2010 Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement

Model-based Argument Analysis for Evolving Security Requirements

Thein Than Tun[1], Yijun Yu[1], Charles Haley[1], Bashar Nuseibeh[12]
[1]*Department of Computing, The Open University, Milton Keynes, UK*
[2]*Lero, Limerick, Ireland*
{*t.t.tun, y.yu, c.b.haley, b.a.nuseibeh*}*@open.ac.uk*

***Abstract*—Software systems are made to evolve in response to changes in their contexts and requirements. As the systems evolve, security concerns need to be analysed in order to evaluate the impact of changes on the systems. We propose to investigate such changes by applying a meta-model of evolving security requirements, which draws on requirements engineering approaches, security analysis, argumentation and software evolution. In this paper, we show how the meta-model can be instantiated using a formalism of temporal logic, called the Event Calculus. The main contribution is a model based approach to argument analysis, supported by a tool which generates templates for formal descriptions of the evolving system. We apply our approach to several examples from an Air Traffic Management case study.**

***Keywords*-Security argumentation; Requirements Engineering; Evolution; Event Calculus; OpenPF**

I. Introduction

Long-lived software systems often evolve over an extended period of time. Evolution of these systems is inevitable as they need to continue to satisfy changing business needs, new regulations/standards and the introduction of novel technologies.

Such evolution may add, remove, or modify the requirements and parts of the system contexts, and migrate the system from one operating platform to another. These changes may result in requirements that were satisfied in a previous release of a system not being satisfied in the newer release of the system. When evolutionary changes violate security requirements, a system may be left vulnerable to attacks.

As a software system evolves, security concerns need to be analysed in order to evaluate the impact of changes on the requirements. Traditionally, changes that could affect the system security have been handled in an ad-hoc way. For instance, changes are often described in an informal language, whilst the information about the existing system design is partial. Analysing the security impact of changes is therefore a complex challenge.

By adopting a model-based engineering methodology, we propose to investigate such changes using a meta-model of Evolving Security Requirements (ESR). The ESR meta-model has the following characteristics.

1) Security problems are examined at the requirements level
2) Security-specific concepts such as attacker, assets and vulnerabilities are made explicit
3) Argumentation is used to describe relationship between formal and informal descriptions of system artefacts in order to show why the system is thought to be secure
4) Evolutionary changes are considered orthogonal to all artefacts

The ESR meta-model draws on the concepts in requirements engineering, security analysis, argumentation, and software evolution.

In this paper, we show how the ESR meta-model can be used to generate templates for formal descriptions of a system, in a way similar to model-driven code generation. Security problems of software systems are described using the problem diagrams [1]. Our tool OpenPF performs the model-to-text transformation to generate Event Calculus descriptions compliant with the ESR meta-model. As a result, changes in the meta-model can be reflected by the generated Event Calculus descriptions before modifying and feeding into a reasoning engine for security analysis. Since the meta-model is rich enough to express core concepts in security requirements, many of the existing RE languages can be mapped to the meta-model so that the argumentation can be performed to analyse the changes in the evolution.

The major advantage of our approach is that it frees the requirements engineers from having to write mundane parts of the formal descriptions as the system evolve: they can focus on the more abstract and critical part of the descriptions instead.

We have applied the transformations to examples from an Air Traffic Management system. The study shows that a large part of the formal descriptions can be generated using the OpenPF tool, whilst improving the interface with the Event Calculus reasoning engine.

The rest of the paper is organized as follows. Section II relates our work to similar approaches in the literature. Section III presents an illustrative example from the Air Traffic Management (ATM) system. Section IV shows the key parts of the ESR meta-model, which captures the key concepts involved in our analysis. Section V gives the syntax and semantics of the Event Calculus formalism, which is the target language for the transformations. The proposed tool, OpenPF is described in Section VI. Section VII uses

978-0-7695-4086-3/10 $26.00 © 2010 IEEE
DOI 10.1109/SSIRI.2010.36

the running example to go through the transformation and reasoning processes and explains the results. Section VIII gives concluding comments.

II. Related Work

By way of putting our discussions into context, this section provides a survey of related work, which covers existing research into meta-models of requirements, security requirements, change management and formalisation of argumentation.

Meta-models of (security) requirements: Gunter et al. [2] formalise the relationships between key artefacts in requirements engineering, namely, requirement, problem world context, specification, program, and computer. They further define the responsibilities of requirements engineers and software developers. Earlier, Parnas and Madey proposed the four-variable model in [3]. Jureta et al. [4] extend the Jackson-Zave framework in order to include concepts such as beliefs, desires, intentions, and attitudes. These meta-models are not explicit about security requirements: they do not distinguish between users and attackers, for instance.

There are several proposals for meta-models of security requirements engineering: Hartong et al describe a meta-model of misuse cases [5]. Susi et al [6] give a meta model of Tropos. van Lamsweerde [7] suggests that KAOS provide necessary concepts for analysing intentional security requirements. Elahi et al. [8] propose a ontology of security requirements which focuses on the notion of vulnerability. Beydoun et al [9] incorporate security issues into a meta-model of multi-agent systems. Basin et al describe a meta-model-based approach to analysing access control problems. A taxonomy of information security is provided by Savolainen et al in [10]. Lee et al [11] propose a domain ontology based on regulatory documents.

Although these proposals are useful in modelling different aspects of security requirements, they do not capture evolutionary nature of security requirements. Furthermore, the fact that security analysis has to draw on the formal and informal descriptions of the system is often overlooked. Our meta-model is geared towards evolutionary security requirements, and argumentation for bringing together formal and informal descriptions.

