A UML Profile for Requirements Analysis of Dependable Software

Denis Hatebur[1,2] and Maritta Heisel[1]

[1] Universität Duisburg-Essen, Germany, Fakultät für Ingenieurwissenschaften, email: maritta.heisel@uni-due.de
[2] Institut für technische Systeme GmbH, Germany, email: d.hatebur@itesys.de

Abstract. At Safecomp 2009, we presented a foundation for requirements analysis of dependable software. We defined a set of patterns for expressing and analyzing dependability requirements, such as confidentiality, integrity, availability, and reliability. The patterns take into account random faults as well as certain attacks and therefore support a combined safety and security engineering.
In this paper, we demonstrate how the application of our patterns can be tool supported. We present a UML profile allowing us to express the different dependability requirements using UML diagrams. Integrity conditions are expressed using OCL. We provide tool support based on the Eclipse development environment, extended with an EMF-based UML tool, e.g., Papyrus UML. We illustrate how to use the profile to model dependability requirements of a cooperative adaptive cruise control system.

1 Introduction

Dependable systems play an increasingly important role in daily life. More and more tasks are supported or performed by computer systems. These systems are required to be safe, secure, available, and reliable. For such systems, it is of utmost importance to thoroughly analyze, understand, and consolidate the requirements.

In an earlier paper [8], we have presented a foundation for requirements analysis of dependable systems, based on problem frames [12]. In this paper, we show how the approach of [8] can be tool supported. To this end, we have defined a Unified Modeling Language (UML) profile [17] that allows us to represent problem frames in UML. This UML profile is then augmented with stereotypes that support the expression of dependability requirements. The stereotypes are complemented by constraints expressed in the Object Constraint Language (OCL) [15] that can be checked by existing UML tools. These constraints express important integrity conditions, for example, that security requirements must explicitly address a potential attacker. By checking the different OCL constraints, we can substantially aid system and software engineers in analyzing dependability requirements.

We work with the following definitions of dependability attributes [8]: *Safety* is the *in*ability of the system to have an undesirable effect on its environment, and *security* is the *in*ability of environment to have an undesirable effect on the system. To achieve safety, systematic and random faults must be handled. For security, in contrast, certain attackers must be considered. Security can be described by confidentiality, integrity and availability requirements. Also for safety, integrity and availability must be considered. For safety, integrity and availability mechanisms have to protect against random (and some systematic) faults. *Reliability* is a measure of continuous service accomplishment. It describes the probability of correct functionality under stipulated environmental conditions.

Dependability requirements must be described and analyzed. Problem frames [12] are a means to describe and analyze functional requirements, but they can be extended to describe also dependability requirements and domain knowledge, as also shown in earlier papers [8,10]. In Section 2, we present problem frames and the parts of the problem frames profile that extends the UML meta-model [17]. We describe the parts of our profile relevant to model dependability features. In Section 3, we show how we build tool support for the problem frame approach and for describing and analyzing dependability requirements. Section 4 contains our profile extension to describe dependability, and it also describes the OCL constraints for applying the elements introduced to describe dependability. Section 5 describes the process to work with our UML profile for problem frames for dependable systems. The case study in Section 6 applies that process to a cooperative adaptive cruise control system. Section 7 discusses related work, and the paper closes with a summary and perspectives in Section 8.

2 UML Profile for Problem Frames

Problem frames are a means to describe software development problems. They were introduced by Jackson [12], who describes them as follows: *"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."*

We describe problem frames using class diagrams extended by stereotypes (see Fig. 1). All elements of a problem frame diagram act as placeholders, which must be instantiated to represent concrete problems. Doing so, one obtains a problem description that belongs to a specific problem class.

The class with the stereotype <<*machine*>> represents the thing to be developed (e.g., the software). The other classes with some domain stereotypes (e.g., <<*CausalDomain*>> or <<*BiddableDomain*>>) represent *problem domains* that already exist in the application environment.

In frame diagrams, *interfaces* connect domains, and they contain *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation *O!E4* means that the phenomena in the set *E4* are controlled by the domain *Operator*. These interfaces are represented as associations, and the name of the associations contain the phenomena and the domains controlling the phenomena.

