A Reuse-Based Approach to Security Requirements Engineering

Laurent A. Hermoye and Axel van Lamsweerde
Département d'ingénierie informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
lahermoye@gmail.com, avl@info.ucl.ac.be

Dewayne E. Perry
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX
perry@ece.utexas.edu

Abstract

The paper presents a reuse-based approach to the modeling, specification and analysis of application-specific security requirements. The method is based on a goal-oriented framework that addresses malicious goals (called anti-goals) set up by attackers to threaten security goals. Threat tree fragments are built systematically through specializations of attack patterns. Attack patterns abstract attacker's malicious goals for known attacks. Once specialized to a specific vocabulary, they are useful to build a complete threat tree. Indeed, the specialized attack pattern is then further refined until leaf nodes are derived that are either software vulnerabilities observable by the attacker or anti-requirements implementable by this attacker. New security requirements are then obtained as specializations of reusable countermeasures associated with the attack pattern. The method is illustrated with an attack pattern for replay attacks that is specialized in an e-commerce system and a mailing system.

1. Introduction

Requirements engineering (RE) is the first step in the system life-cycle. RE aims at identifying goals to be achieved by the system-to-be, operationalizing such goals into services and constraints, and assigning responsibilities for the resulting requirements to agents such as humans, devices, and software [24].

Reports on software vulnerabilities have increased dramatically [1]. They also have been implicated in a large number of accidents [17]. Moreover, technologies used by attackers are rendering software systems incresingly insecure. Identification of software vulnerabilities can thus have a major impact on security. The **first goal** of this paper is to introduce a formal technique to support *high quality* requirements specifications by finding threats, vulnerabilities, malicious goals, and their corresponding countermeasures. By *high quality*, we mean complete, accurate, and secure.

Future breakthroughs in requirements engineering productivity and quality, as well as cost reduction, may well depend on the ability to reuse existing requirements to produce new requirements [15]. The current build-from-scratch techniques that dominate most requirements elicitations must eventually give way to techniques that emphasize their construction from reusable patterns. If not, requirements engineers may reach a limit in generating large, high-quality requirements definitions. These definitions are increasingly important due to a demand for large and complex software systems. The **second goal** of the paper is to provide contributions to requirements reuse.

Our general goal is thus to create a systematic technique to support reuse of requirements in the context of security. The solution proposed in the paper defines reusable abstract threat tree fragments for known attacks (e.g., replay, denial-of-service, password attacks). Once specialized to a specific vocabulary the attack pattern is used to systematically generates malicious goals. The resulting threat tree fragment is then further refined until leaf nodes are derived. Reusable countermeasures are proposed as well in response to such goals.

The organization of the paper will be the following: Section 2 provides the necessary background; Section 3 introduces *attack patterns* as reusable models for threat analysis; Section 4 assesses the technique with an attack pattern for replay attacks specialized in two case studies (e-commerce system, mailing system); Section 5 discusses related work; and Section 6 summarizes contributions, evaluations, and future directions.

2. Background

Goals are prescriptive statements that capture the rationale of the envisioned system by answering *why* the system is needed. Goals are then operationalized into services and constraints, thereby answering *what* the system should do [21]. The responsibilities for goals are finally assigned to agents such as humans, devices and software, thereby capturing *how* the system will be built. Goals are organized in hierarchies with high-level goals being coarse-grained (e.g., "secure transaction"), and low-level goals being more technical (e.g., "encrypted transaction"). Goals are elicited through a refinement/abstraction process. AND-refinements link a goal to a set of subgoals. Satisfying all subgoals in the refinement is a sufficient condition for satisfying the goal. OR-refinements link a goal to an alternative set of reductions. Satisfying one of the reductions is a sufficient condition for satisfying the goal. The specification of a goal can be formally expressed in a realtime linear temporal logic borrowed from [14] using the following temporal operators [19],

$\Diamond$ *(eventually)* $\blacklozenge$ *(some time in the past)*
$\Box$ *(always in the future)* $\blacksquare$ *(always in the past)*

and the following epistemic operators [25],

$KnowsV_{ag}(v) \equiv \exists x : Knows_{ag}(x = v)$ *(knows value)*
$Knows_{ag}(P) \equiv Belief_{ag}(P) \land P$ *(knows property)*

where $Belief_{ag}(P)$ means P is among the properties stored in the local memory of agent ag. Let us illustrate this by a goal in a mailing system:

Goal *Achieve*[MailReceivedInTime]
Refines EffectiveSystem
RefinedTo MailAtPostOfficeInTime, MailDispatchedInTime, MailDeliveredInTime
FormalDef $\forall s : Sender, m : Mail, r : Receiver$
$Posts(s, m, r) \Rightarrow \Diamond_{\leq d} Receives(r, m)$

An object model associated with the goal tree captures agents (*Sender*, *Receiver*), objects (*Mail*), relationships between them (*Posts*, *Receives*), and their corresponding invariants. Goals can be refined through instantiations of generic refinement patterns [3]. The refinement process ends when every goal is realizable by an agent.

Obstacles can be elicited to identify goal violation scenarios [20]. An obstacle is a condition whose satisfaction may prevent a goal from being achieved. More precisely, let G be a goal assumption and Dom be a set of prescriptive statements on the environment. An assertion O is said to be an obstacle to GA iff the following conditions hold:

1. $\{O, Dom\} \vdash \neg GA$ (obstruction)
2. $Dom \nvdash \neg O$ (domain consistency)

Obstacle analysis [27] consists of identifying obstructions by regressing goal negations through domain theory until they are assignable to agents. Regression consists of calculating preconditions from the domain theory for obtaining the negation of a goal assertion. For example, a precondition of the MailReceivedInTime goal previously defined is that the target address should be readable. An obstructing obstacle can thereby be regressed: ImpossibleToReadTargetAddress. The obstruction is resolved using resolution tactics [27], thereby producing more robust requirements (e.g., closing postboxes to avoid rain rendering the target address unreadable).

