$	(S-4) $\dfrac{\texttt{CASAS}\ \mathbf{sat}\ \bigcirc^{\leq n}\omega}{(t : \texttt{CASAS})\ \mathbf{sat}\ \bigcirc^{\leq \mathbf{min}(t,n)}\omega}$
(S-5) $\dfrac{\texttt{CASAS}_1\ \mathbf{sat}\ \psi_1 \quad \mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset}{(\texttt{CASAS}_1 \parallel \texttt{CASAS}_2)\ \mathbf{sat}\ \psi_1}$	(S-6) $\dfrac{\texttt{CASAS}\ \mathbf{sat}\ \bigcirc^{n}\omega \quad n \leq t}{(t : \texttt{CASAS})\ \mathbf{sat}\ \bigcirc^{n}\omega}$
(S-7) $\dfrac{\texttt{CASAS}\ \mathbf{sat}\ \omega\ \mathbf{U}\ \phi}{(\langle b\rangle\texttt{CASAS})\ \mathbf{sat}\ \omega\ \mathbf{U}\ (b \vee \phi)}$	(S-8) $\dfrac{\texttt{CASAS}\ \mathbf{sat}\ \phi\ \mathbf{R}\ \omega}{(\langle b\rangle\texttt{CASAS})\ \mathbf{sat}\ (b \vee \phi)\ \mathbf{R}\ \omega}$
(S-9) $\dfrac{\texttt{CASAS}\ \mathbf{sat}\ \omega\ \mathbf{W}\ \phi}{(\langle b\rangle\texttt{CASAS})\ \mathbf{sat}\ \omega\ \mathbf{W}\ (b \vee \phi)}$	(S-10) $\dfrac{\texttt{CASAS}_1\ \mathbf{sat}\ \psi_1 \quad \texttt{CASAS}_2\ \mathbf{sat}\ \psi_2}{b?\texttt{CASAS}_1 : \texttt{CASAS}_2\ \mathbf{sat}\ \bigcirc\ ((b \wedge \psi_1) \vee (\neg b \wedge \psi_2))}$
(S-11) $\dfrac{\texttt{CASAS}_2\ \mathbf{sat}\ \psi_2}{(\texttt{CASAS}_1; \texttt{CASAS}_2)\ \mathbf{sat}\ \diamond\ \psi_2}\{\texttt{CASAS}_1\ \text{terminates}\}$	(S-12) $\dfrac{\texttt{CASAS}_1\ \mathbf{sat}\ \Box\omega \qquad \texttt{CASAS}_2\ \mathbf{sat}\ \Box\omega}{(\texttt{CASAS}_1; \texttt{CASAS}_2)\ \mathbf{sat}\ \Box\omega}$

The rule S-1 is commonly known as the *consequence rule*. It says that if a system satisfies a property, then it satisfies all the consequences of that property. The rule S-2 says that if a system satisfies a collection of individual properties then that system also satisfies the conjunction of those properties. The rules S-3, S-11 and S-12 can be used to prove properties of a sequential composition. The former states that a sequential composition inherits any prefix property of its left component, while the latter asserts that if the left component terminates then the composition also meets eventually the properties of the right component. A parallel composition of two systems that share no agent satisfies any property of either two systems; this is the essence of the rule S-5. The verification of the parallel composition of two systems sharing a common agent is more complicated and is beyond the scope of this paper. A conditional satisfies the properties of one component system or another depending on whether its condition holds or not (rule S-10); the symbol $\bigcirc$ corresponding to the selection step. Similarly, the rules S-7 to S-9 can be used to prove properties of the *unless* operator. So do the rules S-4 and S-6 for the operators *interrupt*. The soundness of these proof rules is given by Theorem 11.2 (proof in the Appendix).

Theorem 11.2. *The proof rules* S-1 *to* S-11 *are sound.*

12. Related work

Many works have proposed formal models and notations for representing and analysing context-aware and pervasive systems. [35] proposed Context UNITY which extends early work on Mobile UNITY [34] to cater for the representation and formal reasoning of context-aware programs. In this model context is provided through *exposed variables* and existential quantification is used for context discovery. A verification mechanism is provided based on the underlying UNITY [14] proof rules. In addition, access control restrictions can be placed on *exposed variables* to control what external programs can access them. Recently, Gamble et al. [20] extended Context UNITY with additional security features such as encryption and auditing. However Context UNITY, unlike CASAS, does not provide means for composing systems; so does not support compositional reasoning upon large and complex systems.

