International Journal on Advances in Security, vol 2 no 4, year 2009, http://www.iariajournals.org/security/

Analysing security requirements formally and flexibly based on suspicion

Nuno Amálio

Abstract—**Increasingly, engineers need to approach security and software engineering in a unified way. This paper presents an approach to the formal analysis of security requirements that is based on model-checking and uses the concept of suspicion to guide the search for threats and security vulnerabilities in requirements. It proposes an approach to security analysis that favours exploration of a system's state space based on what is abnormal or suspicious to find threats and vulnerabilities, instead of ironclad security proofs that try to demonstrate that a system is secure; as this paper shows, such security proofs can often be misleading. The approach is tested and illustrated by conducting two experiments: one focussing on a system with a confidentiality security property, and another with an integrity security property enforced through the separation of duty principle. One of the advantages of the approach presented here is that threats are derived directly from a model of requirements and no prior knowledge about possible attacks is necessary to perform the analysis. The paper shows that suspicion is an effective search criteria for finding vulnerabilities and security threats in requirements, and that the feedback generated by the analysis helps in elaborating security requirements.**

Index Terms—**Security, requirements, formal analysis, Event-Calculus, planning, confidentiality, separation of duty.**

I. Introduction

Traditional approaches to software engineering and current practice tend to treat security concerns as an after-thought [1]. Security requirements are handled as non-functional requirements and are kept separate from their functional counter-parts until design or implementation-time. This raises problems for the whole software development process because (as demonstrated in this paper) functionality has an impact on security. If security aspects are not treated properly at the requirements phase, then the resolution of the problem will inevitably be deferred but at a much higher cost, which is a well known software engineering problem [2]. This issue can be resolved by integrating security into the requirements engineering phase of the software life-cycle [1], [3]. However, the best way to capture, model and analyse security and system requirements in a unified way is still an open problem.

Security requirements have proved tricky to formulate and reason about [4]. There are several methods for reasoning about properties that a system must satisfy. Traditionally, properties are classified as either *safety* or *liveness* [5], [6]. Safety properties say that something bad must not happen, and liveness properties say that something good must eventually happen. We check safety to ensure that bad states are not reachable, and liveness to ensure that good states are eventually reachable. This is used to check that invariants are preserved, that operations are applicable when certain conditions are met (pre-conditions) and that operations have the desired effect taking the system into a valid state. However, important classes of security requirements are either difficult to express using traditional safety or liveness properties, or they are just not possible to express at all [7], [4], [8].

This paper claims that, from a practical point of view, not only it is important to verify *safety* (that some unsecure state is not reached) and liveness (that the security measures do what is expected from them), but also in finding security vulnerabilities and possible security threats that give attack opportunities to malicious users: we need to look for *what can happen under certain suspicious conditions*. Traditionally, such *possibilistic* properties [7] are hard to formulate and reason about. This paper proposes a practical approach for dealing with such properties.

This paper presents a practical approach to the formal analysis of security requirements based on model-checking, where the search for threats and vulnerabilities in requirements is based on what is *suspicious* from a security point of view. This is inspired by anomaly-based approaches to intrusion detection [9], where the search for intrusions at run-time is driven by *abnormal* (or suspicious) behaviour of system use. The approach presented here takes a formal model of requirements and an analysis goal (a description of suspicious states from the analysis point of view), which are used by the model checker to generate traces of events describing how the analysis goals is reached. Each trace gives a scenario illustrating a possible threat or security vulnerability. The generation of plans is done automatically with tool support. The approach is illustrated with the Event-Calculus temporal logic [10] and the analysis is conducted with tool support using the discrete event calculus reasoner[1] (*decreasoner*).

The remainder of this paper starts by giving a brief introduction to event calculus (section II). Then it presents the approach to the formal analysis of security requirements that is proposed here (section III). After this, the analysis approach is used to conduct two experiments: analysis of a simple health-care system with a confidentiality security requirement (section IV), and analysis of a business system with an integrity requirement enforcing separation of duty (section V). Then, the paper summarises the experimental results that we obtained (section VI), discusses the paper's results (section VII), presents some related work (section VIII), and takes the conclusions.

N. Amálio is with the University of Luxembourg.

[1] http://decreasoner.sourceforge.net/

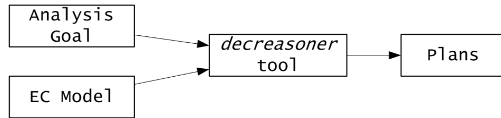

Fig. 1. Planning-based formal analysis. EC model of requirements, and EC description of analysis goal are fed into *decreasoner* tool to obtain a set of plans (scenarios) that achieve the goal.

II. The Event Calculus

Event Calculus (EC) [10], [11], a temporal logic based on first-order predicate calculus designed for common-sense reasoning, enables representation and reasoning about action and change. Its basic ontology comprises *events*, *fluents* and *timepoints*. An *event* is an action that may occur in the world. A *fluent* is a time varying property of the world. A *timepoint* is an instance of time. EC includes a set of basic predicates to describe happening of events, their effects and state of fluents. An EC model is built by describing two types of facts: the fact that an event occurs at a timepoint, and the fact that a property holds at a timepoint [11]. These facts are either true or false.

The basic predicates of EC are as follows:

- $HoldsAt\ (f, t)$ says that fluent f is true at timepoint t.
- $Happens\ (e, t)$ says that event e may occur at timepoint t.
- $Initiates\ (e, f, t)$ says that if event e occurs at timepoint t, then fluent f is true after t.
- $Terminates\ (e, f, t)$ says that if event e occurs at timepoint t, then fluent f is false after t.
- $Initially\ (f)$ says that fluent f holds at timepoint 0.

III. Formal threat analysis by studying reachability

The analysis proposed here is essentially a study of *reachability*: it checks whether certain states are reachable from a model of the requirements. The actual generation of threat scenarios is based on AI planning [12] and uses the *decresoner* tool, which is based on a SAT approach to EC reasoning [13][2]. This is related to what is known in software engineering as *model-checking* [14]: the exploration of all possible states and transitions of a model to determine if a certain property holds or not

Figure 1 depicts the analysis approach followed here. The EC model and EC description of analysis goals are given as inputs to *decreasoner*, which generates a set of plans (or traces) that satisfy the goal. Each plan describes a scenario comprising a sequence of events (a *trace*) that takes the system from the initial state to one of the states described by the goal.

The goal (a predicate) describes a set of states that are interesting from the analysis point of view. Planning generates plans that reach such a state. If there are plans, then the goal is satisfiable: a state as described by the goal can be reached in the model. If no plans can be found, then no state as described by the goal is reachable. If the goal state describes something that should not happen, then the resulting plans (scenarios) describe a sequence of events that reach such a state; thus exposing a way to reach something undesired.

Two strategies are used to formulate the goal. There is a more traditional strategy that does a *safety analysis* by formulating a goal describing states where security is violated and that must not happen; these goals are called *security violation* goals. The other strategy applies suspicion by defining a goal describing *suspicious states deserving investigation* that may expose possible vulnerabilities and threats. These are called *suspicious goals*.

[2]Appendix A shows sample outputs given by decreasoner for the suspicion-based analysis conducted in this paper.

R1	Doctors must be able to access their patient's medical data to provide effective medical care.
R2	A doctor may nominate a substitute who may be able to access the patient's medical data only when main doctor is on leave.

TABLE I
The requirements of Simple Medical Information System.

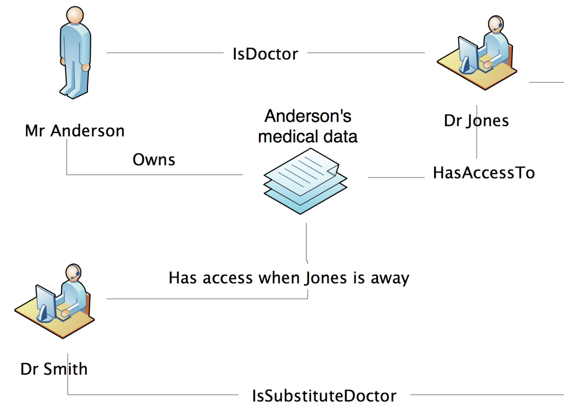

Fig. 2. The SMIS with one patient, Anderson, his doctor Jones, and another doctor, Smith, who is able to replace Jones while he is on leave.

IV. Confidentiality

Confidentiality is about protecting information. It tries to ensure that sensitive information is accessible only to those authorised to access it. Analysis of confidentiality involves checking ways in which confidential information may be accessed by those who are not authorised. Here, confidentiality is studied using a case study of a domain where it is a professional ethical principle: health-care [15].

The case study is a simple medical information system (SMIS) that manages patient information. Due to its sensitive nature, patient information is subject to confidentiality constraints to protect patient's privacy. The requirements of SMIS are summarised in table I. Figure 2 depicts a concrete system scenario of SMIS.

A. EC Model

To satisfy the requirements of SMIS, EC model presented here introduces a protection mechanism based on *credentials*: doctors need to request a credential prior to accessing the data; credentials have a validity period.

The building blocks of an EC model are *sorts*, *events* and *fluents*. EC model of SMIS comprises sort $Delay$, representing

time delays used in credential mechanism, and domain sorts *User*, representing a user of SMIS, *Doctor*, a sub-sort of $User$ that represents doctors using SMIS, and $Patient$, which represents the patients recorded in the system.

EC model of SMIS comprises the following events:

- $AuthoriseAccess\ (d,\ p)$: occurs when doctors (d) request a credential for accessing some patient's data (p).
- $GetMD\ (d,\ p)$: occurs when doctors (d) actually access patient's medical data (p).
- $SetSubstituteDoctor\ (u,\ d_1,\ d_2)$: occurs when a user ($u$) sets some doctor ($d_1$) as substitute of another (d_2).
- $SetDoctorOnLeave\ (u,\ d)$: occurs when a user ($u$) informs system that some doctor (d) is on leave.
- $DoctorNoLongerOnLeave\ (u,\ d)$: occurs when a user ($u$) informs system that a doctor (d) is no longer on leave.