Obstacle analysis appears to be too limited to capture attacks that satisfy attacker's goals based on their capabilities and on the system's vulnerabilities [25]. Threat models, on the other hand, capture these malicious goals, called anti-goals, by (1) systematically negating relevant specification patterns for Security goals [4] instantiated to sensitive attributes, (2) elaborating a threat AND/OR tree, and (3) deriving alternative countermeasures to the threats found. Let us illustrate this in the mailing example:

SpecPattern *Avoid*[SensitiveInfoKnownByUnauthorized]
FormalDef $\forall a : Attacker, ob : Object$
$\neg Authorized(a, ob.Info) \Rightarrow \neg KnowsV_a(ob.Info)$

can be instantiated and negated, yielding:

AntiGoal *Achieve*[MailKnownByThirdParty]
FormalDef $\forall a : Attacker, m : Mail$
$\neg isReceiverOf(a, m.Info) \land KnowsV_a(m.Info)$

A Thief agent could benefit from the above anti-goal. Regression through domain theory and resolution of the anti-goal are illustrated on Figure 1.

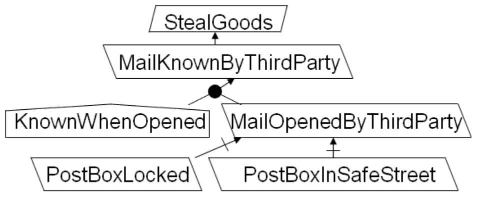

Figure 1. Anti-model for the Mailing System

3. Reusing Attack Patterns

The threat tree building process should be supported by techniques and tools. One technique is to abstract commonalities among known attacks from several case studies. This abstraction results in reusable anti-goals. These are linked together through refinement links to build an attack pattern. The result is a reusable threat tree fragment that captures common objectives of malicious agents for known attacks. Once specialized to a domain-specific vocabulary, the attack pattern provides useful domain properties and anti-goals. The anti-goals of the specialized fragment should then be further refined until leaf nodes are derived that are either software vulnerabilities observable by the attacker or anti-requirements implementable by this attacker. The rationale of root anti-goals should be understood as well to capture attacker's high level intentions.

3.1. Definition of Attack Patterns

Several concepts are important for understanding and defining attack patterns.

- A *specialization* is a binary relation between two concepts. IsA(A, B) means that A is a specialization of B. IsA(A,B) holds iff every instance of concept A is also an instance of concept B, and there are instances of concept A that are not instances of concept B [16].

- An *instantiation* is a binary relation between two concepts. IsOf(A, B) means that A is a particularized occurrence of a meta-concept B.

- A *domain* can be *abstract* or *concrete*. An abstract domain is an abstracted form of a set of concrete domains. The abstraction categorizes concrete domains in abstract domains (for example, the Plane, and the Train domains are Transportation domains). A concrete domain is therefore a specialization (isA) of an abstract domain with knowledge from a real world application area [10].
- *Abstract anti-goals, domain properties* and *predicates* are reusable concepts defined on abstract domains that should be specialized in concrete domains at reuse time.
- An *attack pattern* is a fragment of an anti-model defined on an abstract domain. Attack patterns are built with abstract anti-goals and abstract domain properties that can be formalized in linear temporal logic with abstract predicates. Abstract predicates should be specialized to a concrete domain in order to generate a fragment of an anti-model. The generated fragment isA abstract anti-model fragment.

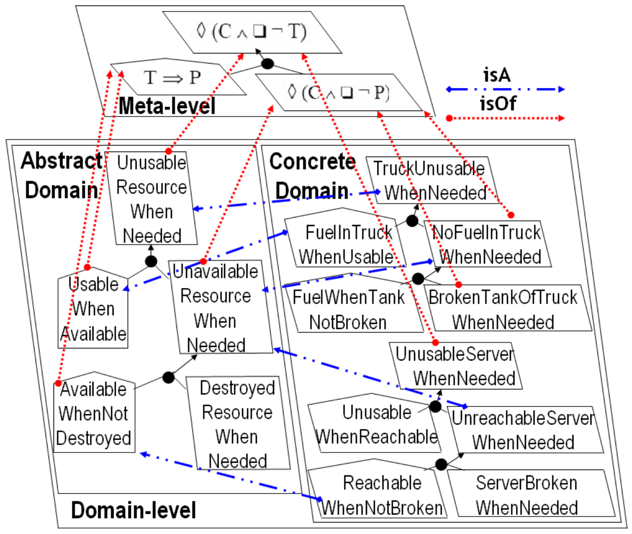

Figure 2. Attack Pattern for *destruction* Attacks

For example, the attack pattern for *destruction* attacks (left of Figure 2) states that the attacker wants some resource needed by an agent to be unusable. The refinements provide a means for rendering the resource unusable by destroying it. The abstract anti-model fragment is specialized in two concrete domains: an e-commerce domain, and a mailing domain. The first specialization states that the attacker wants a server to be unusable by known clients. The refinements provide a means for rendering the server unusable by breaking it. The second specialization states that the attacker wants a truck needed for transporting mail to be unusable. The refinements provide a means to make the truck unusable, by destroying the gas tank.
Let us illustrate one of the regressions of the attack pattern and its corresponding specializations.

AbsAntiGoal *Achieve*[UnusableResourceWhenNeeded]

FormalDef $\Diamond \exists u : User, r : Resource$
$NeedsResource(u, r) \wedge AuthorizedUser(u, r) \wedge$
$\Box_{<M} \neg UsingResource(u, r)$

A necessary precondition for a user to use a resource is that the resource should be available.

AbsDomProp [UsableWhenAvailable]

FormalDef $\forall u : User, r : Resource$
$UsingResource(u, r) \Rightarrow Available(r)$

Instantiating the 1-step regression pattern (above on Figure 2) [3] in order to regress UnusableResourceWhenNeeded through UsableWhenAvailable yields the following anti-goal:

AbsAntiGoal *Achieve*[UnavailableResourceWhenNeeded]

FormalDef $\Diamond \exists u : User, r : Resource$
$NeedsResource(u, r) \wedge AuthorizedUser(u, r) \wedge$
$\Box_{<M} \neg Available(r)$

We can specialize the attack pattern to an e-commerce system, and a mailing system:

- **E-Commerce:** *Client* isA *User*, *WebService* isA *Resource*, *Accesses* isA *NeedsResource*, *KnownClient* isA *AuthorizedUser*, *UsingServer* isA *UsingResource*, and *Reachable* isA *Available*.
- **Mailing:** *Mail* isA *User*, *Truck* isA *Resource*, *NeedsTransport* isA *NeedsResource*, *WithPostage* isA *AuthorizedUser*, *isTransportedBy* isA *UsingResource*, and *HasFuel* isA *Available*.

