A Tool for Managing Evolving Security Requirements[*]

Gábor Bergmann[1], Fabio Massacci[2], Federica Paci[2], Thein Than Tun[3], Dániel Varró[1], and Yijun Yu[3]

[1] DMIS - Budapest University of Technology and Economics
{bergmann,varro}@mit.bme.hu
[2] DISI - University of Trento
{fabio.massacci,federica.paci}@unitn.it
[3] Department of Computing - The Open University
{t.t.tun,y.yu}@open.ac.uk

Abstract. Management of requirements evolution is a challenging process. Requirements change continuously making the traceability of requirements difficult and the monitoring of requirements unreliable. Moreover, changing requirements might have an impact on the security properties a system design should satisfy: certain security properties that are satisfied before evolution might no longer be valid or new security properties need to be satisfied after changes have been introduced. This paper presents SeCMER, a tool for requirements evolution management developed in the context of the SecureChange project. The tool supports automatic detection of requirement changes and violation of security properties using change-driven transformations. The tool also supports argumentation analysis to check security properties are preserved by evolution and to identify new security properties that should be taken into account.

Keywords: security requirements engineering, secure i*, security argumentation, change impact analysis, security patterns.

1 Introduction

Modern software systems are increasingly complex and the environments where they operate are increasingly dynamic. The number and needs of stakeholders are also changing constantly as they adjust to changing environments. A consequence of this trend is that the requirements for a software system are many and they change continuously. To deal with evolution, we need analysis techniques that assess the impact of system evolution on the satisfaction of requirements. Requirements for system security, in particular, are very sensitive to evolution: security properties satisfied before the evolution might no longer hold or new security properties need to be satisfied as result of the evolution.

[*] Work partly supported by the project EU-FP7-ICT-FET-IP-SecureChange.

Another important aspect is that change management process is a complex process that would benefit from tool support. However, changes make the traceability of requirements difficult and the monitoring of requirements unreliable: requirements management is time-consuming and error-prone when done manually. Thus, a semi-automated requirements evolution management environment, supported by a tool, will improve requirement management with respect to keeping requirements traceability consistent, realizing reliable requirements monitoring, improving the quality of the documentation, and reducing the manual effort.

In this paper we present SeCMER[1], a tool developed in the context of the SecureChange European project[2]. The tool supports the different steps of SeCMER methodology for evolutionary requirements [10]. The methodology supports the automatic detection of requirement changes and violation of security properties, and argumentation analysis [16] to check security properties are preserved by evolution and to identify new security properties that should be taken into account.

In the next section we give an overview of the SeCMER methodology; then in Sec. 3 we describe the tool architecture. In Sec. 4 we illustrate the tool features based on an industrial example of evolution taken from the air traffic management domain. After presenting related works in Sec. 5, the results of tool evaluation are discussed in Sec. 6. Finally Sec. 7 concludes the paper.

2 SeCMER Methodology

The SecureChange Methodology for Evolutionary Requirements (SeCMER) [10] supports:

- a conceptual model of security requirements and a process for the elicitation of security goals
- a light-weighted approach to formalizing and reasoning about changing security goals, and
- an approach based on argumentation and model transformation to reason about the impact of change.

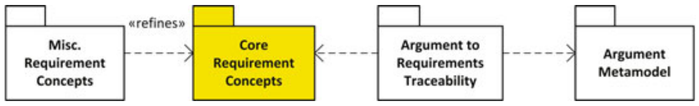
The main output from the methodology is either an assurance that the changes did not make the system violate the existing security properties, or a formulation of new properties to be satisfied by the new design. In the next subsections we will illustrate the main steps of the SeCMER methodology.

2.1 Modeling of Evolving Requirements

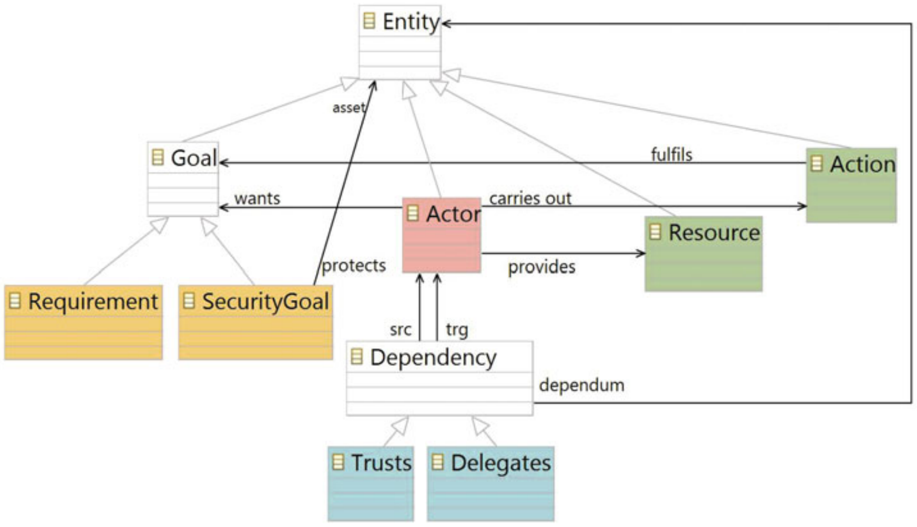
The Modeling of Evolving Requirements step produces two requirements model the *before model* and the *after model* which are an instance of the SeCMER conceptual model [13]. The conceptual model identifies a set of core concepts

[1] A detailed description of the tool implementation is reported in [11].

