

Design Decisions in the Development of a Graphical Language for Risk-Driven Security Testing

Gencer Erdogan¹ and Ketil Stølen^{1,2}

¹ Department for Software and Service Innovation, SINTEF Digital, Norway

² Department of Informatics, University of Oslo, Norway

{gencer.erdogan,ketil.stolen}@sintef.no

Abstract. We have developed a domain-specific modeling language named CORAL that employs risk assessment to help security testers select and design test cases based on the available risk picture. In this paper, we present CORAL and then discuss why the language is designed the way it is, and what we could have done differently.

Keywords: risk-driven security testing, model-based testing, security risk assessment, domain-specific modeling language

1 Introduction

Security testers face the problem of determining tests that are most likely to reveal severe security vulnerabilities. To address this challenge, we have developed a domain-specific modeling language that employs risk assessment to help security testers select and design test cases based on the available risk picture [5]. Our language (CORAL) supports a model-based approach to risk-driven security testing as defined in [2]. The approach is model-based in the sense that graphical models are actively used throughout the whole testing process to support the various testing tasks and activities, and to document the test results.

The intended users of CORAL are security testers. In this paper, we first present CORAL, and then we discuss why the language is designed the way it is, and what we could have done differently. In particular, we motivate our design decisions by discussing five main areas typically considered when developing or evaluating a modeling language: domain appropriateness, comprehensibility appropriateness, participant appropriateness, modeler appropriateness, and tool appropriateness [15]. With respect to what we could have done differently, we discuss alternative design decisions and their consequence in terms of graphical versus textual representations, risk annotations versus tables, choice of modeling notation, CORAL versus attack trees, and CORAL versus formal methods.

The remainder of this paper is organized as follows. In Sect. 2 we present the CORAL language followed by a small example. In Sect. 3 we motivate our

design decisions in context of the five areas mentioned above. In Sect. 4 we discuss alternative design decisions and their consequence. Finally, in Sect. 5 we conclude the paper.

2 The CORAL Language

As shown in Fig. 1, the graphical notation of CORAL is mainly based on the graphical notation of UML sequence diagrams [21]. The graphical icons used to represent risk-related information are based on corresponding graphical icons in the CORAS risk analysis language [17]. With respect to testing concepts, CORAL uses stereotypes from the UML Testing Profile [20]. The constructs in CORAL are grouped into five categories: diagram frame, lifelines, messages, risk-measure annotations, and interaction operators. In the following, we explain each category.

Diagram Frame: The diagram frame is the frame in which a sequence diagram is modeled. A sequence diagram in CORAL may represent the system under test, its environment, as well as threat scenarios that the system under test and its environment are exposed to. The diagram frame is graphically equivalent to the diagram frame in UML used to represent sequence diagrams [21]. Similar to UML, we use the keyword *sd* in a pentagon in the upper left corner of the diagram frame to denote that the diagram is a sequence diagram.

Lifelines: According to UML, a lifeline represents an individual participant in an interaction [21]. As illustrated in Fig. 1, we distinguish between five different lifelines: general lifeline, deliberate threat lifeline, accidental threat lifeline, non-human threat lifeline, and asset lifeline.

The general lifeline is graphically equivalent to a lifeline used in UML sequence diagrams. In CORAL, a general lifeline is used to model the system under test, as well as the environment interacting with the system under test. The name of the lifeline is placed inside the rectangle of the lifeline as illustrated in Fig. 1, and the naming convention is equivalent to the naming convention of lifelines in UML.

The lifelines representing threats are used to model threats that may initiate threat scenarios, which in turn may cause security risks in the system under test. Inspired by CORAS [17], we distinguish between three kinds of threats: deliberate threat, accidental threat, and non-human threat. A deliberate threat is a human threat that has malicious intents. An accidental threat is also a human threat, but this threat is different in the sense that it does not have malicious intents. The non-human threat is a threat that may be anything else except a human. For example, a power failure in a server hall may cause problems with respect to the availability of a system.

In practice, the distinction between a human threat and a non-human threat is sometimes not straight forward. For example, if a hacker exploits a security bug in a source code in order to attack a system, then the threat is the hacker. On the other hand, if the security bug lies dormant in the source code and is triggered at some point during system execution, then the threat is the bug in the

source code, that is, a non-human threat. In other words, the distinction between a human threat and a nonhuman threat depends on the viewpoint from which a threat is regarded. The name of a threat is placed below the icon representing the threat as illustrated in Fig. 1.


