Eliciting Security Requirements by *Mis*use Cases

Guttorm Sindre
Dept of Computer and Info. Sci.
Norwegian Univ. of Sci. and Tech.
guttorm@idi.ntnu.no

Andreas L. Opdahl
Dept of Information Science
University of Bergen, Norway
andreas@ifi.uib.no

Abstract

Use case diagrams have proven quite helpful in requirements engineering, both for eliciting requirements and getting a better overview of requirements already stated. However, not all kinds of requirements are equally well supported by use case diagrams. They are good for functional requirements, but poorer at, e.g., security requirements, which often concentrate on what should ***not*** *happen in the system. With the advent of e- and m-commerce applications security requirements are growing in importance, also for quite simple applications where a short lead time is important. Thus, it would be interesting to look into the possibility for applying use cases on this arena. This paper suggests how this can be done, extending the diagrams with* ***mis****use cases. This new construct makes it possible to represent actions that the system should prevent together with those actions which it should support.*

Keywords: use cases, requirements, security

1. Introduction

Use case diagrams (UCD) [1] have proven quite helpful for the elicitation of requirements [2]. Some research indicates that software development projects where the analysis phase concentrates on UCDs rather than textual requirements may actually be more successful in capturing the user needs [3,4]. This seems to be because simple and intuitive diagrams provide better overview of the functionality of a system, and the explicit focus on actors makes it easier to see each stakeholder's interest in the system, thus aiding the communication with end-users.

However, there are also problems with use case based approaches to requirements engineering. As stated in [5]: "A collection of use cases is not a suitable substitute for a requirements specification". Thus, projects where use cases more or less replace specifications in the more traditional sense, may have serious trouble missing significant requirements. One typical problem with the quick and intuitive kind of modeling associated with use cases is over-simplified assumptions about the problem domain [6]. Another problem associated with use cases is the tendency to go prematurely into design and implementation considerations, especially concerning the user interface [6]. Partly, of course, this may be due to problems with the method applied for use case modeling in the particular projects that those studies were based on, rather than an essential problem with the UCD modeling language itself. But there are also some intrinsic problems with use case diagrams for requirements elicitation, namely that they are not equally representative of all kinds of requirements.

A use case typically describes some function that the system should be able to perform for the user – or, if the model is on a higher level of abstraction where the system boundary has not yet been decided, some business function that the user and system should provide in cooperation. This means that use cases will be good at reflecting so-called functional requirements, but maybe not so good for non-functional requirements.

One example where use cases might not be quite adequate, are security requirements. Traditionally, security has been defined as the system's ability to prevent unauthorized access. E.g., "the prevention of, or protection against, access to information by unauthorized recipients, and intentional but unauthorized destruction or alteration of that information" [8] or "attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data" [9]. But these definitions do not cover all aspects associated with security today. One example is the system's ability to resist *denial of service (DoS)* attacks, which need not involve any unauthorized access at all. One typical style of DoS attack is to flood a system with stupid (but allowed) requests to create a performance bottleneck, thus disrupting its service to honest users. As noted by [10] it is actually quite difficult to come up with a good definition of security. But then again, its exact definition is not vital to this paper. Regardless of the exact definition, our point remains the same: Security requirements will seldom be explicitly stated at the outset of a project. Rather, the stakeholders will state them as *concerns* about things that should *not* happen in the system [11]. Since use cases, by their nature, concentrate on what the system *should* do, they obviously have less to offer when describing the opposite.

The reason why this paper focuses on security requirements and not other non-functional requirements, is twofold:

- With the advent of e-commerce, security requirements are becoming more and more important, even for quite small and simple applications. Moreover, such applications usually have strong demands for short lead times. This makes them a potentially interesting arena for the quick and intuitive approach of use cases.
- Whereas some other classes of non-functional requirements seem clearly beyond the scope of use case diagrams, it is imaginable that some modifications could make use case diagrams helpful for security considerations. After all, functions that the system should *not* allow are still functions, which could potentially be covered by use cases.