EC model defines the following fluents to hold state:

- $IsDoctorOf\ (d,\ p)$ says who is doctor (d) of some patient (p).
- $CredentialMD\ (d,\ p,\ t)$ says that a doctor (d) has been issued a credential to access data of some patient (p) at time-point t.
- $ExposedToAt\ (d,\ p,\ t)$ says that a doctor (d) has seen the medical data of some patient (p) at time-point t.
- $IsSubstituteDoctor(d_1, d_2)$ says that d_1 is substitute doctor of d_2.
- $OnLeave\ (d)$ says that doctor d is on leave.

EC model of SMIS starts by constraining $Duration$ predicate, which gives actual time associated with delays (of $Delay$ sort). Next EC equation says that $Duration$ relation is functional; each delay has at most one duration (a time point):

$$\forall\ d\ :\ Delay;\ t_1, t_2\ :\ Time\ |\ Duration\ (d,\ t_1) \wedge Duration\ (d,\ t_2) \Rightarrow t_1 = t_2 \quad (1)$$

Next EC equation says that $Duration$ is total; all delays of the model must have a duration associated:

$$\forall\ d\ :\ Delay\ |\ (\exists\ t\ :\ Time)\ Duration\ (d,\ t) \quad (2)$$

Next EC equation says that $IsDoctorOf$ is a surjective relation; each patient must have a doctor:

$$\forall\ p\ :\ Patient;\ t\ :\ Time\ |\ (\exists\ d\ :\ Doctor)\ HoldsAt\ (IsDoctorOf\ (d,\ p),\ t) \quad (3)$$

Next EC equation defines predicate $CanAccessMD$, describing conditions ruling doctors' access to patient's data:

$$\begin{aligned}&\forall\ d : Doctor; p : Patient; t : Time\ |\\ &\quad HoldsAt\ (CanAccessMD\ (d,p),t)\\ &\quad\quad \Leftrightarrow HoldsAt\ (IsDoctorOf\ (d,p),t)\\ &\quad\quad \vee ((\exists\ d' : Doctor)\ HoldsAt\ (IsDoctorOf\ (d',p),t)\\ &\quad\quad\quad \wedge HoldsAt\ (IsSubstituteDoctor\ (d,d'),t)\\ &\quad\quad\quad \wedge HoldsAt\ (OnLeave\ (d'),t)) \quad (4)\end{aligned}$$

This says that a doctor can access some patient's medical data at some time point provided doctor is either (a) patient's doctor (fluent $IsDoctorOf$) or (b) substitute doctor of patient's doctor (fluent $IsSubstituteDoctor$) who is on leave at that time (fluent $OnLeave$). This formalises requirements $R1$ and $R2$.

Next EC equation describes how doctors get credentials to access patient's data by describing effect of event $AuthoriseAccess$:

$$\begin{aligned}&\forall\ d\ :\ Doctor;\ p\ :\ Patient;\ t\ :\ Time\ |\\ &\quad HoldsAt\ (CanAccessMD\ (d,p),t)\\ &\quad \Rightarrow Initiates\ (AuthoriseAccess\ (d,p),\\ &\quad\quad CredentialMD\ (d,\ p,\ t),\ t) \quad (5)\end{aligned}$$

This says that some doctor gets a credential to access a medical file (fluent $CredentialMD$ is initiated) upon event $AuthoriseAccess$, provided doctor can access patient's data (predicate $CanAccessMD$ defined above). This formalises the scheme of credential-based protection.

Model objects of $Delay$ sort have a duration, as defined by $Duration$ predicate defined above. Next EC equation defines delay $credentialValidity$ representing validity period of a credential:

$$credentialValidity\ :\ Delay \quad (6)$$

Next EC equation defines validity conditions of credentials, which are captured by predicate $HasValidCredential$:

$$\begin{aligned}&\forall\ d\ :\ Doctor;\ p\ :\ Patient;\ t\ :\ Time\ |\\ &\quad HoldsAt\ (HasValidCredential\ (d,\ p),\ t)\\ &\quad \Leftrightarrow (\exists\ t_2,\ t_3\ :\ Time)\\ &\quad\quad HoldsAt\ (CredentialMD\ (d,\ p,\ t_2),\ t)\\ &\quad\quad \wedge Duration\ (credentialValidity,\ t_3)\\ &\quad\quad \wedge (t_2\ +\ t_3)\ \geq\ t \quad (7)\end{aligned}$$

This says that a credential is valid for the duration period defined by $credentialValidity$.

Next EC equation defines precondition of event $GetMD$, which may happen provided requesting doctor has a valid credential to access requested patient's data:

$$\begin{aligned}&\forall\ d\ :\ Doctor;\ p\ :\ Patient;\ t\ :\ Time\ |\\ &\quad Happens\ (GetMD\ (d,\ p),\ t)\\ &\quad\quad \Rightarrow\ HoldsAt\ (HasValidCredential\ (d,\ p),\ t) \quad (8)\end{aligned}$$

Next EC equation describes how event $GetMD$ initiates (sets to true) fluent $ExposedToAt$ that records exposure to patient's data has been accessed:

$$\forall\ d\ :\ Doctor;\ p\ :\ Patient;\ t\ :\ Time\ |\ Initiates\ (GetMD\ (d,\ p),\ ExposedToAt\ (d,\ p,\ t),\ t) \quad (9)$$

Next equation constrains fluent $IsSubstituteDoctor$ to be non-reflexive; that is, a doctor may not be set as a substitute of himself:

$$\forall\ d\ :\ Doctor;\ t\ :\ Time\ |\ \neg\ HoldsAt\ (IsSubstituteDoctor\ (d,\ d),\ t) \quad (10)$$

Next equation describes effect how event $SetSubstituteDoctor$ initiates fluent $IsSubstituteDoctor$ to record that some doctor is substitute of another:

$$\begin{aligned}&\forall\ u\ :\ User;\ d_1,\ d_2\ :\ Doctor;\ t\ :\ Time\ |\\ &\quad Initiates\ (SetSubstituteDoctor\ (u, d_1, d_2),\\ &\quad\quad IsSubstituteDoctor\ (d_2, d_1), t) \quad (11)\end{aligned}$$

Next equation describes how event $SetDoctorOnLeave$ initiates (sets to true) fluent $OnLeave$; some user (u) informs system that some doctor (d) is on-leave:

$$\forall\ u : User; d : Doctor; t : Time\ |\ Initiates\ (SetDoctorOnLeave\ (u,d), OnLeave\ (d), t) \quad (12)$$

Next equation describes how event $DoctorNoLongerOnLeave$ terminates (sets to false) fluent $DoctorNoLongerOnLeave$; this used so that users u inform system that a doctor (d) is no longer on leave:

$$\begin{aligned}&\forall\ u : User, d : Doctor; t : Time \mid\\&\quad Terminates\ (DoctorNoLongerOnLeave\ (u, d),\\&\qquad OnLeave\ (d), t)\end{aligned} \quad (13)$$

Next equations describe the initial condition of SMIS:

$$\begin{aligned}&\forall\ d : Doctor; p : Patient; t : Time \mid\\&\quad Initially\ (\neg\ ExposedToAt\ (d, p, t)) &(14)\\&\forall\ d_1, d_2 : Doctor \mid\\&\quad Initially\ (\neg\ IsSubstituteDoctor\ (d_1, d_2)) &(15)\\&\forall\ d : Doctor \mid Initially\ (\neg\ OnLeave\ (d)) &(16)\end{aligned}$$

Above, initially no one has been exposed to medical data (fluent $ExposedToAt$), no doctors are set as substitutes, and there are no doctors on leave.

This completes EC model of SMIS's requirements (table I). Next section, formally analyses this model.

B. Model Analysis

Analysis uses configuration depicted in Fig. 2. There are two doctors, Jones and Smith, and a patient of Jones, Anderson. This is formulated in EC as:

$$\begin{aligned}&jones, smith : Doctor &(17)\\&anderson : Patient &(18)\\&Initially\ (IsDoctorOf\ (jones, anderson)) &(19)\end{aligned}$$

Configuration for model analysis also needs to define duration of $credentialValidity$ delay, which is defined as taking three time-points:

$$Duration\ (credentialValidity, 3) \quad (20)$$

Analysis starts by formulating a security violation goal: it asks whether it is possible to reach a state where patient confidentiality is compromised. In the context of SMIS, this happens when some doctor accesses some patient's data without a valid security credential; the goal expressing this is formulated as:

$$\begin{aligned}&\exists\ d : Doctor; p : Patient; t_1, t_2 : Time \mid\\&\quad HoldsAt\ (ExposedToAt\ (d, p, t_2), t_1)\\&\quad \wedge\neg\ HoldsAt\ (HasValidCredential\ (d, p), t_2)\end{aligned} \quad (AG1)$$

For this goal, *decreasoner* does not find any traces (a fragment of decreasoner's output for this goal is given appendix A1). This means that the modelled system cannot reach one of the goal's states. At this point, one could argue that the modelled system is secure because it is not possible to reach an unsecure state, but it isn't so.

Analysis proceeds by applying suspicion. The substitute doctor rule (Requirement $R2$) enables access to patient's data by doctors other than the main patient's doctor. This should occur, but not very often; the situations under which this occurs are suspicious and deserve investigation. Analysis investigates states where those other than the main doctor access the patient's data. This is formulated as the goal:

$$\begin{aligned}&\exists\ d : Doctor;\ p : Patient;\ t_1,\ t_2 : Time \mid\\&\quad HoldsAt\ (ExposedToAt\ (d,\ p,\ t_2),\ t_1)\\&\quad \wedge\neg\ HoldsAt\ (IsDoctorOf\ (d,\ p),\ t_2)\end{aligned} \quad (AG2)$$

R3	Only the doctors themselves or one of their administrators are allowed to nominate a substitute, and inform the system of their absence (on-leave) or return to duty.

TABLE II
REQUIREMENTS EMERGING AFTER ANALYSIS OF SMIS INITIAL REQUIREMENTS.