For example, UnusableServerWhenAccessed isA specialization of UnusableResourceWhenNeeded in the *e-commerce* system.

AntiGoal *Achieve*[UnusableServerWhenAccessed]

Specializes UnusableResourceWhenNeeded

FormalDef $\Diamond \exists c : Client, ws : WebService$
$Accesses(c, ws) \wedge KnownClient(c, ws) \wedge$
$\Box_{<M} \neg UsingServer(c, ws)$

The same abstract anti-goal can be specialized to the *mailing* system. In that context, TruckUnusableWhenNeeded isA specialization of UnusableResourceWhenNeeded.

AntiGoal *Achieve*[TruckUnusableWhenNeeded]

Specializes UnusableResourceWhenNeeded

FormalDef $\Diamond \exists m : Mail, t : Truck$
$NeedsTransport(m, t) \wedge WithPostage(m, t) \wedge$
$\Box_{<M} \neg isTransportedBy(m, t)$

Other abstract anti-goals and abstract domain properties are specialized similarly. Note that the specialized anti-goals should be further refined because the specialization generated only a fragment of anti-model.

3.2. A Library of Attack Patterns

Attack patterns were abstracted from concrete anti-models defined in case studies (mine pump system, electronic vote system, e-commerce system, and mailing system) [9]. The description approach in this paper will proceed instead from abstract to concrete: first an introduction of abstract concepts, and then, specializations of them in application-specific domains.

Reusable libraries are commonly described as follows [26]:
- How to define a library of reusable elements?
- How to reuse such a library?

The library of abstract domain properties is described with this structure in the next sections.

3.2.1 Definition of a Library

The library is composed of attack patterns. Attack patterns are composed of abstract anti-goals and domain properties.

- **Model-specific features:**

- *Context:* A set of circumstances in which the attacker might be able to compromise a system. The circumstances are captured in abstract goals for Security goals that the attacker may be able to compromise. These abstract goals are instances of (isOf) specification patterns for Security goals [25]. Meta-variables of specification patterns are instantiated to attributes/associations defined on abstract domains.

  For example, UsableResourceWhenNeeded previously defined isOf ObjectUsableWhenNeededAndAuthorized (below) [25], and can be specialized to UsableWebServiceWhenAccessed (see Figure 3).

  SpecPattern *Achieve*[ObjectUsableWhenNeededAndAuthorized]

  FormalDef $\forall ob : Object, ag : Agent,$
  $Needs(ag, ob) \wedge Authorized(ag, ob) \Rightarrow$
  $\Diamond_{<M} \neg Using(ag, ob)$

- *Vocabulary:* Concepts used to define the attack pattern.

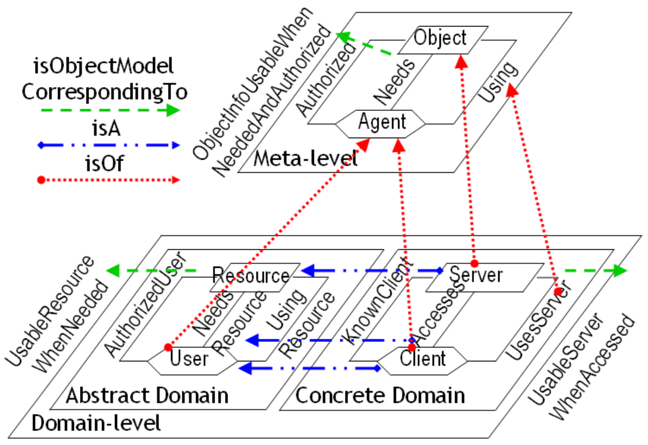

Figure 3. Context for Attack Patterns

- **Goal- and property-specific features:** Formal and informal definitions of abstract anti-goals and abstract domain properties (elements of the attack pattern). Regression of abstract anti-goals through abstract domain properties to build the model are also discussed.

- **Countermeasures:** Reusable anti-goal resolutions.

3.3 Reusing the Library

- **Retrieve:** Get initial anti-goals by negating relevant abstract anti-goals for Security goals. Then, specialize the abstract anti-goals with sensitive objects of the application-specific object model. For example, UnusableResourceWhenNeeded is the negation of an abstract goal and can be specialized to UnusableServerWhenAccessed (see Figure 3).

- **Specialize and Adapt:** Specialize each abstract variable (e.g., objects, agents, relations) of the attack pattern to the specific application domain and adapt if necessary. Specializations must satisfy the following semantic condition:

  Condition $[X(Spec)/P]$ means that the definition of the abstract atomic formula X specialized through $Spec$ covers the definition of the concrete atomic formula P. X is defined on an abstract domain and P is defined on a concrete domain. The definition of $X(Spec)$ is said to cover the definition of P when the set of instances corresponding to the definition of $X(Spec)$ is a superset of the set of instances corresponding to the definition of P.

  For example, let us consider the following definitions:

  – **Resource:** An entity that an agent may request for some usage [2].
    - *Spec: The usage of the resource.*
  – **Server:** A device that a client may request for obtaining network services.
  – **Truck:** A vehicle that transports goods.

  Resource is defined on an abstract domain and can be specialized through *Spec*. Specializations include a resource used for network services or for transporting goods. In the e-commerce domain, $[Resource(Service)/Server]$ holds because the definition of a *Server* covers the definition of a specialized resource that is used to obtain network services (*Spec*=*Services*). In the traffic domain, $[Resource(Transport)/Truck]$ holds because the definition of a *Resource* covers the definition of a truck that is used to transport goods (*Spec*=*Transport*).

