Reasoning About Confidentiality at Requirements Engineering Time

Renaud De Landtsheer and Axel van Lamsweerde
Département d'Ingénierie Informatique, Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
{rdl, avl}@info.ucl.ac.be

ABSTRACT

Growing attention is being paid to application security at requirements engineering time. Confidentiality is a particular subclass of security concerns that requires sensitive information to never be disclosed to unauthorized agents. Disclosure refers to undesired knowledge states of such agents. In previous work we have extended our requirements specification framework with epistemic constructs for capturing what agents may or may not know about the application. Roughly, an agent knows some property if the latter is found in the agent's memory.

This paper makes the semantics of such constructs further precise through a formal model of how sensitive information may appear or disappear in an agent's memory. Based on this extended framework, a catalog of specification patterns is proposed to codify families of confidentiality requirements. A proof-of-concept tool is presented for early checking of requirements models against such confidentiality patterns. In case of violation, the counterexample scenarios generated by the tool show how an unauthorized agent may acquire confidential knowledge. Counter-measures should then be devised to produce further confidentiality requirements.

Categories & Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification - *methodologies, languages, tools*.

General Terms

Languages, Verification, Security, Design.

Keywords

Security requirements, reasoning about confidentiality, bounded model checking, specification patterns.

1. INTRODUCTION

Software applications are increasingly ubiquitous, heterogeneous, mission-critical and vulnerable to unintentional or intentional security incidents [18, 3]. Security is therefore among the non-functional requirements taken the most seriously nowadays.

Our interest is in modeling, specifying, and analyzing security-related requirements and designs [21, 22]. Rather than considering security at the crypto, protocol or system/language levels, as usually done in the literature, we address security concerns at the *application level* where the state of the art is much more limited [31, 30]. At this level, services such as, e.g., web-based banking operations must implement application-specific security requirements in terms of primitives from the lower levels. Typical application-specific security breaches would allow malicious users of a web banking application to know account numbers together with their associated PIN code, or malicious users of an e-commerce application to get some item without having paid for it.

The paper's focus is on *confidentiality* requirements. *Confidentiality* requires that sensitive information be not disclosed to unauthorized recipients [18]. It thus imposes restrictions on what agents may or may not *know* about the application over time. *Agents* are active components forming the system. They can be humans, devices, legacy software, etc.

Information disclosure can be unintentional or intentional. In the former case, confidential knowledge is inadvertently disclosed to some unauthorized agent. In the latter case, the agent gets to know confidential information by proactive exploitation of unprotected data and system knowledge through deductive inferences, calculations, or malicious behaviors. The agent does so in order to satisfy its underlying anti-goals [22].

This paper proposes an extended framework for formally specifying and reasoning about confidentiality requirements in the early phases of software development. Our extension makes it possible to automate checks of requirements models against confidentiality claims and discover confidentiality violations at requirements engineering time. Counter-measures must then be found to yield new confidentiality requirements.

We make no distinction here between unintentional and intentional disclosure. The term "*unauthorized agent*" (UA) will therefore be used to refer to an attacker or an agent exposed to unintentional disclosure.

Our framework is based on logics for reasoning about knowledge [9]. It makes the operational semantics of earlier epistemic constructs, used in [19, 22], more precise through axioms defining how sensitive knowledge may get in and out an agent's memory. The tool is built on top of a constraint solver [29, 6] according to

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

principles borrowed from bounded model checking [1]. It checks instantiated requirements models against confidentiality claims. In case of claim violation, the tool shows a temporal sequence of state transitions leading to information disclosure. Although originally developed in the context of the KAOS method for goal-oriented requirements engineering [20], our specification patterns and checker were designed for use within other declarative frameworks.

The paper is organized as follows. Section 2 introduces some background on temporal and epistemic logics and their use for requirements specification. Our running example, an e-payment system widely used in Belgium, is introduced there as well. Section 3 extends our previous specification framework through a formal model of knowledge of UAs. Section 4 presents a taxonomy of confidentiality requirements patterns that relies on those grounds. Section 5 presents our confidentiality checker and illustrates its use on the running example.

2. Background

2.1. Modeling and Specifying the Target System

To enable some form of automated reasoning about confidentiality, we need some formal apparatus for structuring requirements models and for specifying properties of interest. With respect to the formal language optionally used in KAOS [19], we assume a minimal subset comprising an entity-relationship language for object modeling and a real-time linear temporal logic.

Fig. 1 shows an object model fragment for a smartcard-based e-purse installed on bank cards. Money can be transferred from bank accounts onto e-purses at ATM counters. This requires PIN code authentication like for cash withdrawal. Buyers may use their e-purse at dedicated terminals installed in shops. To perform a payment from an e-purse to a terminal, the buyer needs to insert her bank card in the seller's terminal. The seller enters the amount due in the terminal. This amount is displayed to the buyer so that she can validate it by pressing an "OK" button. The validated amount is transferred from the buyer's e-purse to the seller's terminal. Payment is performed offline and requires no authentication. Terminals are unloaded to bank accounts via a secured network.

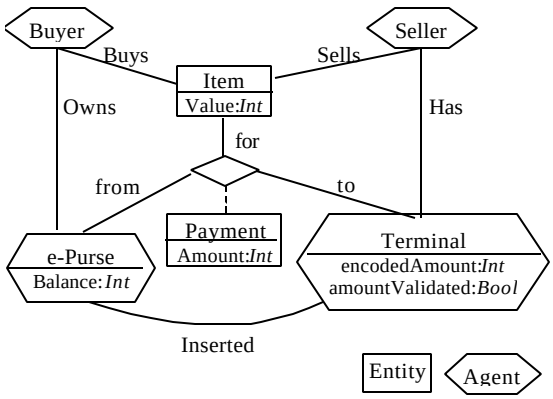

Fig. 1: Object model fragment for an e-purse payment system

In Fig. 1, agents are distinguished from entities as they control behaviors and have knowledge (unlike entities). We expand on this in Section 2.2 below.