For this goal, *decreasoner* generates traces exposing a security vulnerability, which enables doctors to access patient's data in non-legal ways (see appendix A2 for output generated by decreasoner). In some scenarios, the system behaves as intended: $Jones$ sets $Smith$ as his substitute, at a later time $Jones$ informs system that he is on leave, and so $Smith$ is able to access the medical data (model 5 in appendix A2). Other scenarios are more unusual. In some of them, it is possible that $Smith$ himself requests to be the substitute of $Jones$ (model 1 in appendix A2), and that it is $Smith$ who informs the system that $Jones$ is on-leave (model 1 in appendix A2). Obviously this is strange and could be explored by a malicious doctor determined to get some patient's medical data: (a) he sets himself as substitute of another doctor, then (b) he informs the system that main doctor is on-leave and (c) finally he is able to access the patient's data.

Such scenarios are possible because events $IsSubstituteDoctor$ and $SetDoctorOnLeave$ (equations 10 and 11) are unconstrained: any user may execute them, which introduces a loophole providing an opportunity for access to patient's data in ways that are not intended.

We can confirm the vulnerability by posing an analysis question. We want to know if it is possible that some user can nominate himself as some other doctor's substitute and then to be able to access the data of a patient that is not his own. This is formulated as:

$$\begin{aligned}&\exists\ d_1,\ d_2 : Doctor;\ p : Patient;\ t_1,\ t_2,\ t_3 : Time \mid\\&\quad Happens\ (SetSubstituteDoctor\ (d_1, d_2, d_1), t_2)\\&\quad HoldsAt\ (ExposedToAt\ (d_1, p, t_3), t_1)\\&\quad \wedge\neg\ HoldsAt\ (IsDoctorOf\ (d_1, p), t_3)\ \wedge\ t_2 \leq t_3\end{aligned} \quad (AG3)$$

For this goal, descreasoner is able to identify many scenarios, thus confirming the identified vulnerability.

C. Fixing the Model

The analysis' findings are used to elaborate the requirements. We try to remove the vulnerability that has been identified. The requirements that emerge as a result of this elaboration are given in table II; here, $R1$ and $R2$ of table I still hold, and there is new requirement $R3$, which introduces users of type administrators that execute administration tasks on behalf of doctors.

This new requirement needs to be reflected in the EC model. New version of EC model introduces sort $Admin$, subsort of $User$ sort, and which is disjoint from $Doctor$ sort. It also introduces a new fluent:

- $HasAdmin\ (d,\ a)$ indicates administrator user (a) doing administrative tasks on behalf of some doctor d.

This new sort and fluent are used to describe the new requirement. Next EC equation constrains $HasAdmin$ to be a total relation; each doctor must have at least one administrator:

$$\forall\ d\ :\ Doctor;\ t\ :\ Time\ |\ (\exists\ a\ :\ Admin)\ HoldsAt\ (HasAdmin\ (d,\ a),\ t) \quad (21)$$

Next EC equation constrains $HasAdmin$ to be a surjective relation; each administrator must be associated with a doctor:

$$\forall\ a : Admin;\ t\ :\ Time\ |\ (\exists\ d\ :\ Doctor)\ HoldsAt\ (HasAdmin\ (d,\ a),\ t) \quad (22)$$

Next EC equation defines predicate $CanDoAdmin$, which indicates users (u) that can do administrative tasks on behalf of some doctor (d):

$$\forall\ u : User;\ d\ :\ Doctor;\ t\ :\ Time\ |\ HoldsAt\ (CanDoAdmin\ (u,\ d),\ t) \Leftrightarrow\ u\ =\ d\ \vee\ ((\exists\ ad\ :\ Admin\ )\ u = ad \wedge\ HoldsAt\ (HasAdmin\ (d,\ ad), t)) \quad (23)$$

Next EC equations use predicate $CanDoAdmin$ to define a pre-condition for events $SetSubstituteDoctor$, $DoctorOnLeave$ and $DoctorNoLongerNoLeave$. These events may occur provided $CanDoAdmin$ is true; that is, user executing them can do administrative tasks on behalf of affected doctor:

$$\forall\ u\ :\ User;\ d_1, d_2\ :\ Doctor;\ t\ :\ Time\ |\ Happens\ (SetSubstituteDoctor\ (u,\ d_1,\ d_2), t) \Rightarrow HoldsAt\ (CanDoAdmin\ (u,\ d_1),\ t) \quad (24)$$

$$\forall\ u\ :\ User;\ d\ :\ Doctor;\ t\ :\ Time\ |\ Happens\ (SetDoctorOnLeave\ (u,\ d),\ t) \Rightarrow HoldsAt\ (CanDoAdmin\ (u,\ d),\ t) \quad (25)$$

$$\forall\ u\ :\ User;\ d\ :\ Doctor;\ t\ :\ Time\ |\ Happens\ (DoctorNoLongerOnLeave\ (u,\ d),\ t) \Rightarrow\ HoldsAt\ (CanDoAdmin\ (u,\ d),\ t) \quad (26)$$

D. Re-analysing the Model

The analysis configuration is refined by introducing two administrators; one for each doctor:

$$alice,\ sue\ :\ Admin \quad (27)$$
$$Initially\ (HasAdmin\ (jones,\ alice)) \quad (28)$$
$$Initially\ (HasAdmin\ (smith,\ sue)) \quad (29)$$

Under the revised EC model, we re-submit the model to the security violation and suspicion goals:

- For the security violation goal (equation $AG1$ above), we still get no plans (not possible to break confidentiality in an obvious way).
- For the refined suspicious goal (equation $AG3$ above), we no longer get any plans. Meaning that the loophole has been eliminated.
- For the more abstract suspicious goal (equation $AG2$ above) we no longer get obvious security threats, but the results still prompt interesting requirements questions, such as: the system allows doctors to operate the system while they are recorded as being on leave, should this be allowed?

R1	There are two types of users clerks and managers. Managers can performs the tasks that usually the clerks do, but clerks should not usually perform manager's tasks (exception is delegation, below).
R2	Clerks are responsible for starting the refund procedure, and for issuing or cancelling the refund.
R3	The refund shall be issued by a clerk if approved by the managers, or cancelled otherwise.
R4	A refund must by approved by two different managers.
R5	A clerk shall not both prepare and issue or cancel a refund.
R6	Managers can delegate the authority on approval of refunds to one of their administrators.

TABLE III
REQUIREMENTS OF PAYMENT PROCESSING WORKFLOW.

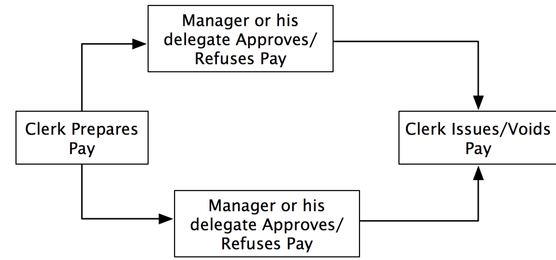

Fig. 3. The payment processing workflow.

V. Separation of Duties

Separation of Duties (SoD) [16], [17] is a security mechanism used to prevent fraud and errors. It aims to prevent a single individual from executing business-critical tasks of some transactions or business processes. SoD requires such tasks to be performed by different users acting in cooperation (e.g by requiring two persons to sign a cheque). Here, SoD is studied using a classical case study: a workflow of payment processing; SoD is used to enable payment authorisations to be performed by different users.

The requirements of this workflow system are given in table III; Fig. 3 depicts underlying workflow. The two tasks involving approval of payments to be carried out by managers, and the tasks *prepare payment* and *issue/void payment*, to be carried out by clerks, are subject to SoD.

A. EC Model

The EC model presented here models a *workflow* as a set of activities. Each activity is made of several alternative tasks; one of the tasks must be carried out to complete the activity. In workflow of Fig. 3, activity *Approve/Refuse Pay* comprises tasks *approve pay* and *refuse pay*, and activity *Issue/Void Pay* comprises tasks *issue pay* and *void pay*. There is always some active activity in some running workflow session.

Workflow tasks are executed by users who have different task execution permissions. Task permissions are defined at the level or rôles; a user is assigned one or more user rôles. In Fig. 3, users that have rôle *clerk* may execute tasks Prepare Pay, Issue Pay and void Pay; users of rôle *manager* may execute tasks Approve Pay and Refuse Pay and all other tasks that clerks do. The model also enables delegation; administrators of managers may also execute tasks on managers' behalf.

The following describes the following elements of EC model: sorts; activities; tasks, rôles and delgation; task execution; and payment processing workflow.

1) Sorts: EC model introduces sorts $Activ$, $Task$, $Session$ and $User$. $Activ$ represents activities of a workflow; $Task$ represents a workflow task that is executed by users; $Session$ represents a running session of a workflow; and $User$ represents users that can execute tasks. Rôles are modelled as sub-sorts of sort $User$; rôles of payment processing workflow are defined below (section V-A5).

2) Activities: Fluent $CurrActiv\ (a,\ s)$ of EC model records current activity (a) of some workflow session (s). Next EC equations constrain fluent $CurrActiv$ to say that there is at most one current activity per workflow session:

$$\begin{array}{l} \forall\ a_1,\ a_2\ :\ Activ;\ s\ :\ Session;\ t\ :\ Time\ | \\ \quad HoldsAt\ (CurrActiv\ (a_1,\ s),\ t) \\ \quad HoldsAt\ (CurrActiv\ (a_2,\ s),\ t) \\ \quad \Rightarrow\ a_1\ =\ a_2 \end{array} \tag{30}$$

Next EC equation gives initial state of fluent $CurrActiv$, saying that initially there are no current activities:

$$\begin{array}{l} \forall\ a\ :\ Activ;\ s\ :\ Session\ | \\ \quad Initially\ (\neg\ CurrActiv\ (a, s)) \end{array} \tag{31}$$

To represent workflow configurations in terms of its constituent activities, EC model uses predicates $OccursBefore$. This indicates the ordering of activities in a workflow. Predicate $OccursBefore$ is defined below for payment processing workflow (section V-A5, equation 50).