In the following, we will investigate in more detail if it is possible to have use case diagrams show both what we want from the system *and* what we want to avoid/prevent. The rest of the paper is structured as follows: In section 2 we take a brief look at various simple alternatives of representing unwanted behavior in use case diagrams. Section 3 gives a more detailed account of our particular approach, representing these negatives together with ordinary positive system behavior. Section 4 outlines a method for using the new constructs. Section 5 discusses the feasibility of the approach and its relation to other work. Section 6 concludes the paper.

2. A brief look at some naïve alternatives

The first alternative to consider is of course the zero alternative. Many projects in the past have indeed captured security requirements, by methods not involving use cases, as discussed e.g. in [19]. Thus, one may go on capturing security requirements by other means and apply use cases primarily for functional requirements. But such use of two separate approaches always involves some overhead and potential problems of checking consistence across representations. Moreover, in many cases it is just the effective functionality provided for legitimate users that also makes it easier for crooks to achieve their goals. One well-known example here is the automatic file name completion in UNIX (although that operating system may not be the one with the biggest security problems). The feature helps legitimate users to work faster, since you only need to write the first couple of letters in a file name and then hit the space bar. But it also helps intruders find files quickly, since their names need only be half-guessed, not known precisely.

Thus, it may be interesting to look at functional requirements and security requirements together, and if this could be done within one method – for instance use case modeling – the situation might be simpler. This makes it interesting to look into possible extensions of use cases for eliciting security requirements.

First, we need a couple of definitions to make the following discussion more precise. As stated in the documentation of UML v.1.3. [12]:

- "The use case construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity".

This definition does not have to be changed, but in addition we state the following:

- A use case generally describes behavior that the system/entity owner wants it to provide.
- A *mis*use case is a special kind of use case, describing behavior that the system/entity owner does not want to occur.

Thus, apart from being something unwanted, a misuse case will have all the same properties as an ordinary use case. The misuse case, too, can be specified in terms of a sequence of actions, including variants. It, too, can have relations to actors, and be at either end of the standard arcs "includes", "extends", "generalizes" defined between use cases.

Similarly, [12] defines an actor to be "a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates". Here, we might also add a definition of our own:

- A *mis*-actor is a special kind of actor who initiates misuse cases.

What is to be investigated, then, is how to represent misuse cases and mis-actors in use case diagrams. The most naïve approach would be depict these in the same manner as ordinary actors and use cases, without changing the diagram language at all. A small and quite incomplete e-commerce example illustrating this is shown in Figure 1. Here the system is supposed to offer some functionality for customers and operators, and we have also included in the model a crook, who might want to acquire the customers' credit card number or infect the system with a virus.

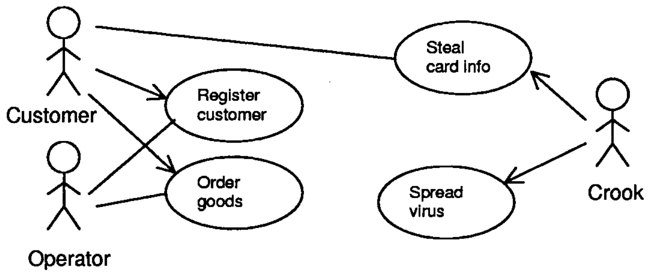

Figure 1: Using the standard diagrams

The only advantage that could be claimed for this approach, is that it was unnecessary to modify the diagram language. The disadvantages clearly outweigh the simplicity of this naive approach. In this simple example it might be clear from the names of the various nodes that the "crook" is a mis-actor, and that "steal card info" and "spread virus" are misuse cases, whereas the other use cases and actors are those that should be supported. But with slightly bigger examples (and possibly even with this one, had the reader not been warned beforehand), there will soon be confusion. Knowing the huge variations in people's abilities to deal with irony, it seems unwise to mingle use and misuse in one diagram without making it absolutely obvious which is which.