The associations can be replaced by interface classes in which the operations correspond to phenomena. The interface classes are either controlled or observed by the connected domains, represented by dependencies with the stereotypes <<*controls*>> or <<*observes*>>. Each interface can be controlled by at most one domain. A controlled interface must be observed by at least one domain, and an observed interface must be controlled by exactly one domain

Problem frames substantially support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described

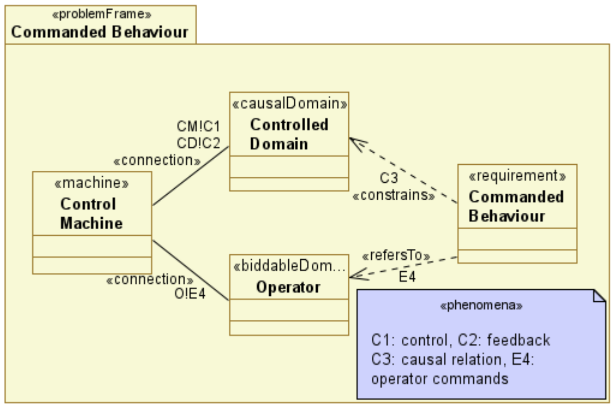

Fig. 1. *Commanded Behaviour* problem frame using UML notation

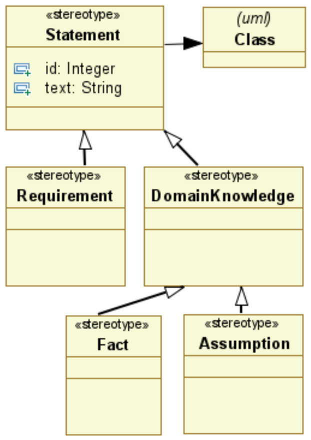

Fig. 2. Requirement stereotype inheritance structure

and reasoned about when analyzing the problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge*. The domain knowledge consists of *assumptions* and *facts*. Assumptions are conditions that are needed, so that the *requirements* are accomplishable. Usually, they describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment regardless of how the machine is built.

Domain knowledge and requirements are special statements. A statement is modeled similarly to a Systems Modeling Language (SysML) requirement [16] as a class with a stereotype. In this stereotype a unique identifier and the statement text are contained as stereotype attributes. Fig. 2 shows that the stereotype <<*Statement*>> extends the metaclass Class of the UML metamodel.

When we state a requirement, we want to change something in the world with the machine to be developed. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype <<*constrains*>>. Such a constrained domain is the core of any problem description, because it has to be controlled according to the requirements. Hence, a constrained domain triggers the need for developing a new software (the machine), which provides the desired control.

A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype <<*refersTo*>>. The referred domains are also given in the requirements description.

In Fig. 1, the Controlled Domain domain is constrained, because the Control Machine has the role to change it on behalf of user commands for achieving the required Commanded Behaviour.

Jackson distinguishes the domain types *biddable domains* that are usually people, *causal domains* that comply with some physical laws, and *lexical domains* that are data representations. The domain types are modeled by the stereotypes <<*BiddableDomain*>> and <<*CausalDomain*>> being subclasses of the stereotype <<*Domain*>>. A lexical domain (<<*LexicalDomain*>>) is modeled as a special case of a causal domain. This kind of modeling allows to add further domain types, such as <<*DisplayDomain*>> being also a special case of a causal domain. In Figure 1, the stereotypes <<*CausalDomain*>> and <<*BiddableDomain*>> indicate the domain types. To describe the problem context, a *connection domain* between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. They are modeled as classes with the stereotype <<*ConnectionDomain*>>. Connection domains are, e.g., video cameras, sensors, or networks.

Other problem frames besides the commanded behavior frame are *required behaviour*, *simple workpieces*, *information display*, and *transformation*. [12]

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements (see Fig. 6 for an example). Then, the problem is decomposed into subproblems. If possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, phenomena, and interfaces. The instantiated frame diagram is called a *problem diagram*.

Successfully fitting a problem to a given problem frame means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. A problem can only be fitted to a problem frame if the involved problem domains belong to the domain types specified in the frame diagram. For example, the Operator domain of Fig. 1 can only be instantiated by persons, but not for example by some physical equipment like an elevator.

Since the requirements refer to the *environment* in which the machine must operate, the next step consists in deriving a *specification* for the machine. The specification describes the machine and is the starting point for its construction.

The different diagram types make use of the same basic notational elements. As a result, it is necessary to explicitly state the type of diagram by appropriate stereotypes. In our case, the stereotypes are <<*ContextDiagram*>>, <<*ProblemDiagram*>>, and <<*ProblemFrame*>>. These stereotypes extend (some of them indirectly) the metaclass Package in the UML meta-model.

3 Tool Support

We have developed a tool called UML4PF to support the requirements engineering process sketched in Section 2. Our tool is integrated into the Eclipse development environment [3] as a plug-in. After the developer has drawn some diagram(s) using some EMF-based editor, for example Papyrus UML [5], UML4PF provides him or her with the following functionality:

– It checks if the developed model is valid and consistent by using our OCL constraints.
– It returns the location of invalid parts of the model.
– It automatically generates model elements, e.g., it generates observed and controlled interfaces from association names.