- **Specialized Target Usage:** Check whether the specialization is relevant by further refining the generated anti-goal fragments.
  – Ask *WHY* questions and understand the high-level motivations of the attacker.
  – Ask *HOW* questions and find out anti-requirements attributable to attackers and software vulnerabilities observable by the attacker.
  – Specialize reusable countermeasures to the generated anti-goals.

4. Attack Pattern for Replay Attacks

Attack patterns can be defined for every type of attack. To demonstrate our approach, we use an example of replay attacks, i.e., attacks on an authentication systems by recording and replaying previously played information [8].

A. Defining the Attack Pattern

A.1 Model-Specific Features

A.1.1 Context

Let us first introduce a definition:

Def An object is *replayed* if it was played in the past.

FormalDef $\forall ob : Object,\ \ Replayed(ob) \triangleq$
$\exists ag : Agent, ag' : Agent,\ \ \blacklozenge Plays(ag, ob, ag')$

The specification pattern that attackers may want to compromise to replay objects is the following:

SpecPattern *Maintain*[ReplayedObjectNeverAccepted]

Def Replayed objects should never be accepted by agents.

FormalDef $\forall ob : Object,\ \ Replayed(ob) \Rightarrow$
$\Box\neg\exists ag : Agent,\ \ Accepts(ag, ob)$

The specification pattern can be instantiated in a Message abstract domain:

Definition Instantiates definition of *replay*.

Def A message is said to be *fresh* if it was never sent in the past.

FormalDef $\forall m : Mess,\ \ Fresh(m) \triangleq$
$\neg\exists s : Sender, r : Receiver,\ \ \blacklozenge Sends(s, m, r)$

AbsGoal *Maintain*[UnfreshMessageNeverAccepted]

Instantiates ReplayedObjectNeverAccepted

Def An unfresh message should never be accepted by a receiver.

FormalDef $\forall m : Mess,\ \ \neg Fresh(m) \Rightarrow$
$\Box\neg\exists r : Receiver,\ Accepts(r, m)$

The specification pattern could also be instantiated in other abstract domains such as a Ticketing abstract domain. Specializations for such an abstract domain could be to replay a used train ticket or a used concert ticket.

In this section, we focus on the Message abstract domain and build an attack pattern that compromises the UnfreshMessageNeverAccepted abstract goal.

A.1.2. Vocabulary

The concepts used to define the attack pattern fragment are illustrated on the state chart of Figure 4. Note that for space limitations, some obvious definitions are not provided.

Channel: Channel through which agents communicate.

- *Spec: The kind of channel (e.g., LAN, post, phone).*

Fresh: Never sent in the past.

- *Spec: The kind of message (e.g., packet, mail, voice).*

Accepts: Message being accepted by some receiver.

- *Spec: The accepting condition (e.g., no virus, fresh).*

Records: Message being recorded by some attacker.

- *Spec: The kind of message.*

Listens: Listening messages on some channel.

- *Spec: The kind of channel.*

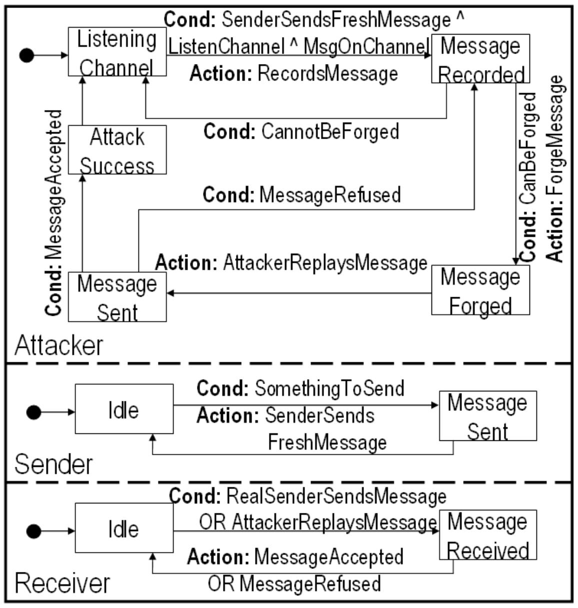

Figure 4. State Chart for Replay Attacks

Forges: Transform a message so that it can be accepted by a receiver even if it is not fresh.

- *Spec: The forging process (e.g., decrypt, modify).*

A.2. Goal- and Property-Specific Features

The attack pattern for replay attacks (see Figure 5) is inspired from the taxonomy proposed by Syverson [23], and is consistent with the list of replay attacks proposed by Gong [6]. By consistent, we mean that the goals of replay attacks defined in [23] and [6] can be modeled with the proposed attack pattern.

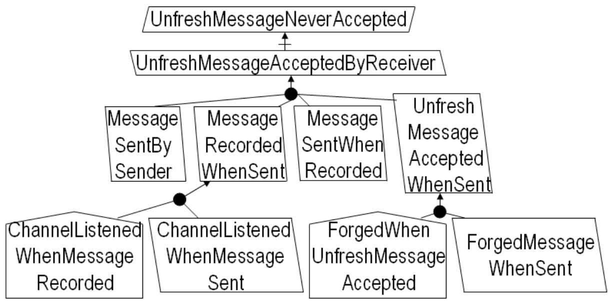
UnfreshMessageNeverAccepted
UnfreshMessageAcceptedByReceiver
Message SentBy Sender
Message Recorded WhenSent
Message SentWhen Recorded
Unfresh Message Accepted WhenSent
ChannelListened WhenMessage Recorded
ChannelListened WhenMessage Sent
ForgedWhen UnfreshMessage Accepted
ForgedMessage WhenSent

Figure 5. Attack Pattern for Replay Attacks

The root of the attack pattern is the negation of the UnfreshMessageNeverAccepted abstract goal:

AbsAntiGoal *Achieve*[UnfreshMessAcceptedByReceiver]

Def An unfresh message is accepted by some receiver.