Another approach might be to separate use and misuse into different diagrams. Thus, there would be one depicting the major use cases of the system, another the major misuse cases. A page heading could inform the reader whether it is one or the other. This approach is shown in Figure 2. Clearly, such strong separation of the positive and negative might sometimes be useful, for instance to prevent diagrams from becoming too big and complex. But sometimes it might also be interesting to model the two together, since the negative and positive are closely connected:

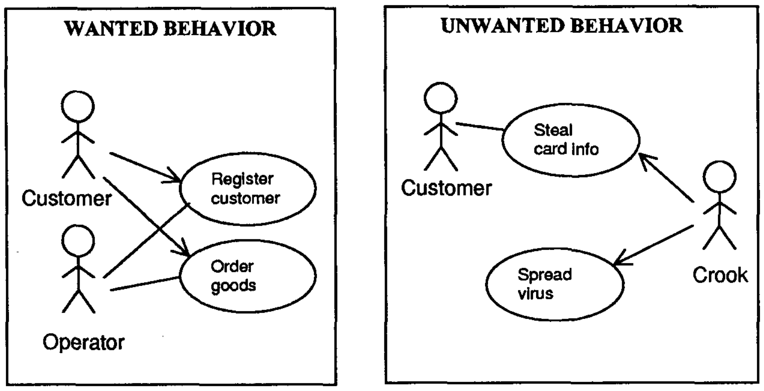

Figure 2: Use and misuse on different sheets

- Intended sabotage may often be achieved by means of ordinary functions that the system has to support – the problem being that somebody got hold of somebody else's password. Then, it may be interesting to place revealing "includes"-associations between misuse cases and use cases, which would be impossible if totally separate diagrams were used.
- Sometimes prohibited actions may be caused or aided by the normal actors of the system, by intent or accident/failure to comply with security standards, e.g., using a password which is too easily guessed. And anyway, normal actors are likely to be stakeholders of prohibited actions. In Figure 1 the customer was noted as a stakeholder of the "steal card info" action, meaning that the customer could be affected by this (for instance suddenly finding his account empty). With the divided approach of Figure 2, this meant that we had to depict the customer in both diagrams. If various parties affected should be registered for all illegal actions, practical models would yield lots of actor node duplication, clearly an inefficient way of modeling.

- For the most likely security threats, a system will usually have to provide functions to deal with these threats, i.e., captured misuse cases will again inspire new use cases. Again, this means that associations between use and misuse can yield quite revealing diagrams.

In many cases, then, it may be interesting to look at use and misuse together, which is our suggested approach. In the spirit of use case diagrams, we still try to keep it simple, the idea being something that almost anyone could have come up with. As in Figure 1, use and misuse will be depicted in the same diagram. But to make it obvious which is which, misuse (and any mis-actors) will be shown in an inverted format, as indicated in Figure 3.

In all the diagrams in this paper it can be noted that we use UML v.1.2's possibility to have arrows between actor and use case, not just undirected lines. This is in accordance with a case study using UML v.1.3 [13], which indicated that the removal of arrows here made the language poorer, losing the ability to show who initiates a use case. In figure 3 this helps to show that it the "Crook" who does "Steal card info", whereas the "Customer" is only affected by it.

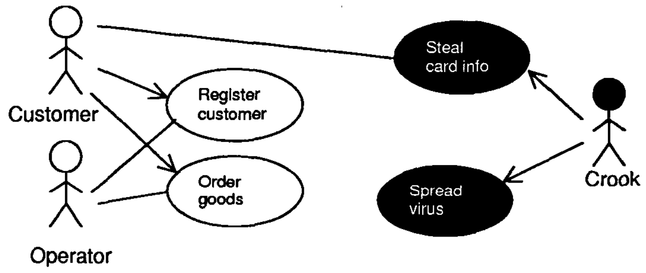

Figure 3: Misuse and mis-actors inverted

3. Co-representing use and misuse

The approach can be illustrated in more detail by the example of Figure 4, where we have also included some new kinds of relations, "detects" and "prevents". This example illustrates part of the functionality for a typical e-commerce system, where customers do not need to log on but simply browse the catalog and order goods, giving their credit card number together with name and address info before the order is submitted. The operator working on the server side has to log on. The operator will also perform other functions not shown in the diagram, such as registering new goods for sale, deleting goods which the company has quit selling, etc. These have been omitted for simplicity.