Security requirements engineering: Several requirements engineering approaches for security engineering have been proposed [12], [13], and many of the approaches have been surveyed in [14], and here we briefly recall some of them to put this work into context.

In [13], precondition calculus has been used to regressively compute obstacles. Label propagation has been used in goal-oriented requirements engineering in order to analyse satisfaction of security requirements [15]. However, the issue of maintaining the security while introducing change to an existing system has not been extensively studied.

Change Management: The importance of managing change in software development has been recognized for a long time [16]. Several approaches for investigating software evolution have been developed, focusing on various issues including: empirical observation of change [17], feature location [18], version control of development artefacts [19], evolving the software systems during the runtime [20], change recommendations based on historical data and heuristics [21], [22], and economic analysis of change [23].

Although many of these issues are present in the evolution of secure software systems, in this work we focus on the unresolved issue of model-based generating of formal descriptions in order to facilitate automated analysis of change.

Arugmentations: Human intuitions about argumentation have been formalized recently using various logics [24]–[29], in which the use of formal and informal argumentations in the context of requirements engineering have been discussed. For example, argumentation has been useful to document the correctness and completeness of goal decomposition using GSN in [29]. Earlier, an argumentation framework to security requirements has been developed and applied in [30]. This work can be considered as an extension to [30] that addresses the argumentation problem together with tool supported change analysis.

III. ATM Example: Sending Weather Data

One of the key services of ATC systems is to maintain a degree of separation distance between aircrafts. This involves the surveillance of aircrafts in airspaces, and the determination of the flight paths and the separation necessary. Separation distance may vary for a number of reasons, including the type of involved aircrafts, and the stages of journey they are at. For all airborne aircraft in a controlled airspace, human air traffic controllers (ATC operators) on the ground need to know where each aircraft is in the airspace in order to determine the flight paths.

One of the main requirements of the ATC systems is to ensure that a certain separation distance (SD) is maintained between aircrafts in the airspace controlled by an ATC system. The SD requirement needs to be satisfied by the system at all times. Furthermore, ATC operators can send various data and directions to the aircrafts using ATM system. Some of the data can potentially change the flight paths and are therefore security-related. One of such requirements considered in the rest of the paper is about sending weather data to the aircraft.

During the analysis of requirements such as this, the requirements engineer will have to describe the behaviour of various parts of the system, including the ATC operator, ATM system, the aircraft, and the pilot, identify the assets and security vulnerabilities in each of the components and their configuration, provide mitigation when necessary and show that the security requirements can be met by the system

through formal/informal arguments. In addition, when the system evolves, the behaviour of the system components and the requirements may change. Therefore, automated generation of partial descriptions of the system components facilitates the analysis process.

IV. META-MODEL

Our ESR meta-model, shown in Figure 1, combines concepts from requirements engineering, evolution and security analysis. Some of the key concepts in the meta-model are explained below.

Evolution Concepts: A model captures a *situation* at a given *time*, which contains a set of *contexts* and a set of *subjects*. The model can generally evolve by modifying the contexts and by introducing new subjects. When the time intervals are sufficiently small, the changes of the model are assumed to be small. Yet the impact of such smaller changes may still cover a large portion of the model. In order to show that important security requirements are satisfied after some change has been implemented, it is necessary to consider how the change should be propagated. In establishing that, the following concepts can be useful.

Requirements Engineering Concepts: A knowledge context has a set of *propositions*. A proposition can be assumed a fact or a rule, which is part of a given *domain* context, or can be supported by a set of propositions in an *argument* context. Both contexts and propositions are concerned with a set of *subject* matters. A subject matter in requirements engineering is either a *resource*, a *process* or an *actor*. A resource is a subject that may have multiple data *states*, a process is a subject that may have control behaviours that can be described by domains representing pre- or post-conditions, and an actor is a subject that may want *requirements* and may conduct some processes. A requirement relates an actor to a number of wanted propositions.

Problem Frames Concepts: The Problem Frames approach [1] emphasises the relationship between three main artefacts: a specification of a system (called machine), within a particular problem world context, satisfies a given requirement. *Fulfils* between a specification S and a requirement R can be represented by a logic entailment relation that $W, S \vdash R$, where W is the contexts in the situation. Some of the domains are physical domains with causal behaviour [1].

Temporal Logic Concepts: The temporal logic we use is a first-order predicate logic with discrete time. It has three main sorts: time, event and fluent (time varying property). Later in the discussion, we will explain how these concepts are used to describe requirements engineering artefacts.

Security and Argumentation Concepts: An *asset* is a resource that has desired value to the stakeholders (actors). It must be protected according to a *security goal* from damages that may be introduced by a potential *attack*. An *attacker* wants to achieve *anti-requirements*, which would *obstruct* the fulfilment of the security goals. An attack exploits *vulnerability* propositions inside the domains. By challenging the domain knowledge with additional propositions, a *rebuttal* is effectively constructed to demonstrate that the security requirements are not achievable under possible attacks.

A *mitigation* may introduce further changes to the domain knowledge such that the satisfaction argument of security requirements is valid again. Both rebuttals and mitigations are forms of arguments in different situations, hereby we choose not to represent them as separate concepts.