Fig. 2 informally states a functional requirement and a confidentiality requirement for that system.

Requirement *Achieve* [PaymentDone**If**AmountValidatedAnd BalanceSufficient]
Def *A payment shall be done for some amount, from a card to a terminal, when the card is inserted in the terminal, the amount is entered by the seller in the terminal, the balance of the card is higher or equal to the amount, and the amount is OK-ed by the buyer on the terminal.*

Requirement *Maintain* [e-PurseBalanceConfidential]
Def *E-purse balances shall be known only to the e-purse holder and the e-purse itself.*

Fig. 2: Two requirements for an e-purse payment system

The requirements to be analyzed are assumed to be formalized in a first-order, real-time linear temporal logic [19, 26]. The following operators are used for temporal referencing:

$\mathbf{o}$	(in the next state)	$\bullet$	(in the previous state)
$\Diamond$	(some time in the future)	$\blacklozenge$	(some time in the past)
$\Box$	(always in the future)	$\blacksquare$	(always in the past)
W	(always in the future *unless*)	U	(always in the future *until*)

$\Diamond_{\leq d}$ (some time in the future within deadline *d*)
$\Box_{\leq d}$ (always in the future up to deadline *d*)

We will also use standard logical connectives such as $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), $\Rightarrow$ (entails), with

$$P \Rightarrow Q \quad \text{iff} \quad \Box(P \rightarrow Q).$$

Entities and agents from the object model may be used as sorts in quantifications. Attributes generally define state variables; they are accessed through the standard "." selector. For example, the assertion

$$(\forall\ ep\text{: e-Purse})\ \Box\ ep.Balance \geq 0$$

says that the balance of any e-Purse instance must always remain positive.

Association classes define predicates over instance variables playing the corresponding role. For example, the predicate

Payment (ep, term, it)

is true in the current state if and only if there is an object Payment[ep,term,it] in that state – that is, an instance of the ternary association *Payment* linking instances *ep*, *term* and *it* of sort *e-Purse*, *Terminal* and *Item*, respectively (see Fig. 1).

The functional requirement in Fig. 2 may now be specified formally as follows:

Requirement *Achieve* [PaymentDone**If**AmountValidatedAnd BalanceSufficient]
FormalSpec $\forall$ ep: e-Purse, term:Terminal, it: Item
Inserted (ep, term) $\wedge$ term.encodedAmount $\leq$ ep.Balance
$\wedge$ term.amountValidated
$\Rightarrow$ **o** (Payment (ep, term, it) $\wedge$
Payment[ep,term,it]**.**Amount = term**.**amountValidated)

2.2. Specifying Requirements On Agent Knowledge

To formalize confidentiality requirements such as the second requirement in Fig. 2, we need epistemic constructs that capture what agents may or may not know at specific states. The epistemic

operator $Knows_{ag}$ was introduced in [19] to capture accuracy requirements and inaccuracy obstacles. It is defined as follows:

$Knows_{ag}(P) \equiv Belief_{ag}(P) \wedge P$ ("knows property")

The operational semantics of the epistemic operator $Belief_{ag}(P)$ is: *"P is among the properties currently stored in the local memory of agent ag"*. An agent thus *knows a property* if that property is found in its local memory and it is indeed the case that the property holds.

The following particularization of this construct was used in [22] to capture malicious violations of confidentiality goals:

$KnowsV_{ag}(x) \equiv \exists v: Knows_{ag}(x = v)$ ("knows value")

This particular construct expresses that agent *ag* currently knows the value of state variable *x*. A very first specification pattern for confidentiality goals was introduced in [22] based on this construct:

Goal *Avoid* [SensitiveInfoKnownByUnauthorizedAgent]
FormalSpec $\forall$ ag: Agent, ob: Object
$\neg$ Authorized (ag, ob.Info) $\Rightarrow \neg KnowsV_{ag}$ (ob.Info),

where the *Authorized* predicate is generic and has to be instantiated through an application-specific definition.

This pattern may not be adequate in various situations. It only says that *exact* knowledge of the value of some state variable is prohibited to UAs. Let us assume that we use it for specifying our confidentiality requirement in Fig. 2; the knowledge by some third party that the e-purse balance is between $99 and $101 would not violate the corresponding requirement specification (as only *exact* knowledge of the current balance is prohibited). However, in such a situation the balance would be almost completely disclosed. Such disclosure might be sufficient in the context of satisfying an anti-goal of some malicious agent – such as, e.g., stealing e-purses that are sufficiently worth taking the risk. We should therefore specify stronger restrictions on knowledge of e-purse balances. We come back to this in Section 3.2.

The semantics of $Knows_{ag}(P)$ should also make it precise under which circumstances properties *P* do appear/disappear in/from the agent's memory. We come back to this in Section 3.1.

2.3. Logics for Reasoning about Knowledge

Different axiomatic systems have been proposed for defining the semantics of knowledge operators found in epistemic logics [9]. The *S5* system is the strongest in that it gives agents the most powerful reasoning capabilities. This system is summarized in Fig. 3, where *K* denotes the knowledge operator, *P* and *Q* are assertion placeholders and *ag* is an agent instance. The axiom schemas hold for all possible *P*, *Q* and *ag*.

Other systems are generally variants of *S5* where one or more axioms are removed or weakened. For instance, a formal model of agents with resource-bounded reasoning capabilities is described in [14].

Knowledge logics have also been combined with temporal logics such as LTL or CTL [9, 11]. For LTL integration, the *S5* axioms are strengthened with the ð-operator to hold at any time. Additional axioms capture the interaction between knowledge and time, typically, by defining memory-related capabilities of agents. Several combined systems are available dependent on whether the agents have a "synchrony" capability of measuring time or not. The two most common axioms in a synchronous system are given in Fig. 4.