We defined a set of stereotypes in a profile that extends the UML meta-model. The most important stereotypes are presented in Section 2. This UML profile can be extended independently from our tool. Our tool is based on the Eclipse Modeling Framework (EMF [4]) and should be inter-operable with other EMF-based UML development tools being extendable using UML-profiles [17].

Our plugin UML4PF checks (using the Eclipse Modeling Framework) that the stereotypes are used correctly according to integrity conditions, e.g., that each statement constrains at least one domain. The tool is an open source tool under development and is free for download from `http://swe.uni-due.de/en/research/`.

4 Dependability Extension

We developed a set of patterns for expressing and analyzing dependability features (requirements and domain knowledge). Our patterns consist of UML classes with stereotypes and a set of rules describing possible relations to other model elements. The stereotype contains specific properties of the dependability feature (e.g. the probability to be achieved), a unique identifier, and a textual description that can be derived from the properties and the relations to other model elements. The patterns can be directly translated into logical predicates [8]. These predicates are helpful to analyze conflicting requirements and the interaction of different dependability requirements, as well as to find missing dependability requirements.

An important advantage of our patterns is that they allow dependability requirements to be expressed without anticipating solutions. For example, we may require data to be kept confidential during transmission without being obliged to mention encryption, which is a means to achieve confidentiality. The benefit of considering dependability requirements without reference to potential solutions is the clear separation of problems from their solutions, which leads to a better understanding of the problems and enhances the re-usability of the problem descriptions, since they are completely independent of solution technologies.

The dependability features can be described independently from the functional description. This approach limits the number of patterns, and allows one to apply these patterns to a wide range of problems. For example, the functional requirements for data transmission or automated control can be expressed using a problem diagram. Dependability requirements for confidentiality, integrity, availability and reliability can be added to that description of the functional requirement.

```
Class.allInstances()->select(
(getAppliedStereotypes().name->includes('Confidentiality') or
getAppliedStereotypes().name->includes('Integrity') or
getAppliedStereotypes().name->includes('Availability') or
getAppliedStereotypes().name->includes('Reliability') )
and getAppliedStereotypes().name->includes('Requirement'))
->forAll(clientDependency->select(d |
    d.oclAsType(Dependency).getAppliedStereotypes().name ->
        includes('supplements'))
    .oclAsType(Dependency).target.getAppliedStereotypes().name
        -> includes('Requirement')->count(true)>=1 )
```

Listing 1.1. Each Dependability Statement Supplements a Requirement

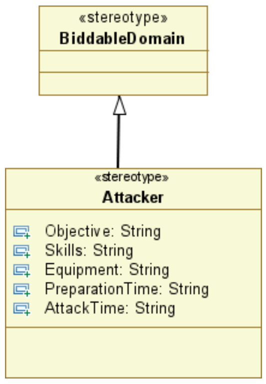

Fig. 3. Attacker in UML Problem Frames Profile

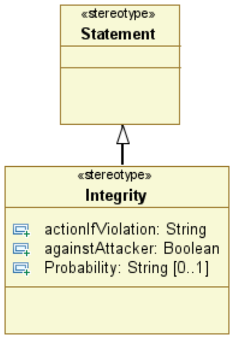

Fig. 4. Integrity Statement in UML Problem Frames Profile

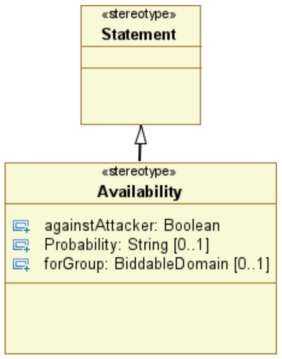

Fig. 5. Availability Statement in UML Problem Frames Profile

A dependability requirement always supplements (stereotype <<*supplements*>>) a functional requirement. This can be validated with the OCL expression in Listing 1.1. In this OCL expression, all classes with a stereotype indicating a dependability statement (e.g., <<*Integrity*>> or <<*Availability*>>) and additionally the stereotype <<*Requirement*>> are selected in Lines 1-6. In all of these requirement classes, it is checked that their dependencies (Line 7) with the stereotype <<*supplements*>> (Line 8) point to at least one class with the stereotype <<*Requirement*>> (Line 9).

Our patterns help to structure and classify dependability requirements. For example, requirements considering integrity can be easily distinguished from the availability requirements. It is also possible to trace all dependability requirements that refer to a given domain.

The patterns for integrity, reliability, and availability considering random faults are expressed using probabilities, while for the security requirements no probabilities are defined. We are aware of the fact that no security mechanism provides a 100 % protection and that an attacker can break the mechanism to gain data with a certain probability. But in contrast to the random faults considered for the other requirements, no probability distribution can be assumed, because, e.g., new technologies may dramatically increase the probability that an attacker is successful. For this reason we suggest to describe a possible attacker and ensure that this attacker is not able to be successful in a reasonable amount of time.