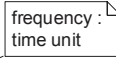







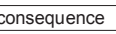



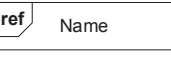


Diagram frame		Lifelines					Risk-measure annotations		
Node type	Notation	Node type	General lifeline	Deliberate threat lifeline	Accidental threat lifeline	Non-human threat lifeline	Asset lifeline	Node type	Notation
Frame		General message						Frequency	
		New message						Conditional ratio	
		Altered message		Name	Name	Name	Name	Consequence	
		Deleted message							
		Unwanted incident message							
Messages		Interaction operators							
Node type	Notation	Node type	Notation						
Potential alternatives		Referred interaction							
Parallel execution		Loop							

Fig. 1. CORAL graphical notation.

The name of a threat typically represents a threat profile which is described by the tester. A threat may be named, for example, “hacker” (deliberate threat), “database administrator” (accidental threat), or “computer virus” (non-human threat).

In CORAL, risk assessment is carried out with respect to security assets we want to protect. Security assets are represented by the asset lifeline shaped as a moneybag, and the asset name is placed below the moneybag icon. Examples of security assets are “availability of customer data” or “integrity of bank transactions”. What is meant by “customer data” and “bank transactions” has to be described by the tester.

Messages: According to UML, a message defines a particular communication between lifelines of an interaction [21]. UML distinguish between complete, lost and found messages. Complete messages have both a sender and a receiver lifeline. A lost message has a sender lifeline, but not a receiver lifeline. A found message has a receiver lifeline, but not a sender lifeline. The graphical notation for these messages are different. However, lost and found messages are often unnecessary and are used in rare situations [25]. Furthermore, UML categorize complete messages into synchronous and asynchronous messages. The

synchronous and asynchronous messages have different graphical notations. A synchronous message is used to call an operation, and the lifeline transmitting a synchronous message always expects a responding message. An asynchronous message, on the other hand, is used to send a signal which may or may not be responded. Synchronous messages are therefore syntactically more strict than asynchronous messages because they require a corresponding response message for each operation call. However, at a logical level, sending a signal and calling an operation are similar. Both types of messages involve a communication from a sender to a receiver [25].

In CORAL we are interested in expressing complete interactions between two lifelines. Moreover, because synchronous and asynchronous messages are similar at a logical level, it is not necessary to express both in CORAL. For this reason, we choose to treat all messages as asynchronous messages. The graphical notation for messages in CORAL are therefore based on the graphical notation for the asynchronous message in UML. As illustrated in Fig. 1, we distinguish between five messages in CORAL: general message, new message, altered message, deleted message, and unwanted incident message.

The general message is graphically equivalent to the asynchronous message in UML, and it is used to model the expected behavior between lifelines representing the system under test and its environment, that is, the interaction between general lifelines (recall that general lifelines are used to model the system under test and its environment). The signature of a message, that is, the content of a message, is placed above the arrow representing the message. Signatures are written using the same convention as given for messages in UML. In addition, we represent the risk related information, in the signatures, using a red-colored, bold, and italic font to distinguish between the expected behavior and the risk-related information.

New, altered, deleted and unwanted incident messages are used in combination to represent threat scenarios. A new message is a message initiated by a threat. This may be a deliberate human threat, an accidental human threat, or a non-human threat. A new message is represented by a red triangle which is placed at the transmitting end of the message. An altered message is a message in the system under test that has been altered by a threat to deviate from its expected behavior. Altered messages are represented by a triangle with red borders and white fill. A deleted message is a message in the system under test that has been deleted by a threat. Deleted messages are represented by a triangle with red borders and a red cross in the middle of the triangle. Finally, an unwanted incident message is a message modeling that an asset is harmed or its value is reduced. Unwanted incidents are represented by a yellow explosion sign.

Risk-Measure Annotations: Risk-measure annotations are used to annotate messages for the purpose of estimating and evaluating security risks. As illustrated in Fig. 1, we distinguish between three kinds of risk-measure annotations: frequency, conditional ratio, and consequence.

The frequency annotation represents either the frequency of the transmission or the frequency of the reception of a message. The graphical notation of

a frequency annotation is equivalent to the graphical notation of a comment generally used in UML [21]. The connector on the frequency annotation is attached on either the transmission-end or the reception-end of a general, new, or altered message. It may also be attached on the transmission-end of an unwanted incident message to convey how often an unwanted incident harms a certain security asset. A frequency annotation may not be attached on a deleted message because the message represents a complete deletion. That is, if a message is deleted, then it is not transmitted and therefore not received. It therefore does not make sense to estimate how often a message is *not* received, given that it is *not* transmitted. A message is either deleted, or it is not. Also, in the context of testing, we are interested in testing the messages that may *cause* the deletion of other messages. The frequency is written inside the comment frame, in terms of an interval followed by a time unit, as illustrated in Fig. 1.

The conditional ratio annotation represents the ratio by which a message is received, given that it is transmitted. Conditional ratios may be attached on general, new, or altered messages, and may not be attached on deleted messages because they represent complete deletion. In addition, conditional ratios may not be attached on unwanted incidents because their purpose is to model that an asset is harmed or reduced in value.