- **Distribution**
  $[K_{ag}(P) \wedge K_{ag}(P \rightarrow Q)] \rightarrow K_{ag}(Q)$
  (Agents can deduce new knowledge from their knowledge)
- **Knowledge generalization**
  If *P* is valid then $K_{ag}(P)$ (Agents know all valid formulas)
- **Truth axiom**
  $K_{ag}(P) \rightarrow P$ (Agents have no false knowledge)
- **Positive introspection**
  $K_{ag}(P) \rightarrow K_{ag}(K_{ag}(P))$ (Agents know that they know)
- **Negative introspection**
  $\neg K_{ag}(P) \rightarrow K_{ag}(\neg K_{ag}(P))$ (Agents know that they don't know)

Fig. 3: Axioms of the *S5* epistemic logic system

- **Perfect Recall**
  $K_{ag}(\mathbf{o}\, P) \Rightarrow \mathbf{o}\, K_{ag}(P)$ (Agents never forget past knowledge)
- **No Learning**
  $\mathbf{o}\, K_{ag}(P) \Rightarrow K_{ag}(\mathbf{o}\, P)$ (Agent knowledge does not increase over time)

Fig. 4: Axioms for temporal knowledge

3. WHAT DO UNAUTHORIZED AGENTS KNOW?

To automate reasoning about (non-)disclosure of sensitive information to UAs, we need to make our $Knows_{ag}$ construct further precise by addressing questions such as the following:

- When and how do assertions get in and out of the memory of an UA?
- What do such agents know about the target system?
- Can such agents have contradictory beliefs?
- Can such agents reason on their beliefs and make deductions? In other words, does an agent's memory merely consist of a set of assertions or of its deductive closure?
- Can such agents know that time has elapsed?

For security-critical systems the answer to such questions should be driven by worst-case principles. To get robust requirements about confidentiality, we need to pessimistically consider an environment with malicious agents that are able to make calculations and deductions, even unanticipated ones. If such agents cannot violate the confidentiality requirements, any other will not. This worst-case principle is directly inspired from the *Most Powerful Attacker* model (MPA) used in cryptographic protocol analysis [17, 25, 2, 15, 4]. We use it to choose axioms that further define our $Knows_{ag}$ construct. The axioms are stated just informally here. Their relevance is illustrated on our e-purse example.

- **Perfect System Knowledge:** *UAs know all the requirements the software implements and all the properties of the domain.* In the e-purse system, the seller knows that payment is denied in case of insufficient balance.
- **Maximal Input:** *An agent at any time knows the value of any state variable that is not explicitly specified as confidential to her. The agent at any time also knows any past value of any state variable if that past value is not explicitly specified as*

confidential in the current state. In our e-purse system, the seller knows the amount that was entered, the amount that was validated, etc. This axiom ensures that the requirements model has no implicit assumptions on the confidentiality of variables.

- **Distribution** (taken from *S5*): *UAs are able to make deductive inferences.* In our example, if a payment is denied due to insufficient balance in the e-purse, the seller is able to infer that the balance of the e-purse is lower than the amount entered in the terminal.
- **Knowledge generalization** (taken from *S5*): *UAs know all tautologies*. In our example, the seller knows that the assertion
$$(e.Balance \geq \$5 \wedge true) \rightarrow false$$
reduces to
$$e.Balance < \$5.$$
- **Truth axiom** (taken from *S5*): *UAs have no false knowledge.* This axiom supports our first definition of $Knows_{ag}$ in terms of $Belief_{ag}$, see Section 2.2.
- **Perfect Recall** (taken from Fig. 4): *UAs always remember facts and properties they used to know in the past.* In our running example, the seller remembers, in the state where payment should take place, that the card was inserted in the terminal, that the amount was entered, and that this amount was validated.
- **Synchrony** (see Fig. 4): *UAs are able to measure time.* In our example, the seller may notice that time has elapsed since the card was inserted and since the amount was validated.
- **Introspection** (taken from *S5*): *UAs know that they know and know that they do not know*. This capability is taken to ensure some desired theoretical properties of the axiomatic system (e.g., the relation behind the epistemic modality is an equivalence relation). It also makes UAs more knowledgeable.
- **Closure assumption:** *All the knowledge of UAs directly or indirectly comes from the above axioms*. This axiom is taken to keep models realistic enough by excluding some "deus ex machina". We need it when we want to show that confidentiality requirements cannot be violated within the requirements model considered.

The closure assumption and the "Perfect System Knowledge" axiom together make the axiomatic system *non-monotonic* [10, 8], that is, we can have

$$P \models Q \quad \text{and} \quad P \wedge R \not\models Q$$

A confidentiality requirement might be satisfied by a requirements model but violated by some extended model where, for example, some functionality has been added (e.g., an alarm ringing on the terminal when e-purse balances are insufficient). Moreover, the confidentiality requirement might be satisfied by the requirements model but not by the final product due to information leaks introduced at implementation time [12].

4. SPECIFICATION PATTERNS FOR CONFIDENTIALITY REQUIREMENTS

Section 2.2 introduced the $KnowsV_{ag}$ construct that captures knowledge of the *exact* value of some state variable. Although appropriate in some situations, confidentiality of exact values may be too weak in others, as illustrated in the requirement on e-purse balances. In that specific example, agents who are not the cardholder should not know that there is *at least* $\$v$ in the e-purse (whatever the value of v is). Moreover, knowing that the e-purse contained at least $\$v$ six months ago is of no particular appeal to a potential card stealer; knowledge should refer to the current state in this case. The confidentiality requirement should thus rather look like:

$$\forall \text{ ag: Agent, ep: e-Purse}$$
$$\neg \text{ Authorized (ag, ep.Balance)}$$
$$\Rightarrow \forall v \in \mathbf{ran}(ep.Balance): \neg Knows_{ag}(ep.Balance \geq v)$$

where $\mathbf{ran}(ep.Balance)$ denotes the range of values of state variable *ep.Balance*.