The diagram has one mis-actor, called "crook", who may perform misuse cases such as "steal card info", "flood system" (a typical DoS attack), "tap communication", and "acquire password". We have also indicated some "includes"-relations between misuse cases, just the same way as these may exist between ordinary use cases. For instance, "steal card info" could use "tap communication", acquiring the customer name and credit card number as it is transferred from client to server. If the operator is not sitting directly at the server but connected to this via some network, "obtain password" could use "tap communication", too. But the password could also be obtained by repeated guesses, here we see a misuse case using a normal use case, "Log on". The "Flood system" misuse could possibly be obtained by massively repeated attempts to register customers (real or fictitious, the same over and over again or with some name changes for each iteration). Again we see a misuse case using a normal use case, and this time without any unauthorized access.

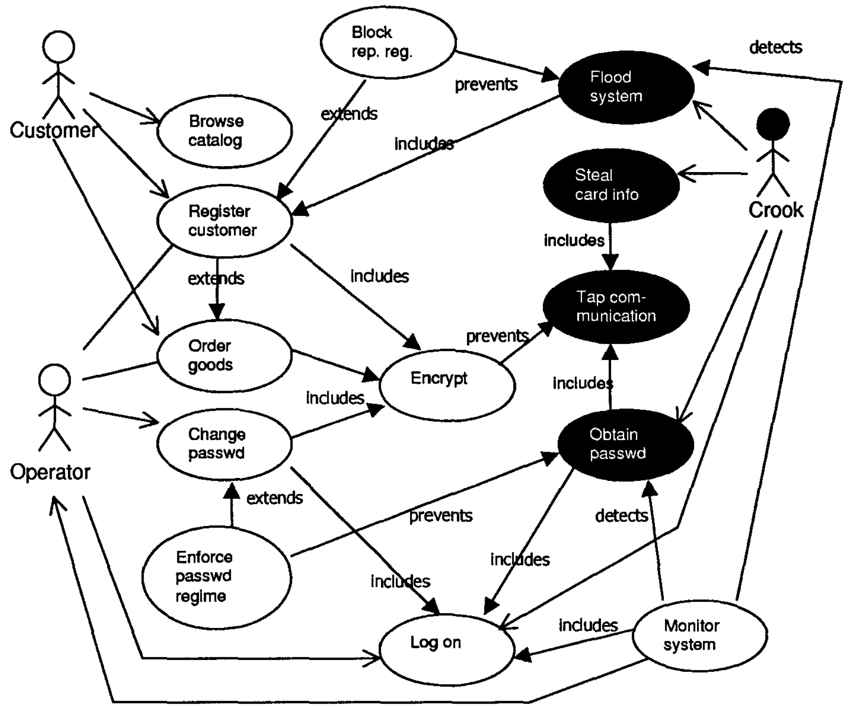

Figure 4: A bigger example

In addition to the standard "includes" and "extends" relations, we have introduced some new relations in the diagram. These are particularly interesting with respect to use cases dealing with misuse cases and are called "detects" and "prevents". The meaning of these should be pretty clear from their names:

- "prevents": the function provided by the use case that the arrow originates from, prevents the activation of the misuse case that the arrow is directed towards, at least in some cases.
- "detects": the function provided by the use case that the arrow originates from, detects the activation of the misuse case that the arrow is directed towards, at least in some cases.