Predicate $IsStartActiv$ indicates start activity of a workflow; it is defined from predicate $OccursBefore$ as:

$$\begin{array}{l} \forall\ a_1\ :\ Activ\ | \\ \quad IsStartActiv\ (a_1) \\ \quad \Leftrightarrow\ \neg\ ((\exists\ a_2\ :\ Activ)\ OccursBefore\ (a_2,\ a_1)) \end{array} \tag{32}$$

In model proposed here, separation of duties is enforced on activities. In workflow of Fig. 3, we have a SoD constraint between activities *Approve/Refuse Pay 1* and *Approve/Refuse Pay 2* and *Prepare Pay* and *Issue/Void Pay*. In EC model, such constraints are represented using predicate SoD, which is defined for each workflow that is to be described and analysed (described below for payment processing workflow in equation 51).

To define a precondition for event $StartWrkf$, EC model introduces predicate $Started$, which indicates whether some session has been started or not:

$$\begin{array}{l} \forall\ s\ :\ Session;\ t\ :\ Time\ | \\ \quad HoldsAt\ (Started\ (s),\ t) \\ \quad \Leftrightarrow\ (\exists\ a\ :\ Activ;\ t_2\ :\ Time)\ t_2 \geq t \\ \quad\quad \wedge\ HoldsAt\ (CurrActiv\ (a,\ s),\ t_2) \end{array} \tag{33}$$

Event $StartWrkf$ starts a workflow session. Next EC equation defines pre-condition of event $StartWrkf$; a workflow session may start if it has not already been started:

$$\begin{array}{l} \forall\ s\ :\ Session;\ t\ :\ Time\ | \\ \quad Happens\ (StartWrkf\ (s),\ t) \\ \quad \Rightarrow\ \neg\ HoldsAt\ (Started\ (s),\ t) \end{array} \tag{34}$$

Next EC equation says how event $StartWrkf$ initiates fluent $CurrActiv$; when a workflow starts, current activity becomes workflow's start activity (predicate $IsStartActiv$):

$$\begin{array}{l} \forall\ s\ :\ Sesstion;\ a\ :\ Activ;\ t\ :\ Time\ | \\ \quad IsStartActiv\ (a) \\ \quad \Rightarrow\ Initiates\ (StartWrkf\ (s), CurrActiv\ (a,\ s),\ t) \end{array} \tag{35}$$

3) Tasks, rôles and delegation: As said above, tasks are associated with activities. This association is defined through predicate $IsTaskOfActiv$; this predicate is defined for each workflow being described and analysed (it is defined in equation 49, below, for payment processing workflow).

To know whether some task can be executed in a workflow, EC model introduces predicate $IsTaskOfCurrActiv$, which indicates whether some task belongs to the current activity of some workflow session. Next EC equation defines this predicate; it says that some task belongs to current activity of some session if it is a task of session's current activity:

$$\begin{array}{l} \forall\ ta\ :\ Task; s\ :\ Sesstion;\ t\ :\ Time\ | \\ \quad HoldsAt\ (IsTaskOfCurrActiv\ (ta,\ s),\ t) \\ \quad \Leftrightarrow\ (\exists\ a\ :\ Activ)\ HoldsAt\ (CurrActiv\ (a,\ s),\ t) \\ \quad\quad \wedge\ IsTaskOfActiv\ (ta,\ a) \end{array} \tag{36}$$

Predicate $CanDo\ (u,\ t)$ indicates the rôles (r) that are allowed to execute workflow tasks (t). Certain users may delegate their rôles; predicate $MayDelegTo\ (u_1,\ u_2)$ says that some user ($u1$) is allowed to delegate his rôles to another user (u_2). Both $CanDo$ and $MayDelegTo$ are defined for each workflow being described and analysed; it is described below for payment processing workflow in equations 52 and 53.

Event $DelegsTo$ occurs whenever a rôle delegation takes place. Next EC equation defines this event's pre-condition; users delegate to others provided they are allowed to do so:

$$\begin{array}{l} \forall\ u_1, u_2\ :\ User;\ t\ :\ Time\ | \\ \quad Happens\ (DelegsTo\ (u_1,\ u_2),\ t) \\ \quad \Rightarrow\ MayDelegTo\ (u_1, u_2) \end{array} \tag{37}$$

Fluent $Delegated\ (u_1,\ u_2)$ says that some user (u_1) has delegated to another. Next EC equation says that event $DelegsTo$ initiates (sets to true) fluent $Delegated$ to enable system to keep track of rôle delegations:

$$\begin{array}{l} \forall\ u_1, u_2\ :\ User;\ t\ :\ Time\ | \\ \quad Initiates\ (DelegsTo\ (u_1,\ u_2), Delegated\ (u_1,\ u_2),\ t) \end{array} \tag{38}$$

Next EC equation defines initial condition of fluent $Delegated$; initially, no rôle delegations have taken place:

$$\begin{array}{l} \forall\ u_1,\ u_2\ :\ User\ | \\ \quad Initially\ (\neg\ Delegated\ (u_1,\ u_2)) \end{array} \tag{39}$$

To capture permissions related with delegations, next EC equation introduces predicate $CanExecAsDelegate$, which indicates whether some user can execute a task as delegate:

$$\begin{array}{l} \forall\ u\ :\ User;\ ta\ :\ Task;\ t\ :\ Time\ | \\ \quad HoldsAt\ (CanExecAsDelegate\ (u,\ ta),\ t) \\ \quad \Leftrightarrow\ \neg\ CanDo\ (u,\ ta) \\ \quad\quad \wedge\ ((\exists\ u_2\ :\ User)\ HoldsAt\ (Delegated\ (u_2,\ u),\ t) \\ \quad\quad \wedge\ CanDo\ (u_2,\ ta)) \end{array} \tag{40}$$

This predicate is used to define predicate $HasPerm$, which indicates whether some user has the required permissions to

execute some workflow task; this is true if user has required permissions to execute task with his rôle or if he has been delegated the rôle of someone else with required permissions:

$$\forall\, u : User;\ ta : Task;\ t : Time \mid HoldsAt\ (HasPerm\ (u,\ ta),\ t) \Leftrightarrow\ CanDo\ (u,\ ta) \vee HoldsAt\ (CanExecAsDelegate\ (u,\ ta),\ t) \quad (41)$$

4) Task execution: Event $ExecTask$ happens whenever some workflow task is executed. Next EC equation defines predicate $ExecutedTaskOfActiv$, which indicates whether some user executed a task of some activity in some workflow session:

$$\forall\, u : User;\ a : Activ;\ s : Session;\ t : Time \mid HoldsAt\ (ExecutedTaskOfActiv\ (u,\ a,\ s),\ t) \Leftrightarrow (\exists\, ta : Task;\ t_2 : Time)\ t_2 < t \wedge Happens\ (ExecTask\ (ta,\ u,\ s),\ t_2) \wedge HoldsAt\ (CurrActiv\ (a,\ s),\ t_2)) \quad (42)$$

Predicate $ExecutedTaskOfActiv$ is used to define predicate $BreachesSoD$, which indicates whether some user can breach SoD in some workflow session. $BreachesSoD$ is true whenever some user executed some task of some activity for which there is a SoD constraint with current activity in some workflow session; it defined as:

$$\forall\, u : User;\ s : Session;\ t : Time \mid HoldsAt\ (BreachesSoD\ (u,\ s),\ t) \Leftrightarrow (\exists\, a_1,\ a_2 : Activ)\ HoldsAt\ (CurrActiv\ (a_1, s), t) \wedge (SoD\ (a_1, a_2) \vee SoD\ (a_2, a_1)) \wedge HoldsAt\ (ExecutedTaskOfActiv\ (u, a_2, s), t)) \quad (43)$$

When event $ExecTask$ happens, current activity changes to be next activity in the workflow; this goes on until the workflow session finishes. There are several restrictions associated with execution of tasks in a workflow; these are modelled as event pre-conditions of event $ExecTask$: (a) the task belongs to the current activity (predicate $IsTaskOfCurrActiv$, equation 36), that (b) the user has the required permissions to execute the task (predicate $HasPerm$, equation 41), and that (c) the execution of the task by the user does not break separation of duties (predicate $BreachesSoD$, equation 43). This is defined in EC by the equation:

$$\forall\, u : User;\ ta : Task;\ s : Session;\ t : Time \mid Happens\ (ExecTask\ (ta,\ u,\ s),\ t) \Rightarrow HoldsAt\ (IsTaskOfCurrActiv\ (ta,\ s), t) \wedge HoldsAt\ (HasPerm\ (u,\ ta),\ t) \wedge \neg\ HoldsAt\ (BreachesSoD\ (u,\ s),\ t) \quad (44)$$

As said above, when a task is executed the current activity must change. This requires an EC equation defining a terminates predicate to set the current activity to false if there is a current activity. It also requires an initiates predicate to set the current activity to the next activity of the workflow. These are defined as:

$$\forall\, u : User;\ a : Activ;\ ta : Task;\ s : Session;\ t : Time \mid HoldsAt\ (CurrActiv\ (a,\ s),\ t) \Rightarrow Terminates\ (ExecTask\ (ta,\ u,\ s),\ CurrActiv(a,\ s),\ t) \quad (45)$$

$$\forall\, u : User;\ a_1,\ a_2 : Activ;\ ta : Task;\ s : Session;\ t : Time \mid HoldsAt\ (CurrActiv\ (a_1,\ s),\ t) \wedge OccursBefore\ (a_1,\ a_2)) \Rightarrow Initiates\ (ExecTask\ (ta,\ u,\ s),\ CurrActiv\ (a_2,\ s),\ t) \quad (46)$$

5) Payment processing workflow: EC equations above define infrastructure necessary to describe workflows with SoD constraints. The following EC equations actually define the payment processing system workflow of Fig. 3.

We start by defining the rôles of the workflow. Rôles are modelled as sub-sorts of sort $User$; and so we have $User$ sub-sorts $Clerk$ and $Manager$. Rôle administrator is modelled as the predicate $IsAdminOf$.

Next EC equation defines the tasks and activities of the workflow:

$$prepPay,\ approvePay1,\ approvePay2,\ FinPay : Activ \quad (47)$$

$$tPrepPay, tApprovePay, tRefusePay, tIssuePay, tVoidPay : Task \quad (48)$$

This says that the workflow activities are those identified in Fig. 3, prepare payment ($prepPay$), approve payment ($approvePay1$ and $approvePay2$), and finalise payment ($FinPay$), and that the tasks are also those of Fig. 3, prepare pay ($tPrepPay$), approve payment ($tApprovePay$), refuse payment ($tRefusePay$), issue payment ($tIssuePay$) and void payment ($tVoidPay$).