In the spirit of [7, 5, 23], we have explored specification patterns that could help specifiers write recurring properties more easily as pattern instances. In our case the properties refer to confidentiality requirements. Such patterns may guide requirements elicitation; the analyst may browse the pattern catalog and ask herself whether instantiations to sensitive application objects are relevant requirements for the system. Pattern instances may also be used as input to analysis tools.

Our pattern catalog was built systematically along two dimensions:

- the degree of approximate knowledge to be kept confidential,
- the timing according to which that knowledge should be kept confidential.

Degree of approximate knowledge

Patterns along this dimension include the following.

- **Confidential exact value**: An agent should not know the exact value of some state variable:
$$val\text{-}Confidential_{ag}(x) \equiv$$
$$\forall v \in \mathbf{ran}(x): \neg Knows_{ag}(x = v)$$
- **Confidential lower/upper bound:** An agent should not know that the value of some state variable is below/above some treshold:
$$lb\text{-}Confidential_{ag}(x) \equiv$$
$$\forall v \in \mathbf{ran}(x): \neg Knows_{ag}(x \geq v)$$
$$ub\text{-}Confidential_{ag}(x) \equiv$$
$$\forall v \in \mathbf{ran}(x): \neg Knows_{ag}(x \leq v)$$
$$betw\text{-}Confidential_{ag}(x) \equiv$$
$$\forall v1, v2 \in \mathbf{ran}(x): \neg Knows_{ag}(v1 \leq x \leq v2)$$
- **Fully confidential value:** An agent should not be able to infer any property about the value of some state variable (e.g., lower/upper bound, parity, order of magnitude, etc.). In other words, the agent should not be able to exclude any value within the variable's range:
$$full\text{-}Confidential_{ag}(x) \equiv$$
$$\forall v \in \mathbf{ran}(x): \neg Knows_{ag}(x \neq v)$$
This is the strongest confidentiality requirement.
- **Confidentiality of order of magnitude:** An agent should not know the order of magnitude of some state variable. In other words, the agent will always consider that the order of magnitude of that variable can be any possible value. The order of magnitude might be defined, e.g., as the decimal logarithm rounded down. The pattern is then:
$$om\text{-}Confidential_{ag}(x) \equiv$$
$$\forall v \in 0..\lfloor \log_{10}(\max(\mathbf{ran}(x)) \rfloor:$$
$$\neg Knows_{ag}(\lfloor \log_{10}(x) \rfloor \neq v)$$

Timing of confidentiality

To be fully precise, the specifier should also make it explicit at what time and how long should confidentiality be enforced.

- **Confidential now:** In the current state, an agent should not know about some state variable:

  $Y\text{-Confidential-now}_{ag}(x) \equiv Y\text{-Confidential}_{ag}(x)$
  with $Y \in \{val, lb, ub, betw, full, om\}$

  Example of use: In our e-purse system, we might only be concerned by UAs knowing the value of an e-purse balance in the current state where the balance has *that* value – without necessarily caring for such agents to know tomorrow that the balance had that value today.

- **Confidential until expiration date:** An agent should not know about some state variable until some delay has expired:

  $Y\text{-Confidential-upTo}_{ag}(x; d) \equiv$
  $\forall w: x = w \rightarrow \Box_{\leq d}\ Y\text{-Confidential}_{ag}(w)$
  with $Y \in \{val, lb, ub, betw, full, om\}$

  Example of use: A classified NSA archive must be kept confidential for 60 years.

- **Confidential unless/until condition:** An agent should not know about some state variable unless or until some condition becomes true:

  $Y\text{-Confidential-unless}_{ag}(x; Cond) \equiv$
  $\forall w: x = w \rightarrow Y\text{-Confidential}_{ag}(w)\ \boldsymbol{W}\ Cond$
  $Y\text{-Confidential-until}_{ag}(x; Cond) \equiv$
  $\forall w: x = w \rightarrow Y\text{-Confidential}_{ag}(w)\ \boldsymbol{U}\ Cond$
  with $Y \in \{val, lb, ub, betw, full, om\}$

  Example of use: In a paper publication process, submissions must be kept confidential unless they are accepted.

- **Confidential forever:** An agent should never know about some state variable:

  $Y\text{-Confidential-always}_{ag}(x) \equiv$
  $\forall w: x = w \rightarrow \Box\ Y\text{-Confidential}_{ag}(w)$
  with $Y \in \{val, lb, ub, betw, full, om\}$

  Example of use: In some banking applications, daily account balances may be required to be kept confidential forever.

The combination of patterns defined along the knowledge approximation and the timing dimensions yields a pattern space of 30 generic specifications to be considered during requirements elicitation. The catalog is of course extendable along those two dimensions. Retrieving the appropriate pattern from the catalog is fairly simple; the degree of approximate knowledge is determined first and then the required timing is selected.

5. CHECKING REQUIREMENTS MODELS AGAINST CONFIDENTIALITY CLAIMS

We have built a proof-of-concept tool that checks requirements models against confidentiality claims that are expressed in terms of our specification patterns. The tool is named CONCHITA (CONfidentiality CHecker for Incremental Threat Analysis). It uses bounded model checking (BMC) and constraint solving techniques to search for counter-example scenarios leading to violations of confidentiality claims. The difference from standard BMC is that (a) the system is described at a more abstract level in terms of temporal logic assertions (instead of operational state machines), and (b) the tool reasons on epistemic logic constructs.

5.1 CONCHITA as seen from outside

The following *inputs* are provided to the tool.