Graphical notations have been proposed for describing context-aware systems. Milner [25,29,23] introduced Bigraphical Reactive Systems (BRSs) as a unifying framework for designing models of concurrent and mobile systems. These reactive systems are defined as a set of rewriting rules together with an initial bigraph on which the rules operate. A bigraph is a particular kind of graphs which allows for the representation of communication among nodes as well as their spatial configuration (nodes may be nested within each other). In [11], Birkedal et al. attempt the modelling of context-aware systems using bigraphical reactive systems (BRSs). They reach the conclusion that BRSs are not suitable for directly modelling context queries and propose plato-graphical models as an alternative. While in plato-graphical models of contexts are queried through a proxy (also modelled as a BRS), in our model *context agents* provide context information to the actions in which they participate. Although graphical notations are great visual aid for understanding small size systems, they are not quite helpful for analysing large systems.

Process calculi have also been proposed as a formalism for specifying context-aware and pervasive systems. [40] proposed a context-awareness calculus which features a hierarchical structure similar to Mobile Ambients [13], and a generic multi-agent synchronisation mechanism inspired from the join-calculus. This work was extended in [12] to enable ambients to publish context information upwards in the hierarchy of ambients. The Calculus of Context-aware Ambients (CCA) [36] is another extension of the Mobile Ambients where a process can be guarded with a context-expression so that this process executes only if the environment satisfies that context expression. In this way, an ambient can be aware of its context and executes specific processes as a result of changes in its environment. Process calculi are very useful for simulation and type systems can be devised to verify safety and security properties of systems. Compare to CASAS, process calculi are customarily stateless and use message passing as communication mechanism. The concept of *joint action* used in CASAS is more expressive and can be used to model synchronisation, shared memory communication as well as message passing as shown in [5]. The execution model of CASAS is event-driven just like in the calculus of Communicating Sequential Processes (CSP) [21]. However, CASAS provides notations for expressing security policies and a mechanism to enforcing them continuously like in the UCON model.

Lazouski et al. [27] address the problem of continuous usage control of multiple copies of data objects in distributed systems. They propose an architecture, a set of workflows, a set of policies and a mechanism for the distributed enforcement. The policies include access and usage rules, and also specify the parties that will be involved in the decision process. The enforcement requires collaboration of several entities because the access decision might be evaluated on one site, enforced on another, and the attributes needed for the policy evaluation might be stored in many distributed locations. A prototype implementing main features of the approach was integrated into OpenNebula (One), an open-source middleware for managing Cloud resources and providing Cloud IaaS services. The state partitioning among multiple sites and parties is modelled in CASAS by the notion of *agent* and the collaborative enforcement relies on the concept of

joint action (see Definition 3.7) involving many concurrent agents as participants. Anggorojati et al. [4] propose an access delegation method with security considerations based on Capability-based Context Aware Access Control (CCAAC) model intended for federated machine-to-machine communication in Internet of Things (IoT) networks. The access delegation is realised by means of a capability propagation mechanism, and incorporates the context information as well as secure capability propagation under federated IoT environments. CCAAC is a special case of UCON (which is the security model in CASAS) where capabilities are modelled as subjects and objects attributes; and can be propagated through mutable attributes. Both these works focus on the security requirements in isolation; unlike CASAS which in addition enables the specification of functional requirements in a uniform way.

In [1,30,10,7] event–condition–action (ECA in short) rules are used to model the behaviours of context-aware applications. An ECA rule has the general form: ON *event* IF *cond* DO *action*, where *event* signals a change in the external environment, *cond* is a Boolean expression about the state of the system and *action* is a computation. The meaning of this rule is: when the *event* occurs, if *cond* is true then the *action* is performed. In CASAS, such a rule can be represented as an *action definition*:

action a **on** O_a **by** U_a **context** C_a : $(event \wedge cond) \rightarrow action$

where

- *event* is a Boolean expression upon the states of the *context agents* in C_a;
- *cond* is a Boolean expression upon the states of the *subject agents* in U_a and *object agents* in O_a; and
- *action* is written in the grammar of Table 1.