The consequence annotation represents the impact an unwanted incident has on an asset. Thus, consequences may therefore be attached only on unwanted incident messages.

Interaction Operators: In sequence diagrams, messages may be combined in rectangles containing special keywords in order to convey a particular relationship between the combined messages. The rectangle encapsulating the messages is referred to as a combined fragment, while the keyword is referred to as an interaction operator. An interaction operator specifies the operation that defines the semantics of the combination of messages [21]. As illustrated in Fig. 1, CORAL makes use of four interaction operators inherited from UML: potential alternatives (keyword *alt*), referred interaction (keyword *ref*), parallel execution (keyword *par*), and loop (keyword *loop*). All interactions in sequence diagrams are by default encapsulated within an implicit combined fragment that makes use of an interaction operator named weak sequencing (keyword *seq*). The *seq* operator is the implicit composition mechanism of interactions. However, because the *seq* operator is always implicitly included in all sequence diagrams, it is generally not modeled explicitly. The reader is referred to UML for further information on interaction operators [21].

2.1 Example-Driven Explanation of the CORAL Approach

We carry out risk-driven security testing in three consecutive phases: test planning, security risk assessment, and security testing. The method takes as input a description of the system to test, and provides a test report as output. The description may be in the form of system diagrams and models, use case documentation, source code, executable versions of the system, and so on.

In Phase 1 we prepare the system model, identify security assets to be protected, define frequency and consequence scales, and define the risk evaluation matrix based on the frequency and consequence scales.

In Phase 2 we carry out risk modeling in three consecutive steps. First, we identify security risks by analyzing the system model with respect to the security assets, and then we identify threat scenarios that may cause the security risks. Second, we estimate frequency and consequence of the identified risks by making use of the predefined frequency and consequence scales, respectively. Third, we evaluate the risks with respect to their frequency and consequence estimates and select the most severe risks to test.

In Phase 3 we conduct security testing in three consecutive steps. First, for each risk selected for testing we select its associated threat scenario and specify a test objective for that threat scenario. To obtain a test case fulfilling the test objective, we annotate the threat scenario with stereotypes from the UML Testing Profile [20] according to the test objective. Second, we carry out security testing with respect to the test cases. Finally, based on the test results, we write a test report.

The example in Fig. 2 is a small fragment taken from an industrial case study, which is thoroughly documented in [4]. The system under test is a feature in a web-based e-business application designed to deliver streamlined administration and reporting of all forms of equity-based compensation plans. The feature is named Exercise Options and it is used for buying shares in a company.

Phase 1 (test planning): We modeled Exercise Options from a black-box perspective by observing its behavior. That is, we executed Exercise Options using a web browser, observed its behavior, and created the model based on that (Fig. 2a). Together with the system owners we decided to focus on security risks that may be introduced via the application layer. Thus, the threat profile is someone who has access to Exercise Options, but who resides outside the network boundaries of the service provider. We identified security assets by consulting the system owners. The security asset identified for Exercise Options was *integrity of data*.

We also defined a frequency scale and a consequence scale together with the system owner. The frequency scale consisted of five values (Certain, Likely, Possible, Unlikely, and Rare), where each value was defined as a frequency interval. For example, the frequency interval for likelihood Possible was $[5,20):1y$, which means “from and including 5 to less than 20 times per year”. The consequence scale also consisted of five values (Catastrophic, Major, Moderate, Minor, and Insignificant), where each value described the impact by which the security asset is harmed. For example, consequence Major with respect to security asset *integrity of data* was defined as “the integrity of customer shares is compromised”. The scales were also used to construct a 5×5 risk evaluation matrix used to evaluate risks in Phase 2.

Phase 2 (security risk assessment): We identified security risks by analyzing the model in Fig. 2a with respect to security asset *integrity of data*. We did this by first identifying unwanted incidents. Then we identified alterations