[2] www.securechange.eu


Misc. Requirement Concepts
«refines»
Core Requirement Concepts
Argument to Requirements Traceability
Argument Metamodel

(a) Package-level overview (extract)



(b) Package of core SeCMER concepts (extract)

Fig. 1. SeCMER conceptual model

that link the empirical security knowledge such as information about assets, security goals and threats to the stakeholders' security goals. To create this link, the conceptual model amalgamates concepts from Problem Frames (PF) [12] and SI* requirements engineering methodology [14] with traditional security concepts such as security goal and asset.

SI* [14] extends the i* framework which allows to model the stakeholders for a given project, their goals and their social inter-dependencies. In SI*, actors have goals, and own resources and tasks. The Problem Frames [12] approach (PF) instead explores the relationship between the machines, the physical domains in the problem world context and the requirements. Concepts from Problem Frames diagrams are similar to SI* concepts. For instance, the notion of biddable domain is similar to the notion of actor. Other types of domains such as lexical domains and causal domains are similar to resource and asset. The notion of phenomena in Problem Frames is generic enough to cover action, event and state.

The combination of the two security goals engineering approaches has several advantages: with SI* analysis, malicious intentions of attackers can be identified through explicit characterization of social dependencies among actors; with PF security goals analysis, valuable assets that lie within the system boundary can be identified through explicit traceability of shared phenomena among physical domains and the machine itself.

The SecMER conceptual model is illustrated in Figure 1. Figure 1 b) shows the core requirement concepts that are relevant for change detection and security analysis based on argumentation:

– An *actor* is an entity that can act and intend to want or desire something.
– An *action* is a means to achieve a goal.
– A *resource* is an entity without intention or behavior, and can be provided by actors.
– An *asset* is any entity of value for which protection is required.
– A *goal* is a proposition an actor wants to make true.
– A *requirement* is a goal that could be satisfied by a software system.
– A *security goal* is a goal to prevent harm to an asset through the violation of security properties.

The conceptual model also includes a set of relationships between concepts which include do-dependency, can-dependency and trust-dependency adopted from SI*. The *protects* relationship is a relationship between a security goal and a resource, action or goal – that denotes they are assets. For a complete list of all the possible relationships supported in SeCMER conceptual model the reader is referred to [13].

2.2 Change Detection Based on Security Patterns

The SeCMER methodology includes a lightweight automated analysis step that evaluates requirements-level compliance with security principles. These security principles are declaratively specified by an extensible set of *security patterns*.

A security pattern expresses a situation (a graph-like configuration of model elements) that leads to the violation of a security property. Whenever a new match of the security pattern (i.e. a new violation of the security property) emerges in the model, it can be automatically detected and reported. The specification of security patterns may also be augmented by automatic remedies (i.e. templates of corrective actions) that can be applied in case of a violation to fix the model and satisfy the security property once again.

SeCMER includes extension facilities that allow plug-ins to contribute the declarative definition of *security pattern*s in a high-level change-driven language [9] based on the notion of graph patterns. Automated solution templates (defined programmatically) can also be contributed. The tool then detects violations of these security patterns, which will appear as problem markers (warnings). The suggested solutions appear as Quick Fix rules offered for the problem marker.

Although the set of security patterns is extensible, the main focus points of security patterns are the following: *trust* (which can be explicitly modeled, and interpreted transitively), *access* (which can also be granted / delegated transitively), and *need* (expressed by carrying out an action that consumes a resource). The patterns are further characterised by the following:

– The security patterns only consider *assets* that are protected by security goals.
– If a trusted actor performs an action that is known to fulfill the security goal, then no further investigation is required.
– If there is access to an asset without trust (regardless of need), then it is considered a violation of the *trusted path* property.
– If there is access to an asset without the need thereof (regardless of trust), then it is considered a violation of the *least privilege* property.
– If there is need for an asset but no actual access, then the model is reported as inconsistent / incomplete.
– Security violation reports can be suppressed by manual arguments supporting the satisfaction of the security goal.

Example: The *trusted path* security pattern finds security violations where an asset is communicated via an untrusted path. The pattern has the following structure: if a concerned actor wants a security goal that expresses that a resource must be protected, then each actor that the resource is delegated to must be trusted (possibly transitively) by the concerned actor. An exception is made if a trusted actor performs an action to explicitly fulfill the security goal, e.g. digital signature makes the trusted path unnecessary in case of an integrity goal.