In the following, we present a selection of our dependability analysis patterns. More patterns and details are given in our technical report [9].

4.1 Confidentiality

A typical confidentiality statement is to

> Preserve confidentiality of *StoredData / TransmittedData* for *Stakeholder*s and prevent disclosure by a certain *Attacker*.

A statement about confidentiality is modeled as a class with the stereotype <<*Confidentiality*>> in our profile. This stereotype is a specialization of the stereotype <<*Statement*>>, described in Section 2. Three dependencies must be specified for a confidentiality requirement:

1. A causal domain representing the *StoredData* or *TransmittedData* must be constrained (using <<*constrains*>>). Even if data is usually modeled using lexical domains, we derive *StoredData* or *TransmittedData* from *CausalDomain*, because in some cases the storage device and not the data is modeled.
2. The statement needs to refer to the considered attacker. This attacker must be described in detail. We suggest to describe at least the attacker's objective, its skills, equipment, knowledge, and the time the attacker has to prepare and to perform the attack. A similar kind of description is suggested in the Common Methodology for Information Technology Security Evaluation (CEM) [1]. As shown in Fig. 3, the stereotype <<*Attacker*>> is a specialized <<*BiddableDomain*>>. The reference to an *Attacker* is necessary, because we can only ensure confidentiality with respect to an *Attacker* with given properties.
3. A confidentiality statement also needs to refer to the data's stakeholder. The *Stakeholder* is referred to, because we want to allow the access only to *Stakeholder*s with legitimate interest. The instances of *Stakeholder* and *Attacker* must be disjoint. The corresponding OCL expression requires that each confidentiality statement refers to at least one biddable domain that is not the attacker.

It is possible to generate the statement text from other model information: In the typical confidentiality statement the *StoredData / TransmittedData* can be obtained from the names of the domains constrained by this statement, the *Attacker* can be instantiated with the name of the domain referred to with the stereotype <<*Attacker*>>, and the *Stakeholder* can be instantiated with the name of the referred domain with the stereotype <<*BiddableDomain*>> (or a subtype different from <<*Attacker*>>). Additionally, the names of supplemented functional requirements can be added to the statement text if they exist.

A confidentiality requirement is often used together with functional requirements for data transmission and data storage.

4.2 Integrity

Typical integrity statements considering random faults are:

> With a probability of P_i, one of the following things should happen: service (as described in the functional statement) *with influence on / of* the *domain constrained in the functional requirement* must be correct, or *a specific action* must be performed.

Typical security integrity statements are:

> The *influence (as described in the functional statement) on / content of domain constrained in the functional statement* must be either correct, or in case of any modifications by some *Attacker a specific action* must be performed.

In contrast to the dependability statement considering random faults, this requirement can refer to the content of a domain (instead of the functionality), because security engineering usually focuses on data. For security the domain constrained in the functional requirement is usually a display or some plain data. The specific action could be, e.g.:

- write a *log* entry
- switch off the *actuator*
- do *not* influence the *domain constrained in the functional statement*

– perform the same action as defined in the functional statement on *domain constrained in the functional statement*. In this case we talk about reliability.
– inform *User*

Integrity statements are modeled as classes with the stereotype <<*Integrity*>>. In our profile, this stereotype is a specialization of the stereotype <<*Statement*>>, as shown in Fig. 4.

The domain mentioned in the specific action must be constrained by the integrity statement. The last specific action directly refers to the *User*. The *User* is a biddable domain and cannot be directly constrained. Therefore, the *User* must be informed by some technical means that can be constrained, e.g. a display. The assumption that the *User* sees the *Display* (being necessary to derive a specification from the requirements) must be checked later for validity.

An integrity requirement needs to refer to the domain constrained by the supplemented functional requirement. The class defining the stereotype <<*integrity*>> also has attributes. The attribute ActionIfViolation in Fig. 4 contains the textual description of the specific action as a string. The boolean attribute againstAttacker shows that the statement is a security statement, or if it is set to *false* that it is a statement considering random faults. In that case also the Probability must be specified. For all integrity statements (Lines 1-3 of Listing 1.2), it is checked if the inverted value of the stereotype attribute againstAttacker (Lines 4-6) implies that the value Probability is set, i.e not equal to *null* (Lines 8-10).

```
1  Class.allInstances()->select(oe |
2          oe.oclAsType(Class).getAppliedStereotypes().name ->
               includes('Integrity'))
3  ->forAll(c |
4          not c.oclAsType(Class).getValue(c.oclAsType(Class)
               .getAppliedStereotypes() -> select(s |
5              s.oclAsType(Stereotype).name ->
                   includes('Integrity') )
6              ->asSequence()->first(),'againstAttacker')
                   .oclAsType(Boolean)
7          implies
8              c.oclAsType(Class).getValue(c.oclAsType(Class)
                   .getAppliedStereotypes() -> select(s |
9              s.oclAsType(Stereotype).name ->
                   includes('Integrity') )
10             ->asSequence()->first(),'Probability') <> null )
```

Listing 1.2. Integrity Statements Contain Probabilities

The probability is a constant, determined by risk analysis. The standard ISO/IEC 61508 [11] provides a range of failure rates for each defined safety integrity level (SIL). The probability P_i could be, e.g., for SIL 3 systems operating on demand $1 - 10^{-3}$ to $1 - 10^{-4}$.