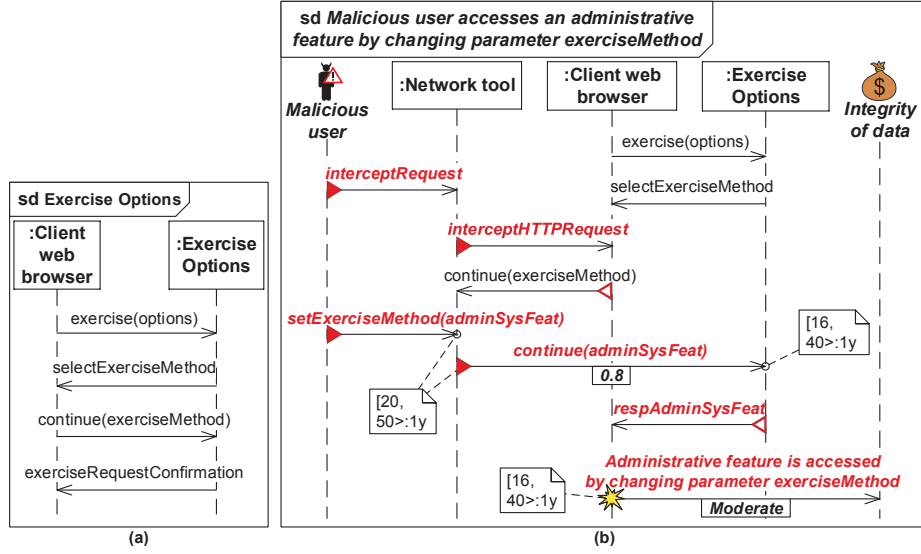


Fig. 2. (a) Black-box model of Exercise Options. (b) Threat scenario.

that have to take place in the messages in order to cause the unwanted incidents. Finally we identified messages initiated by the threat which in turn could cause the alterations.

Let us consider a threat scenario for the black-box model of Exercise Options. Assume that a malicious user attempts to access an administrative feature by altering certain parameters in the HTTP request sent to Exercise Options. The malicious user could achieve this, for example, by first intercepting the request containing the message `continue(exerciseMethod)` using a network proxy tool such as OWASP ZAP [22], and then altering the parameter `exerciseMethod` in the message as an attempt to access other features in the system. This alteration, could in turn give the malicious user access to an administrative feature. This unwanted incident occurs if the alteration is successfully carried out, and Exercise Options responds with an administrative feature instead of the expected message `exerciseRequestConfirmation`. Thus, the unwanted incident may occur after the reception of message `exerciseRequestConfirmation` (Fig. 2a). The resulting threat scenario is shown in Fig. 2b.

In order to estimate how often threat scenarios may occur, in terms of frequency, we based ourselves on knowledge data bases such as OWASP [22], reports and papers published within the software security community, as well as expert knowledge within security testing. We see from Fig. 2b that the malicious user successfully alters the parameter `exerciseMethod` with frequency $[20, 50):1y$. Given that parameter `exerciseMethod` is successfully altered and transmitted, it will be received by Exercise Options with conditional ratio 0.8 . The conditional ratio causes the new frequency $[16, 40):1y$ for the reception of message

$continue(adminSysFeat)$. This is calculated by multiplying $[20,50):1y$ with 0.8 . Given that message $continue(adminSysFeat)$ is processed by Exercise Options, it will respond with an administrative feature. This, in turn, causes the unwanted incident (security risk) to occur with frequency $[16,40):1y$. The unwanted incident has an impact on security asset *integrity of data* with consequence *Moderate*. Having identified and estimated a set of risks, we evaluated the risks by plotting them into the predefined risk evaluation matrix with respect to their frequency and consequence.

Phase 3 (security testing): Based on the risk evaluation we chose to test the risk in Fig. 2b. The test objective for this threat scenario was defined as: “Verify whether the malicious user is able to access an administrative feature by changing parameter *exerciseMethod* into a valid system parameter”. Based on this test objective, we annotated the threat scenario with the stereotypes *SUT*, *TestComponent*, *ValidationAction*, and *Verdict* as defined in the UML Testing Profile [20]. The resulting test is illustrated in Fig. 3. The security tester takes the role as “malicious user” in the test case. We carried out the test manually by following the interaction in Fig. 3, and used the OWASP Zed Attack Proxy tool [22] to intercept the HTTP requests and responses.

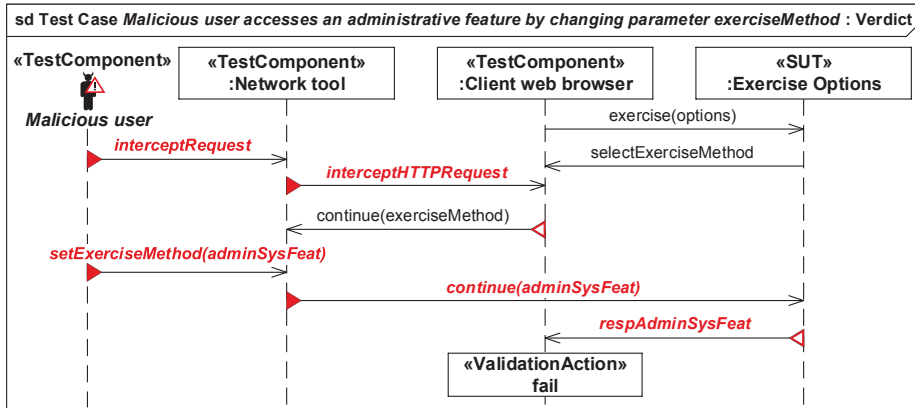


Fig. 3. Security test w.r.t. the threat scenario in Fig. 2b.