Attribute-based access control (ABAC) and variants of role-based access control (RBAC) models and languages have been proposed in [31,19,18,17] for the specification of access control in open systems. [15] proposed an adaptive access control scheme which extend RBAC to enable dynamic user and permission assignment based on changes in context. They proposed algorithms for adaptive role assignment, delegation and revocation. Usage control (UCON) model [32] enhances the traditional access control models to enable mutability of subject and object attributes, and continuity of control on usage of resources. It is shown in [32] that previous access control models such as RBAC and Digital Rights Management (DRM) can be represented in UCON. Further extension of UCON [2,3] enables adaptation when access is denied or stopped due to changes in the environmental context. In these works security is addressed separately, generally as an add-on after the system is developed. Our approach proposes a single formalism for specifying both functional and security requirement and reason about them in a uniform manner. In particular, the usage conditions are attribute-based (ABAC) and are continuously checked at run-time (UCON).

Recently, researchers have been interested in developing a fine-grained access control model for the Android platform [8,16] based on the UCON model. This is motivated by the fact that current Android permission model does not allow user to revoke nor change the permissions of an application once the application is installed; users can neither specify context-dependent policies to grant permissions to applications only in specific situations. [8] proposed a context-aware usage control (ConUCON) for Android that enables the specification of spatial and temporal context information in policies to control how permissions are granted to applications based on time and location. They then extend the existing security mechanism of Android to support ConUCON and enable revocation and modification of an application's permissions at run-time. They later applied ConUCON to enhance security of services in Web of Things (WoT) [9]. Likewise, [16] considered a subset of the UCON model and modified the Android implementation to enable the enforcement of context-dependent policies. Their system, CRêPE, enables the users to specify own policies to regulate the behaviour of the applications installed on their smart devices. While these works are concrete implementation of the UCON model on a specific platform (Android), CASAS is designed for modelling and analysing UCON based context-aware systems prior to their actual implementations on any platform. Hence our work is complementary to theirs.

13. Conclusion

This work proposed a new programming paradigm called CASAS where the functional, security and context-awareness requirements of pervasive computing systems can be specified and reasoned about in a uniform manner. This enables the integration of these three types of system requirements using a single formalism. The syntax of CASAS was presented, based on the key concepts of agent, action, context and security policy. A notion of system consistency was defined: a consistent CASAS is one that is well-formed and enforces its security policy correctly. An algorithm for checking this property statically was proposed.

In order to enable formal reasoning, a formal semantics of CASAS was given in LTL. Furthermore, a set of operators for building complex systems from simpler ones in a compositional manner was proposed; these operators are formally proven to be closed under system consistency, i.e. *a system built from well-formed and secure components is well-formed and secure*. Finally, the algebraic properties of these operators were established and sound compositional proof rules were proposed to support a syntactical approach to system verification which is more accessible to programmers and system designers. The pragmatics of the proposed approach is demonstrated using a number of examples.

In future work, a refinement technique and software tools will be developed to support the proposed approach. In particular, an interpreter will be implemented to enable the execution of CASAS programs for simulation purposes. We will also investigate how LTL off-the-shelf tools such as the model-checker SPIN [22] can be used to verify CASAS programs. For this task, the proof rules of Section 11 will be very useful because they enable to infer properties of a large system from those of their smaller components that have been initially checked separately using a model-checker. As to the implementation, we envision that the structure of the formalism will make it easy to be implemented on a smart device. An agent representing a physical sensor (e.g. GPS, clock, or accelerometer) will be refined into the corresponding physical component on the smart device. The one representing a logical sensor will be implemented as a function over (physical) sensors' interfaces. An action will be refined into a function whose parameters are the objects mapped to the participating agents of that action. We will adopt an architecture similar to that of CRêPE [9] and ConUCON [16] for the enforcement of the access control policy. Their works have shown that such an architecture bears little overhead on energy consumption, storage and responsiveness. An appropriate optimisation technique would improve the performance even further.

Appendix. Proofs

In this appendix we give the proofs of all the theorems enounced in this paper.