See Lst. 1 for the simplified definition of the pattern using the declarative model query language of EMF-IncQuery [8]. According to the pattern definition, a violation of the trusted path property is characterized by a quadruplet of model elements *ConcernedActor*, *SecGoal*, *Asset*, *UntrustedActor*, provided that they satisfy a list of criteria (graph constraints listed between the pair of braces). Regarding the type and configuration of these elements, as enforced by the two edge and one node constraint in lines 2-4, *SecGoal* is an instance of type Security Goal that is wanted by an actor *ConcernedActor* and expresses the protection of the element *Asset*. Lines 5-6 state that *Asset* is provided by some actor *ProviderActor* (not exposed as a parameter of the pattern), and - through a chain of transitive delegation - is eventually possessed by the *UntrustedActor*; thanks to the pattern composition language feature, the latter is expressed by a helper pattern *transitiveDelegation* (defined elsewhere). A negated condition on line 7 ensures that *UntrustedActor* is not trusted (transitively) by the *ConcernedActor*. A second negative constraint (line 8) expresses that the *SecGoal* is not fulfilled explicitly by any action that is trusted by the *ConcernedActor*.

The security pattern in Lst. 1 can be applied to enforce a security property such as integrity. Requirements engineers are further assisted by a set of suggested fixes that can be applied on violations of the security property. In fact, each of these suggestions can be implemented as automated corrective actions to be applied to the model in order to re-establish the security property. The requirements engineers can then choose one of the suggestions, or come up with their own solution. Possible examples of corrective actions include:

Listing 1. Pattern to capture violations of the trusted path property

```
shareable pattern untrustedPath(ConcernedActor,SecGoal,Asset,UntrustedActor)={
    Actor.wants(ConcernedActor,SecGoal);
    SecurityGoal(SecGoal);
    SecurityGoal.protects(SecGoal, Asset);
    Actor.provides(ProviderActor,Asset);
    find transitiveDelegation(ProviderActor,UntrustedActor,Asset);
    neg Actor.trust*(ConcernedActor,UntrustedActor);
    neg find trustedFulfillment(ConcernedActor,AnyActor,AnyTask,SecGoal);
}
```

– Add a trust relationship from *ConcernedActor* to *UntrustedActor* to reflect that the security decision was that there must be trust between these actors (e.g. by establishing a liability contract between them).
– Alternatively, an action can be created that explicitly fulfills *SecGoal*, such as introducing a policy or technological process that makes it impossible for *UntrustedActor* to abuse the situation (e.g. digital signature to ensure the security goal of data integrity).

These solution templates can be attached to the security pattern so that they are offered whenever a violation of the corresponding security property is detected. The solutions can be implemented by arbitrary program code, typically short snippets that manipulate the model according to the description of the solution.

2.3 Argumentation-Based Security Analysis

In this step of the SeCMER methodology, the developers check whether there are new security properties to be added or to be removed (Δ Security Properties) as a result of changes in the requirement model. This phase is supported by argumentation analysis.

As shown in the meta-model of the SeCMER arguments in Figure 2, an argument diagram may have several arguments linked to each other. An *argument* contains one and only one claim. It also contains facts and warrants. A *claim* is a predicate whose truth-value will be established by an argument. A *fact* is a true proposition (an argument with a claim only). A *warrant* links facts in an argument to the claim. Since facts and warrants can themselves be arguments, arguments can be nested. Every argument has an optional timestamp, which indicates the time (or the round) during the argumentation process at which the argument is introduced.

As well as nesting of arguments, arguments may be related to each other through rebuttal and mitigation/restore relationships. A rebuttal argument is a kind of argument whose purposes are to establish the falsity of their associated argument or make them inconsistent. Similarly, mitigations are another special kind of arguments following the iteration of rebuttals in order to reestablish the truth-value of the associated original claims. Mitigations may or may not negate the claims of the rebuttals: sometimes they add further facts overlooked by the rebuttals.

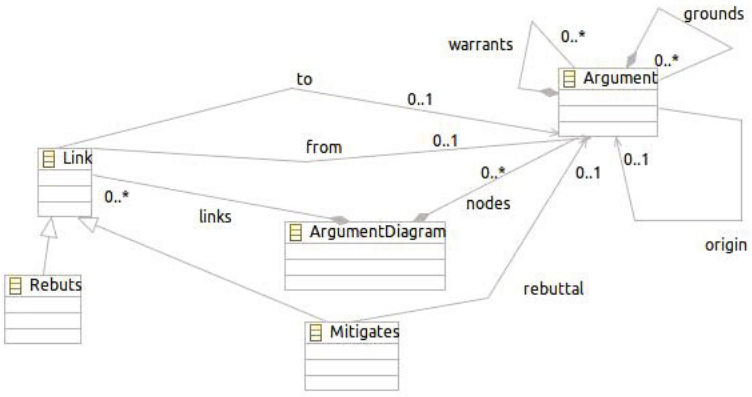
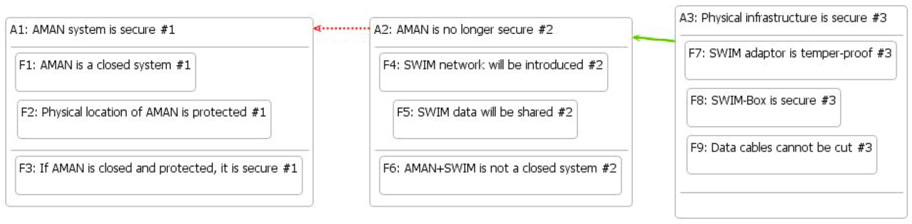


Fig. 2. Meta-Model of SeCMER Arguments

Figure 3 illustrates the visual syntax of SeCMER argument diagrams. Graphically, an argument is represented by a box with three compartments: the claim is written in the top compartment, the fact(s) in the middle compartment and the warrant(s) in the bottom compartment. Rebuttal and mitigation links are represented by the red and green arrows respectively.