3 Why we designed CORAL as we did?

There are five main areas to consider when developing or evaluating a modeling language: domain appropriateness, comprehensibility appropriateness, participant appropriateness, modeler appropriateness, and tool appropriateness [15]. Organizational appropriateness may also be considered [15], but this is outside the scope of this paper because CORAL is not developed for a specific organi-

zation. Thus, in the following, we elaborate on why we designed CORAL as we did with respect to the first five aforementioned areas.

3.1 Domain Appropriateness

Domain appropriateness relates the modeling language to the domain it targets [15]. The purpose is to evaluate expressiveness of the language in relation to the domain. This also includes considering whether the language miss any constructs (construct incompleteness), and whether the language expresses anything that is not in the domain (construct excess).

CORAL employs constructs that are well known within the domain of testing, security, and risk assessment. The conceptual foundation of CORAL is leading international standards. Concepts related to testing are based on the software testing standard ISO 29119 [12] and the UML Testing Profile [20]. Concepts related to security are based on the information security standard ISO 27000 [10]. Concepts related to security risk assessment are based on the information security risk management standard ISO 27005 [11]. Moreover, the graphical notation of CORAL is based on UML sequence diagrams, which are among the top three modeling languages within the model-based testing community [19], and often used for testing purposes [27]. In addition, constructs inherited from UML sequence diagrams are annotated with risk-related information such as threat, unwanted incident, and security asset, which in turn brings security risk assessment to the work bench of testers without the burden of a separate risk analysis language. The CORAL process of risk assessment involves security risk modeling. The resulting risk models are used as a basis for designing and subsequently executing security tests.

The above standards and guidelines consist of a large number of concepts relevant for their respective domains. When developing CORAL we selected and related concepts which we found necessary for security testers to carry out risk-driven security testing. Security testers may carry out a complete run of risk-driven security testing using constructs provided in CORAL. This is backed up by an empirical evaluation in which we discovered that CORAL is effective in terms of producing valid risk models and identifying security tests [4].

3.2 Comprehensibility Appropriateness

Comprehensibility appropriateness relates the language to the social actor interpretation [15]. This is often evaluated with respect to design principles referred to as semiotic clarity, perceptual discriminability, complexity management, cognitive integration, visual expressiveness, dual coding, and graphic economy [18].

Each graphical symbol in CORAL is designed to represents only one semantic construct in the language. For example, the graphical icon shaped as a human with “devil horns” represents a deliberate threat, and may not be used to represent accidental threats or non-human threats, and so on. Moreover, each semantic construct in CORAL is represented by only one graphical symbol. For example, an unwanted incident is only represented by the unwanted incident message, and

may not be represented by other messages. This means that CORAL fulfills the principle of semiotic clarity [18].

The principle of perceptual discriminability states that different symbols should be clearly distinguishable from each other. To achieve this we employ distinct shapes and colors. Although the conditional ratio symbol and the consequence symbol are rectangular, they are easily distinguishable because conditional ratios may be assigned to general, new, and altered messages, while consequences are assigned only to unwanted incident messages. In addition, conditional ratios are always represented as nonnegative real numbers, while consequences are always represented textually. However, the new, altered, and deleted messages are similar in the sense that they all have a triangular shape at the transmission end, but they are distinguishable with respect to the coloring inside the triangles. In our experience, CORAL risk models typically contain a greater number of new messages compared to the number of altered and deleted messages. In some cases, particularly in large risk models, this makes it somewhat difficult to spot the altered/deleted messages. We may mitigate this by using different shapes at the transmission end on new, altered and deleted messages. However, the reason why we use triangles (in combination with the color red) is to support semantic transparency, which is discussed in Sect. 3.3.

With respect to the principle of complexity management, our experience shows that the *ref* construct is sufficient to manage the complexity of CORAL risk models [3–5]. Because of its modular property, the *ref* construct may also be used to support cognitive integration, i.e., to support integration of information from different diagrams. Although the information in a *ref* construct is limited to abstract descriptions of the referred interaction, it is sufficient for constructing high-level risk models, which are useful to obtain an overview of the various threat scenarios and their relationships. Thus, in CORAL we may divide complex risk models into simpler risk models, as well as compose high-level risk models, by making use of the *ref* construct.

The principles of visual expressiveness and dual coding refer to the usage of the full range and capacities of visual variables, and the usage of text to complement graphics, respectively. To achieve this we use a red colored, bold, and italic font to highlight the risk-related information (text) on messages. This comes in addition to symbols that are distinguishable with respect to shape and color. Based on our experience, this convention is useful for new and altered messages, as well as unwanted incidents. The text on new messages is always formatted as risk-related information because these messages are initiated by threats. The text on altered messages is formatted as risk-related information when highlighting the alteration in the message. This could be part of the text or the complete text on the altered message. The text on unwanted incidents are always formatted as risk-related information because they represent that assets are harmed or reduced in value. This formatting strengthens the visual expressiveness and helps security testers keep track of and distinguish between risk-related and non risk-related information on the messages.