Theorem 9.1

Proof. Let $\texttt{CASAS}_1$ and $\texttt{CASAS}_2$ be two consistent systems. We shall prove that the system $\texttt{CASAS}_1$; $\texttt{CASAS}_2$ is also consistent. The new agent *watchdog* has a single local variable initialised to *false*. So the agent definition Eq. (25) is consistent according to Definition 3.1. The action definition Eq. (26) is also consistent from Definition 3.5 for the following reasons:

- Requirement 1 holds because any variable that occurs in the guard or the body of this action is part of the state of some participating agent; no variable occurs more than once in the left hand-side of the assignment statement in the body of the action *switch*; and the left hand-side list of the assignment statement is of the same size as the right hand-side list.
- Requirement 2 holds because the state of no agent in $\mathcal{P}_1 \setminus \mathcal{P}_2$ is modified in the body of the action *switch*.
- Requirement 3 holds because the right hand-side of the assignment in the body of the action *switch* is a list of constant values. So there is no flow of information between agents through this action.
- Requirement 4 holds as well due to no flow of information (in the sense of Definition 3.4) between agents in the body of the action *switch*.

The consistency of all the action in Eq. (28) and Eq. (30) follows from that of the actions in Eq. (27) and Eq. (29), respectively. We can then conclude that $\texttt{CASAS}_1$; $\texttt{CASAS}_2$ is consistent.

Theorem 9.2

Proof. The proof is straightforward from Definition 3.6.

Theorem 9.3

Proof. We assume that $\texttt{CASAS}_1$ and $\texttt{CASAS}_2$ are consistent and shall prove that the conditional $b?\texttt{CASAS}_1 : \texttt{CASAS}_2$ is also consistent. To build the conditional of two systems, a new agent *choice* – see Eq. (31) – and a new action *select* – see Eq. (32) – have been introduced. It can be seen that these are consistent with respect to Definitions 3.1 and 3.5, respectively. Finally, the consistency of the actions in Eqs. (34), (36) and (38) follows from that of the actions in Eqs. (33), (35) and (37), respectively.

Theorem 9.4

Proof. Similar to the proof of Theorem 9.3.

Theorem 9.5

Proof. Similar to the proof of Theorem 9.3.

Theorem 11.2

Proof.

- (S-1): Suppose $\texttt{CASAS}$ **sat** ψ_1. By Definition 11.1, this means that $\models \mathcal{M}[\![\texttt{CASAS}]\!] \Rightarrow \bigcirc\psi_1$. From the semantics of LTL given in Table 3, it follows that if $\models \psi_1 \Rightarrow \psi 2$ then $\models \bigcirc\psi_1 \Rightarrow \bigcirc\psi_2$. So by transitivity we deduce that $\models \mathcal{M}[\![\texttt{CASAS}]\!] \Rightarrow \bigcirc\psi_2$, i.e. $\texttt{CASAS}$ **sat** ψ_2.
- (S-2): Assume $\texttt{CASAS}$ **sat** ψ_1 and $\texttt{CASAS}$ **sat** ψ_2. It follows from Definition 11.1 that $\models \mathcal{M}[\![\texttt{CASAS}]\!] \Rightarrow \bigcirc\psi_1$ and $\models \mathcal{M}[\![\texttt{CASAS}]\!] \Rightarrow \bigcirc\psi_2$. From the semantics of LTL given in Table 3, one can conclude that $\models \mathcal{M}[\![\texttt{CASAS}]\!] \Rightarrow \bigcirc(\psi_1 \wedge \psi_2)$, i.e. $\texttt{CASAS}$ **sat** $(\psi_1 \wedge \psi_2)$.
- (S-3): Suppose that $\texttt{CASAS}_1$ **sat** $\bigcirc^{\leq} \omega$. Let $initial_1$ and $initial'$ be the initial state of $\texttt{CASAS}_1$ and $\texttt{CASAS}_1$; $\texttt{CASAS}_2$, respectively; defined as in Eq. (15). One can see that $\models initial' \Rightarrow initial_1$. Since $\texttt{CASAS}_1$ **sat** $\bigcirc^{\leq} \omega$, we have $\models initial_1 \Rightarrow \omega$ (i.e. ω holds in the initial state) and therefore $\models initial_1 \Rightarrow \bigcirc^{\leq} \omega$ (because $\models \omega \Rightarrow \bigcirc^{\leq} \omega$ from the definition in Table 4). It follows thus that $\models initial' \Rightarrow \bigcirc^{\leq} \omega$ and therefore $\models \mathcal{M}[\![\texttt{CASAS}_1; \texttt{CASAS}_2]\!] \Rightarrow \bigcirc\bigcirc^{\leq} \omega$ (from Eq. (23)), i.e. $\texttt{CASAS}_1$; $\texttt{CASAS}_2$ **sat** $\bigcirc^{\leq} \omega$.
- (S-4): Suppose that $\texttt{CASAS}$ **sat** $\bigcirc^{\leq n} \omega$, for some non-negative integer n and state formula ω. We shall prove that $t : \texttt{CASAS}$ **sat** $\bigcirc^{\leq \mathbf{min}(t,n)} \omega$, where t is also a non-negative integer. First of all, note that the action *tick* (see Eq. (45)) shares no synchronisation agent with any other action and so executes continuously in parallel with the system $\texttt{CASAS}$ until its guard $(\neg final_1 \wedge clock.time < t)$ ceases to hold. On one hand, when the expression $clock.time < t$ does not hold, neither does the guard of any other action defined as in Eq. (48); so the system $t : \texttt{CASAS}$ terminates. On the other hand, when $final_1$ holds – i.e. when the system $\texttt{CASAS}$ terminates – the action *tick* is not enabled (see Eq. (45)) and consequently the system $t : \texttt{CASAS}$ also terminates because all the agents are ready but no action is enabled (see Eq. (16)). We shall distinguish two cases: $t \leq n$ and $t > n$.
  - For $t \leq n$: Then the system $t : \texttt{CASAS}$ terminates when $clock.time = t$. Initially, the variable $clock.time$ is set to 0 and continuously incremented by 1 in the action *tick*. Since the action *tick* does not modify the state of (nor synchronise with) any agent in $\texttt{CASAS}$ and the restriction $clock.time < t$ on the guards of the actions (see Eq. (48)) continuously holds till termination, we conclude that $t : \texttt{CASAS}$ **sat** $\bigcirc^{\leq t} \omega$.
  - For $t > n$: For the same reason that the execution of the action *tick* does not interfere with the execution of the system $\texttt{CASAS}$ as long as the condition $clock.time < t$ continues to hold, it follows that $t : \texttt{CASAS}$ **sat** $\bigcirc^{\leq n} \omega$.