A1: AMAN system is secure #1
F1: AMAN is a closed system #1
F2: Physical location of AMAN is protected #1
F3: If AMAN is closed and protected, it is secure #1
A2: AMAN is no longer secure #2
F4: SWIM network will be introduced #2
F5: SWIM data will be shared #2
F6: AMAN+SWIM is not a closed system #2
A3: Physical infrastructure is secure #3
F7: SWIM adaptor is temper-proof #3
F8: SWIM-Box is secure #3
F9: Data cables cannot be cut #3

Fig. 3. Visual Syntax of SeCMER Arguments

Since argumentation is a costly manual process, it is preferable to avoid its re-execution after each small change of the requirement model. However, some arguments may be invalidated by evolution and require attention from security experts. Therefore, if a change affects one of the elements that was recorded as an evidence for an argument, then the argument is marked for re-examination. This relies on traceability links that can be established between the argument and requirement models.

3 SeCMER Tool Architecture and Implementation

SeCMER is an Eclipse-based heterogeneous modeling environment for managing evolving requirements models. It has the following features:

- **Modeling of Evolving Requirements**. Requirement models can be drawn in SI*, Problem Frames or SeCMER. Traceability and bidirectional synchronization is supported between SeCMER and SI* requirements models.
- **Change detection based on security patterns**. Violations of formally defined static security properties expressed as security patterns can be automatically identified. Detection of formal or informal arguments that has been invalidated by changes affecting model elements that contributed to the argument as evidence is also supported.
- **Argumentation-based security analysis**. Reasoning about security properties satisfaction and identification of new security properties is supported.

These capabilities of the tool are provided by means of the integration of a set of EMF-based [15] Eclipse plug-ins written in Java, relying on standard EMF technologies such as GMF, Xtext and EMF Transaction. The components of the tool are:

- Eclipse plug-ins of OpenPF including (a) the implementation of the SecMER conceptual model, (b) the argumentation model and tools, as well as (c) external modeling tools for Problem Frames [17],
- SI* (requirements engineering tool [14]),
- traceability models to represent the relationship between corresponding model elements in different languages, e.g. the SecMER conceptual model and SI*,
- run-time platform components of EMF-IncQuery (incremental EMF model query engine) for change-driven transformations,
- model query plug-ins automatically generated from (a) security patterns or (b) transformation specification by the development-time tools of EMF-IncQuery,
- integration code developed solely for this tool, including User Interface commands and the Java definition of the action parts of (a) quick fixes and (b) model synchronization.

The relationship of the most important model management components are depicted on Fig. 4, focusing on the SI* and SeCMER models in particular, as well as the traceability model established between them. User Interface components are omitted from this diagram.

All the involved EMF models are accessed through a common EMF `ResourceSet` and edited solely through the corresponding `TransactionalEditingDomain` (from the EMF Transaction API). Consequently, all modifications are wrapped into EMF Transactions, including those carried out by manual editing through the User Interface (e.g. the SI* diagram editor) as well as changes performed by automated mechanisms such as model transformation. As one of the benefits, concurrent modifications are serialized and therefore conflict-free. Furthermore, the commit process of the transactions provides an opportunity for triggering change-driven actions.

The incremental query mechanism provided by EMF-IncQuery plays a key role in the functionality of the tool. Incremental query evaluation code is generated automatically at development time by EMF-IncQuery, from a graph

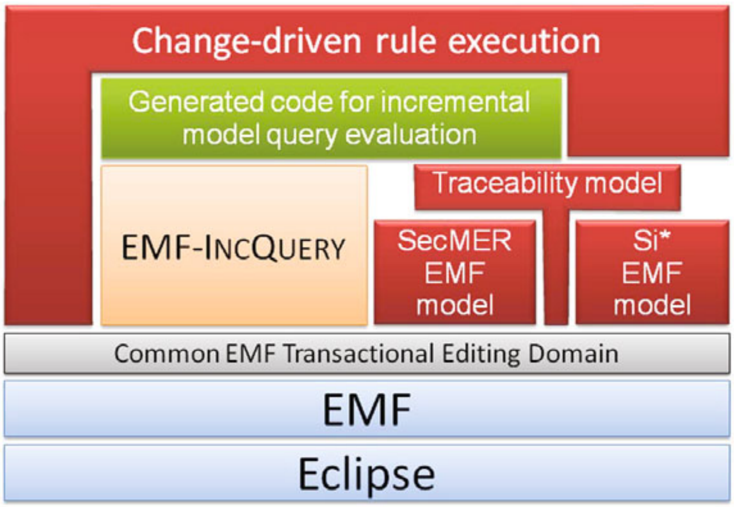

Change-driven rule execution
Generated code for incremental model query evaluation
EMF-IncQuery
Traceability model
SecMER EMF model
Si* EMF model
Common EMF Transactional Editing Domain
EMF
Eclipse