Table I: Elementary Predicates of the Event Calculus

Predicate	Meaning
Happens(a, t)	Action a occurs at time t
Initiates(a, f, t)	Fluent f starts to hold after action a at time t
Terminates(a, f, t)	Fluent f ceases to hold after action a at time t
HoldsAt(f, t)	Fluent f holds at time t
$t1 < t2$	Time point $t1$ is before time point $t2$

$$Clipped(t1, f, t2) \stackrel{\text{def}}{\equiv} \exists a, t[Happens(a, t) \wedge t1 \leq t < t2 \wedge Terminates(a, f, t)] \quad \text{(EC1)}$$

$$Declipped(t1, f, t2) \stackrel{\text{def}}{\equiv} \exists a, t[Happens(a, t) \wedge t1 \leq t < t2 \wedge Initiates(a, f, t)] \quad \text{(EC2)}$$

$$HoldsAt(f, t2) \leftarrow [Happens(a, t1) \wedge Initiates(a, f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)] \quad \text{(EC3)}$$

$$\neg HoldsAt(f, t2) \leftarrow [Happens(a, t1) \wedge Initiates(a, f, t1) \wedge t1 < t2 \wedge \neg Declipped(t1, f, t2)] \quad \text{(EC4)}$$

$$HoldsAt(f, t2) \leftarrow [HoldsAt(f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1, f, t2)] \quad \text{(EC5)}$$

$$\neg HoldsAt(f, t2) \leftarrow [\neg HoldsAt(f, t1) \wedge t1 < t2 \wedge \neg Declipped(t1, f, t2)] \quad \text{(EC6)}$$

Figure 2: Event Calculus Domain Independent rules

V. THE EVENT CALCULUS

t First introduced by Kowalski and Sergot [31], the Event Calculus (EC) is a system of logical formalism, which draws from first-order predicate calculus. It can be used to represent actions, their deterministic and non-deterministic effects, concurrent actions and continuous change [32]. We chose EC as our formalism, because it is suitable for describing and reasoning about event-based temporal systems such as the Air Traffic Management systems. Several variations

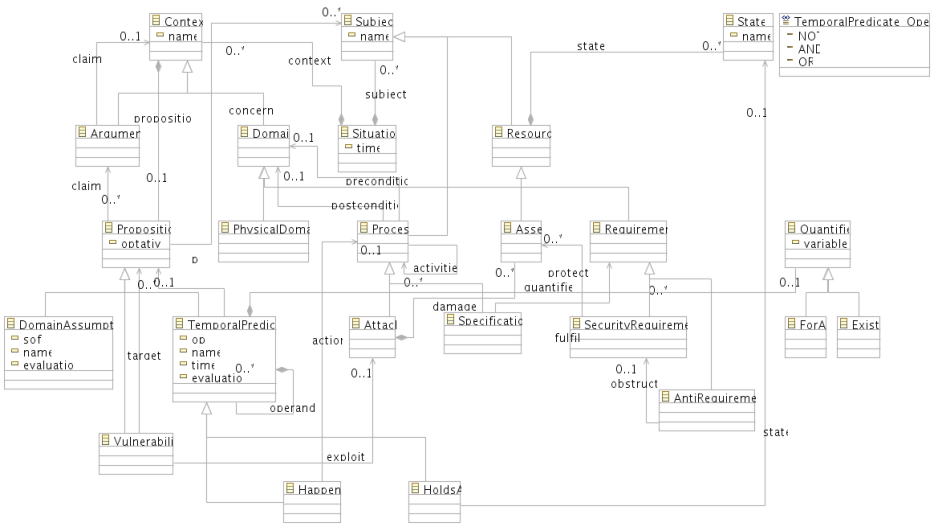


Figure 1: A ESR meta-model for Analysing Evolving Security Requirements

of EC have been proposed, and the version we have adopted here is based on the discussions in [33].

The calculus relates events and event sequences to *fluents*, or time-varying properties, which denote states of a system. Table 1, based on [33], gives the meanings of the elementary predicates of the calculus we use in this paper. The domain-independent rules in Fig. 2, taken from [33], state that: $Clipped(t1, f, t2)$ is a notational shorthand to say that the fluent f is terminated between times $t1$ and $t2$ (EC1), $Declipped(t1, f, t2)$ is another notational shorthand to say that the fluent f is initiated between times $t1$ and $t2$ (EC2), fluents that have been initiated by occurrence of an event continue to hold until occurrence of a terminating event (EC3), fluents that have been terminated by occurrence of an event continue not to hold until occurrence of an initiating event (EC4), and truth values of fluents persist until appropriate initiating and terminating events occur (EC5 and EC6).

Following Shanahan, we assume that all variables are universally quantified except where otherwise shown. We also assume linear time with non-negative integer values. In EC, we follow the rules of circumscription in formalizing commonsense knowledge [34], by assuming that all possible causes for a fluent are given in the database and our reasoning tool cannot find anything except those causes.

A. Describing ESR Models using the EC

In our approach to specifying event-based systems, requirements are constraints on the combinations of fluents capturing the required states of resources. Problem world domains capture behaviours of processes that are described by causality between pre- and post-conditions, called domain obligations. We now define them more formally.

Definition 5.1: Observations consist of a finite conjunction of $(\neg)HoldsAt$ predicates. *Reference phenomena* (Γ) are observations describing the given state of the resources or given action of the processes, while *controlled phenomena* (Γ') are observations describing the desired state of the resources or triggering of actions. They are also captured by the pre/post conditions of the process. A *requirement* is expressed either as

- ground observations Γ', without any reference to the given state of the resource or given action of the processes, or
- as a relationship between the reference and the controlled phenomena, such as a *constraint* of the form $\Gamma \rightarrow \Gamma'$, or an *action precondition axiom* of the form $(\neg)Happens(f1, t) \rightarrow \Gamma'$ where the antecedent is an occurrence of an action in the system (for example, to say that when an event a1 happens at time t, the fluent $f1$ must be true at $t1$).