FormalDef $\Diamond\exists m : Mess, r : Receiver$
$\neg Fresh(m) \wedge \Diamond Accepts(r, m)$

If an attacker wants to replay a message that is accepted by some receiver, (G1) a legitimate sender has to send a message to some receiver, (G2) the attacker has to record that message, (G3) the attacker has to replay the message to some receiver, (G4) the receiver has to accept the replayed message. These four goals are milestones if an attacker wants to successfully replay a message. Instantiating the milestone pattern [9] to sensitive objects of the replay context, we formally refine the UnfreshMessageAcceptedByReceiver abstract anti-goal:

AbsAntiGoal *Achieve*[MessageSentBySender]
Def A message is sent by a sender to some receiver.
FormalDef $\Diamond \exists m : Mess, s : Sender,$
$r : Receiver, \ \ Sends(s, m, r)$

AbsAntiGoal *Achieve*[MessageRecordedWhenSent]
Def A message is recorded by an attacker when it is sent by a sender to some receiver.
FormalDef $\exists m : Mess, s : Sender, r : Receiver$
$Sends(s, m, r) \Rightarrow \Diamond \exists a : Attacker, Records(a, m)$

AbsAntiGoal *Achieve*[MessageSentWhenRecorded]
Def An attacker sent a message to some receiver when he recorded it.
FormalDef $\exists m : Mess, a : Attacker, r : Receiver$
$Records(a, m) \Rightarrow \Diamond Sends(a, m, r)$

AbsAntiGoal *Achieve*[UnfreshMessAcceptedWhenSent]
Def An unfresh message should be accepted when an attacker sent it to some receiver.
FormalDef $\exists m : Mess, a : Attacker, r : Receiver$
$Sends(a, m, r) \Rightarrow \neg Fresh(m) \wedge \Diamond Accepts(r, m)$

Let us now refine the MessageRecordedWhenSent abstract anti-goal by regression. A necessary precondition for the target predicate $Records(a, m)$ is that the attacker should listen messages that are on a channel. This precondition is stated in the following abstract domain property:

AbsDomProp ChannelListenedWhenMessRecorded
Def An attacker listens some channel on which messages are transferred if he records a message.
FormalDef $\exists m : Mess, a : Attacker$
$Records(a, m) \Rightarrow \exists c : Channel$
$Listens(a, m) \wedge On(m, c)$

Instantiating the 1-step regression pattern [27] to regress MessageRecordedWhenSent through ChannelListenedWhenMessageRecorded yields the following abstract anti-goal:

AbsAntiGoal *Achieve*[ChannelListenedWhenMessSent]
Def A channel is listened when a sender sends a message to some receiver.
FormalDef $\exists m : Mess, s : Sender, r : Receiver$
$Sends(s, m, r) \Rightarrow \Diamond \exists a : Attacker, c : Channel$
$Listens(a, c) \wedge On(m, c)$

Let us now refine the UnfreshMessageAcceptedWhenSent abstract anti-goal by regression. A necessary precondition for the acceptance of an unfresh message is that the attacker can transform the message so that it is accepted by a receiver (i.e., it can be forged). This precondition is stated in the following abstract domain property:

AbsDomProp ForgedWhenUnfreshMessageAccepted
Def A message was forged by some attacker in the past messages if a receiver accepts an unfresh message.
FormalDef $\exists m : Mess, r : Receiver$
$\neg Fresh(m) \wedge \Diamond Accepts(r, m) \Rightarrow$
$\exists a : Attacker, \ \ \blacklozenge Forges(m, a)$

Instantiating the 1-step regression pattern to regress UnfreshMessageAcceptedWhenSent through ForgedWhenUnfreshMessageAccepted yields the following abstract anti-goal:

AbsAntiGoal *Achieve*[ForgedMessageWhenSent]
Def A message was forged by an attacker in the past when he sent a message to some receiver.
FormalDef $\exists m : Mess, a : Attacker, r : Receiver$
$Sends(s, m, r) \Rightarrow \blacklozenge Forges(m, a)$

A.3. Countermeasures

Countermeasures against replay attacks include the following [23]: (1) freshness mechanisms (e.g., messages should be tied with a unique identifier), (2) introduce asymmetry (e.g., avoid the man-in-the-middle attack due to protocol symmetry), (3) tying messages to a particular use at a particular time (e.g., messages should be tied to a particular protocol run rather than to a particular epoch. Messages from different protocol runs would therefore be revealed), (4) authentication of message's emitter and recipient (e.g., cryptographically bind the name of a message originator to the message).

B. Reusing the Attack Pattern

Let us now illustrate two specializations: an e-commerce domain (Section B.1) and in the mailing domain (Section B.2). We also specialized the attack pattern in other case studies [9] not illustrated in this paper due to space limitations.

B.1. E-Commerce Domain

Let us consider the e-commerce system, and more specifically the ordering procedure. A client orders a product via a transaction sent to the account manager. The account manager then chooses to commit or to reject the sent transaction. An objective of the system is to avoid that transactions are committed twice. A malicious attacker might want to compromise that goal by replaying a modified transaction so that a known client pays for his transaction.

B.1.1. Retrieve

The definition of freshness is first specialized to the E-commerce domain:

Definition Specializes definition of *fresh*.
Def A transaction is said to be *new* if it was never used for an order of some client in the past.
FormalDef $\forall t : Trans, \ \ New(t) \triangleq \neg \exists c : Client,$
$tm : TrManager, \ \blacklozenge Orders(c, t, tm)$

Let us get an initial anti-goal by specializing the UnfreshMessageNeverAccepted abstract anti-goal to the e-commerce domain.