Fig. 4. Architectural overview of model management components

pattern-based declarative description of EMF model queries. Through this incremental evaluation functionality, change-driven rules can be efficiently triggered by changes captured as graph patterns. The implementation currently supports detecting the presence, appearance and disappearance of graph patterns. A more advanced formalism for capturing changes is already defined [9], but support is not implemented yet.

The core trigger engine plug-in offers an Eclipse extension point for defining change-driven rules. Multiple constituent plug-ins contribute extensions to register their respective set of rules. The graph pattern-based declarative guard of the rules is evaluated efficiently (see measurements in [8]) by the incremental graph pattern matcher plug-ins automatically generated from the declarative description by EMF-IncQuery. At the commit phase of each EMF transaction, the rules that are found to be triggered will be executed to provide their reactions to the preceding changes. These reactions are implemented by arbitrary Java code, and they are allowed to modify the model as well (wrapped in nested transactions) and could therefore be reacted upon.

So far, there are three groups of change-driven rules as extension points:

- transformation rules that realize the on-the-fly synchronization between multiple modeling formalisms,
- security-specific rules that detect the appearance of undesired security patterns, raise alerts and optionally offer candidate solutions.
- rules for invalidating arguments when their ground facts change.

Another key feature is a bi-directional synchronizing transformation between SI* and the SeCMER model with changes propagated on the fly, interactively. Since the languages have different expressive power, the following challenges arise:

1. some concepts are not mapped from one formalism to the other or vice versa,
2. some model elements may be mapped into multiple (even an unbounded amount of) corresponding model elements in the other formalism, and finally
3. it is possible that a single model element has multiple possible translations (due to the source formalism being more abstract); one of them is created as a default choice, but it can later be changed to the other options, which are also tolerated by the transformation system.

4 Illustrative Example

We now illustrate the features supported by our tool using the ongoing evolution of ATM (Air Traffic Management) systems as planned by the ATM 2000+ Strategic Agenda [7] and the SESAR Initiative.

Part of ATM system's evolution process is the introduction of the Arrival Manager (AMAN), which is an aircraft arrival sequencing tool to help manage and better organize the air traffic flow in the approach phase. The introduction of the AMAN requires new operational procedures and functions that are supported by a new information management system for the whole ATM, an IP based data transport network called System Wide Information Management (SWIM) that will replace the current point to point communication systems with a ground/ground data sharing network which connects all the principal actors involved in the Airports Management and the Area Control Centers.

We have chosen to illustrate the following steps of the SeCMER methodology based on the above evolutionary scenario.

1. **Requirements evolution**. We show how SeCMER supports the representation of the evolution of the requirement model as effect of the introduction of the SWIM.
2. **Change detection based on security patterns.**
   a *Detection of a security property violation based on security patterns.* We show how the tool detects that the integrity security property of the resource MD "Meteo Data" is violated due to the lack of a trusted path.
   b *Automatically providing corrective actions based on security patterns.* We show how violations of the integrity security property, as detected by a security pattern, may have corrective actions associated with them.
3. **Argumentation-based security analysis**. We show how argumentation analysis [16] can be carried to provide evidence that the information access property applied to the meteo data is satisfied after evolution.

The entities involved in the simple scenario are the AMAN, the Meteo Data Center (MDC), the SWIM-Box and the SWIM-Network. The SWIM-Box is the core of the SWIM information management system which provides access via defined services to data that belong to different domain such as flight, surveillance, meteo, etc. The introduction of the SWIM requires suitable security properties to be satisfied: we will show how to protect information access on meteo data and how to ensure integrity of meteo data.

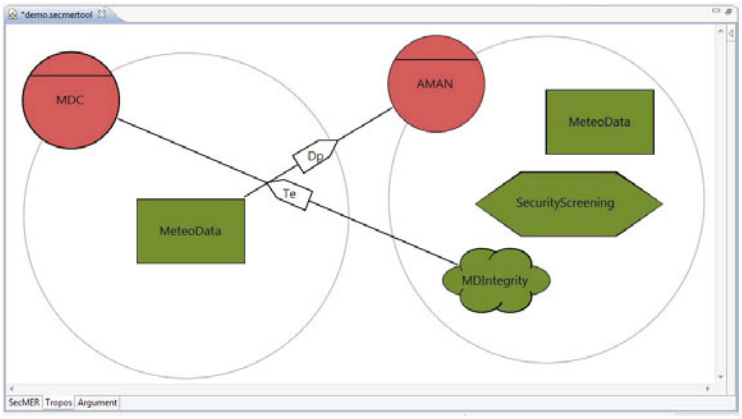


Fig. 5. Requirement Model before evolution (Si* syntax)

Requirements evolution. Figure 5 shows the before requirement model which consists of two actors the *AMAN* and *MDC*: MDC provides the asset Meteo Data (*MD*) to the AMAN. The AMAN has an integrity security goal *MDIntegrity* for MD, and MDC is entrusted with this goal. AMAN also performs an Action, *SecurityScreening*, to regularly conduct a background check on its employees to ensure that they do not expose to risk the information generated by the AMAN.