- (S-5): $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$ means that the systems $\mathtt{CASAS}_1$ and $\mathtt{CASAS}_2$ do not share any variable, action or agent. So in any state, an action that is enabled in $\mathtt{CASAS}_1$ or $\mathtt{CASAS}_2$ is also enabled in $\mathtt{CASAS}_1 \parallel \mathtt{CASAS}_2$. Similarly, in any state an action that is not enabled in $\mathtt{CASAS}_1$ or $\mathtt{CASAS}_2$ is also not enabled in $\mathtt{CASAS}_1 \parallel \mathtt{CASAS}_2$. From Eq. (15), it follows that $\models \mathit{initial} \Leftrightarrow (\mathit{initial}_1 \wedge \mathit{initial}_2)$, i.e. the initial state *initial* of the parallel composition is the conjunction of the initial states of its immediate components. Similar observation is valid for Eqs. (18)–(22). Therefore $\models \mathcal{M}[\![\mathtt{CASAS}_1 \parallel \mathtt{CASAS}_2]\!] \Leftrightarrow (\mathcal{M}[\![\mathtt{CASAS}_1]\!] \wedge \mathcal{M}[\![\mathtt{CASAS}_2]\!])$. Hence, if $\mathtt{CASAS}$ **sat** ψ then $\mathtt{CASAS}_1 \parallel \mathtt{CASAS}_2$ **sat** ψ.
- (S-6): Suppose $\mathtt{CASAS}$ **sat** $\bigcirc^n \omega$ for some state formula ω, and non-negative integers n and t such that $n \leq t$. We shall prove that $t : \mathtt{CASAS}$ **sat** $\bigcirc^n \omega$. As explained in the proof of (S-4), the action *tick* (see Eq. (45)) shares no synchronisation agent with any other action and so executes continuously in parallel with the system $\mathtt{CASAS}$ until its guard $(\neg \mathit{final}_1 \wedge \mathit{clock.time} < t)$ ceases to hold. The condition $\mathit{clock.time} < t$ restricts the guards of actions as stated in Eq. (48). So for $n \leq t$, the nth state of the execution of the system $t : \mathtt{CASAS}$ coincides with that of the execution of the system $\mathtt{CASAS}$. We conclude that $t : \mathtt{CASAS}$ **sat** $\bigcirc^n \omega$.
- (S-7): Suppose that $\mathtt{CASAS}$ **sat** ψ **U** ϕ. We shall prove that $\langle b \rangle \mathtt{CASAS}$ **sat** ψ **U** $(b \vee \phi)$. From Eqs. (40) and (39), it is clear that the new action *detect* is not enabled until the Boolean expression b holds; till then, only the system $\mathtt{CASAS}$ executes and the state of the agent *detector* is kept unchanged (see Eq. (17)). Once b holds, the action *detect* is executed to set the variable $\mathit{detector.dummy}$ to *true* and so falsifies the guards of all the actions defined as in Eq. (43). The system $\langle b \rangle \mathtt{CASAS}$ then terminates. So, the system $\langle b \rangle \mathtt{CASAS}$ behaves exactly like $\mathtt{CASAS}$ until the condition b holds. For that reason, we can conclude that if $\mathtt{CASAS}$ **sat** ψ **U** ϕ then $\langle b \rangle \mathtt{CASAS}$ **sat** ψ **U** $(b \vee \phi)$.
- (S-8): Similar to the proof of (S-7).
- (S-9): Similar to the proof of (S-7).