For example, the requirement $HoldsAt(AircraftOnGround, t) \wedge 0 \leq t \leq 9$ says that the aircraft must be on the ground between the timepoints 0 and 9 range; the requirement $HoldsAt(Airborne, t) \rightarrow HoldsAt(TransponderOn, t)$ says that as long as the aircraft remains airborne, the transponder must be on; and the requirement $Happens(BreachSD, t) \wedge \neg Happens(Clearance, t2) \wedge t \leq t1 \leq t2 \rightarrow HoldsAt(AlarmRaised, t1)$ says that as soon as the separation distance is breached, the alarm should be raised until the clearance happens.

Since requirements tend to be about desired properties of the system over time, they will be formulated in terms of fluents holding, rather than in terms of (instantaneous) event occurrences. In other words, they specify 'what' the

system should achieve rather than 'how' the specification or processes achieve it.

Anti-requirements are requirements of an attacker, and the system must ensure that those requirements are not satisfiable. Since anti-requirements are also requirements, they can expressed in a similar way. For instance, $\exists t \cdot HoldsAt(AircraftsCollide, t)$, is a requirement of an attacker who wants to collide some aircraft.

B. Specifications/Obligations

We assume that a domain has the *potential* to generate instances of events it controls: it may generate all, some or none, of the event instances, even if that leads to undesired states.

In order to see its significance, we will briefly discuss some possible alternatives. In one other option, we may assume that the specification by default does not generate any event: In that case, if the specification describes events that must be generated, the specification is closed; if the specification describes events that may be generated, it is impossible to prove any "liveness" property. Similarly, we may assume that the specification by default generates all events, and specifications should restrict certain event occurrences. This again is not satisfactory for reasons similar to those given above. Therefore we have to categorize events into "must", "must not" and "may".

We therefore recognize three modes of describing specifications: in the *Act* mode, we describes events that must be generated (using Happens); and in the *Prohibit* mode, we describe events must not be generated (using Prohibit).

$$Prohibit(a, t1, t2) \stackrel{\text{def}}{\equiv} \neg\exists t \cdot Happens(a, t) \wedge t1 \leq t \leq t2 \quad \text{(EC7)}$$

In the third implicit mode, all other possible events are left undescribed because their occurrence or non-occurrence is assumed not to affect the requirement satisfaction.

In a *closed* specification, the union of "must" and "must not" covers all possible event sequences of the software (there is no event that may or may not happen). In a *partially open* specification, occurrence or non-occurrence of at least one event is not described: therefore there can be more than one specification that fulfil the requirement.

Definition 5.2: A *specification* is expressed as a finite conjunction of the *event occurrence constraints* (Ψ) of the form $(\neg)Happens(a_1, t) \wedge (\neg)HoldsAt(f, t) \rightarrow (\neg)Happens(a_2, t)$ where a_1, a_2, t, and f are terms for the action, time point, and fluent respectively.

Definition 5.3: A *domain description* in our approach to ATM is expressed as event-to-condition and condition-to-event causality. The first causality deals with what happens to the fluents when events occur, and the second causality deals with the domain properties that lead to the occurrence of certain events. In the Event Calculus, the event-to-condition causality is described as a finite conjunction *positive effect axioms* and *negative effect axioms* (Σ) of the form $Initiates(a, f, t) \leftarrow \Pi$ or $Terminates(a, f, t) \leftarrow \Pi$ where Π has the form $(\neg)HoldsAt(f_1, t) \wedge \cdots \wedge (\neg)HoldsAt(f_n, t)$ and t, and f_1 to f_n are terms for the time and fluents respectively. The condition-to-event causality is described as a finite conjunction of *trigger axioms* (Δ_2) of the form $Happens(a, t) \leftarrow \Pi$. For example, the following statement says that if the aircraft has transponder, an occurrence of the event interogateTransponder has an effect of making BroadcastACInfo true.

$$Initiates(interogateTransponder, BroadcastACInfo, t) \leftarrow HoldsAt(HasTranspnder, t)$$

Similarly, the following statement says that the fluent OperatorHasWeatherInfo on becoming true, generates the event sendWeatherInfo because of the functionality `SendWeatherInfo`.

$$Happens(sendWeatherInfo, t) \leftarrow HoldsAt(OperatorHasWeatherInfo, t) \wedge \neg HoldsAt(OperatorHasWeatherInfo, t-1)$$

Note that the condition $\neg HoldsAt(OperatorHasWeatherInfo, t-1)$ is necessary to prevent stuttering of the event sendWeatherInfo when the fluent OperatorHasWeatherInfo holds continuously.

C. Important Properties

Before we describe the important properties, we will make certain assumptions clear. First, these have to rely on the consistency of the domain theory Σ and observations Γ and Γ'. Second, we assume uniqueness of fluent and event names, meaning that no two names denote the same thing. This uniqueness axiom is represented by Ω.

A simple specification in this approach is a proactive specification that addresses a subtype of problem known as *Required Behaviour*. In this type of problem, a specification is required to bring about certain states in the system resources, without relying on the feedback from the other processes. In such cases, the basic property of the descriptions we want is:

$$\Sigma \wedge \Psi \models \Gamma'$$

That is, given a theory of physical domains (Σ), a specification (Ψ), and an appropriate deductive system, we want to show that the requirements are satisfied non-trivially.

In more common cases, the system has to rely on the feedback from the environment (Δ_2) and observations about the environment (Γ) .

$$\Sigma \wedge \Gamma \wedge \Delta_2 \wedge \Psi \models \Gamma'$$

D. Analysis

Finding vulnerabilities is done through logical abduction.