AbsGoal *Maintain*[OldTransactionsNeverCommitted]
Specializes UnfreshMessageNeverAccepted
Def An old transaction should never be committed by the transaction manager.
FormalDef $\forall t : Trans,\ \neg New(t) \Rightarrow \Box\neg\exists tm : TrManager,\ Commits(tm, t)$

B.1.2. Specialize and Adapt

The dynamic behavior of the system is illustrated on Figure 6 (Figure 6 isA Figure 4). Specializations of concepts defined in Section A.1.2. are the following:

[Channel(Network)/LAN]
[Accepts(isTransaction)/Commits]
[Records(Transaction)/Sniffes]
[Listens(Network)/Eavesdrops]
[Fresh(Transaction)/New]
[Forges(ModifyTransactionNumber)/Modifies]

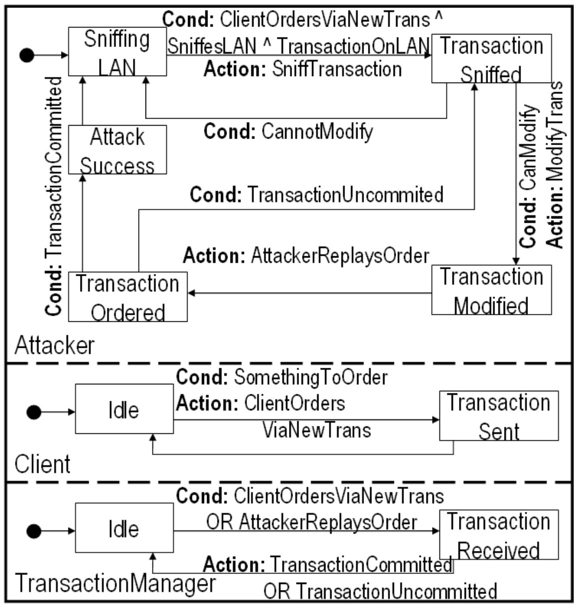
Cond: ClientOrdersViaNewTrans ^ SniffesLAN ^ TransactionOnLAN
Action: SniffTransaction
Sniffing LAN
Transaction Sniffed
Attack Success
Cond: CannotModify
Cond: TransactionUncommited
Cond: TransactionCommitted
Cond: CanModify
Action: ModifyTrans
Action: AttackerReplaysOrder
Transaction Ordered
Transaction Modified
Attacker
Cond: SomethingToOrder
Action: ClientOrders ViaNewTrans
Idle
Transaction Sent
Client
Cond: ClientOrdersViaNewTrans OR AttackerReplaysOrder
Idle
Transaction Received
Action: TransactionCommitted OR TransactionUncommitted
TransactionManager

Figure 6. Specialized State Chart

The anti-model fragment specialized from the attack pattern of Figure 5 is illustrated on Figure 7.

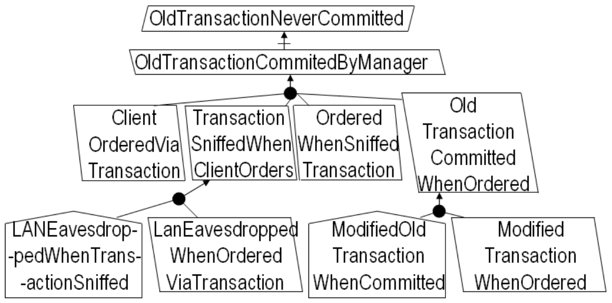
OldTransactionNeverCommitted
OldTransactionCommitedByManager
Client OrderedVia Transaction
Transaction SniffedWhen ClientOrders
Ordered WhenSniffed Transaction
Old Transaction Committed WhenOrdered
LANEavesdrop--pedWhenTrans--actionSniffed
LanEavesdropped WhenOrdered ViaTransaction
ModifiedOld Transaction WhenCommitted
Modified Transaction WhenOrdered

Figure 7. Specialized Attack Pattern

The attack pattern was proved once and for all complete. Specializing the attack pattern is a sufficient condition to build a complete anti-model fragment. Let us illustrate this though one regression of Figure 7:

AntiGoal *Achieve*[TransSniffedWhenClientOrder]
Specializes MessageRecordedWhenSent
Def Transactions that the client sent for ordering should be sniffed.
FormalDef $\exists t : Trans, c : Client, tm : TrManager$ $Orders(c, t, tm) \Rightarrow \Diamond\exists a : Attacker,\ Sniffs(a, t)$

A necessary precondition for the fact that the attacker sniffs a transaction is that he can eavesdrop the LAN where transactions are transmitted. This precondition is stated in the following domain property:

DomProp LANEavesdroppedWhenTransactionSniffed
Specializes ChannelListenedWhenMessageRecorded
Def An attacker eavesdrops a LAN where transactions are transmitted if he sniffs transactions.
FormalDef $\forall t : Trans, a : Attacker$ $Sniffs(a, t) \Rightarrow \exists l : LAN$ $Eavesdrops(a, l) \wedge TransmittedOn(t, l)$

Instantiating the 1-step regression pattern to regress TransactionSniffedWhenClientOrder through LAN-EavesdroppedWhenTransactionSniffed yields the following anti-goal:

AntiGoal *Achieve*[LANEavesdropWhenOrderedViaTrans]
Specializes ChannelListenedWhenMessageSent
Def An attacker eavesdrops a LAN where transactions are transmitted when a client sent a transaction.
FormalDef $\exists t : Trans, c : Client, tm : TrManager$ $Orders(c, t, tm) \Rightarrow \Diamond\exists l : LAN, a : Attacker$ $Eavesdrops(a, l) \wedge TransmittedOn(t, l)$

B.1.3 Specialized Target Usage

The generated fragment of anti-model must now be further refined. We must understand *why* an attacker would want to replay a transaction, *how* he would do it, and *what* are the countermeasures on such attack.

- ***Why?*** An attacker may want the client to pay several times (e.g., revenge or to smeer the reputation of e-commerce). He may also want a valid client to pay his orders.
- ***How?*** Normally a machine on a LAN accepts messages only destined for itself, but when the machine is in promiscuous mode, it reads all information, regardless of its destination. This enables the attacker to read transactions on the LAN.
- ***What?*** (a) Enforce transactions to be certified by a third party so that the client is sure that the transaction is not replayed by an attacker. (b) Forbid promiscuous modes on the LAN (c) Bind transactions to time and protocol run.

B.2 Mailing System

Let us consider the mailing system, and more specifically the dispatching procedure at the post office. An objective of the system is to avoid dispatching mail of senders that did not pay their postage.

B.1.1 Retrieve

The definition of freshness is first specialized to the mailing domain:

Definition Specializes definition of *fresh*.

Def Postage is said to be *paid* if a mail with that postage was never posted in the past.