The "at least in some cases" part is important to notice, since it is seldom the case that one countermeasure will detect or prevent all possible ways that a certain misuse can be performed. This is illustrated by several of the "prevents" and "detects" relations shown in fig. 3. For instance, "Enforce password regime" can prevent the crook from easily obtaining passwords by repeated guessing. But if some employee has written his password on a piece of paper and the crook gets hold of this, it all falls to pieces. It could be possible to distinguish diagrammatically between certain and less certain countermeasures. For instance, one could use "prevents" and "detects" about the former, "may prevent" and "may detect" about the latter. Or one could direct the "prevents" and "detects" arrows towards other arrows instead of misuse cases, e.g., directing the "detects" arrow from "Monitor system" not to the misuse case "Obtain password", but to the includes-relation between this and "Log on". This would signify that the monitor function will not detect all attempts to obtain a password, only those applying repeated guesses. However, this would complicate the diagrams too much. It seems better to take the view that no countermeasure is a 100% failsafe and rather describe the exact details about what security it provides in the text accompanying the use case - then the diagrams can still be kept simple, which will support communication with the stakeholders.

The mis-actor concept could be discussed in some more detail. As stated previously, the ordinary actor construct denotes a comprehensive set of roles played towards the system. As described in standard UML literature [14] not all actors need persons or groups of persons as role players, some may be played by other systems which the current one must interact with. The same goes for mis-actors, since obviously, a security threat may also be caused by failure in another system whose cooperation we are somehow relying on. Actually, mis-actors may be even more abstract than ordinary actors, for instance "Bad luck" or "Devil". Such mis-actors will be particularly needed in cases where security threats arise from unexpected equipment failure, sudden operator illness and the like. Thus, the elicitation of security requirements will not be limited to threats caused by hostile persons.

Another aspect concerning mis-actors is the case when an ordinary actor behaves in a treacherous manner, either accidentally or by intent. Should these then be modeled as two different actors, or as the same actor being connected both to use cases and misuse cases? The latter alternative will easily lead to situations where a huge number of use cases are connected to each actor — then it might be tidier to use duplicate actors. Giving them the same name (e.g. "Operator") will still signal that we are talking about the same role in the organization, just that the ordinary actor denotes the operator performing as prescribed, whereas the corresponding mis-actor denotes an operator whose behavior is not in accordance with security regulations. These latest points are illustrated in Figure 5.

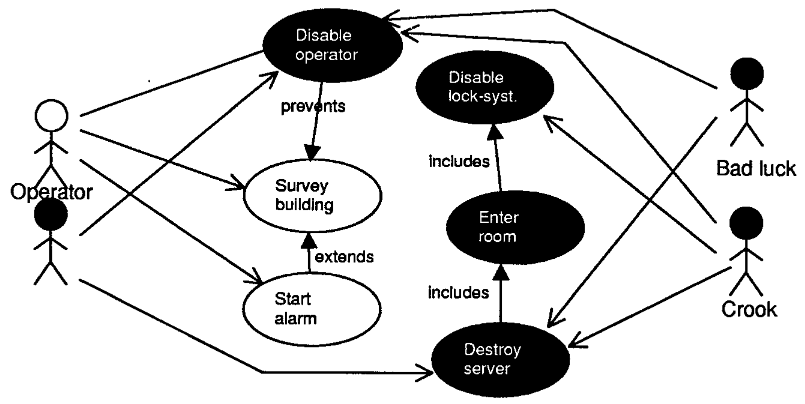

Figure 5: Some special mis-actors

Here, the mis-actor "Bad luck" is introduced for easier elicitation of unlucky incidents that nobody is really guilty of, but which may still jeopardize security. The operator may be disabled, for instance by a sudden health condition such as a heart attack, and thus be prevented from carrying out normal duties required to ensure system security. As the figure indicates, though, the operator's disability need not be bad luck. It may also be caused on purpose by the crook (poisoning?) or even by the operator himself, closing an eye to assist the crook. Similarly, destruction of the server may result from some accidental equipment failure (bad luck), but also from crook or operator manipulation (in the latter case, either by accident or intent). Here, it can be seen that the previously mentioned possibility of arrows to show initiation of use cases, is quite important. It is certainly helpful to know whether "Disable operator" is something done to the operator by others (the line between the actor and the misuse case) or done by the operator himself (the arrow from the mis-actor to the misuse case).