1) We first pose a *logical abduction* problem in order to find all constructive hypotheses (Δ_1) explaining how, given the domain theory ($\Sigma \wedge \Gamma \wedge \Delta_2$), the requirement ($\Gamma'$) can be satisfied, i.e.

$$CIRC[\Sigma; Initiates, Terminates] \wedge$$
$$CIRC[\Delta_1 \wedge \Delta_2; Happens] \wedge \Gamma \wedge \Omega \models \Gamma'$$

where Δ_1 is consistent with the domain theory. Δ_1 is a partially ordered sequences of event occurrences that, given the physical domains, leads to the requirement being satisfied. The circumscription operator assumes that no events other than those by Δ_1 and Δ_2 may occur (otherwise the requirement is not satisfied). Therefore, Δ_1 tells us events that must happen and that may happen. Event occurrences that do not appear in Δ_1 must not happen.

However, some of the hypotheses in Δ_1 may not be "realistic": for example, a scenario may assume competence and co-operation of users to a level that cannot be guaranteed. Furthermore, Δ_1 may also contain stuttering events that can be eliminated without affecting requirements satisfactions[1].

2) The developer then identifies the 'unrealistic' hypotheses in Δ_1 and eliminates them by providing further information about the problem world domains. Similarly, event stuttering is removed by adding further constraints (which is then used to weaken the specifications). These are assertions, or mitigation, we want the tool to consider.
3) When no vulnerabilities can be found, there are no "internal" vulnerabilities[2].

   Notice that Ψ is not circumscribed: it will have to make explicit all events that must not happen. Therefore, Ψ describe all events that must happen, and all events that must not happen, the remainder being events that may happen.

E. Event Caclulus Reasoner

We choose Decreasoner to implement the verification for the generated EC rules. Decreasoner translates the EC rules into SAT formulae automatically, and invokes `Rel-Sat` solver to check whether they are satisfiable, given the bounded time range. In principle, abductive process may not terminate if the goal cannot be satisfied; however, since the time range is discrete and bounded, the tool forces a termination when the time limit is reached. Therefore, it is important to choose a reasonably large time range. This range, however, is not enlarged if a counter example can already be found.

[1]This is often called "invariance under stuttering" (for example [35]). Model checking techniques, known as "partial order reduction", exploit this property to reduce state space [36].

[2]These are vulnerabilities that can be found with the model.

```
1  grammar uk.ac.open.problem.Problem with uk.ac.open.Istar
2  import "platform:/resource/openome_model/model/openome_model.ecore" as
       openome_model
3  generate problem "http://open.ac.uk/problem"
4
5  ProblemDiagram: ("problem" ':' description=STRING)?
6          ((nodes+=Node|links+=Link))*;
7
8  Node:
9    name=ID (type=NodeType)?
10   (':' description=STRING)?
11   ("{" (subproblem=ProblemDiagram | "see" "domain" problemRef=[Node] |
12                 istar=Model | "see" "intention" istarRef=[openome_model::
                       Intention] |
13                 (hiddenPhenomena+=Phenomenon (',' hiddenPhenomena+=Phenomenon)*)
14   ) "}")?;
15
16 enum NodeType:
17   REQUIREMENT="R" | MACHINE="M" | BIDDABLE="B" | LEXICAL="X" | CAUSAL="C" |
           DESIGNED="D" | PHYSICAL="P";
18
19 Phenomenon:
20   (type=PhenomenonType)? name=ID (':' description=STRING)?;
21
22 enum PhenomenonType:
23   UNSPECIFIED="phenomenon" | EVENT="event" | STATE="state";
24
25 Link:
26   from=[Node] (type=LinkType) to=[Node] ('{' phenomena+=Phenomenon (',' phenomena
         +=
27   Phenomenon)* '}')? (':' description=STRING)?;
28
29 enum LinkType:
30   INTERFACE="-->" | REFERENCE="~~" | CONSTRAINT="~>";
```

Figure 3: Partial listings of the concrete syntax of the the ESR meta-model in Figure 1

VI. OpenPF

This section explains how Event Calculus templates are generated from Problem Frame diagrams using OpenPF.

A. Concrete Syntax

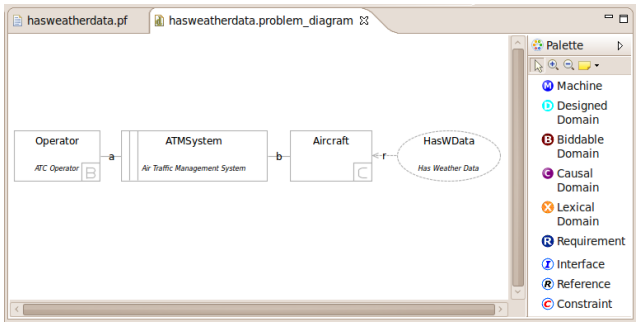
Figure 3 lists the most of concrete syntax for the Problem Frames concepts in the EMF meta-model shown in Figure 1. This concrete syntax is given in order to show the textual representations of the meta-model, and also to indicate that the root concept in the representation is the problem diagram.

The syntax is composed of a number of BNF-like rules, each defines one non-terminal at the left hand side and a number of refinement parsable elements, including both non-terminals and terminals. The words occurring as strings in the rules are treated as keywords, which is only necessary in the concrete syntax. For the same abstract syntax in Figure 1, there can be more than one way to express the concrete syntax. In this example, we try to express it using intuitive keywords consistently. Comparing the graph-based meta-model with the tree-based concrete syntax, one useful feature of `xtext` is to use `ID` to provide shared references inside other types.