- A set of functional requirements about the system (for example, the one given in Fig. 2). These are taken as system specification *and* as UA's knowledge of the system according to the *Perfect System Knowledge* axiom.
- A set of confidentiality requirements, taken as *claims*. These requirements are specified in terms of the patterns in Section 4. For our running example, we might submit the following claim (see Fig.2):

  $\forall$ ep: e-Purse, ag: Agent
  $\neg$ Owns (ag, ep) $\wedge$ ag $\neq$ ep
  $\Rightarrow$ full-Confidential-forever$_{ag}$ (ep.balance)

- A (possibly empty) set of other confidentiality requirements, specified with the same patterns, and taken as *assumptions*.
- An entity-relationship model that declares the objects referred to in those various assertions (see Fig. 1).
- Directives for instantiating the object model in order to propositionalize those various assertions. For example, we might bound our model to one single instance of Buyer, Seller and Terminal and introduce corresponding names for reference in the counter-example trace (if any):

  ```
  Buyer:[alice]
  Seller:[bob]
  Terminal:1
  ```

- A maximal length for the output trace.
- Optional hints for constraining the output trace to make it more "interesting". For example, the user might require that the balance of e-purses is a least 3, to avoid simple traces with zero-balance.

CONCHITA displays a counter-example trace, if one is found, or a message "no confidentiality violation found". The output trace leads to disclosure of one of the claimed-to-be-confidential state variables. Disclosure takes place at the last position of the trace. It is characterized by the following items:

- the state variable that was claimed to be confidential,
- the agent instance that was claimed to know nothing about this variable,
- a position in the trace and a value for the variable such that, at the end of the trace, the agent knows that the variable is different from this value at that position.

The counter-example trace delivered for our running example is shown in Fig. 5. A graphical representation of it is shown in Fig. 6.

The trace is displayed as a sequence of states. Each state shows the value of the attributes of entity/agent instances together with their associations. A bracketed attribute in the trace expresses that its value was assumed to be confidential and thus hidden to the agent instance (see *e-Purse1.Balance* in state 1).

As mentioned before, unviolated confidentiality claims are taken as assumptions. The claim is split into one confidentiality claim per position in the trace because of the $\Box$-operator implicit in the strong implication of the claim. The stars around *ePpurse1.Balance* in state 0 express that this state of the variable

is disclosed at the end of the trace. The UA is bob. At the end of the trace, he gets to know that the balance of alice's e-purse was different from 4 at time 0 (see the message generated in state 1 in Fig. 5 or the bubble at state 1 in Fig. 6). Bob's reasoning goes as follows: (a) in state 0, the *e-Purse1* is inserted in *Terminal1*, the encoded amount is 4, and this amount is validated; (b) in state 1, no payment takes place; hence the balance of *e-Purse1* can be inferred to be strictly lower than 4.

```
---------------------------------- State 0
alice  (Buyer agent){}
bob  (Seller agent){}
e-Purse1  (e-Purse agent){
    ** Balance: 3 ** (DISCLOSED) }
Terminal1  (Terminal agent){
    amountValidated: true
    encodedAmount: 4 }
Inserted<docked: e-Purse1
        docker: Terminal1>
Owns<owned: e-Purse1
      owner: alice>
---------------------------------------- State 1
alice  (Buyer agent){}
bob  (Seller agent){}
e-Purse1  (e-Purse agent){
    [ Balance: 3 ] (HIDDEN)}
Terminal1  (Terminal agent){
    amountValidated: false
    encodedAmount: 0}
Owns<owned: e-Purse1
      owner: alice>
At this state, bob knows that e-Purse1.Balance is
different from 4 at time 0
-------------------------------------------------
```

Fig. 5: Counter-example trace produced by CONCHITA

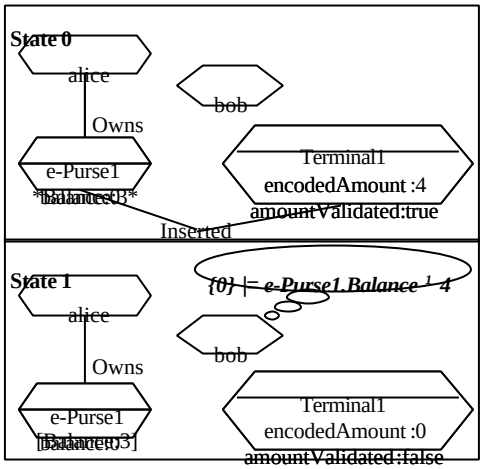

Fig. 6: Graphical view of the output trace

In the current state of implementation, CONCHITA searches for violation of the most typical patterns only, that is, the *full-Confidential-now*$_{ag}$, *full-Confidential-until*$_{ag}$ and *full-Confidential-forever*$_{ag}$ patterns. Therefore, it only finds one value that Bob can eliminate from the set of possible values for *e-Purse1*'s balance. However, as explained before, Bob can infer more properties about this value.

CONCHITA also explains the reasoning performed by the UA that enabled him to infer that the claimed-to-be-confidential variable is different from the given value. The explanation consists of a minimal set of knowledge fragments about the system that are necessary to perform the deduction. Knowledge fragments include requirements and domain properties (including association multiplicities), in which universal quantifications over objects and time have been instantiated to corresponding object instances and trace positions, respectively.