Next equation defines the relation that exists between tasks and activities by defining predicate $IsTaskOfActiv$:

$$\forall\, ta : Task; a : Activ \mid IsTaskOfActiv\ (ta,\ a) \Leftrightarrow (a = PrepPay \wedge ta = tPrepPay) \vee ((a = ApprovePay1 \vee a = ApprovePay2) \wedge (ta = tApprovePay \vee ta = tRefusePay)) \vee (a = IssueOrVoidPay \wedge (task = tIssuePay \vee task = tVoidPay)) \quad (49)$$

This says that prepare payment activity is made of prepare payment task, approve payment activities ($ApprovePay1$ and $ApprovePay2$) are composed of tasks *approve pay* and *refuse pay*, and that $FinPay$ activity is composed of tasks *issue pay* and *void pay*.

Next equation defines the $OccursBefore$ predicate. It Says that $Prepay$ must occur before $ApprovePay1$, which must occur before $approvePay2$, and that $approvePay2$ must occur before $issueOrVoidPay$:

$$\forall\, a_1,\ a_2 : Activ \mid OccursBefore\ (a_1, a_2) \Leftrightarrow (a_1 = PrepPay \wedge a_2 = ApprovePay1) \vee (a_1 = approvePay1 \wedge a_2 = approvePay2) \vee (a_1 = approvePay2 \wedge a_2 = issueOrVoidPay) \quad (50)$$

Next equation defines the SoD constraints of the payment processing workflow by defining predicate SoD. It says that there is a SoD constraint between activities $PrepPay$ and

$IssueOrVoidPay$, and between activities $ApprovePay1$ and $ApprovePay2$:

$$\begin{aligned}&\forall\, a_1,\ a_2\ :\ Activ \mid SoD\ (a_1,\ a_2)\\&\quad \Leftrightarrow\ (a_1\ =\ PrepPay\ \wedge\ a_2\ =\ IssueOrVoidPay)\\&\qquad \vee\ (a_1\ =\ ApprovePay1\ \wedge\ a_2\ =\ ApprovePay2)\end{aligned}\quad (51)$$

Next equation defines the permissions of workflow tasks by defining predicate $CanDo$. Equation says that mnagers can execute any task and that clerks can execute tasks prepare, issue and void payments:

$$\begin{aligned}&\forall\, u\ :\ User;\ ta\ :\ Task \mid CanDo\ (u,\ ta)\\&\quad \Leftrightarrow\ ((\exists\, ma\ :\ Manager)\ u\ =\ ma)\\&\qquad \vee\ ((\exists\, cl\ :\ Clerk)\ (ta\ =\ tPrepPay\\&\qquad\quad \vee\ ta\ =\ tIssuePay \vee ta\ =\ tVoidPay))\end{aligned}\quad (52)$$

Next EC equation defines the delegation rule of the payment processing workflow. It says that managers may delegate tasks to their administrators:

$$\begin{aligned}&\forall\, u_1,\ u_2\ :\ User \mid MayDelegTo\ (u_1,\ u_2)\\&\quad \Leftrightarrow\ ((\exists\, ma\ :\ Manager)\ u_1\ =\ ma\\&\qquad \wedge\ IsAdminOf\ (u_2,\ u_1))\end{aligned}\quad (53)$$

A payment is issued provided both managers approve it. Next EC equation defines predicate $PayApproved$, which defines what it means for a payment to be approved:

$$\begin{aligned}&\forall\, s\ :\ Session;\ t\ :\ Time \mid\\&\quad HoldsAt\ (PayApproved\ (s),\ t)\\&\quad \Leftrightarrow\ ((\exists\, u_1,\ u_2\ :\ User;\ t_2,\ t_3\ :\ Time;\ ta\ :\ Task)\\&\qquad u_1 \neq u_2 \wedge\ t_2\ <\ t \wedge t_3 < t\\&\qquad \wedge\ ta\ =\ tApprovePay\\&\qquad \wedge\ Happens\ (ExecTask\ (ta,\ u_1,\ s),\ t_2)\\&\qquad \wedge\ Happens\ (ExecTask\ (ta,\ u_2,\ s), t_3))\end{aligned}\quad (54)$$

This says that a payment is approved provided task $tApprovePay$ has been executed at two different time-points by two different users in context of a workflow session.

Next two EC equations define the constraints associated with tasks $tIssuePay$ and $tVoidPay$. They say that task $tIssuePay$ may be executed provided the payment has been approved, and that task $tVoidPay$ may be executed provided it has not been approved:

$$\begin{aligned}&\forall\, u\ :\ User;\ s\ :\ Session;\ t\ :\ Time \mid\\&\quad Happens\ (ExecTask\ (tIssuePay, u, s),\ t)\\&\qquad \Rightarrow\ HoldsAt\ (PayApproved\ (s),\ t)\end{aligned}\quad (55)$$

$$\begin{aligned}&\forall\, u\ :\ User;\ s\ :\ Session;\ t\ :\ Time \mid\\&\quad Happens\ (ExecTask\ (tVoidPay,\ u,\ s),\ t)\\&\qquad \Rightarrow\ \neg\ HoldsAt\ (PayApproved\ (s), t)\end{aligned}\quad (56)$$

This completes EC model of payment processing workflow requirements (table III). Next section analyses this model.

B. Model Analysis

Analysis is conducted in a configuration made of three managers, Bob, John and Martin, and three clerks Sam, Alice and Sue; Sue also works as an administrator for John. This is defined in EC as:

$$bob, john, martin : Manager \quad (57)$$

$$alice, sam, sue : Clerk \quad (58)$$

$$\begin{aligned}&\forall\, u_1,\ u_2\ :\ User \mid IsAdminOf\ (u_1,\ u_2)\\&\quad \Leftrightarrow u_1\ =\ sue \wedge u_2\ =\ john\end{aligned}\quad (59)$$

Analysis starts with a security violation goal to know if it is possible to reach a state where SoD is breached. Next EC equation defines what it means to breach SoD; that is, a user executed tasks belonging to activities constrained under SoD:

$$\begin{aligned}&\forall\, u\ :\ User;\ s\ :\ Session;\ t\ :\ Time \mid\\&\quad HoldsAt\ (BreachedSoD\ (u,\ s),\ t)\\&\quad \Leftrightarrow\ ((\exists\, a_1,\ a_2)\ a_1\ \neq\ a_2\\&\qquad \wedge\ HoldsAt\ (ExecutedTaskOfActiv\ (u,\ a_1,\ s),\ t)\\&\qquad \wedge\ HoldsAt\ (ExecutedTaskOfActiv\ (u,\ a_2,\ s),\ t)\\&\qquad \wedge\ (SoD\ (a_1,\ a_2)\ \vee\ SoD\ (a_2,\ a_1)))\end{aligned}\quad (60)$$

This predicate is used to formulate the goal:

$$\begin{aligned}&\exists\, u\ :\ User;\ s\ :\ Session;\ t\ :\ Time \mid\\&\quad HoldsAt\ (BreachedSoD\ (u,\ s),\ t)\end{aligned}\quad (AG4)$$

For this goal, *decreasoner* does not find any plans. This means that it is not possible to reach a state where SoD is breached. Again, one could argue that SoD is preserved and the system is secure, but it isn't so.

Analysis proceeds by investigating the suspicious space. Although a user is allowed to execute more than one task in some workflow session, this should not happen very often and is suspicious. The idea is to explore this somehow suspicious or abnormal situation in order to find clues that help in finding security vulnerabilities. First, we define a predicate describing the suspicious system condition of having a user executing two tasks in a workflow session:

$$\begin{aligned}&\forall\, u : User; s : Session; t : Time \mid\\&\quad HoldsAt\ (ExecutedTwoTasks\ (u, s), t)\\&\quad \Leftrightarrow \exists\, ta_1, ta_2 : Task; t_2, t_3 : Time \mid\\&\qquad \wedge\ Happens\ (ExecTask\ (ta_1, u, s), t_2)\\&\qquad \wedge\ Happens\ (ExecTask\ (ta_2, u, s), t_3)\\&\qquad \wedge\ ta_1 \neq ta_2 \wedge t_2 \leq t \wedge t_3 \leq t\end{aligned}\quad (61)$$

Since we are interested in scenarios involving complete workflow runs, next EC equation defines predicate $IsWrkfComplete$, which says whether some workflow session is complete or not:

$$\begin{aligned}&\forall\, s : Session; t : Time \mid\\&\quad HoldsAt\ (IsWrkfComplete\ (s),\ t)\\&\quad \Leftrightarrow\ HoldsAt\ (Started\ (s),\ t)\\&\qquad \wedge \neg\ ((\exists\, a\ :\ Activ)\ HoldsAt\ (CurrActiv(a,\ s),\ t))\end{aligned}\quad (62)$$

From these two predicates, we define the suspicious goal by describing states where some user executes two different tasks in some workflow run:

$$\begin{aligned}&\exists\, u : User; s : Session; t : Time \mid\\&\quad HoldsAt\ (IsWrkfComplete\ (s), t)\\&\quad \wedge\ HoldsAt\ (ExecutedTwoTasks\ (u, s), t)\end{aligned}\quad (AG5)$$

For this goal, *decreasoner* generates interesting plans. We have scenarios where a manager prepares the payment and then approves it, or that he approves and then issues the payment. This happens because managers may act as clerks and there is no SoD constraint between tasks that managers do and clerks do. As this may give a fraud opportunity, it is important to clarify the requirements regarding this issue.

R6'	Managers can delegate authority on approval of refunds to one of their administrators, but when administrators executes such tasks system should consider that they have been executed on behalf of manager and are is manager had executed them.
R7	The same person may perform tasks as either manager or clerk, but not both, in any workflow session.

TABLE IV
REQUIREMENTS RULING THE PROCESSING OF TAX REFUNDS THAT EMERGED AFTER ANALYSIS.