If the stereotype attribute againstAttacker is true, it is necessary that the statement refers to an attacker. The attacker must be described in the same way as for confidentiality in Section 4.1.

4.3 Availability

A typical availability statement considering random faults is:

The service (described in the functional statement) *with influence on / of* the *domain constrained in the functional statement* must be available (for *User*s) with a probability of P_a.

When we talk about availability in the context of security, it is not possible to provide the service to everyone due to limited resources. The availability statement considering an attacker is expressed as follows:

The service (described in the functional statement) *with influence on / of* the *domain constrained in the functional statement* must be available for *User*s even in case of an attack by *Attacker*s.

Availability statements are modeled as classes with the stereotype <<*Availability*>>. In our profile, this stereotype is a specialization of the stereotype <<*Statement*>>, shown in Fig. 5.

Availability requirements constrain the domains constrained by the supplemented functional requirement. The stereotype class for availability contains the attributes againstAttacker, Probability, and forGroup. If againstAttacker is *false*, the stereotype attribute Probability must be specified. This can be checked in the same way as for integrity, described in Section 4.2.

If againstAttacker is *true*, the stereotype attribute forGroup must be specified, and an attacker must be referred to. Both conditions can be expressed similarly as described in Listing 1.2.

4.4 Reliability

Reliability is defined in a similar way as availability (see Section 4.3). The same failure rates as for integrity (see Section 4.2) can be used.

5 Procedure to Use the Dependability Extension

This section describes how to work with the UML profile for problem frames for dependable systems. To use our profile and apply the dependability patterns, we assume that *hazards and threats are identified, and a risk analysis* has been performed. The next step is to *describe the environment*, because dependability requirements can only be guaranteed for some specific intended environment. For example, a device may be dependable for personal use, but not for military use with more powerful attackers or a non-reliable power supply. The *functional requirements are described* for this intended environment using problem frames (see Section 2). The requirements describe how the environment should behave when the machine is in action. The requirements should be expressed in terms of domains and phenomena of the context diagram. From hazards and threats an *initial set of dependability requirements can be identified*. These requirements supplement the previously described functional requirements.

For each dependability requirement, a pattern from Section 4 should be selected. After an appropriate pattern is determined, is must be connected with the concrete domains from the environment description. The connected domains must be described. For an attacker, at least the attributes of the stereotype must be defined (objective, equipment, skill, time to attack, time to prepare). Via these assumptions, *threat models* are integrated into the development process using dependability patterns. The values for probabilities can be usually extracted from the risk analysis.

Our paper [8] describes how to *find missing, interacting, and related requirements or domain knowledge* by selecting generic mechanisms. New requirements and new domain knowledge is described using the same notation as used for the initial requirements and analyzed in the same way.