The principle of graphic economy states that the number of different graphical symbols should not exceed 6 in order to be cognitively manageable. However, if a language consists of more than 6 symbols, which is the case in CORAL, then one can deal with graphic complexity by increasing visual expressiveness. As explained above, to achieve this we format text to complement the graphics, which in turn strengthens the visual expressiveness. In particular, we position the symbols representing new, altered, deleted, and unwanted incident messages so that they are horizontally aligned with the message, as well as correctly oriented with respect to the message direction. These two visual variables give an additional increase to the visual expressiveness [18].

3.3 Participant Appropriateness

Participant appropriateness relates the participant knowledge to the language [15]. This is often evaluated with respect to the design principle referred to as semantic transparency [18].

The principle of semantic transparency states that symbols should use visual representations whose appearance suggests their meaning. To achieve this, we base the risk-related symbols used in CORAL on corresponding symbols used in the CORAS risk analysis language [17]. The graphical symbols in CORAS have been empirically shown to be cognitively effective [8], and these concepts are also commonly used in security testing [23], which is why we use similar symbols in CORAL.

3.4 Modeler Appropriateness

Modeler appropriateness relates the language to the knowledge of the one doing the modeling [15]. This is often evaluated with respect to the design principle referred to as cognitive fit [18].

The principle of cognitive fit states that the language should use different visual dialects for different tasks and audiences. CORAL is mainly to be used by security testers, for the purpose of risk-driven security testing. This implies that CORAL must provide concepts and a corresponding graphical notation necessary to carry out security risk assessment, as well as security testing. As discussed above, we provide this by basing CORAL on state of the art standards and guidelines. However, this also means that CORAL requires testers to be familiar with security risk assessment. Security testers usually have this required background, because they often have to carry out activities related to security risk assessment, such as creating security abuse/misuse cases, performing architectural risk analysis, and building risk-driven security test plans [23].

3.5 Tool Appropriateness

Tool appropriateness relates the language to the interpretation from the technical audience (tools) [15]. A prerequisite for tool interpretation is that the language

must have a syntax and semantics that are formally defined. CORAL is accompanied by an abstract syntax as well as a schematically defined natural-language semantics [3]. Testers may use the abstract syntax in order to create risk models that are syntactically correct, and the natural-language semantics in order to clearly and consistently document, communicate and analyze risks.

4 What we could have done differently?

This section discusses what we could have done differently with respect to the design of CORAL.

4.1 Graphical Versus Textual

A model may be either two-dimensional or one-dimensional³. A graph is two-dimensional while text is one-dimensional. CORAL is obviously two-dimensional. A one-dimensional alternative would be to replace the UML diagrams by actual code and the specific graphical annotations of CORAL by textual annotations. One argument in favor of such an approach is that it would be sufficient for the tester to know the source-code language. However, the price to pay would be no abstraction. The tester would have to create a mental model of how security risks are caused including the chain of events and how they may affect the system to test, and based on that describe the risk picture. Moreover, the tester would have to read through code from top to bottom to capture details such as unwanted incidents, frequencies, conditional probabilities, consequences and so on.

Using UML sequence diagrams we cover a scenario that occurs multiple times in the source code by a single diagram. Moreover, UML sequence diagrams capture the interaction between independent actors and processes in a manner not possible using source code. Finally, UML sequence diagrams allow us to describe the behavior of human actors including working procedures as well as threat behavior.

Finding the right balance between text and graphics in annotations is non-trivial. As argued in [6], text labels are often preferred over graphical means. Hence, finding the right balance is essential. The recently completed EMFASE project arrived at similar conclusions [1].

4.2 Risk Annotations Versus Tables

An alternative to the CORAL approach of representing risk-related information on top of sequence diagrams is to document the risk-related information separately using tables. The most commonly used table-based risk assessment approach is Hazard and Operability (HazOp) analysis [9]. Many risk-driven testing approaches use table-representations inspired from HazOp analysis [2]. HazOp makes use of guide words to identify risks, their causes, as well as possible

³ A model may of course also be three-dimensional, but that is not relevant in this paper.

treatments. Figure 4 illustrates a typical HazOp table in which we represent the threat scenario in Fig. 2b.

Element	Characteristics	Guide word	Deviation	Possible causes	Consequences	Safeguards	Comments	Action required
Exercise Options form	Selling or buying shares in a company	No (not)	HTTP requests not sanitized	Web-application form has no input validation	An admin-feature is accessed	No	HTTP requests may have been tampered to access restricted features	Implement input validation mechanism

Fig. 4. HazOp table representing the threat scenario in Fig. 2b.