C. Clarifying the requirements

Clarification of the issue exposed by the analysis results in new requirement $R7$ (table IV), which says that a person can execute tasks under at most one rôle in any workflow session. To take this new requirement into account, EC model introduces predicate $RolesRequiredDiffer$, which says which workflow tasks require different rôles to execute them. This predicate is defined for workflow of payment processing as:

$$\begin{aligned} &\forall\, ta_1,\ ta_2\ :\ Task \mid \\ &\quad RolesRequiredDiffer\ (ta_1,\ ta_2) \\ &\quad \Leftrightarrow\ (ta_1\ =\ tPrepPay \vee\ ta_1\ =\ tIssuePay \\ &\qquad \vee\ ta_1\ =\ tVoidPay) \\ &\qquad \wedge\ (ta_2\ =\ tApprovePay\ \vee\ ta_2\ =\ tRefusePay) \quad (63) \end{aligned}$$

This predicate says that the roles required for tasks $tPrepPay$, $tIssuePay$ and $tVoidPay$ is different for those of tasks $tApprovePay$ and $tRefusePay$.

Predicate $RolesRequiredDiffer$ is used to state the required requirement by constraining event $ExecTask$. Next EC equation describes this constraint by saying that if some user executes two different tasks then roles required to execute them must not differ:

$$\begin{aligned} &\forall\, u\ :\ User;\ s\ :\ Session;\ ta_1,\ ta_2\ :\ Task; \\ &\qquad t_1,\ t_2\ :\ Time \mid \\ &\quad Happens\ (ExecTask\ (ta_1,\ u,\ s),\ t_1) \\ &\quad \wedge\ Happens\ (ExecTask\ (ta_2,\ u,\ s),\ t_2) \\ &\quad \wedge\ ta_1\ \neq\ ta_2 \\ &\quad \Rightarrow\ \neg\ RolesRequiredDiffer\ (ta_1,\ ta_2) \qquad (64) \end{aligned}$$

D. Re-Analysing the model

After the fix, the vulnerability identified above that that could give a fraud opportunity is no longer allowed. We resubmit the analysis goal above and decreasoner no longer generates scenarios with those possible fraudulent behaviours.

Analysis turns to delegation, which is known to generate security vulnerabilities. Someone executing a task on behalf of someone is legal but suspicious and deserves investigation. Again, the idea is too look in behaviours involving delegation for clues on possible system vulnerabilities. We introduce a predicate to say whether some user executed some task as delegate; next two EC equations define this predicate:

$$\begin{aligned} &\forall\, u : User; ta : Task; s : Session; t : Time \mid \\ &\quad HoldsAt\ (DelegExecutedFor\ (u, ta, s), t) \\ &\quad \Leftrightarrow\ ((\exists\, t_2\ :\ Time)\ t_2 < t \\ &\qquad \wedge\ DelegExecutedForAt\ (u,\ ta,\ s,\ t_2)) \qquad (65) \\ &\forall\, u\ :\ User;\ ta :\ Task;\ s\ :\ Session;\ t\ :\ Time \mid \\ &\quad DelegExecutedForAt\ (u,\ ta,\ s,\ t) \\ &\quad \Leftrightarrow\ ((\exists\, u_2\ :\ User) \\ &\qquad Happens\ (ExecTask\ (ta,\ u_2, s), t) \\ &\qquad \wedge\ HoldsAt\ (Delegated\ (u,\ u_2),\ t) \\ &\qquad \wedge\ HoldsAt\ (CanExecAsDelegate\ (u_2,\ ta),\ t)) \quad (66) \end{aligned}$$

Above, predicate $DelegExecutedFor$ says whether some task was executed by delegate for some user. This is defined from predicate $DelegExecutedForAt$, which says whether some user executed the task as delegate.

Goal is defined from $DelegateExecutedFor$ by describing states of complete workflow runs where someone executes a task on behalf of someone else. This results in the goal:

$$\begin{aligned} &\exists\ u : User; ta, : Task; s : Session; t : Time \mid \\ &\quad HoldsAt\ (IsWrkfComplete\ (s),\ t) \\ &\quad \wedge\ HoldsAt\ (DelegExecutedFor\ (u,\ ta,\ s),\ t) \quad (AG6) \end{aligned}$$

Plans generated by *decreasoner* result in what is normally expected under delegation (someone executes a task on behalf of someone else), but they also result in plans that may be possible frauds: a delegate approves a payment on behalf of the manager and the same manager also approves the same payment.

E. Clarifying and elaborating the requirements

From this, we elaborate the requirements, and we get $R6'$ (table IV) an elaboration of requirement $R6$. This says that system must consider tasks executed by administrators acting as delegates as if they had been executed by the managers themselves.

To accommodate this new requirement, we introduce the predicate $ExecutedTask$, which indicates whether some user executed some task, either directly or indirectly through a delegate. This is defined as:

$$\begin{aligned} &\forall\, u\ :\ User;\ ta\ :\ Task;\ s\ :\ Session;\ t\ :\ Time \mid \\ &\quad HoldsAt\ (ExecutedTask\ (u,\ ta,\ s),\ t) \\ &\quad \Leftrightarrow\ (\exists\, t_2\ :\ Time)\ t_2\ <\ t \\ &\qquad \wedge\ ExecutedTaskAt\ (u,\ ta,\ s,\ t_2)) \qquad (67) \\ &\forall\, u\ :\ User;\ ta\ :\ Task;\ s\ :\ Session;\ t\ :\ Time \mid \\ &\quad ExecutedTaskAt\ (u,\ ta,\ s,\ t) \\ &\quad \Leftrightarrow\ Happens\ (ExecTask\ (ta,\ u,\ s),\ t) \\ &\qquad \vee\ DelegExecutedForAt\ (u, ta, s, t) \qquad (68) \end{aligned}$$

Predicate $ExecutedTask$ defined above is used to redefine predicate 'ExecutedTaskOfActiv' equation 42). New formulation of this predicate is defined by EC equation:

$$\begin{aligned} &\forall\, u\ :\ User;\ a\ :\ Activ;\ s\ :\ Session;\ t : Time \mid \\ &\quad HoldsAt\ (ExecutedTaskOfActiv\ (u,\ a,\ s),\ t) \\ &\quad \Leftrightarrow\ (\exists\, ta\ :\ Task;\ t_2\ :\ Time)\ t_2\ <\ t \\ &\qquad \wedge\ HoldsAt\ (ExecutedTask\ (u, ta, s), t_2) \\ &\qquad \wedge\ HoldsAt\ (CurrActiv\ (a,\ s),\ t_2)) \qquad (42') \end{aligned}$$

In this revised EC model, the possible fraudulent behaviour identified above is no longer allowed.

VI. EXPERIMENTAL RESULTS

In both experiments presented above, formal analysis verified a straightforward safety security property, which could mislead analysts in concluding that an unsecure state would not be reached in the modelled system. However, suspicion-based analysis demonstrated that the modelled systems were in fact not secure.

Section IV analyses a simple medical information system that includes a confidentiality requirement. Following the

Case Study	Anaysis Goal	Time
SMIS	Security Violation (AG1)	3.6s
SMIS	Suspicion goal 1 (AG2)	4.9s
SMIS	Suspicion goal 2 (AG3)	12.6s
SMIS	Security Violation (AG1), after fix	14.s
SMIS	Suspicion goal 1 (AG2), after fix	15.8s
SMIS	Suspicion goal 2 (AG3), after fix	21.1s
Workflow	Security violation (AG4)	235.9s (3.9m)
Workflow	Suspicion goal 1 (AG5)	600.7s (10.0m)
Workflow	Suspicion goal 1, after fix (AG5)	476.9s (7.9m)
Workflow	Suspicion goal 2 (AG6)	619.2s (10.3m)
Workflow	Suspicion goal 2 (AG6), after fix	611.11s (10.2m)

TABLE V
RUNNING TIMES FOR ANALYSIS OF EC MODELS WITH *decreasoner*. TABLE INDICATES CASE STUDY, ANALYSIS GOAL AND TIME TAKEN TO GENERATE PLANS.

traditional route of safety analysis, it was not possible to find ways in which confidentiality would be compromised: without a valid credential it would not be possible to obtain the patient's medical data. Analysis based on suspicion then uncovered a security vulnerability (or loophole) that would enable a malicious user to obtain the required credentials in a non-legal way.

Section V analyses a business process whose security requirements included two integrity requirements enforced through SoD. Again, SoD could be breached, but not in an obvious way. Following the traditional safety analysis route, we checked that it was not possible that the same user would be able to execute two different tasks protected by SoD. Analysis-based on suspicion, however, uncovered several problems: the same user could execute different tasks in a workflow session under different roles, and delegation introduced a *loophole* that would enable users to indirectly breach SoD.

Table V presents the running times of the formal analysis based on planning with *decreasoner*[3]. For each case study, it shows how much time it took to carry out the analysis for each analysis goal. We can see that the analysis of the workflow model of section V takes substantially longer than SMIS model because it is more complex.

VII. DISCUSSION

This paper proposes *suspicion* as a concept driving the analysis of security requirements. Through experiments, it argues that, from a practical point of view, in security the interesting question is not only to verify the in-existence of a state compromising some security property (safety), but also to look for what is suspicious in order to find security vulnerabilities and threats. The experiments conducted in the context of the EC temporal logic, planning and the *decreasoner* tool. They demonstrate the usefulness of suspicion. The traditional safety analysis route, which checks whether some security property is violated, would not expose any security issues; this can mislead analysts in concluding that the system being analysed is secure. Analysis based on suspicion uncovered security vulnerabilities and threats; such findings drive elaboration of the requirements.

One of the advantages of the approach presented here is that security threats can be derived directly from a model of requirements. The analysis that does not need prior knowledge about possible attacks to the modelled system, and so no need to enrich the model with attacker or intruder models. Instead, using the suspicion-based approach proposed here, it is possible to derive threats from a requirements model by posing the model questions based on what is suspicious.