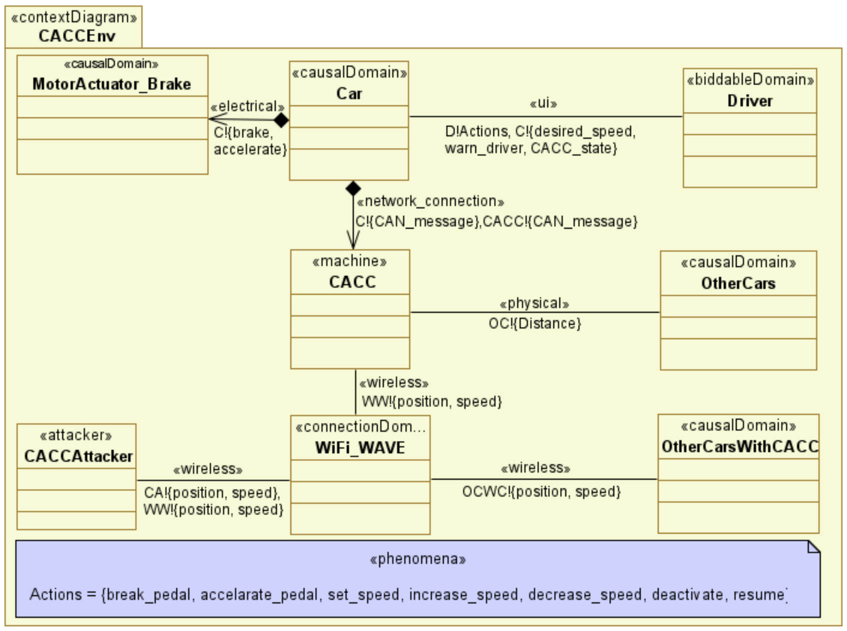

Fig. 6. CACC Context Diagram

6 Case Study

The approach is illustrated by the same case study as presented in [8]; the development of a cooperative adaptive cruise control (CACC) maintaining string stability. Such a system controls the speed of a car according to the desired speed given by the driver and the measured distance to the car ahead. It also considers information about speed and acceleration of the car ahead which is sent using a wireless network. The *hazard* to be avoided is an unintended acceleration or deceleration (that may lead to a rear-end collision). The considered *threat* is an attacker who sends wrong messages to the car in order to influence its speed.[1] Examples for domain knowledge of the CACC in the *described environment* are physical properties about acceleration, braking, and measurement of the distance (relevant for safety). Other examples are the assumed intention, knowledge and equipment of an attacker. We assume here that the attacker can only access the connection domain *WiFi_WAVE interface*. The context diagram for the CACC is shown in Fig. 6. It also contains the type of connection as stereotypes at the associations between domains (e.g. <<*wireless*>> for wireless connections). These connection types are not considered in this paper.

The *functional requirement* for the CACC is to maintain string stability:

R1 The CACC should accelerate the car if the desired speed set by the driver is higher than the current speed, the CACC is activated and the measured distance and the calculated distance to the car(s) ahead are safe.

[1]The *risk analysis* is left out here.

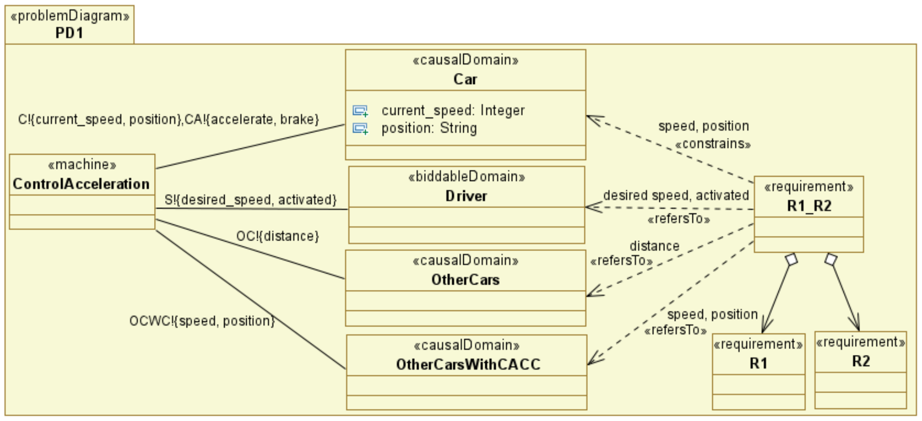

Fig. 7. CACC Problem Diagram for Control Acceleration and Brake

R2 The CACC should brake the car if the desired speed set by the driver is much (30 km/h) lower as the current speed, the CACC is activated and the measured or calculated distance to the car(s) ahead is decreasing towards the safe limit.

As an example, the problem diagram for R1 and R2 is depicted in Fig. 7. The problem diagram describes the interfaces between the machine and the environment necessary to implement requirements R1 and R2, e.g., it describes that the machine (a submachine of the CACC in the context diagram) can accelerate the car (CA!{accelerate}), and it describes the relation of the requirements R1 and R2 to the domains in the environment. The requirements constrain the current speed of the car and therefore indirectly its position. The requirements refer to the information in the domains necessary for the described decision, e.g., the desired speed and the distance to the car ahead.

The next step is to *identify an initial set of dependability requirements*. For the functional requirements R1 and R2, the following security requirement can be stated using the textual pattern from Section 4.2:

> The influence (as described in R1 and R2) on the car (brake, accelerate) must be either correct, or in case of any modifications by CACCAttacker the car (MotorActuator_Brake) should <u>not</u> be influenced (no brake, no accelerate).

A problem diagram including this integrity requirement is depicted in Fig. 8. It supplements the requirements R1_R2. It refers to an attacker (the CACCAttacker) and also refers to the domain constrained by R1_R2 (the Car). The Car is constrained because the MotorActuator_Brake as part of the car should <u>not</u> be influenced.

All OCL constraints defined for the profile were checked. With checking these constraints, we detected several minor mistakes (e.g., wrong names), and we detected that the original version of our problem diagram did not refer to the domain constrained in the requirement.