Fig. 7 shows the explanation displayed for Bob's reasoning. In this case it includes one fragment only, taken from the requirement

Achieve[PaymentDone**If**AmountValidatedAndBalanceSufficient]

formalized in Section 2.1, where the universally quantified variables have been instantiated to *Teminal1* and *e-Purse1*, respectively, and where the $\Box$-operator implicit in the "$\Rightarrow$" entailment has been instantiated to state 0, as mentioned on the left of the '|=" symbol; the '^' suffix is used for reference to the previous value of the state variable.

```
The unauthorized agent used the following
knowledge fragments in his inference.

Source: Achieve[PaymentDoneIfAmountValidated
AndBalanceSufficient]

Instantiation:

[term/Terminal1, ep/e-Purse1]

0 |= Inserted<docked:e-Purse1 docker:Terminal1>
    & Terminal1.amountValidated
  & Terminal1.encodedAmount ≤ e-Purse1.Balance
  -> o Payment<amount: Terminal1.encodedAmount^
               from: e-Purse1
               to:Terminal1>
```

Fig. 7: Explaining the reasoning for disclosure

5.2. CONCHITA's bones and guts

Our tool is based on bounded model checking and finite instantiation algorithms [13, 1]. It is structured into four layers. The top three layers translate their input problem into a simpler problem. The bottom layer performs the computation using constraint-programming techniques [28, 6, 29]. It also contains a constraint satisfaction diagnosis algorithm used for generating the UA's reasoning. The solution found by the bottom layer percolates up through the upper layer; it is gradually translated into a solution to the top input problem. The overall architecture of the tool is shown in Fig. 8. The *system specification* there is made of the requirements, the domain properties and the hints. The *UA's knowledge* is made of the requirements and the domain properties. Layers are organized into a "*use*" hierarchy.

The propositionalization and time bounding layers are not semantics-preserving as they restrict the expressiveness of the model. To ensure tool soundness, we therefore *strengthen* the translation of the system specification and confidentiality assumptions while *weakening* the translation of the confidentiality claims and of the UAs's knowledge. Such dual translations are known as expanding and contracting abstractions [27]. Intuitively, it means that we will be more demanding on generated traces while UAs will know a little bit less about the system. For example, we want to avoid UAs to know that there is only one single e-Purse instance and use this in their reasoning.

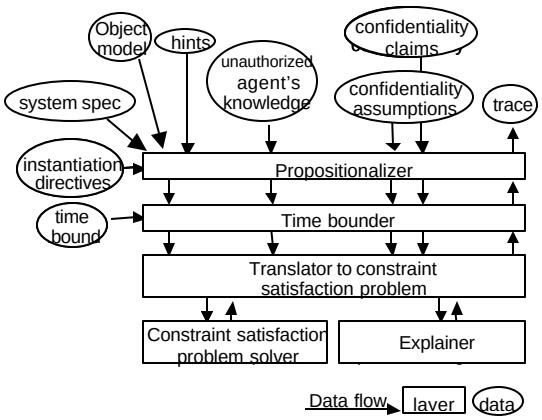

Figure 8: CONCHITA's overall architecture

The four layers of the tool are now briefly described.

Layer 1: Propositionalizing the input model

The first-order input model is propositionalized by instantiation of the object model according to the instantiating directives provided. The system specification and assumptions are propositionalized with strengthening by application of standard transformation rules [13], e.g.,

$(\forall$ ep: e-Purse) $P \wedge$ e-Purse = {e-Purse1, e-Purse2}
~~> P [ep / e-Purse1] $\wedge P$ [ep / e-Purse2]

$(\exists$ e: e-Purse) $P \wedge$ e-Purse = {e-Purse1, e-Purse2}
~~> P [ep / e-Purse1] $\vee P$ [ep / e-Purse2]

On the other hand, the UA's knowledge and the confidentiality claims are propositionalized with weakening by addition of an abstract instance, denoted by *Others*, that represents all other instances. The same translation rules then yield:

$(\forall$ ep: e-Purse) $P \wedge$ e-Purse = {e-Purse1, e-Purse2, *Others*}
~~> P [ep / e-Purse1] $\wedge P$ [ep / e-Purse2] $\wedge P$ [*ep / Others*]

$(\exists$ e: e-Purse) $P \wedge$ e-Purse = {e-Purse1, e-Purse2, *Others*}
~~> P [ep / e-Purse1] $\vee P$ [ep / e-Purse2] $\vee P$ [ep / *Others*]

The assertion P [ep / *Others*] is actually translated into *true* or *false* dependent on whether the translation of P is performed with weakening or strengthening, respectively. (A translation with strengthening must be foreseen for propositionalization with abstract domains as the negation of P is translated with weakening into the negation of the translation of P with strengthening.)

The propositionalization layer also makes a switch to a mono-agent model with one single UA denoted by a rigid variable "*Unauthorized*" whose value does not change along the trace. The confidentiality claims are translated accordingly, e.g.,

Unauthorized = 'bob' $\rightarrow$
full-Confidential-forever (Unauthorized, e-Purse1.Balance)

Unauthorized = 'Terminal1' $\rightarrow$
full-Confidential-forever (Unauthorized, e-Purse1.Balance)

As all claims refer to the same agent we drop agent references.

Layer 2: Bounding time

The propositional temporal logic assertions are transformed into non-temporal logic versions by setting the length of traces to the given time bound. Non-temporal variables are introduced to represent the value of each state variable at each position in the trace. All temporal references are resolved along this finite trace. The system specification and UA's knowledge are transformed with strengthening and weakening, respectively, by application of rules such as the following:

bound ("? P", t, s) **~~>** $\bigwedge_{t \le i \le end}$ bound("P", i, s)

bound ("**o** P", t, strong)
~~> **if** t = end **then** *false* **else** bound ("P", t+1,strong)

bound ("**o** P", t, weak)
~~> **if** t = end **then** *true* **else** bound ("P", t+1, weak)

In the above rules, *t* is a position in the trace, *end* is the last position, and *s* takes the value *strong* for a transformation with strengthening and *weak* for a translation with weakening.

Temporal references in confidentiality claims and assumptions are resolved by evaluating the UA's knowledge at the last position in the trace. We use the *full-Confidential* pattern for confidentiality in non-temporal models. Our first claim above is transformed into the following two non-temporal claims:

Unauthorized = 'bob'
$\rightarrow$ full-Confidential (e-Purse1.Balance$_0$)
Unauthorized = 'bob
$\rightarrow$ full-Confidential (e-Purse1.Balance$_1$) ,

where subscripts denote the position of the non-temporal variables in the trace.