Tables may also be regarded as two-dimensional because a cell in the table can be identified by pairs of row and column headings. Moreover, tables present all information consistently with respect to the headings. This guides the reader to the relevant information in a structured manner. Finally, information in tables are generally presented as text. This removes the need to interpret semantics of graphical symbols when looking for certain information.

CORAL is based on the hypothesis that it is advantageous to annotate risk information on the locations where it belongs in a style corresponding to the underlying modeling language. For example, the transmission frequency of message *continue(adminSysFeat)* in Fig. 2b is attached to the transmission-end of the message, while the reception frequency is attached to the reception-end. Moreover, a particular location in the diagram may convey more than one kind of information. For example, the transmission-end of message *continue(adminSysFeat)* in Fig. 2b simultaneously conveys that the message is a *new message*, that the transmission occurs with frequency $[20, 50]:1y$, and that it is transmitted by *Network tool*. This information would normally be found in separate columns in a table.

On the other hand, whether tables are better than graphs or the other way around is far from obvious [16]. The answer depends probably on the context of use and the complexity of the information to be presented.

4.3 Sequence Diagrams Versus Other UML Representations

The graphical notation of CORAL could have been based on modeling languages other than UML sequence diagrams. According to Dias-Neto et al. [19], the three most common modeling notations (not including UML sequence diagrams) used in model-based testing are UML state machines/finite state machines, class diagrams, and use-case diagrams.

State machines are specifications of sequences of states that an object or an interaction goes through in response to events during its life, together with its responsive effects [21]. These sequence of states correspond to events that occur chronologically on a particular lifeline in CORAL. In principle, it is possible to

represent the same risk-related information as in CORAL using state machines. The advantage of sequence diagrams when compared to state machines is that we may easily isolate particular scenarios without having to consider the behavior for other scenarios. This is very much in the spirit of testing and an important reason for the development of sequence diagram notation.

Class diagrams capture the static view of a system as a collection of declarative (static) model elements with contents and relationships [21]. To this end, class diagrams are useful to describe the structure of the system to test. However, the kind of dynamic behavior that CORAL address cannot be specified using class diagrams.

Use-case diagrams show the relationships among actors and use cases within a system [21]. Their high-level nature is useful for capturing high-level threat scenarios a system may be exposed to. A threat can be modeled as an actor, while a high-level scenario corresponds to use-case. Misuse cases [26] is a well-known notation based on use-case diagrams used to capture high-level threat scenarios. However, in the context of testing, high-level threat scenarios are only useful as a starting point to design detailed security tests. CORAL addresses the latter, that is, designing detailed security tests.

4.4 CORAL Versus Attack Trees

The annotation of sequence diagrams may be thought of as augmenting the sequence diagrams with an attack tree on which there is a huge literature [14]. CORAL allows the representation of sequential conjunction [13] and disjunction, but not ordinary conjunction, for which we have not seen any real need. Instead of embedding the CORAL annotations within the sequence diagrams we could of course use attack trees in addition to sequence diagrams in the same way as some approaches to risk-driven testing use tables in addition to the system documentation. However, as for tables, we think intended users benefit from an integrated approach.

4.5 CORAL Versus Formal Methods

CORAL is supported by an abstract textual syntax formalized in EBNF [3]. The semantics of CORAL is defined by a schematic translation of any syntactically correct CORAL expression into English sentences. The target audience of the natural-language semantics is security testers, and the purpose is to help testers clearly and consistently document, communicate and analyze security risks.

Although formal in the sense described above, CORAL is not formal in the classical meaning of formal methods. This would require a mathematical semantics as well as a formalization of the natural-language expressions used to annotate CORAL diagrams. A mathematical semantics would indeed be useful, not to replace the natural-language semantics which targets the users of CORAL, but to allow tool and method developers building on CORAL prove soundness of the rules and heuristics.

Formalizing the natural-language expressions would on the other hand be counter-productive. We believe it is a strength of CORAL that security testers freely can augment their diagrams without being constrained by formal concerns.

To summarize, a formal semantics would be beneficial and we hope to provide this in the future, for example, based on STAIRS [7] or probabilistic STAIRS [24] for sequence diagrams. Formalizing the natural-language expressions would probably alienate the CORAL approach from its intended users, namely security testers.

5 Conclusion

In this paper we have presented the CORAL language for risk-driven security testing, motivated some of the major design decisions on which it builds, and discussed what we could have done differently with respect to the design of the language.

The target audience of CORAL is security testers. We have tried to explain why we think CORAL is comprehensible to security testers and why it is appropriate to use for risk-driven security testing.

With respect to what we could have done differently we considered various alternatives and their impact on CORAL. In particular, we discussed why we decided to develop CORAL as a graphical language instead of augmenting code, why we embed risk-related annotations in sequence diagrams instead of using separate tables or attack trees, why we do not build on other UML notations instead of sequence diagrams, and why formalizing the natural-language expressions in CORAL diagrams is counter-productive.