4. Method guidelines

The fact that the examples shown in Figures 3 and 4 are quite incomplete (i.e., there are many other use and misuse cases which could also be thought of, as well as additional relations between them), illustrates that diagrams quickly get complex. This may of course also be a problem with ordinary use case diagrams, but the addition of misuse cases and new relations has of course contributed to the complexity. Since one of the main advantages of use cases is their simplicity, this means that the added diagram features should be used soberly, and complexity should be added progressively, so that at each level the diagram is ideal for a certain purpose. A step by step method can be outlined as follows:

1. First concentrate on the normal actors and the main use cases requested by these, i.e., the services that the users want, regardless of any security considerations. Describe actors and use cases in the normal way suggested by UML methodology.
2. Then introduce the major mis-actors and misuse cases, i.e., threats that are reasonably likely. If possible, there could be more mis-actors than given in our examples, and they could be more precisely named. E.g., if the procuring company has a special concern that their competitors might sabotage the system, one mis-actor could be called competitor. This would give a clearer picture of that mis-actor's motivation.
3. Investigate the potential relations between misuse cases and use cases, in terms of potential "includes"-relations. This step is quite important since many threats to a system can largely be achieved by using that system's normal functionality, as for instance the "Flood system" misuse case of our example.
4. Introduce new use cases with the purpose to detect or prevent misuse cases.
5. Continue with a more detailed requirements documentation.

Of course, this need not be a sequential process, but could also be done in much more iterative manner – finding in one step that something was forgotten in a previous step. But generally, the progression 1-5 seems sensible. The end-users and system owners basically want some services to be provided – and without services there is no system and thus no potential misuse anyway. Thus, step 1 will simply be ordinary, high-level use case modeling without any of the extensions suggested in this paper. In step 2-3 the misuse cases are introduced and their relationship with the initial use cases investigated. The new use cases introduced in step 4 can somehow be seen as secondary use cases. "Monitor system", "Enforce password regime" etc. are not services that the users really wanted or needed as such, but services necessitated by security threats.

The indicated progression may also allow for different sets of participants in various steps of the analysis. Steps 1-2 and possibly 3 could easily involve end-users. During step 3 and particularly 4, discussions would become more technical. Now, security engineers are more likely participants in the modeling process. With a more technically trained group of participants it may not be such a problem after all that the diagrams become more detailed during these stages.

Step 5 indicates that there are still lots of work to be done after the modeling of use and misuse cases. As mentioned earlier, use case diagrams should complement more precisely formulated requirements, not replace them. For security requirements it is obvious that there are many considerations which cannot be expressed in our diagrams, such as

- Who are the potential mis-actors, what is their motivation, knowledge, and destructive capability? The mis-actors can be of many kinds, such as deliberate criminals, challenge-seeking youngsters, dissatisfied employees, dubious comptetitors – and, as seen in previous section, just bad luck. Although the diagrams provide a nice way of identifying the various mis-actors, a more precise description and analysis must be made in other forms.
- What is the likelihood of various threats, the cost of the potential damage – not only in direct economical losses, but also in terms of lost trust and goodwill from customers.
- What are the possible countermeasures towards the various threats, and the costs of these – which must be feasible in light of the threats' costs and probabilities.
- To what extent can it be verified that the selected countermeasures really deal with the threats.

Here, there is obviously room for more formal approaches. Our misuse diagrams must only be seen as a support for eliciting threats and corresponding security requirements up front, not as a complete method to document and analyze these requirements.

5. Discussion and related work

The previous section ended with a statement that only a small part of the job would be done by our misuse cases, and that more formal approaches would be needed in addition. Thus, one might question the usefulness of our approach altogether. Might we not just as well have been content that use cases are most suitable for functional requirements, and use other techniques from the very beginning when other kinds of requirements have to be captured? Then, use case diagrams would be kept simple, and elsewhere, more formal methods could be used. For safety-critical systems formal methods have traditionally had a high standing [15]. Can it be said about our approach that it is still to informal to be of any help with security requirements, thus falling between two chairs, only achieving some unnecessary complications of use case diagrams?