All the vulnerabilities exposed by suspicion-based analysis are related with delegation or passing of capabilities, which are known in security as non-interference properties [18]. The formulation and verification of such properties have proved to be far from trivial [4]. The analysis conducted in this paper confirms that delegation can be trick and hard to get right. Suspicion-based analysis helped in identifying security problems with delegation, and in elaborating the security requirements in order to eliminate such problems. The paper also shows that proof of a straightforward safety property related with security does not deem a system secure. Often, as shown in this paper, the *secure* question is more involved and requires more in-depth knowledge of the requirements. As this paper shows, it can be more revealing to analyse the system in order to explore the consequences of the requirements, which leads to a better understanding of the security needs and issues of the modelled system, rather than trying to prove that a system is secure. For the delegation-related issues explored in these two experiments, such proofs are far from trivial.

The approach presented here generates automatically possible scenarios of misuse (threats) from a statement describing some security violation or suspicious condition (the goal). This provides a flexible and illuminating scheme to the analysis of security requirements. Rather then finding themselves possible threats, analysts describe instead what would constitute a violation of security or a suspicious system condition. Analysis goals require an understanding of the requirements domain, and should be described with some security asset in mind.

The security vulnerabilities exposed by the analysis illustrate the sort of vulnerabilities that attackers exploit to intrude into today's software systems. The vulnerabilities identified in the health-care system give insiders the opportunity to perpetrates attacks on the system; the *insider threat* has been identified as one of the main sources of attacks in the medical domain [15]. Once a source of threats is identified in the requirements, two decisions can be made: (a) introduce further constraints by elaborating the requirements so that the source of threats is eliminated, or (b) do nothing in terms of requirements, but take the problem into account in terms of run-time intrusion and threat detection which then has to judge whether some uses of the system are malicious or not. The latter must be considered because it is not possible to eliminate all possible security threats; doing so could result in a system design that is rigid and over-constrained. The approach presented here enables the detection of threats or vulnerabilities in the system, which also constitutes valuable information for run-time intrusion and threat detection.

The experiments conducted here confirm the importance

[3]Model analysis carried out on an Apple *iMac*, with a 2.93 Ghz Intel Core 2 Duo processor and 4GB memory RAM.

of modelling and analysing security together with system requirements. Both case studies show how a functionality of the system, delegation, have a serious impact on security and how it was necessary to further elicit and elaborate the requirements in order to eliminate threats.

Formal security analysis with tool support is capable of exposing many unexpected situations, providing a level of assurance not guaranteed by semi-formal approaches. The drawback of the analysis with *decreasoner* lies in the efficiency of the tool: as models get more complex, the solution to analysis problems take more time to the point that the analysis becomes unpractical. The workflow model of section V is a simplification of an earlier model to enable practical analysis. As usual, the secret is in getting the right abstraction in order to analyse the property of interest.

It is interesting to comment on the usability of the approach presented here. The process of defining analysis goals may require domain knowledge and skill in building and analysing models. However, the process can be partially or fully automated by following the pattern-based approach proposed of [19], which uses the *Formal Template Language* [20], [21] to represent patterns of EC models together with their associated security monitoring goals. [19] defines templates security violation goals, but patterns of suspicious goals can also be defined if we know in advance what can arouse suspicion. In our experiments, *delegation* was the focus of our suspicious goals; this is something that can be known in advance and captured using patterns. Following [19], we can have goals that capture what is known to arouse suspicion; actual suspicious goals would then be automatically generated from templates. [19] also uses UML models to enable intuitive requirements modelling; the same approach can also be used to enhance the usability of the approach proposed here.

VIII. Related Work

This paper is a revised and extended version of the work presented in [22]. It shows in detail the EC models that are used to illustrate the analysis based on suspicion with EC, and provides a more in-depth discussion on the verification of security properties, such as the one explored in the paper.

The results of this paper argue against ironclad proofs of security and how one needs to be careful in interpreting formal demonstrations of security properties. This theme is not new; in [23], McLean refutes what used to be a widely held belief: that the security model of Bell and LaPadula [24] and its *basic security theorem* would capture the essence of security and that implementations following it would be secure[4]. This refutation was done by stating a similar theorem for a model that is clearly not secure. Both experiments of this paper demonstrated that the modelled system would not breach a straightforward safety property of security (confidentiality and separation of duty), and how that could mislead analysts in concluding that the system was secure. However, more flexible means of analysis exposed vulnerabilities showing that the systems being analysed were in fact not secure; the paper suggests analysis lead by what is suspicious in order to find security vulnerabilities in models of requirements.

The case studies used in this paper illustrate behaviours that are usually tricky to be verified using *safety* or *liveness* arguments. Most vulnerabilities identified in sections IV and V are related to delegation, which has traditionally proved to be tricky; [18] introduces *non-interferance*, a confidentiality policy that deals with delegation-based functionality. The verification of non-interference is far from trivial, and requires a simplified model of a system that is difficult to obtain when modelling requirements. Such behaviours or properties have also been termed *possibilistic* [7] and it is known that certain security policies cannot be expressed using safety or liveness properties represented as sets of traces [7], [8]. It is also known that, in general, non-interferance policies cannot be expressed as safety or liveness properties [8]. In [8], the authors propose *hyperproperties*, which are defined as sets of properties (sets of sets of traces), to represent what is not normally captured with traditional properties. This paper provides a pragmatic approach to formally find security vulnerabilities involving delegation in models of security requirements.

There has been substantial interest on security requirements threat analysis [25], [3], [26], [27]. In [26], [27], specifiers need to explicitly identify scenarios or use-cases of abuse and misuse; here, such scenarios are generated automatically from a description of suspicious states (the goal).

[25] proposes a method based on the more flexible abuse frames, specifying undesirable phenomena that the system should prevent from happening. The approach presented here enables the specification of such undesirable phenomena as goals (here called security violation goals). However, it does not only consider what should not happen, but also considers specification of flexible suspicious conditions that (as shown here) have the potential of exposing unnown threats.

[3] proposes a goal-based method that is similar to the approach presented here. Security goals, such as confidentiality, integrity and availability, are negated to obtain goals that specify what should not happen (our security violation goals). Then, these negations are refined to obtain more flexible goals. The approach presented here is more flexible in that it does not only allow specification of goals that come from negation and refinement security goals, but also leaves the specifier the flexibility of defining what constitutes a suspicious condition. Since it is based on tool support, the specifier can use the feedback coming from the tool to either refine existing analysis goals or specify entirely new ones. Essentially, the work presented here is complementary to the body of work on security requirements threat analysis. It explores automated formal analysis (missing in the works above) and provides experimental evidence to the usefulness and effectiveness of security threat analysis.

Suspicion is ubiquitous in *intrusion detection* [9]. Anomaly-based approaches to intrusion detection [9] are so called because the search for intrusions is driven by abnormal (or suspicious) behaviour patterns of system use. [28] proposes the inclusion of suspicion as a concept driving the models of *misuse-based* intrusion detection; it proposes models based on suspicious activities that may lead to an attack, as opposed

[4]This was not claimed by the authors of [24], but others that interpreted their work believed that that was the case.

to models based on actual attacks. Instead, the approach presented here detects abuse by identifying suspicious states in a model of normal system behaviour.

The approach presented here emerges from its preceding work on threat detection [19]. [19] uses an EC model of requirements and planning to find threats at run-time. In [19], however, the goals used to detect threats are more rigid than the ones used here; they say with absolute certainty whether there is an attack or not when the goal is satisfied. In the approach presented here, there is no certainty of attack if a suspicious goal is satisfied, the plans that reach the goal just give us threats (possible attacks). It would be possible to incorporate our approach based on suspicion as a strategy in looking for threats at run-time. Then, the probabilistic component of such a system (like the one of [19]) would try to assign a probability to the computed threats. Here we use suspicion to look for threats in requirements to avoid systems with security vulnerabilities.

The approach presented here analyses requirements automatically using a tool based on SAT-solving. The advantage of this method with respect to other approaches based on theorem-proving [29], [21] is that the reasoning is automatic, avoiding the need for user-intervation as it is usually the case with theorem proving. The disadvantages are that only a portion of the state space is analysed, and that the models that can be handled need to be small; this problem can be mitigated by using abstraction to produce smaller models enabling analysis of property of interest. Another disadvantage to the work in [29], [21] is that there is no visual description of requirements and properties to check; the user needs to be an expert in the formal language (here EC).

IX. Conclusions

This paper proposes a practical approach to the formal analysis of security requirements based on planning guided by the concept of suspicion. One of the advantages of the approach presented here is that threats can be detected directly from a requirements model, where no prior knowledge about possible attacks is needed to perform the analysis. Instead, the analysis derives threats automatically by posing the model questions based on what is suspicious. The approach was illustrated using the EC and the *decreasoner tool* by performing two experiments: one involving a simple health-care system with a confidentiality requirement and another a business system with an integrity requirement enforced through SoD. It showed that the more obvious way of analysing security, by doing the traditional safety verification would not give any useful results: following this path analysts could be mislead in concluding that an unsecure state could not be reached. However, it was through more flexible analysis based on suspicion that we could obtain useful results exposing subtle security vulnerabilities.

The main contributions of this paper are: (a) the proposal of suspicion as a driving concept in the analysis of security requirements, and (b) the experimental confirmation that, from a practical point of view, it is important to use flexible criteria for the security analysis in order to find vulnerabilities in system requirements (suspicion was proposed as basis for such criteria). The paper also provides experimental evidence to certain claims made in the security requirements literature: (a) it confirmed that it is important to model security requirements together with other functional requirements because functionality impacts on security; (c) it confirmed the existence of security relevant phenomena that is hard or impossible to capture as safety or liveness properties; and (d) demonstrated the importance of formality and tool support and usefulness of automated reachability analysis of requirements.

References

[1] P. T. Devanbu and S. Stubblebine, “Software engineering for security: A roadmap,” in *The Future of Software Engineering*. ACM, 2000, pp. 227–239.

[2] B. W. Boehm, “Software engineering,” *IEEE Transactions on Computers*, pp. 1266–1241, 1976.

[3] A. van Lamsweerde, “Elaborating security requirements by construction of intentional anti-models,” in *Proc. ICSE’04*, 2004, pp. 148–157.

[4] J. Rushby, “Security requireemnts specifications: How and what? (extended abstract),” in *Symp. on Requirements Engineering for information security*, 2001.