As the communication between the AMAN and MDC is mediated by the SWIM, the before model evolves as follows (see Figure 6):

- The Actors *SWIM*, *SWIMBox_MDC* and *SWIMBox_AMAN* are introduced in the SI* model
- As the meteo data is no longer directly provided by MDC to AMAN, the delegation relation between the two is removed.
- Delegation relationships are established between the Actors MDC, SWIM-Box_MDC, SWIM, SWIMBox_AMAN, AMAN.
- As the SWIM network can be accessed by multiple parties, the AMAN has a new security goal *MDAccessControl* protecting MD resource.

Detecting violations of security properties based on security patterns. According to the pattern of Lst. 1, the integrity property for MD is violated because AMAN entrusts MDC with the integrity security goal, but not the communication intermediary actors SWIMBox_MDC, SWIM and SWIMBox_AMAN. The violation (i.e. a match of the pattern) is detected and reported by the tool, as shown on Fig. 7.

Automatic corrective actions based on security patterns. The following quick fix suggestions are associated with the security pattern:

- Add a trust relationship between MDC and SWIM Network having the integrity security goal as dependum.

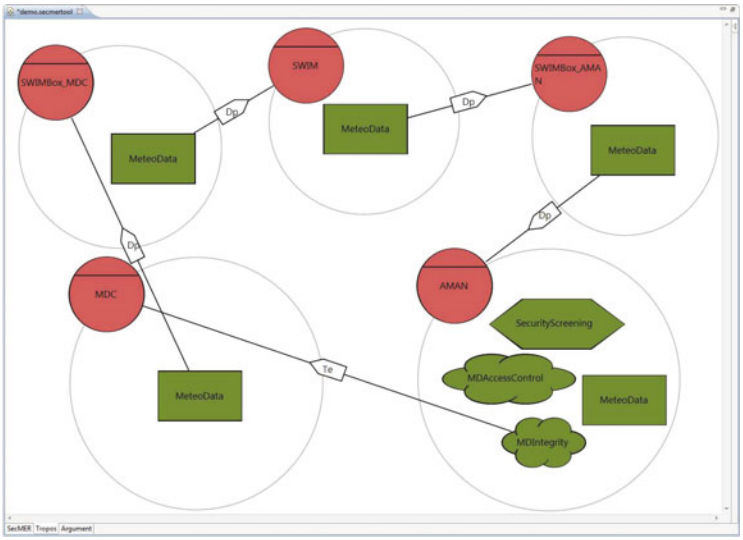
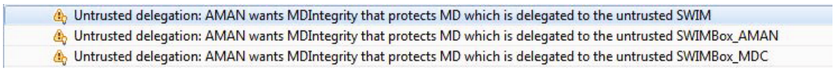
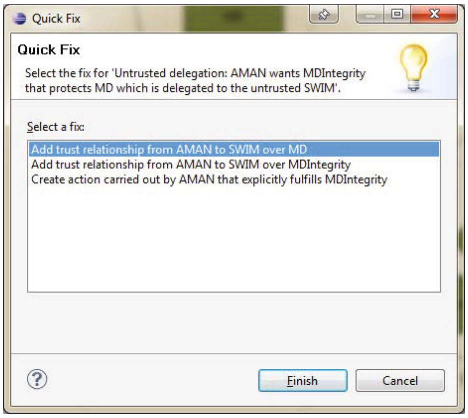


Fig. 6. Requirement after evolution (Si* syntax)


Untrusted delegation: AMAN wants MDIntegrity that protects MD which is delegated to the untrusted SWIM
Untrusted delegation: AMAN wants MDIntegrity that protects MD which is delegated to the untrusted SWIMBox_AMAN
Untrusted delegation: AMAN wants MDIntegrity that protects MD which is delegated to the untrusted SWIMBox_MDC

(a) Detected Security Issues


Quick Fix

Quick Fix

Select the fix for 'Untrusted delegation: AMAN wants MDIntegrity that protects MD which is delegated to the untrusted SWIM'.