The transformation of assumptions yields the annotation of whether or not a non-temporal variable is hidden to the UA. If the value of a temporal variable at some position may be disclosed at any subsequent position in the trace, then its corresponding non-temporal variable is taken as disclosed in the non-temporal model, because of the *perfect recall* axiom.

Layer 3: Translation into a constraint satisfaction problem

The confidentiality checking problem is translated into an equivalent constraint satisfaction problem by encoding our confidentiality patterns as quantifications over traces envisioned by the UA, given his knowledge of the system (H_w) and his knowledge of the variables that are not claimed nor assumed to be confidential. The system specification is denoted H_s. The constraint satisfaction problem submitted to the *Oz* constraint solver [29] can be paraphrased as follows:

find a value *c* and interpretation *Int* **such that**:
(a) *Int* |= H_s
(b) there is no interpretation *J* **such that**
J |= H_w
and non-confidential variables have same values in *Int* and *J*
and claimed-confidential variable in *Int* is equal to *c*

Constraint (a) expresses that the interpretation found must be admissible with respect to the system specification translated with strengthening. Constraint (b) expresses that there is no other interpretation that the UA could imagine of such that the claimed confidential variable is equal to some value, given UA's knowledge (i.e., the system specification translated with weakening and the value of each non-confidential variable). The UA is therefore able to find out that the claimed confidential variable is different from this value.

Layer 4: Solving the constraint satisfaction problem

The constraint satisfaction problem is solved by means of constraint programming techniques built on top of the *Oz* constraint solver [28, 6, 29].

Explaining the UA's reasoning

The interpretation *Int* discovered by the tool has the property that the following problem has no solution:

find interpretation J **such that**:

$J \models H_w$
and non-confidential variables have same values in *Int* and J
and claimed-confidential variable in *Int* is equal to c

The tool finds the knowledge fragments from H_w that cause the failure by applying a two-step process.

Step 1: *Decompose H_w into as many conjuncts as possible.* This is actually performed by layers 1 and 2. For example, a universal quantification yields one fragment per instance, a ?-operator yields one fragment per position between the position at which it is evaluated and the last position in the trace, and a conjunction yields two fragments. Fragmentation is performed recursively up to the first operator that cannot be fragmented; generally an implication or a disjunction.

Step 2: *Find a minimal set of such fragments that causes the problem to have no solution.* In constraint programming terms, this amounts to searching for a minimal conflict set. We use the *QuickXPlain* algorithm for this [16].

5.3. Properties of CONCHITA

Our checker is *sound*; if it finds a trace, the trace does violate some confidentiality claim while matching the system specification. In other words, it will never produce false positives.

Like any bounded model checker, CONCHITA only searches within the given time bounds and object instances. Assigning too restrictive bounds may therefore result in failure to find a trace when there is a longer one or one with more instances. CONCHITA is however *bounded-complete*: if there is a trace within the given bounds, it will find it. Bounded-completeness is achieved by ensuring that the top two layers perform as little strengthening/weakening as possible. (Completeness issues do not arise at the two lower layers which perform exact translations and resolutions of their input problem.)

6. CONCLUSION

Confidentiality is an important class of security concerns that refers to restrictions on agent knowledge. The earliest security leaks are detected, the best; countermeasures can then be devised early as new requirements for a more robust system. Our formal framework makes it possible to specify and analyze confidentiality concerns at requirements engineering time. Non-intentional and malicious disclosure of sensitive information are both supported. Our framework has three components.

- An axiomatic system, built on top of epistemic logics [9], captures how the knowledge of UAs evolves over time. In a way similar to the "Most Powerful Attacker" hypothesis used in cryptographic protocol analysis, we assume UAs to be clever and "forgetless".
- A catalog of confidentiality specification patterns captures different types of confidentiality requirements on state variables. The patterns are expressed in terms of knowledge operators from this axiomatic system. The catalog is built on top of previous patterns [22]. It has a two-dimensional structure according to the degree of approximate knowledge and knowledge timing.
- A proof-of-concept tool prototype checks requirements models against confidentiality requirements expressed in terms of such epistemic patterns. The tool separately maintains UA's knowledge according to the epistemic axiomatic system. It combines bounded model checking and constraint programming techniques to generate counter-example traces leading to violation of confidentiality requirements.

Our framework combines formality, grounded on epistemic logics, and lightweightedness through specification patterns.

Formality is a prerequisite for a well-defined semantics of specification constructs and for in-depth analysis of interactions between functional and security requirements. The analysis performed by the tool is incremental as it is applied to declarative model fragments. It is made efficient by instantiation of our models to small worlds, as done by bounded model checkers, and by pruning of the search space, as done by constraint solvers. The latter more efficiently support complex specifications, e.g., over arithmetic expressions.

The patterns, in the spirit of [7, 19], considerably facilitate the task of writing specifications with knowledge operators. They can also be used as a means for eliciting confidentiality requirements and making them more precise.

Our framework makes minimal assumptions about the requirements model. It essentially needs a simple form of UML class diagram and a set of assertions. It might therefore be grafted at low cost on recent extensions of goal-oriented and agent-oriented methods for threat modeling [24, 22]. Moreover, CONCHITA can be used in the context of goal-oriented requirements engineering [20] to ensure that some high-level goal is enforced by lower-level requirements. In a banking application, for example, we should ensure that bank accounts are confidential on the basis that information flows between ATMs and the central bank is kept confidential; in such a case the high-level goal would be submitted as a confidentiality claim whereas the lower-level requirements are submitted as confidentiality assumptions.

In situations where confidentiality violations are not felt critical, it might be worth considering less clever UAs who know less about the system or are forgetful. In such situations we should then explicitly provide application-specific axioms that specify the conditions under which specific system knowledge fragments appear/disappear in/from UA's memory.