[5] L. Lamport, “Proving the correctness of multiprocess programs,” *IEEE Trans. on Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977.

[6] B. Alpern and F. B. Schneider, “Recognizing safety and liveness,” *Distributed Computing*, vol. 2, pp. 117–126, 1987.

[7] J. McLean, “A general theory of composition for a class of “possibilistic” properties,” *IEEE Trans. on Software Engineering*, vol. 22, no. 1, pp. 53–66, 1996.

[8] M. R. Clarkson and F. B. Schneider, “Hyperproperties,” in *Computer Security Foundations Symposium*. IEEE, 2008.

[9] D. Denning, “An intrusion detection model,” *IEEE Trans. on Software Engineering*, vol. 13, no. 2, pp. 222–232, 1987.

[10] M. Shanahan, “The event calculus explained,” in *Artificial Intelligence Today*, ser. LNCS. Springer, 1999, vol. 1600, pp. 409–430.

[11] E. T. Mueller, “Automating commonsense reasoning using the event calculus,” *Communications of the ACM*, vol. 52, no. 1, pp. 113–117, 2009.

[12] J. Allen, J. Hendler, and A. Tate, Eds., *Readings in planning*. Morgan Kaufmann, 1990.

[13] E. T. Muller, “Event calculus reasoning through satisfiability,” *Journal of Logic and Computation*, vol. 14, no. 5, pp. 703–730, 2004.

[14] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 1999.

[15] R. J. Anderson, “A security policy model for clinical information systems,” in *Proc. of SP ’96*. IEEE, 1996.

[16] D. D. Clark and D. R. Wilson, “A comparison of commercial and military computer security policies,” in *Proc. IEEE Symp. Research in Security and Privacy*, 1987, pp. 184–194.

[17] M. J. Nash and K. R. Poland, “Some conundrums concerning separation of duty,” in *Proc. IEEE Symp. Research in Security and Privacy*, 1990, pp. 201–207.

[18] J. A. Goguen and J. Mesenguer, “Security policies and security models,” in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[19] N. Amálio and G. Spanoudakis, “From monitoring templates to security monitoring and threat detection,” in *Proc. of SECURWARE ’08*. IEEE, 2008, pp. 185–192.

[20] N. Amálio, S. Stepney, and F. Polack, “A formal template language enabling meta-proof,” in *FM 2006*, ser. LNCS, vol. 4085. Springer, 2006, pp. 252–267.

[21] N. Amálio, “Generative frameworks for rigorous model-driven development,” Ph.D. dissertation, Dept. Computer Science, Univ. of York, 2007.

[22] ——, “Suspicion-driven formal analysis of security requirements,” in *SECURWARE’2009*. IEEE, 2009, pp. 217–223.

[23] J. McLean, “A comment on the “basic security theorem” of bell and lapadula,” *Information Processing Letters*, vol. 20, pp. 67–70, 1985.

[24] D. E. Bell and L. J. Padula, “Secure computer systems: a mathematical model,” Mitre Corporation, Bedford, MA, Tech. Rep. MTR-2547 Vol. II, 1996.

[25] L. Lin, B. Nuseibeh, D. Ince, and M. Jackson, “Using abuse frames to bound the scope of security problems,” in *Proc. RE ’04*. IEEE, 2004, pp. 354–355.

[26] I. Alexander, "Misuse cases: Use cases with hostile intent," *IEEE Software*, vol. 20, no. 1, pp. 58–66, 2003.
[27] J. McDermott and C. Fox, "Using abuse case models for security requirements analysis," in *Annual computer security applications conference*. IEEE, 1999.
[28] T. Hollebeek and R. Waltzman, "The role of suspicion in model-based intrusion detection," in *Proc. of NSPW '04*. ACM, 2004, pp. 87–94.
[29] N. Amálio, S. Stepney, and F. Polack, "Formal proof from UML models," in *Proc. ICFEM 2004*, ser. LNCS, vol. 3308. Springer, 2004, pp. 418–433.

APPENDIX

A. Sample outputs of decreasoner

This appendix presents sample outputs generated by the *decreasoner* tool, while carrying out the model analysis presented in this paper.

1) Security Violation Goal (AG1): The output generated by decreasoner for the security violation goal (AG1) of section IV-B is:

```
no models found
```

This means that decreasoner could not find any solutions for the analysis goal.

2) Suspicion Goal 1 (AG2): The following output of decreasoner shows one sample solution for the security violation goal (AG1) of section IV-B is:

```
---
model 1:
0
CanAccessMD(Jones, Anderson).
IsDoctorOf(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 0).
1
+CredentialMD(Jones, Anderson, 0).
+HasValidCredential(Jones, Anderson).
Happens(SetDoctorOnLeave(Smith, Jones), 1).
2
+OnLeave(Jones).
Happens(SetSubstituteDoctor(Smith, Jones,
Smith), 2).
3
+CanAccessMD(Smith, Anderson).
+IsSubstituteDoctor(Smith, Jones).
Happens(AuthoriseAccess(Smith, Anderson), 3).
4
-HasValidCredential(Jones, Anderson).
+CredentialMD(Smith, Anderson, 3).
+HasValidCredential(Smith, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 4).
5
+CredentialMD(Jones, Anderson, 4).
+HasValidCredential(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 5).
6
+CredentialMD(Jones, Anderson, 5).
Happens(GetMD(Smith, Anderson), 6).
7
-HasValidCredential(Smith, Anderson).
+ExposedToAt(Smith, Anderson, 6).
+GoalSatisfied().
```

The sample above says that event $AuthoriseAccess$ $(Jones,\ Anderson)$ happens at timepoint 0, and that $SetDoctorOnLeave$ $(Smith,\ Jones)$ happens at timepoint 1. The analysis goal is satisfied by event $GetMD(Smith, Anderson)$ that happens at timepoint 6.

The remaining 4 sample solutions generated by decreasoner for analysis goal AG2 are as follows:

```
---
model 2:
0
CanAccessMD(Jones, Anderson).
IsDoctorOf(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 0).
1
+CredentialMD(Jones, Anderson, 0).
+HasValidCredential(Jones, Anderson).
Happens(SetDoctorOnLeave(Smith, Jones), 1).
2
+OnLeave(Jones).
Happens(SetSubstituteDoctor(Jones, Jones,
Smith), 2).
3
+CanAccessMD(Smith, Anderson).
+IsSubstituteDoctor(Smith, Jones).
Happens(AuthoriseAccess(Smith, Anderson), 3).
4
-HasValidCredential(Jones, Anderson).
+CredentialMD(Smith, Anderson, 3).
+HasValidCredential(Smith, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 4).
5
+CredentialMD(Jones, Anderson, 4).
+HasValidCredential(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 5).
6
+CredentialMD(Jones, Anderson, 5).
Happens(GetMD(Smith, Anderson), 6).
7
-HasValidCredential(Smith, Anderson).
+ExposedToAt(Smith, Anderson, 6).
+GoalSatisfied().
---
model 3:
0
CanAccessMD(Jones, Anderson).
IsDoctorOf(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 0).
1
+CredentialMD(Jones, Anderson, 0).
+HasValidCredential(Jones, Anderson).
Happens(SetDoctorOnLeave(Jones, Jones), 1).
2
+OnLeave(Jones).
Happens(SetSubstituteDoctor(Jones, Jones,
Smith), 2).
3
+CanAccessMD(Smith, Anderson).
+IsSubstituteDoctor(Smith, Jones).
Happens(AuthoriseAccess(Smith, Anderson), 3).
4
-HasValidCredential(Jones, Anderson).
+CredentialMD(Smith, Anderson, 3).
+HasValidCredential(Smith, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 4).
5
+CredentialMD(Jones, Anderson, 4).
+HasValidCredential(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson),
5).
6
+CredentialMD(Jones, Anderson, 5).
Happens(GetMD(Smith, Anderson), 6).
7
```

```
−HasValidCredential(Smith, Anderson).
+ExposedToAt(Smith, Anderson, 6).
+GoalSatisfied().
———
model 4:
0
CanAccessMD(Jones, Anderson).
IsDoctorOf(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 0).
1
+CredentialMD(Jones, Anderson, 0).
+HasValidCredential(Jones, Anderson).
Happens(SetDoctorOnLeave(Jones, Jones), 1).
2
+OnLeave(Jones).
Happens(SetSubstituteDoctor(Smith, Jones,
Smith), 2).
3
+CanAccessMD(Smith, Anderson).
+IsSubstituteDoctor(Smith, Jones).
Happens(AuthoriseAccess(Smith, Anderson), 3).
4
−HasValidCredential(Jones, Anderson).
+CredentialMD(Smith, Anderson, 3).
+HasValidCredential(Smith, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 4).
5
+CredentialMD(Jones, Anderson, 4).
+HasValidCredential(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 5).
6
+CredentialMD(Jones, Anderson, 5).
Happens(GetMD(Smith, Anderson), 6).
7
−HasValidCredential(Smith, Anderson).
+ExposedToAt(Smith, Anderson, 6).
+GoalSatisfied().
———
model 5:
0
CanAccessMD(Jones, Anderson).
IsDoctorOf(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 0).
1
+CredentialMD(Jones, Anderson, 0).
+HasValidCredential(Jones, Anderson).
Happens(SetSubstituteDoctor(Jones, Jones,
Smith), 1).
2
+IsSubstituteDoctor(Smith, Jones).
Happens(SetDoctorOnLeave(Jones, Jones), 2).
3
+CanAccessMD(Smith, Anderson).
+OnLeave(Jones).
Happens(AuthoriseAccess(Smith, Anderson), 3).
4
−HasValidCredential(Jones, Anderson).
+CredentialMD(Smith, Anderson, 3).
+HasValidCredential(Smith, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 4).
5
+CredentialMD(Jones, Anderson, 4).
+HasValidCredential(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 5).
6
+CredentialMD(Jones, Anderson, 5).
Happens(GetMD(Smith, Anderson), 6).
7
−HasValidCredential(Smith, Anderson).
+ExposedToAt(Smith, Anderson, 6).
+GoalSatisfied().
```