Select a fix:

Add trust relationship from AMAN to SWIM over MD
Add trust relationship from AMAN to SWIM over MDIntegrity
Create action carried out by AMAN that explicitly fulfills MDIntegrity

(b) Possible Corrective Actions

Fig. 7. Detection of Security Issues

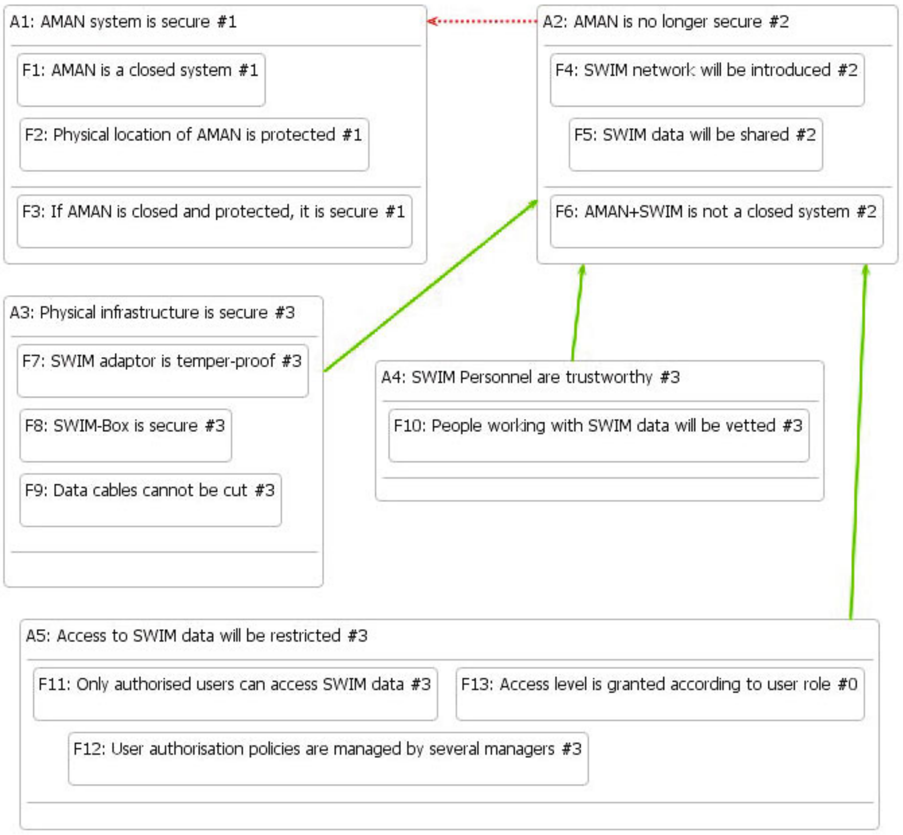

A1: AMAN system is secure #1
F1: AMAN is a closed system #1
F2: Physical location of AMAN is protected #1
F3: If AMAN is closed and protected, it is secure #1
A2: AMAN is no longer secure #2
F4: SWIM network will be introduced #2
F5: SWIM data will be shared #2
F6: AMAN+SWIM is not a closed system #2
A3: Physical infrastructure is secure #3
F7: SWIM adaptor is temper-proof #3
F8: SWIM-Box is secure #3
F9: Data cables cannot be cut #3
A4: SWIM Personnel are trustworthy #3
F10: People working with SWIM data will be vetted #3
A5: Access to SWIM data will be restricted #3
F11: Only authorised users can access SWIM data #3
F13: Access level is granted according to user role #0
F12: User authorisation policies are managed by several managers #3

Fig. 8. A fragment of the argumentation model

– Alternatively, an Action such as "MD is digitally signed" can be created to protect the integrity of MD even when handled by untrusted actors.

Argumentation for the information access property. Fig. 8 shows the different rounds of the argumentation analysis that is carried out for the information access security property applied to MD resource. The diagram says that the AMAN system is claimed to be secure before the change (Round #1), and the claim is warranted by the facts the system is known to be a close system (F1), and the physical location of the system is protected (F2). This argument is rebutted in Round #2, in which another argument claims that the system is no longer secure because SWIM will not keep AMAN closed. The rebuttal argument is mitigated in Round #3 by three arguments, which suggest that the AMAN may still be secure given that the physical infrastructure is secure, personnel are trustworthy and access to data is controlled.

5 Related Works

There are many requirement engineering tools available but only some of them support specific capabilities for requirement change management. CASE Spec [1] makes easy to generate traceability reports and perform impact analysis with built-in visual and tabular traceability tools. Dimensions RM [2] allows enterprises to effectively manage change in requirements during the project lifecycle. In particular, DimRM facilitates the understanding of the impact of requirement changes and the creation of reports on requirements definition, baselines, change impact, and traceability. IBM Rational DOORS [3] has powerful capabilities for capturing, linking, analyzing and managing changes to requirements and their traceability. IBM Rational RequisitePro [4] is a requirements management tool that incorporates a powerful database infrastructure to facilitate requirements organization, integration, traceability and analysis. Moreover, it provides detailed traceability views that display parent/child relationships and shows requirements that may be affected by upstream or downstream changes. MKS Integrity 2009 [5] provides reuse and requirements change management capabilities coupled with meaningful (and traceable) relationships to downstream code and testing assets, which ensure communication of change, conformance to requirements and compliance with applicable governance or regulations.

Reqtify [6] is an interactive requirement traceability and impact analysis tool which can trace requirement from system, program and project levels to the entire levels of software or hardware component development lifecycle.