An example of the ESR model is given in Figure 4. As the example shows, the syntax for writing a problem diagram is straightforward. First, the name of the problem diagram ("Has Weather Data") is defined (Lines 1-2). "Has Weather Data" is a requirement node (indicated by R) and is identified by HasWData. Aircraft is a causal domain with no

```
problem:
"Has_Weather_Data"

HasWData R: "Has_Weather_Data"

Aircraft C

ATMSystem M: "Air_Traffic_Management_System"

Operator B: "ATC_Operator"

Operator -> ATMSystem: "a"

ATMSystem -> Aircraft: "b"

HasWData -> Aircraft: "r"
```

Figure 4: Full listings of one concrete example of the ESR model



a: O!{InputWeatherData}
b: AS!{SendWeatherData}
r: A!{HasWeatherData}

Figure 5: Problem Diagram: Has Weather Data

description. Air Traffic Management System is a machine domain, whilst ATC Operator is a biddable domain. The interface between the operator and ATM System is named a, and the interface between ATM System and aircraft is named b. The requirement constrains the aircraft domain, and is named r.

B. Diagramming

OpenPF also provides editor and automatic diagramming of problem diagrams from the syntax given above. A problem diagram for the problem of sending weather data is shown in Figure 5, together with a description of the events.

C. Generating Event Calculus Templates

Generating Event Calculus templates from the problem diagram is done using the OMG Text-to-Model (Xtext) and Model-to-Text transformation (Acceleo) framework inside the Eclipse modelling Project. The expression syntax of the Model-to-Text transformation is similar to that of OCL, and the concrete syntax resembles the JavaScript language, except that code generation tags are enclosed by square brackets rather than by XML brackets.

First, it is necessary to determine the output file name and generate a common header to include the predefined

```
1  [module generate('http://open.ac.uk/problem')/]
2  [template public generate(d: ProblemDiagram)]
3  [file (d.description.toString().concat('.ec'), false)]
4  load foundations/Root.e
5  load foundations/EC.e
6      [for (dom: Node | d.nodes)]
7        [for (hidden: Phenomenon | dom.hiddenPhenomena)]
8         [if (hidden.type.toString() ='state')]
9  fluent [hidden.name/]_[dom.name/]()
10        [/if]
11        [/for]
12       [/for]
13       [for (interface: Link | d.links)]
14       [for (shared: Phenomenon | interface.phenomena)]
15                [if (shared.type.toString() = 'event')]
16 event [shared.name/]_[interface.description/]()
17         [/if]
18                [if (shared.toString() = 'state')]
19 fluent [shared.name/]_[interface.description/]()
20         [/if]
21       [/for]
22     [/for]
23     [for (dom: Node | d.nodes)]
24      [if (dom.type.toString() <> 'R')]
25 ;--[dom.name/]--
26 ['['/]time[']'/]
27       [for (interface: Link | d.links)]
28        [if (interface.to = dom)]
29         [for (shared: Phenomenon | interface.phenomena)]
30                  [if (shared.type.toString() = 'event')]
31 Happens([shared.name/]_[interface.description/](),time),
32          [/if]
33                [if (shared.toString() = 'state')]
34 HoldsAt([shared.name/]_[interface.description/](),time),
35         [/if]
36         [/for]
37         [/if]
38        [/for]
39         ->
40       [for (interface: Link | d.links)]
41        [if (interface.from = dom)]
42         [for (shared: Phenomenon | interface.phenomena)]
43                  [if (shared.type.toString() = 'event')]
44 Happens([shared.name/]_[interface.description/](),time+1),
45          [/if]
46                [if (shared.toString() = 'state')]
47 HoldsAt([shared.name/]_[interface.description/](),time+1),
48         [/if]
49         [/for]
50         [/if]
51       [/for].
52      [/if]
53    [/for]
54 range time 0 3
55 range offset 1 2
56 [/file]
57 [/template]
```

Figure 6: Part of listings of the code generation rules: transforming from the ESR meta-model into EC

EC rules (Line 3-5). Names of fluents that are internal to the domains are identified, suffixed with the domain names, and declared (Line 9). Similarly, names of events and fluents shared between domains are also identified, suffixed with the domain names, and declared (Line 16, 19). This suffixing of the fluent and event names ensures that it is possible to trace the results of the reasoning tool back to specific parts of the problem diagrams.

Apart from the common footer (Lines 54-55), the rest of the template is pattern-based. Some of the patterns are now discussed. A typical instance of the domain specification (Definition 5.2) is if event e1 happens at time t, then another event e2 happens at a time after t, t+1 (Lines 25-53). Given the diagram matching this pattern, a transformation is applied to generate the following rule snippet for the template:

```
; -- Domain --
[time] Happens(e1, t) ->
       Happens(e2, t + 1).
```

Here the "[time]" denotes a universally quantified variable. Note that this is a generic template in which one can change the time delay between the two event occurrences. Generally speaking, this pattern will put all events and fluents a domain observes on the left-hand side, and all events and fluents a domain controls on the right-hand side of an implication.

Formulae for event-to-fluent and fluent-to-event causalities are also common (Definition 5.3). For this causality, typically there is a domain with a fluent and two events, where one of the events is observed by the domain, and the other event is controlled by the domain. In such cases, the pattern will produce Initiates and Terminates formulae, and Happens formulae.

As a result of these transformations, syntactically correct EC descriptions are generated, which can be further modified by the tool users before analysing them.

VII. Example Analysis and Results

We now work through the example introduced in Section III. First, we describe the abstract domain behaviour in terms of its obligations. We begin by drawing a simple diagram showing the problem world domains, their relationships and the property the system needs to hold. The diagram (Figure 5) shows that the operator sends weather information to the aircraft using the ATM system.