We also defined the problem diagrams and the predicates for the other initial dependability requirements (integrity considering random faults, availability, and reliability). Details can be found in our technical report [9].

To *find missing, interacting, and related requirements or domain knowledge*, we used the table with dependability predicates presented in [8]. This analysis resulted in a

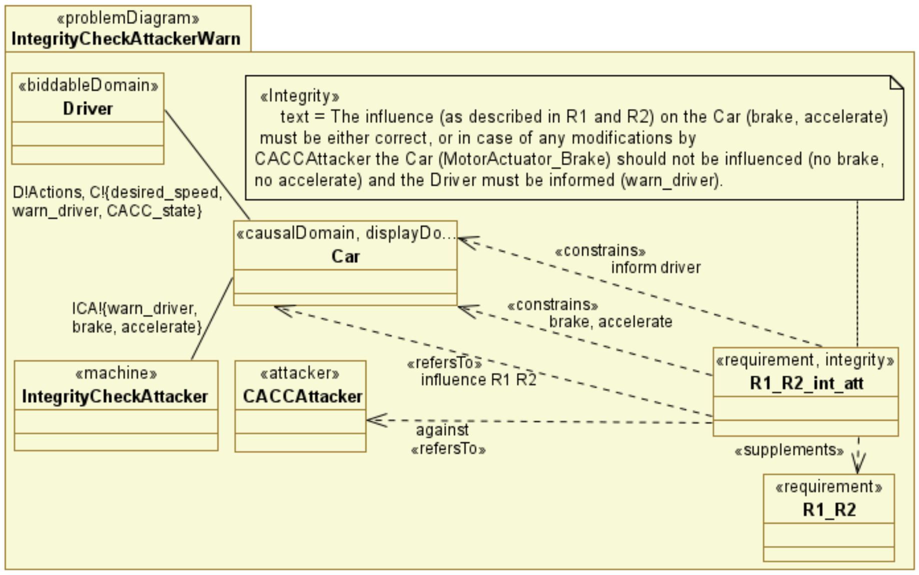
«problemDiagram»
IntegrityCheckAttackerWarn
«biddableDomain»
Driver
«Integrity»
text = The influence (as described in R1 and R2) on the Car (brake, accelerate) must be either correct, or in case of any modifications by CACCAttacker the Car (MotorActuator_Brake) should not be influenced (no brake, no accelerate) and the Driver must be informed (warn_driver).
D!Actions, C!{desired_speed, warn_driver, CACC_state}
«causalDomain, displayDo...
Car
«constrains»
inform driver
«constrains»
brake, accelerate
ICA!{warn_driver, brake, accelerate}
«machine»
IntegrityCheckAttacker
«attacker»
CACCAttacker
«refersTo»
influence R1 R2
«requirement, integrity»
R1_R2_int_att
against
«refersTo»
«supplements»
«requirement»
R1_R2

Fig. 8. CACC Problem Diagram for Integrity Checks considering an Attacker

set of additional requirements and additional domain knowledge. For example, to preserve integrity considering the described attacker, we need a protection of the messages sent using a wireless interface. To protect the messages, we chose Message Authentication Codes (MAC). For creating and validating MACs, session secrets are necessary. These secrets and the processing data in the machine itself must be kept confidential. The pattern also requires to refer to the stakeholder (here: the Manufacturer) and the attacker. The statement about protection of the secrets should not be realized by the software to be built and is therefore considered to be domain knowledge.

The complete case study consists of 40 classes, 50 associations, and about 150 dependencies. A total of 65 OCL constraints were checked using our tool, 15 of which concerned dependability. The OCL constraints showed for previous versions of the CACC case study, e.g., that our integrity requirement did not refer to an attacker. As a final result, the CACC model has been successfully validated.

7 Related Work

This paper extends the patterns for requirements (and domain knowledge) presented in [8] by a formal metamodel to provide tool support.

The Common Criteria [2], Part 2 define a large set of so-called *Security Functional Requirements (SFRs)* as patterns for requirements. But some of these SFRs directly anticipate a solution, e.g. the SFR ***c****ryptographic* ***op****eration* in the class ***f****unctional requirements for* ***c****ryptographic support* (FCS_COP) specifies the cryptographic algorithm, key sizes, and the assigned standard to be used. The SFRs in the Common Criteria are limited to security issues. In a technical report [9], we relate some of the CC SFRs to our patterns for dependability statements.

Lencastre et al. [13] define a metamodel for problem frames using UML. In contrast to our metamodel, it only consists of a UML class model. Hence, the OCL integrity conditions of our metamodel are not considered in their metamodel.

Hall et al. [7] provide a formal semantics for the problem frame approach. Their model focuses on a formal specification language to describe problem frames and problem diagrams.