We think not. First of all, as long as security threats are not being considered, our method suggests that the added features will not be used. Thus, there is no complication of the diagrams until actually needed. And an alternative approach, where ordinary functional requirements are analyzed by use cases but security requirements only in other, textual documents, will not necessarily be less complex – then the connections between use cases and potential misuses of these would have to be registered some other way.

Moreover, there are several arguments why an approach based on use cases could be helpful in the early elicitation of security requirements even if more formal approaches have to be used afterwards:

- User/customer assurance. Although the end-users and management of the procuring organization may not be able to discuss the technical details of security threats and countermeasures, they will have concerns about various threats. Seeing these captured in the use case diagram will reassure them that the problem is actually being dealt with – in a much more direct manner than a mere textual discussion of security threats which they have problems understanding.
- Analyst creativity. Computer criminals are surely going to be creative in their attempts to fool the system. Thus analysts need to be equally creative to identify the relevant threats beforehand. For all the positive things that can be said about formal methods, their rigidity may not inspire creativity in a brainstorming sense. In an initial phase where the knowledge about the modeled domain is vague, this may actually hamper the process – analysts seem to need languages allowing for some vagueness and incompleteness at this stage [16]. Diagrams showing use and misuse cases side by side may be helpful for the analysts' creativity. For instance, just seeing the "Flood system" misuse case together with lots of normal use cases, the analysts are triggered to think about various ways that the system could potentially be flooded. Could it happen by repeated registering of customer info (like noted in Figure 4)? Could it happen by excess catalog browsing? Could it happen by repeated ordering of existing or non-existing goods? The explicit modeling of mis-actors in the diagram, will make it easier for the analysts to see the system from the perspective of potential crooks (let's only hope that it does not also inspire them to act like crooks, as happens to the murder investigator in the movie [17]).
- Traceability. The diagrams show in a straightforward manner the motivation for the secondary use cases introduced to deal with security threat, e.g. that a system monitoring function is required to enable the system to detect attempts at repeated log-ins and flooding. Whatever techniques are chosen for requirements analysis, the problems of traceability must somehow be addressed [11,18]. It would thus be nice if use cases could be extended to give an overview not only of the functions wanted from the system, but also of the security threats to be avoided.

An additional advantage, of course, is that the application of use case diagrams also for security-intensive systems, makes it possible to build on all the ongoing work providing easy transitions between this and object-oriented design, while other techniques for security requirements may have more problems in that respect.

As for related work, our graphical extensions to use case diagrams are believed to be novel. However, from a more abstract conceptual or methodological viewpoint, we have by no means revolutionized requirements engineering. The representation of misuse cases which then lead to the representation of secondary use cases to deal with these, are quite similar to previous approaches addressing the same kind of traceability, for instance [19], which couples non-functional requirements with more abstract concerns. A misuse case like, say, "Flood system" would here correspond to a concern that the system be flooded. Similar parallels could be drawn to any approach coupling detailed requirements to higher level goals, such as GBRAM (Goal-Based Requirements Analysis Method) [20,21,22,23]. Compared to these more elaborated approaches, ours is of course much more limited – while they look at goal structures in general, we limit ourselves to those resulting from the coupling between misuse cases and the subsequent use cases introduced to deal with these. On the other hand, our approach has the advantage of being visually intuitive, achieved through some fairly small additions to UML use case diagrams. Thus, our approach can be easily adopted in projects where use case modeling is applied anyway.

6. Conclusions and further work

Standard use case diagrams are often good for eliciting functional requirements, but not so good for security requirements, which often relate to activities which should *not* be possible in the system (and thus not modeled as use cases). In this paper we have suggested to extend the diagrams with some additional concepts, most notably misuse cases and mis-actors to capture security threats to the system.