A. Describing Specifications/Obligations

We then describe the behaviour of the domains in terms of the events they observe and events they control. OpenPF templates mentioned above produces the following descriptions.

```
; -- AS1 --
[time] Happens(InputWeatherData_a(),time) ->
       Happens(SendWeatherData_b(), time+1).
```

When the ATM system observes the input, it will send the weather information to the aircraft at the next time point. Similar descriptions can be generated for the other domains. For the operator, we may assume that when the operator is told about weather information is available, s/he inputs the information to the ATM system at the next time point.

```
; O1 -- Operators --
[time] Happens(TellWeatherInfo,time) ->
       Happens(InputWeatherInfo, time+1).

; A1 -- Aircraft --
[time] Initiates(SendWeatherInfo_f(),
                 HasWeatherInfo_Aircraft(), time).
```

When the aircraft observes the weather information being sent, the aircraft will has the aircraft information.

B. Describing Domain Behaviour

Let us suppose that the operator has the following behaviour.

```
; O2
[time] Initiates(ReceiveWeatherData,
                 WeatherData_Known, time).
; O3
[time] !HoldsAt(WeatherData_Known,time)
         & HoldsAt(WeatherData_Known,time +1) ->
           Happens(InputWeatherData,time + 1).
```

The first statement says that receiving weather data by the operator means that the weather data is known to the operator. The second statement says that as soon as the operator knows weather information, the InputWeatherData event is generated.

In order to define the partial Behaviour of the aircraft, we first define a few additional sorts.

```
fluent Collided(ac,ac)
fluent At(ac,pos)
event Move(ac,pos,pos)
fluent Avoid(pos)
```

The fluent Collided(ac,ac) is true when two aircraft collided; the fluent At(ac,pos) is true when the aircraft is at the position; the fluent Avoid(pos) is true when no aircraft is at the position; and Move(ac,pos,pos) says that the aircraft moves from one position to another.

```
[time,ac,pos,pos1]
Initiates(Move(ac,pos,pos1),At(ac,pos1),time).

[time,ac,pos,pos1]
Terminates(Move(ac,pos,pos1),At(ac,pos),time).

[time,ac,pos,pos1]
Happens(Move(ac,pos,pos1),time) -> (pos<pos1).

[time,pos,pos1,ac]
HoldsAt(At(ac,pos),time) &
HoldsAt(At(ac,pos1),time) -> (pos=pos1).

[time,ac,ac1,pos,pos1]
HoldsAt(At(ac,pos),time+1) & (ac!=ac1) ->
Initiates(Move(ac1,pos1,pos),
          Collided(ac,ac1),time).

[time,ac,ac1,pos,pos1]
!HoldsAt(At(ac,pos),time+1) & (ac!=ac1) ->
Terminates(Move(ac1,pos1,pos),
           Collided(ac,ac1),time).

[time,ac,ac1]
HoldsAt(Collided(ac, ac1),time) -> (ac!=ac1).

[time,ac,ac1]
HoldsAt(Collided(ac, ac1),time) <->
HoldsAt(Collided(ac1, ac),time).

[time,ac,ac1,pos,pos1]
(time=0) -> (HoldsAt(At(ac,pos),time) &
HoldsAt(At(ac1,pos1),time) & (ac!=ac1) ->
(pos!=pos1)).

[time,ac,pos] HoldsAt(Avoid(pos),time) ->
!HoldsAt(At(ac,pos),time).
```

The above formulae describe how planes move along paths, and when happens when planes converge on a position. Depending on the weather information received from the operator, various constraints can be placed on the flight by stating positions that need to be avoided. For instance, the following says that the position 1 should not be on the flight path.

```
HoldsAt(Avoid(1),1).
```

C. Analysing Domain Obligation

First, we can check that there is at least one model of the operator behaviour (O2 and O3) that satisfies its specification (O1). Here the tool finds several models including the following:

```
0
Happens(ReceiveWeatherData(t), 0).
1
Happens(InputWeatherData(g), 1).
2
Happens(InputWeatherData(g), 2).
Happens(SendWeatherData(f), 2).
3
+HasWeatherData_Aircraft().
P
```

At time 1, the operator receives the weather information, which the operator inputs at the next time point. The weather information is sent at time 2, and the aircraft has the weather information at time point 3.

Next, we can check whether the operator behaviour can fail to satisfy its obligation. Again, the tool finds several models showing how the operator can fail to satisfy his/her obligations, including the following:

```
0
WeatherDataKnown_Operator().
Happens(ReceiveWeatherData(g), 0).
1
Happens(ReceiveWeatherData(g), 1).
2
Happens(ReceiveWeatherData(g), 2).
3
P
```

In one of the models, the operator may know the weather information, and still receive weather information, but fails to input the information to the ATM system. This is a case of operator witholding the information. This is a rebuttal generated by the tool.

The rebuttal shows that the domain behaviour allows the operator to input the weather information without being told. This of course poses a security risk, if the operator has malicious intent. This calls for a strengthening of the domain behaviour by stating that the operator will send if and only if s/he was told about the weather information.

```
; O3'
[time] !HoldsAt(WeatherData_Known,time)
        & HoldsAt(WeatherData_Known,time + 1) <->
          Happens(InputWeatherData,time+1).
```

In this case, the domain obligation is weaker than the domain behaviour. Finally as a mitigation to this problem, we can strengthen the domain obligations.

```
; O1'
[time] Happens(TellWeatherData,time) <->
       Happens(InputWeatherData, time+1).
```

Once the domain behaviour and the obligations are strengthened, it is no longer possible to show that the operator can fail to satisfy his/her obligation. The strengthening is mitigation.

The security requirement to prevent collision can be checked in the same way.

VIII. Conclusion

We have investigated some of the challenges of analysing the security impact of evolutionary changes made to software systems. First, we applied a meta-model of evolving security requirements, which draws on concepts in requirements engineering, security analysis, argumentation and software evolution. We instantiated the meta-model using a formalism of temporal logic, called the Event Calculus. We have proposed a tool called OpenPF that generates templates for Event Calculus descriptions of the evolving system, and analyse them using a reasoning tool called Decreasoner. The approach is illustrated with a simple example from an Air Traffic Management system.