FormalDef $\forall m : Mail,\ Paid(m.postage) \triangleq \neg\exists s : Sender, r : Receiver, p : Post\ \blacklozenge(Posts(s,m,r) \wedge InChargeOf(p,m))$

Let us get an initial anti-goal by specializing the UnfreshMessageNeverAccepted abstract anti-goal to the mailing domain.

AbsGoal *Maintain*[UnpaidMailNeverDispatched]

Specializes UnfreshMessageNeverAccepted

Def A mail with unpaid postage should never be dispatched by the post.

FormalDef $\forall m : Mail,\ \neg Paid(m.postage) \Rightarrow \Box\neg\exists p : Post,\ Dispaching(p,m)$

B.1.2. Specialize and Adapt

The dynamic behavior of the system is illustrated on Figure 8 (Figure 8 isA Figure 4). Note that some adaptation was necessary to match the attack pattern.

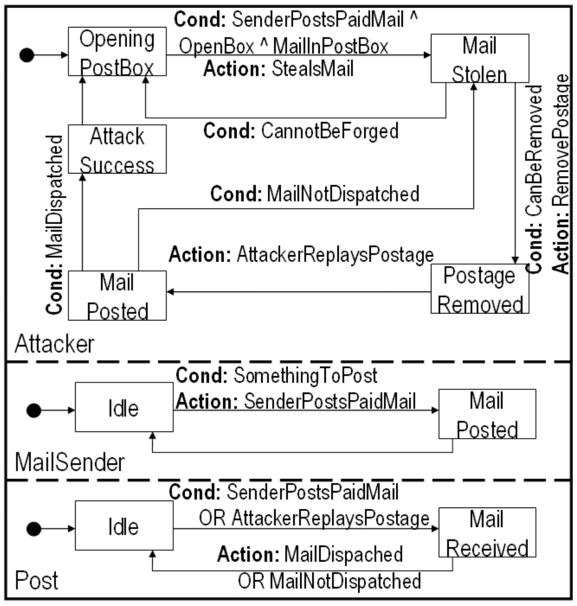

Figure 8. Specialized State Chart

The specializations of concepts defined in Section A.1.2 are the following:

[Channel(PostBox)/PostBox]
[Accepts(isPaid)/Dispatching]
[Records(Mail)/Steals]
[Listens(PostBox)/Opens]
[Fresh(Mail)/Paid]
[Forges(RemovePostage)/Removes ∧¬ Obliterated]

The fragment of anti-model specialized from the attack pattern of Figure 5 is illustrated on Figure 9. Refinements are similar to those of the attack pattern.

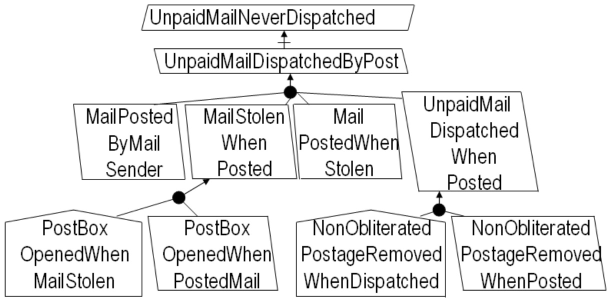
UnpaidMailNeverDispatched
UnpaidMailDispatchedByPost
MailPosted ByMail Sender
MailStolen When Posted
Mail PostedWhen Stolen
UnpaidMail Dispatched When Posted
PostBox OpenedWhen MailStolen
PostBox OpenedWhen PostedMail
NonObliterated PostageRemoved WhenDispatched
NonObliterated PostageRemoved WhenPosted

Figure 9. Specialized Attack Pattern

B.1.3. Specialized Target Usage

The generated fragment of the anti-model must now be further refined. We must answer the *why*, *how*, and *what* questions.

- ***Why?*** An attacker may want to replay postage for multiple reasons: don't pay postage anymore, wants the postoffice to lose money, want to smeer the reputation of the postoffice, want to sell the postage to individuals to make money.
- ***How?*** In order to open the postbox, the attacker may break it, or put his hand in the mail slot. To remove the postage, the attacker may put the mail in water such as philatelists do.
- ***What?*** (a) Enable companies to pre-pay their mails, (b) Lock postboxes (c) Postboxes should not be placed in dangerous streets in order to avoid vandalism, (d) Use sticking postages (they are more difficult to remove without altering them), (e) Raise the price of postage often (this reduces anti-goal occurrences because the global value of mails in postboxes is less attractive), (f) Train post employees to see if the postage are replays.

5. Related Work

The HAZOP (hazard and operability) methodology relies on a set of seven guidewords [18]. They should be applied to all the important parameters of the process under study to identify failure nodes in risk trees. For example, the guideword `MORE` means that there is more of any relevant physical property than there should be. Each guideword has specializations for particular application domains. These specializations are called deviations. The main differences between their technique and ours is (1) hazards are not regressed through reusable properties, (2) rationale of hazards is not obvious in HAZOP.

Many desirable properties of requirements specifications for process control systems have been defined by Jaffe et al. [12]. These reusable properties are listed in safety checklists. The objective is to formally define important properties that must be reflected in the requirements specifications. If the properties are satisfied by the current requirements, the system is said to be safe. If some properties are not satisfied, one should determine criteria that would imply the unsatisfied properties. Properties and their corresponding criteria are reusable. They should be specialized to an application-specific domain at reuse time. The main difference between safety checklists and attack patterns is that lists are not formally structured and that malicious intentions cannot be captured.

Attack trees [22] provide a formal way of describing attacks that could be done on a system. Basically, one represents attacks against a system in a tree structure, with the threat as the root node. Different ways of causing that threat are represented as leaf nodes. When the tree is constructed, one should assign boolean values to each node (e.g., possible/impossible, easy/difficult, expensive/inexpensive, intrusive/nonintrusive.) Attack trees can be reused in multiple contexts and provide a way to think about security. The main difference between attack trees and our technique is that (1) attack trees are not goal-anchored, it is therefore difficult to understand malicious intentions, (2) attack trees have no support for identifying the root threat node, and (3) threats used to build the tree are application-specific, they are thus not meant to be reused.