Acknowledgments. This work has been conducted as part of the EMFASE project funded by SESAR Joint Undertaking (SESAR WP-E project, 2013-2016) managed by Eurocontrol, and the AGRA project (236657) funded by the Research Council of Norway under the BIA research programme.

References

1. Empirical Framework for Security Design and Economic Trade-Off (EMFASE). <https://securitylab.disi.unitn.it/doku.php?id=emfase>, 2016. Accessed April 21, 2016.
2. G. Erdogan, Y. Li, R.K. Runde, F. Seehusen, and K. Stølen. Approaches for the combined use of risk analysis and testing: a systematic literature review. *International Journal on Software Tools for Technology Transfer*, 16(5):627–642, 2014.
3. G. Erdogan, A. Refsdal, and K. Stølen. Schematic Generation of English-prose Semantics for a Risk Analysis Language Based on UML Interactions. Technical Report A26407, SINTEF Information and Communication Technology, 2014.
4. G. Erdogan, K. Stølen, and J.Ø. Aagedal. Evaluation of the CORAL Approach for Risk-driven Security Testing based on an Industrial Case Study. In *Proc. 2nd International Conference on Information Systems Security and Privacy (ICISSP'16)*, pages 219–226. SCITEPRESS, 2016.

5. Gencer Erdogan. *CORAL: A Model-Based Approach to Risk-Driven Security Testing*. PhD thesis, University of Oslo, 2015.
6. I.H. Grøndahl, M.S. Lund, and K. Stølen. Reducing the Effort to Comprehend Risk Models: Text Labels Are Often Preferred Over Graphical Means. *Risk Analysis*, 31(11):1813–1831, 2011.
7. Ø. Haugen, K.E. Husa, R.K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Software & Systems Modeling*, 4(4):355–357, 2005.
8. I. Hogganvik and K. Stølen. A Graphical Approach to Risk Identification, Motivated by Empirical Investigations. In *Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*, pages 574–588. Springer, 2006.
9. International Electrotechnical Commission. *IEC 61882, Hazard and Operability studies (HAZOP studies) - Application guide*, 2001.
10. International Organization for Standardization. *ISO/IEC 27000:2009(E), Information technology - Security techniques - Information security management systems - Overview and vocabulary*, 2009.
11. International Organization for Standardization. *ISO/IEC 27005:2011(E), Information technology - Security techniques - Information security risk management*, 2011.
12. International Organization for Standardization. *ISO/IEC/IEEE 29119-1:2013(E), Software and system engineering - Software testing - Part 1: Concepts and definitions*, 2013.
13. R. Jhawar, B. Kordy, S. Mauw, S. Radomirovic, and R. Trujillo-Rasua. Attack Trees with Sequential Conjunction. In *Proc. 30th International Conference on ICT Systems Security and Privacy Protection (SEC'15)*, pages 339–353. Springer, 2015.
14. B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer. DAG-based attack and defense modeling: Dont miss the forest for the attack trees. *Computer Science Review*, 13-14:1–38, 2014.
15. J. Krogstie. *Model-Based Development and Evolution of Information Systems - A Quality Approach*. Springer, 2012.
16. K. Labunets, Y. Li, F. Massacci, F. Paci, M. Ragosta, B. Solhaug, K. Stølen, and A. Tedeschi. Preliminary Experiments on the Relative Comprehensibility of Tabular and Graphical Risk Models. In *Fifth SESAR Innovation Days*, pages 1–7. SESAR WPE, 2015.
17. M.S. Lund, B. Solhaug, and K. Stølen. *Model-Driven Risk Analysis: The CORAS Approach*. Springer, 2011.
18. D.L. Moody. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *Transactions on Software Engineering, IEEE*, 35(6):756–779, 2009.
19. A.C. Dias Neto, R. Subramanyan, M. Vieira, and G.H. Travassos. A Survey on Model-based Testing Approaches: A Systematic Review. In *Proc. 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07)*, pages 31–36. ACM, 2007.
20. Object Management Group. *UML Testing Profile (UTP), version 1.2*, 2013. OMG Document Number: formal/2013-04-03.
21. Object Management Group. *Unified Modeling Language (UML), version 2.5*, 2015. OMG Document Number: formal/2015-03-01.
22. Open Web Application Security Project. https://www.owasp.org/index.php/Main_Page, 2016. Accessed April 20, 2016.
23. B. Potter and G. McGraw. Software Security Testing. *Security & Privacy, IEEE*, 2(5):81–85, 2004.

24. A. Refsdal, R.K. Runde, and K. Stølen. Stepwise refinement of sequence diagrams with soft real-time constraints. *Journal of Computer and System Sciences*, 81(7):1221–1251, 2015.
25. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, 2005.
26. G. Sindre and A.L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.
27. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.