- (S-10): Assume that $\mathtt{CASAS}_1$ **sat** ψ_1 and that $\mathtt{CASAS}_2$ **sat** ψ_2. Let *initial* denotes the initial state of $b?\mathtt{CASAS}_1 : \mathtt{CASAS}_2$; defined as in Eq. (15). From Eq. (31), it follows that $\models \mathit{initial} \Rightarrow \mathit{choice.lock} = 0$. So it can be seen from Eq. (33), Eqs. (36) and (38) that *select* is the only enabled action in the initial state and so will be executed immediately to set the variables $\mathit{choice.lock}$ to 1 and $\mathit{choice.cond}$ to the value of the condition b in the initial state. From there on, the variables $\mathit{choice.lock}$ and $\mathit{choice.cond}$ are kept unchanged (see Eq. (17)), and the system continues like $\mathtt{CASAS}_1$ if the value of $\mathit{choice.cond}$ is *true* or like $\mathtt{CASAS}_2$ otherwise, as specified in Eqs. (36) and (38). This is reflected via the action guards in Eqs. (16) and (22). So we can conclude that $b?\mathtt{CASAS}_1 : \mathtt{CASAS}_2$ **sat** $\bigcirc((b \wedge \psi_1) \vee (\neg b \wedge \psi_2))$, where $\bigcirc$ corresponds to the execution of the action *select*.
- (S-11): Assume that $\mathtt{CASAS}_2$ **sat** ψ and that $\mathtt{CASAS}_1$ terminates; let the final state of $\mathtt{CASAS}_1$, defined as in Eq. (16), be final_1. During the execution of $\mathtt{CASAS}_1; \mathtt{CASAS}_2$, only the actions of $\mathtt{CASAS}_1$ are enabled until the final state final_1 is reached. This is because the flag $\mathit{watchdog.flag}$ initialised to *false* in Eq. (25) restricts the guards of the actions as in Eqs. (28) and (30). Once final_1 becomes *true*, the action *switch* is the only action enabled and so will be executed to set the flag $\mathit{watchdog.flag}$ to *true* and (re-)initialise the state of $\mathtt{CASAS}_2$ (see Eq. (26)). Thereafter, the flag $\mathit{watchdog.flag}$ is kept to *true* (see Eq. (17)) and the system continues like $\mathtt{CASAS}_2$ following the restriction of the action guards as in Eqs. (28) and (30). So the property ψ is eventually met, i.e. $\mathtt{CASAS}_1; \mathtt{CASAS}_2$ **sat** $\diamond \psi$.
- (S-12): Assume that $\mathtt{CASAS}_1$ **sat** $\Box\omega$ and $\mathtt{CASAS}_2$ **sat** $\Box\omega$. There are two cases. Case 1: $\mathtt{CASAS}_1$ does not terminate; in this case the result follows immediately. Case 2: $\mathtt{CASAS}_1$ terminates and let the final state of $\mathtt{CASAS}_1$, defined as in Eq. (16), be final_1. During the execution of $\mathtt{CASAS}_1; \mathtt{CASAS}_2$, only the actions of $\mathtt{CASAS}_1$ are enabled until the final state final_1 is reached. This is because the flag $\mathit{watchdog.flag}$ initialised to *false* in Eq. (25) restricts the guards of the actions as in Eqs. (28) and (30). By hypothesis, during the execution of $\mathtt{CASAS}_1$ the formula ω holds in every single state. Once final_1 becomes *true*, the action *switch* is the only action enabled and so will be executed to set the flag $\mathit{watchdog.flag}$ to *true* and (re-)initialise the state of $\mathtt{CASAS}_2$ (see Eq. (26)). Thereafter, the flag $\mathit{watchdog.flag}$ is kept to *true* (see Eq. (17)) and the system continues like $\mathtt{CASAS}_2$ following the restriction of the action guards as in Eqs. (28) and (30). By hypothesis, also during the execution of $\mathtt{CASAS}_2$ the formula ω holds in every single state. So we conclude that $\mathtt{CASAS}_1; \mathtt{CASAS}_2$ **sat** $\Box\omega$.