Seater et al. [14] present a metamodel for problem frame instances. They formalize requirements and specifications. Their integrity conditions focus on correctly deriving specifications from requirements.

Charfi et al. [6] use a modeling framework called *Gaspard2* to design high-performance embedded systems-on-chip. They use model transformations to move from one level of abstraction to the next. To validate that their transformations have been correctly performed, they use the OCL language to specify the properties that must be checked in order to be considered as correct with respect to Gaspard2. We have been inspired by this approach. However, we do not focus on high-performance embedded systems-on-chip. Instead, we target dependable systems development.

SysML [16] also provides the stereotype <<*Requirement*>> for classes. It can be used to express dependabilites between requirements and the relation to realization and tests (e.g., with the stereotypes <<*refine*>>, <<*trace*>>, <<*satisfy*>>). We relate the requirements to domains of the environment to make their pupose explicit and provide support for requirements interaction analysis.

8 Conclusions and Future Work

In this paper, we have presented an extension to our UML profile for problem frames to describe dependability. In this profile, we defined a set of stereotypes for dependability requirements and domain knowledge. We set up 65 OCL constraints for requirements engineering, 15 of which concern dependability. These constraints show how functional requirements can be supplemented by dependability requirements.

In summary, our concept has the following advantages:

- Artifacts from the analysis development phase that are part of a model created with our profile can be re-used in later phases in the software development process.
- The notation is based on UML. UML is commonly used in software engineering, and many developers are able to read our models.
- The concept is not tool-specific. It can be easily adapted to other UML2 tools that allow to specify new stereotypes.
- The dependability statements are re-usable for different projects.
- A manageable number of statement types can be used for a wide range of problems, because they are separated from the functional requirements.
- Statements expressed using our profile refer to the environment description and are independent from solutions. Hence, they can be easily re-used for new product versions.
- A generic textual description of the requirement or the domain knowledge can be generated form other model elements.
- Statements expressed using our profile help to structure and classify the dependability requirements. For example, integrity statements can be easily distinguished from availability statements. It is also possible to trace all dependability statements that refer to one domain.

In the future, we plan to extend our tool to support the identification of missing and interacting requirements. We also want to support traceability links to trace our (dependability) requirements to artifacts developed later, e.g. components in the software architecture.

Acknowledgment. We thank Isabelle Côté for her valuable comments on this work.

References

1. Common Methodology for Information Technology Security Evaluation, August 2005. http://www.commoncriteriaportal.org/public/expert/.
2. Common Criteria for Information Technology Security Evaluation, Version 3.1, September 2006. http://www.commoncriteriaportal.org/public/expert/.
3. Eclipse - An Open Development Platform, May 2008. http://www.eclipse.org/.
4. Eclipse Modeling Framework Project (EMF), May 2008. http://www.eclipse.org/modeling/emf/.
5. Papyrus UML Modelling Tool, Jan 2010. http://www.papyusuml.org/.
6. A. Charfi, A. Gamatié, A. Honoré, J.-L. Dekeyser, and M. Abid. Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. In *4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 08)*, 2008.
7. J. G. Hall, L. Rapanotti, and M. Jackson. Problem frame semantics for software development. *Software and System Modeling*, 4(2):189–198, 2005.
8. D. Hatebur and M. Heisel. A foundation for requirements analysis of dependable software. In B. Buth, G. Rabe, and T. Seyfarth, editors, *Proc. of the Int. Conference on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS 5775, pages 311–325. Springer, 2009.
9. D. Hatebur and M. Heisel. A UML profile for requirements analysis of dependable software (technical report). Technical report, Universität Duisburg-Essen, 2010. http://swe.uni-due.de/techrep/depprofile.pdf.
10. D. Hatebur, M. Heisel, and H. Schmidt. A pattern system for security requirements engineering. In B. Werner, editor, *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*, IEEE Transactions, pages 356–365. IEEE, 2007.
11. International Electrotechnical Commission IEC. Functional safety of electrical/electronic/programmable electronic safty-relevant systems, 2000.
12. M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
13. M. Lencastre, J. Botelho, P. Clericuzzi, and J. Araújo. A meta-model for the problem frames approach. In *WiSME'05: 4th Workshop in Software Modeling Engineering*, 2005.
14. R. Seater, D. Jackson, and R. Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, 12(2):77–102, 2007.
15. "UML Revision Task Force". *OMG Object Constraint Language: Reference*, May 2006. http://www.omg.org/docs/formal/06-05-01.pdf.
16. "UML Revision Task Force". *OMG Systems Modeling Language (OMG SysML)*, November 2008. http://www.omg.org/spec/SysML/1.1/.
17. "UML Revision Task Force". *OMG Unified Modeling Language: Superstructure*, February 2009. http://www.omg.org/docs/formal/09-02-02.pdf.