The serious weakness of this work is that it has not been evaluated in practical software development projects, only on what must be called toy examples. The next step will thus be to try it out in a realistic setting. We feel quite confident that some industry partners may be interested in this, because:

- The approach is close to standard UML use case diagrams, which are heavily used in industry anyway.
- Security requirements are increasingly important with the advent of e- and m-commerce, and the problem that use case diagrams deal poorly with these, *will* be an industrial problem.
- The suggested extensions to use case diagrams are small, and simple to implement in a tool prototype (which will be necessary, unless OMG should decide to include our humble proposal into the UML language...)

Indeed, we have recently been contacted by a Norwegian software development company generally interested in a research cooperation on process improvement in software engineering. The elicitation of security requirements by misuse cases is one topic that we will suggest to investigate in that context.

[1] I. Jacobson et al., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[2] J. Rumbaugh, "Getting Started: Using use cases to capture requirements", Journal of Object-Oriented Programming, September 1994, pp. 8-23.

[3] M. Arnold et al., "Survey on the Scenario Use in Twelve Selected Industrial Projects", technical report, RWTH Aachen, Informatik Berichte, Nr. 98-17, 1998.

[4] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenario Usage in System Development: A Report on Current Practice", IEEE Software, 15(2): 34-45, March/April 1998.

[5] A. I. Antón, J. H. Dempster, and D. F. Siege, "Deriving Goals from a Use Case Based Requirements Specification for an Electronic Commerce System", Proc. REFSQ'2000, 5-6 Jun 2000.

[6] J. Arlow, "Use Cases, UML Visual Modelling and the Trivialisation of Business Requirements", Requirements Engineering Journal, 3(2):150-152, 1998.

[7] S. Lilly, "Use Case Pitfalls: Top 10 Problems from Real Projects Using Use Cases", Proc. TOOLS-USA'99, pp.174-183, 1-5 Aug 1999.

[8] Dictionary of Computing, 4th ed., Oxford University Press, 1999.

[9] J. Sanders and E. Curran, Software Quality, A Framework for Success in Software Development and Support, Addison-Wesley, 1994.

[10] S. T. Ross, Unix System Security Tools, McGraw-Hill, 1999.

[11] G. Kotonya and I. Sommerville, Requirements engineering: Processes and Techniques, Wiley, 1997.

[12] Object Management Group, OMG Unified Modeling Language Specification, version 1.3, June 1999.

[13] K. Cox and K. Phalp, A Case Study Implementing the Unified Modeling Language Use-Case Notation Version 1.3, Proc. REFSQ'2000, 5-6 Jun 2000.

[14] M. Fowler and K. Scott, UML Distilled: Applying The Standard Object Modelling Language, Pearson Education, 1997.

[15] A. Hall, "Seven Myths of Formal Methods", IEEE Software, 7(5):11-19, September 1990.

[16] P.E. London and M.S. Feather, "Implementing specification freedoms", Readings in Artificial Intelligence and Software Engineering, pp. 285-305, Morgan Kaufmann, 1986.

[17] L. von Trier and N. Vørsel: "The element of crime" (screenplay, original title in Danish: Forbrydelsens element), directed by L. von Trier, 1984.

[18] A. M. Davis, Software Requirements: Objects, Functions, and States, Prentice Hall, 1993.

[19] P. Loucopoulos and V. Karakostas, Systems Requirements Engineering, McGraw-Hill, 1995.

[20] A. I. Antón, "Goal-Based Requirements Analysis", Proc. ICRE'96, pp. 136-144, 1996.

[21] N. Maiden, S. Minocha, K. Manning, and M. Ryan, "CREWS-SAVRE: Systematic Scenario Generation and Use", Proc. ICRE'98, pp.148-155, 1998.

[22] C. Potts, "A ScenIC: A Strategy for Inquiry-Driven Requirements Determination", Proc. RE'99.

[23] C. Rolland, C. Souveyet, and C. Ben Achour, "Guiding Goal Models Using Scenarios", IEEE Transactions on Software Engineering, 24(12): 1055-1071, Dec 1998.