References

[1] E.E. Almeida, J.E. Luntz, D.M. Tilbury, Event-condition-action systems for reconfigurable logic control, IEEE Trans. Knowl. Data Eng. 4 (2) (2007) 167–181.
[2] A. Almutairi, F. Siewe, Ca-ucon: A context-aware usage control model, in: Proceedings of the 5th ACM International Workshop on Context-Awareness for Self-Managing Systems, CASEMANS'11, ACM, New York, NY, USA, 2011, pp. 38–43.
[3] A. Almutairi, F. Siewe, Formal specification of ca-ucon model using cca, in: Science and Information Conference, 2013, pp. 369–375.
[4] B. Anggorojati, P. Mahalle, N. Prasad, R. Prasad, Capability-based access control delegation model on the federated iot network, in: 15th International Symposium on Wireless Personal Multimedia Communications WPMC, 2012.
[5] R. Back, R. Kurki-Suonio, Decentralization of process nets with centralized control, in: 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, ACM, Montreal, 1983, pp. 131–142.
[6] R. Back, R. Kurki-Suonio, Distributed co-operation with action systems, ACM Trans. Program. Lang. Syst. 10 (1984) 513–554.
[7] J. Bae, H. Bae, S.-H. Kang, Y. Kim, Automatic control of workflow processes using ECA rules, IEEE Trans. Knowl. Data Eng. 16 (8) (2004) 1010–1023.
[8] G. Bai, L. Gu, T. Feng, Y. Guo, X. Chen, Context-aware usage control for android, in: S. Jajodia, J. Zhou (Eds.), Security and Privacy in Communication Networks, in: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 50, Springer, Berlin Heidelberg, 2010, pp. 326–343.
[9] G. Bai, L. Yan, L. Gu, Y. Guo, X. Chen, Context-aware usage control for web of things, Security and Communication Networks.
[10] T. Beer, J. Rasinger, W. Höpken, M. Fuchs, H. Werthner, Exploiting E-C-A rules for defining and processing context-aware push messages, Lecture Notes in Comput. Sci. 4824/2007 (2007) 199–206.
[11] L. Birkedal, S. Debois, E. Elsborg, T. Hildebr, H. Niss, Bigraphical models of context-aware systems, in: IT University of Copenhagen, Springer-Verlag, 2006, pp. 187–201.
[12] D. Bucur, M. Nielsen, Secure data flow in a calculus for context awareness, in: Concurrency, Graphs and Models, in: Lecture Notes in Computer Science, vol. 5065, Springer, 2008, pp. 439–456.
[13] L. Cardelli, A.D. Gordon, Mobile ambients, Theoret. Comput. Sci. 240 (2000) 177–213.
[14] K. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, NY, USA, 1988.
[15] J.H. Choi, S.D.H. Khang, H. Jang, Y.I. Eom, Adaptive access control scheme utilizing context awareness in pervasive computing environments, in: IEEE International Performance, Computing and Communications Conference, 2008, IPCCC 2008, IEEE, 2008, pp. 491–498.
[16] M. Conti, B. Crispo, E. Fernandes, Y. Zhauniarovich, CRêPE: A system for enforcing fine-grained context-related policies on android, IEEE Trans. Inf. Forensics Secur. 7 (5) (2012) 1426–1438.
[17] M.J. Covington, P. Fogla, Z. Zhan, M. Ahamad, A context-aware security architecture for emerging applications, in: Proceedings of the 18th Annual Conference on Computer Security Applications, 2002, pp. 249–258.
[18] M.J. Covington, W. Long, S. Srinivasan, A.K. Dev, M. Ahamad, G.D. Abowd, Securing context-aware applications using environment roles, in: Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies, SACMAT'01, ACM, New York, NY, USA, 2001, pp. 10–20.
[19] J. Crampton, C. Morisset, Ptacl: A language for attribute-based access control in open systems, in: Proceedings of the First International Conference on Principles of Security and Trust, POST'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 390–409.