Acknowledgement

Financial support of the SecureChange project, funded by the European Union, is gratefully acknowledged. We thank the industrial and academic partners of the SecureChange project for providing case studies and feedback on ideas leading to this work.

References

[1] M. Jackson, *Problem Frames: Analyzing and structuring software development problems*. Addison Wesley, 2001.

[2] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *IEEE Softw.*, vol. 17, no. 3, pp. 37–43, 2000.

[3] D. L. Parnas and J. Madey, "Functional documents for computer systems," *Sci. Comput. Program.*, vol. 25, no. 1, pp. 41–61, 1995.

[4] I. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the core ontology and problem in requirements engineering," in *RE '08: Proceedings of the 2008 16th IEEE International Requirements Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 71–80.

[5] M. Hartong, R. Goel, and D. Wijesekera, "Meta-models for misuse cases," in *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*. New York, NY, USA: ACM, 2009, pp. 1–4.

[6] A. Susi, A. Perini, J. Mylopoulos, and P. Giorgini, "The tropos metamodel and its use," *Informatica (Slovenia)*, vol. 29, no. 4, pp. 401–408, 2005.

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

[8] G. Elahi, E. S. K. Yu, and N. Zannone, "A modeling ontology for integrating vulnerabilities into security requirements conceptual foundations," in *ER*, ser. Lecture Notes in Computer Science, A. H. F. Laender, S. Castano, U. Dayal, F. Casati, and J. P. M. de Oliveira, Eds., vol. 5829. Springer, 2009, pp. 99–114.

[9] G. Beydoun, G. Low, H. Mouratidis, and B. Henderson-Sellers, "A security-aware metamodel for multi-agent systems (mas)," *Inf. Softw. Technol.*, vol. 51, no. 5, pp. 832–845, 2009.

[10] P. Savolainen, E. Niemela, and R. Savola, "A taxonomy of information security for service-centric systems," in *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 5–12.

[11] S.-W. Lee, R. Gandhi, D. Muthurajan, D. Yavagal, and G.-J. Ahn, "Building problem domain ontology from security requirements in regulatory documents," in *SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems*. New York, NY, USA: ACM, 2006, pp. 43–50.

[12] P. Giorgini, F. Massacci, and N. Zannone, "Security and trust requirements engineering," in *Foundations of Security Analysis and Design III*, 2005, pp. 237– 272.

[13] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA*, 2004, pp. 148– 157.

[14] A. Nhlabatsi, B. Nuseibeh, and Y. Yu, "Security requirements engineering for evolving software systems: A survey," *Journal of Secure Software Engineering*, vol. 1, pp. 54–73, 2009.

[15] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with goal models," in *Conceptual Modeling ER 2002*, 2003, pp. 167–181.

[16] R. S. Arnold, "Software change impact analysis," *Computer*, 1996.

[17] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Trans. Softw. Eng.*, vol. 25, pp. 493–509, 1999.

[18] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 210–224, 2003.

[19] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Comput. Surv.*, vol. 30, pp. 232–282, 1998.

[20] Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture- based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, pp. 54–62, 1999.

[21] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 574–586, 2004.

[22] A. Egyed, "Fixing inconsistencies in uml design models," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

[23] C. Jones, "Software change management," *Computer*, vol. 29, pp. 80–82, 1996.

[24] P. Besnard and A. Hunter, "Argumentation based on classical logic," in *Argumentation in Artificial Intelligence*, 2009, pp. 133–152.

[25] ——, "Practical first-order argumentation," in *Proceedings of the 20th national conference on Artificial intelligence - Volume 2 Pittsburgh, Pennsylvania: AAAI Press*, 2005.

[26] C. I. Chesnevar, A. G. Maguitman, and R. P. Loui, "Logical models of argument," *ACM Comput. Surv.*, vol. 32, pp. 337–383, 2000.

[27] G. Governatori, M. J. Maher, G. Antoniou, and D. Billington, "Argumentation semantics for defeasible logic," *J Logic Computation*, vol. 14, pp. 675–702, 2004.

[28] I. Jureta, J. Mylopoulos, and S. Faulkner, "Analysis of multi-party agreement in requirements validation," in *17th IEEE International Requirements Engineering Conference (RE'09), Atlanta, GA, USA, 2009,*, 2009, pp. 57 – 66.

[29] I. Habli, W. Wu, K. Attwood, and T. Kelly, "Extending argumentation to goal-oriented requirements engineering," in *Advances in Conceptual Modeling Foundations and Applications*, 2007, pp. 306–316.

[30] C. B. Haley, R. C. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Trans. Software Eng.*, vol. 34, pp. 133–153, 2008.

[31] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Gen. Comput.*, vol. 4, no. 1, pp. 67–95, 1986.

[32] M. Shanahan, "The event calculus explained," *Lecture Notes in Computer Science*, vol. 1600, pp. 409–430, 1999.

[33] R. Miller and M. Shanahan, "The event calculus in classical logic - alternative axiomatisations," *Electronic Transactions on Articial Intellligence*, vol. 3, pp. 77–105, 1999.

[34] J. McCarthy, *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex, 1990.

[35] L. Lamport, "What good is temporal logic?" in *IFIP Congress*, 1983, pp. 657–668.

[36] D. Peled and T. Wilke, "Stutter-invariant temporal properties are expressible without the next-time operator," *Inf. Process. Lett.*, vol. 63, no. 5, pp. 243–246, 1997.