Compared with the above tools, SecMER provides support to the requirement engineer for handling security related changes. The tool supports automatic detection of requirement changes that lead to violation of security properties using change-driven transformations and suggests possible corrective actions. The tool also supports argumentation analysis to check security properties are preserved by evolution and to identify new security properties that should be taken into account.

6 Tool Evaluation

The SecMER tool has been validated during a workshop with Air Traffic Management experts. We had a total of fifteen participants: four requirement analysts and eleven ATM experts who were air traffic controllers and the others were Deep Blue[3] consultants. The participants were divided in three groups. Each group has to first create the before and after requirement models for the illustrative scenario introduced in Sec. 4; then, check security violations and select a possible suggested quick fix; and build an argument model for the after requirement model. The domain experts were given wild cards to provide feedbacks related to the application of the methodology steps and on the usability and reliability of

[3] Deep Blue is a human factors, safety and validation consultancy providing solutions throughout industry and the public sector in the field of transportation (http://www.dblue.it/)

the tool. The validation session had a duration of one hour and thirty minutes. During the validation session each group was observed by a requirement analyst. At the end of the validation session the requirement analysts gave a questionnaire to be filled out by the participants. Useful feedbacks have been provided by the domain experts that have been used to improve the tool usability and reliability. Each group reported that was not clear how to create before and after models and how to maintain the history of changes. The experts suggested to have a guideline or a source of help that explains when to use the most critical concepts; and the possibility of saving in the same project the before and after models. These issues have been addressed since. Moreover, for the participants was confusing to have different views of the same model - SeCMER view and SI* view. Since the *protects* relationship is not part of the standard SI* conceptual model but only of the SeCMER conceptual model, the participants were required to switch from the SI* view to the SeCMER view and add the relationship to the model. In order to improve the usability of the tool, the *protects* information is now made part of SI*, and does not require manual effort from the final user; we have also named the SI* concepts in the palette of the SI* view as the mapped concepts in the SeCMER view to converge the terminologies of the two views. About the automatic detection of violation of security properties, the participants suggested that more guildelines should be given about the state of security modeling even when no violations are detected. The tool now guides the user in creating the first security goal, as well as in identifying the protected assets of security goals.

7 Conclusions

This paper has presented SeCMER, a tool for managing evolving requirements. As shown by the ATM-based illustrative scenario, the tool supports visual modeling of security requirements. Additionally, argument models can be constructed manually to investigate the satisfaction of security properties; the tool detects invalidated arguments if the requirements model evolves. Finally, the tool performs continuous and automatic pattern-based violation detection of security properties, with optional "quick fix" corrective actions.

We plan to extend the tool in order to support other sets of security patterns to automate the detection and handling of security violations in a wider range of application scenarios. We plan also to realize a tighter integration with additional modeling formalisms (Problem Frames) and industrial tools e.g DOORS-TREK.

References

1. CASE Spec, `http://www.analysttool.com/`
2. Dimensions RM, `http://www.serena.com/products/rm/index.html`
3. IBM Rational DOORS, `http://www-01.ibm.com/software/awdtools/doors/`
4. IBM Requisite Pro, `http://www-01.ibm.com/software/awdtools/reqpro/`
5. IMKS Integrity (2009), `http://www.mks.com/`

6. Reqtify, http://www.geensoft.com/en/article/reqtify
7. EUROCONTROL ATM Strategy for the Years 2000+ Executive Summary (2003)
8. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010)
9. Bergmann, G., et al.: Change-Driven Model Transformations. Change (in) the Rule to Rule the Change. Software and System Modeling (2011) (to appear)
10. Bergmann, G., et al.: D3.2 Methodology for Evolutionary Requirements, http://www.securechange.eu/sites/default/files/deliverables/-%20Methodology%20for%20Evolutionary%20Requirements_v3.pdf
11. Bergmann, G., et al.: D3.4 Proof of Concept Case Tool, http://www.securechange.eu/sites/default/files/deliverables/D3.4 tt%20Proof-of-Concept%20CASE%20Tool%20for%20early%20requirements.pdf
12. Jackson, M.: Problem Frames: Analyzing and structuring software development problems. ACM Press, Addison Wesley (2001)
13. Massacci, F., Mylopoulos, J., Paci, F., Tun, T.T., Yu, Y.: An Extended Ontology for Security Requirements. In: Salinesi, C., Pastor, O. (eds.) CAiSE Workshops 2011. LNBIP, vol. 83, pp. 622–636. Springer, Heidelberg (2011)
14. Massacci, F., Mylopoulos, J., Zannone, N.: Computer-aided support for secure tropos. Automated Software Engg. 14, 341–364 (2007)
15. The Eclipse Project: Eclipse Modeling Framework, http://www.eclipse.org/emf
16. Tun, T.T., et al.: Model-based argument analysis for evolving security requirements. In: Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, pp. 88–97. IEEE Computer Society, Washington, DC (2010)
17. Yu, Y., Tun, T.T.: OpenPF - The Open Requirements Engineering Lab, http://computing-research.open.ac.uk/trac/openre