[20] M.T. Gamble, R.F. Gamble, M.L. Hale, Security policy foundations in context UNITY, in: the 7th International Workshop on Software Engineering for Secure Systems (SESS'11), ACM, Waikiki, Honolulu, USA, 2011, pp. 8–14.
[21] C. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[22] G.J. Holzmann, The model checker SPIN, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–295.
[23] O.H. Jensen, R. Milner, Bigraphs and Mobile Processes (revised), Tech. Rep. UCAM-CL-TR-580, University of Cambridge, 2004.
[24] D.E. Knuth, Semantics of context-free languages, Math. Syst. Theory 2 (2) (1968) 127–145.
[25] J. Krivine, R. Milner, A. Troina, Stochastic bigraphs, in: Proc. of MFPS'08, 24th Conference on the Mathematical Foundations of Programming Semantics, ENTCS, Elsevier, 2008.
[26] D.S. Kyle, J.C. Brustoloni, Uclinux: a linux security module for trusted-computing-based usage controls enforcement, in: The Second ACM Workshop on Scalable Trusted Computing, 2007.
[27] A. Lazouski, G. Mancini, F. Martinelli, P. Mori, Architecture, workflows, and prototype for stateful data usage control in cloud, in: IEEE Security and Privacy Workshops, 2014.
[28] Z. Manna, A. Pneuli, Verification of concurrent programs: the temporal framework, in: The Correctness Problem in Computer Science, in: International Lectures Series in Computer Science, Academic Press, London, 1981.
[29] R. Milner, Pure bigraphs: structure and dynamics, Inform. and Comput. 204 (2006) 60–122.
[30] A. Moon, H. Kim, H. Kim, S. Lee, Context-aware active services in ubiquitous computing environments, ETRI J. 29 (2) (2007) 169–178.
[31] OASIS, extensible access control markup language (xacml) version 3.0, 2010.
[32] J. Park, R. Sandhu, The UCON_{ABC} usage control model, ACM Trans. Inf. Syst. Secur. 7 (1) (2004) 128–174.
[33] A. Pneuli, The temporal logic of programs, in: Proceedings of the 18th Symposium on Foundations of Computer Science, IEEE, Providence, 1977, pp. 46–57.
[34] G. Roman, McCann, A notation and logic for mobile computing, Formal Mthods Syst. Des. 20 (1) (2002) 47–68.
[35] G.-C. Roman, C. Julien, J. Payton, Modeling adaptive behaviors in context UNITY, Theoret. Comput. Sci. 376 (3) (2007) 185–204. Fundamental Aspects of Software Engineering.
[36] F. Siewe, H. Zedan, A. Cau, The calculus of context-aware ambients, J. Comput. System Sci. 77 (4) (2011) 597–620.
[37] M. Weiser, The computer for the 21st century, Sci. Am. 265 (3) (1991) 94–104.
[38] M. Weiser, Some computer science issues in ubiquitous computing, Commun. ACM 36 (7) (1993) 75–84.
[39] M. Xu, X. Jiang, R. Sandhu, X. Zhang, Towards a vmm-based usage control framework for os kernel integrity protection, in: 12th ACM symposium on Access Control Models and Technologies, 2007.
[40] P. Zimmer, A Calculus for Context-awareness, Tech. rep, BRICS, 2005.



Dr. François Siewe is currently a senior research fellow at De Montfort University (DMU) in Leicester, in United Kingdom. He received his Ph.D. in computer science from the Software Technology Research Laboratory at DMU in 2005. He obtained his B.Sc. in Mathematics and Computer Science, M.Sc. and Doctorat de Troisieme Cycle in Computer Science from the University of Yaounde I, Cameroon, in 1990, 1992 and 1997 respectively. His research interests include computer security, context-aware systems, ubiquitous and pervasive computing, and formal verification.