Threat descriptions are composed with a problem frames [11] representation of functional requirements, giving vulnerabilities [7]. Vulnerabilities are ameliorated by security requirements. Security requirements are expressed as constraints on the functional requirements. They are thereby a part of the specification process, comparable with other constraints. The main differences between their technique and attack patterns is that (1) threat descriptions are not meant to be reused, (2) they are informally defined, (3) there is no model of attacker agents, (4) there is no understanding of high-level malicious intentions, and (5) there are no heuristics for threat derivation.

Cheng et al. [13] describe how an approach similar to architectural design patterns [5] can be applied to reuse problem frames [11], and object-oriented models. They term their reusable patterns *requirements patterns*. The term *requirements patterns* is confusing because their patterns are akin to design patterns and not requirements patterns. The main difference between requirements patterns and our technique is that requirements patterns are domain-level patterns. At the other hand, abstract domain properties are defined on an abstract domain and can be specialized at reuse time.

6. Conclusions

This paper has presented formal reuse-based techniques that support safe, and secure goal-oriented specifications. Goal-oriented specifications model systems in terms of concepts including goals and domain properties. Commonalities between these concepts have been extracted from several case studies. The extracted commonalities have been captured in models defined on abstract domains. The main contribution has been to build up systematic techniques to reuse these models in the context of threat analysis. Once specialized to application-specific vocabulary, these models provide a useful basis for building *high quality* specifications, that is, complete, accurate, and secure specifications. Further, this paper enriches the KAOS framework through the following contributions:

- We have developed a description pattern for defining a library of reusable attack patterns.
- We have proposed an attack pattern for replay attacks. Reusable countermeasures were proposed as well. Other attack patterns were developed in [9].
- We have assessed the quality of our reusable models on several case studies.

Attack patterns have the following strengths:

- Obstacle analysis is complete iff we have a complete set of obstacles. Attack patterns enhance the completeness of a set of obstacles because one can start modeling the world with a predefined set of domain properties that can potentially be specialized.
- Attack patterns are complete because they are derived using proven complete refinement patterns [27].
- The underlying mathematics for proving completeness at specialization time are hidden.
- The techniques can improve the effectiveness and time interval of the requirements elicitation process. Indeed, one starts the elicitation with predefined attack patterns.
- For each model, we described the context in which it should be considered for specialization.
- We defined a semantic matching condition to systematically derive specializations from abstract entities, agents, and relationships.

The presented techniques have, however, the following weaknesses:

- Abstract domain properties and abstract anti-model fragments cannot be specialized in all domains and are not universally applicable for all contexts.
- Reusable specifications are subject to interpretation. They can lead to confusions and ambiguities.
- There exist assumptions about the specialization, the context, the domain, or the way obstacles/anti-goals must be regressed that might not perfectly fit with a particular real-world problem.

- Completeness of the definitions is impossible to prove. They may be insufficient in some contexts, which will result in overlooking important obstacles/anti-goals.

The techniques presented in [9] require further extensions in several directions.

- Enlarge the libraries and assess their quality on case studies.
- Define attack patterns for each anti-goal that compromises a specification pattern for Security goals [25].
- Develop techniques that enable one to automatically determine if a reusable specification can be specialized to a specific system.
- Organize abstract attack patterns in categories derived from high-level malicious intentions. The goal would be to facilitate reuse.
- Define a list of keywords for theories and abstract anti-model fragments in order to facilitate reuse.

References

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

[2] R. Darimont. *Process Support for Requirements Elaboration*. Phd Thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, 1995.

[3] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Foundations of Software Engineering*, pages 179–190, 1996.

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. *In Proceedings of the 21st International Conference on Software Engineering*, (UM-CS-1998-035), May 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional; 1st edition, 1995.

[6] L. Gong. Variations on the themes of message freshness and replay – or the difficulty of devising formal methods to analyze cryptographic protocols. In *Proceedings of the Computer Security Foundations Workshop VI*, pages 131–136, Los Alamitos, California, 1993. IEEE Computer Society Press.

[7] C. B. Haley, R. C. Laney, and B. Nuseibeh. Deriving security requirements from crosscutting threat descriptions. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 112–121, New York, NY, USA, 2004. ACM Press.

[8] N. Haller and R. Atkinson. On internet authentication, October 1994.

[9] L. A. Hermoye. Master Thesis, A Reuse-Based Approach To Security Requirements Engineering, June 2006.

[10] N. Iscoe, G. B. Williams, and G. Arango. Domain modeling for software engineering. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 340–343, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[11] M. Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[12] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Trans. Softw. Eng.*, 17(3):241–258, 1991.

[13] S. Konrad and B. H. C. Cheng. Requirements patterns for embedded systems. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 127–136, Washington, DC, USA, 2002. IEEE Computer Society.

[14] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[15] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[16] E. Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. Phd Thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, May 2001.

[17] N. G. Leveson. Software safety: why, what, and how. *ACM Comput. Surv.*, 18(2):125–163, 1986.

[18] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[19] Z. Manna and A. Pnueli. *The Temporal Logic of Concurrent and Reactive Systems: Specification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.

[20] C. Potts. Using schematic scenarios to understand user needs. In *DIS '95: Proceedings of the conference on Designing interactive systems*, pages 247–256, New York, NY, USA, 1995. ACM Press.

[21] D. T. Ross. Structured analysis: A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(1):16–34, 1977.

[22] B. Schneier. *Secret and Lies: Digital Security in a Networked World*. John Wiley and Sons, 2000.

[23] P. Syverson. A taxonomy of replay attacks. In *Proceedings of the Computer Security Foundations Workshop VII, Franconia NH*, Los Alamitos, CA, USA, 1994. IEEE CS Press.

[24] A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 5–19, New York, NY, USA, 2000. ACM Press.

[25] A. van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 148–157, Washington, DC, USA, 2004. IEEE Computer Society.

[26] A. van Lamsweerde. *Goal-Oriented Requirements Engineering – From System Objectives to UML Models to Software Specifications*. Wiley, to appear, 2007.

[27] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, 2000.