Our approach requires the analyst to specify all necessary confidentiality requirements (in view of the maximal input axiom). We might alleviate this by imposing a *least privilege principle* [30] as additional requirement on some agents – e.g., software agents. This principle would state here that the agents may only see those state variables which they need to monitor/control for achieving the requirements they are responsible for.

Our pattern catalog is currently limited to confidentiality of state variables. Although it was built in a systematic way along two

structuring dimensions, its coverage needs to be better assessed through further validation studies. Our current implementation also needs to be extended to support the entire catalog, and then experimented on larger-scale applications. The feedback provided by the tool should also be easier to understand. Our plan is to represent violation scenarios graphically (see Fig. 6 in comparison with Fig. 5), and only show attributes that are effectively used in UA's reasoning.

There is an interesting connection between confidentiality and trust we are also currently exploring. To enforce confidentiality of classified information, one must ensure that secret holders are trustworthy agents; they should not reveal classified information to unauthorized agents. This issue might be resolved by inclusion of some appropriate model of trustworthiness in requirements models.

Acknowledgements. The work reported herein was partially supported by the Belgian Fonds National de la Recherche Scientifique (FNRS) and the Regional Government of Wallonia (MILOS project, RW Conv. 114856).

7. REFERENCES

[1] A. Biere, A.Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking", *Advances in Computers*, 58, 2003.

[2] D. Bolignano, "Towards a mechanization of cryptographic protocol verification", *Proc. 9th International Computer Aided Verification Conference*, 1997, 131-142.

[3] http://www.cert.org/stats/cert_stats.html.

[4] I. Cervesato, "Data access specification and the most powerful symbolic attacker in msr", In *ISSS 2002: Software Security - Theories and Systems*, LNCS 2609, Springer-Verlag, November 2003, 384-416.

[5] M. Chechik and D. O. Paun, "Events in property patterns", In *Theoretical and Practical Aspects of* SPIN *Model Checking*, LNCS 1680, Springer-Verlag, 1999, 154-167.

[6] R. De Landtsheer, "Solving CSPs including universal quantifications", *Proc. of the 2nd Int. Mozart/Oz Conference*, 2004.

[7] M.B. Dwyer, G. S. Avrunin and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification", *Proc. ICSE'99 - 21st Intl. Conf. Softw. Eng.*, May 1999.

[8] J. Engelfriet, "Monotonicity and persistence in preferential logics", *J. Artif. Intell. Res.* 8, 1998, 1-21.

[9] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[10] J. Halpern and Y. Moses, "Towards a Theory of Knowledge and Ignorance: Preliminary Report", In *Logics and Models of Concurrent Systems*, Springer-Verlag, 1985, 459-476.

[11] Halpern J., van der Meyden R., and Vardi. Complete axiomatizations for reasoning about knowledge and time. 1997.

[12] J. Jacob, "On the derivation of secure components", In *Proc. of 1989 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1989.

[13] D. Jackson, "Automating first-order relational logic", *Proc. FSE'2000: 8th ACM SIGSOFT Intl Symp. Foundations of Software Engineering*, San Diego, 2000.

[14] M. Jago, N. Alechina, and B. Logan, "A complete and decidable logic for resource bounded agents", *P*roc. *AAMAS 04*, New York, July 2004, 606 - 613.

[15] S. Jha, E.M. Clarke and W. Marrero, "Verifying security protocols with Brutus", *ACM Trans. Software Engineering and Methodology (TOSEM)*, October 2000, 443-487.

[16] U. Junker, "QUICKXPLAIN: Conflict Detection for Arbitrary Constraint Propagation Algorithms", *Proc. IJCAI'01 Workshop on Modeling and Solving Problems with Constraints*, 2001.

[17] R. Kemmerer, C. Meadows, and J. Millen, "Three systems for cryptographic protocol analysis", *Journal of Cryptology* 7(2), 1994, 79-130.

[18] R. Kemmerer, "Cybersecurity", *Proc.ICSE'03 - 25th Intl. Conf. on Softw. engineering*, Portland, 2003, 705 - 715.

[19] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering", *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26(10), October 2000, 978 - 1005.

[20] A. van Lamsweerde, "Goal-oriented requirements engineering: A guided tour". *Proc.RE'01 - 5th IEEE International Symposium on Requirements Engineering*, Toronto, August 2001, 249-263.

[21] A. van Lamsweerde, "From System Goals to Software Architecture", In *Formal Methods for Software Architectures*, M. Bernardo & P. Inverardi (eds), LNCS 2804, Springer-Verlag, 2003, 25-43.

[22] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models", *Proc. ICSE'04: 26th Intl. Conf. on Software Engineering*, IEEE, 2004, 148-157.

[23] E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", *Proc. FSE'10:* 10th *ACM SIGSOFT Symp. Foundations of Software Engineering*, Charleston, November 2002.

[24] L. Liu, E. Yu and J. Mylopoulos, "Security and Privacy Requirements Analysis within a Social Setting", *Proc.RE'03: 11th IEEE International Requirements Engineering Conference*, Monterey, Sept. 2003.

[25] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *TACAS'96: Tools and Algorithms for Construction and Analysis of Systems*, 1996.

[26] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[27] A. Pnueli, "Verification by Finitary Abstraction", *Proc. SPIN'98: 4th Intl. SPIN Workshop*, Paris, Nov. 1998.

[28] Ch. Schulte. *Programming Constraint Services*. Lecture Notes in Artificial Intelligence Vol. 2302,. Springer-Verlag, Berlin, 2002.

[29] P. Van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

[30] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.

[31] J. Wing, "A Symbiotic Relationship Between Formal Methods and Security", Proc. NSF Workshop on Computer Security, Fault Tolerance, and Software Assurance: From Needs to Solution. December 1998.