A Foundation 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. We present patterns for expressing dependability requirements, such as confidentiality, integrity, availability, and reliability. The paper considers random faults as well as certain attacks and therefore supports a combined safety and security engineering. The patterns - attached to functional requirements - are part of a pattern system that can be used to identify missing requirements. The approach is illustrated on 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, reliable, and maintainable.

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 [16]. 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. **Confidentiality** is the absence of unauthorized disclosure of information. **Integrity** is the absence of improper system, data, or a service alterations [15]. **Availability** is the readiness for service (up-time vs. down-time) [14][1]. 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 [14]. A safety-critical system has to perform its safety-functions with a defined reliability (or integrity) [2]. In this case, reliability describes the probability of correct functionality under stipulated environmental conditions [4]. This paper shows that reliability requirements can be defined not only from a safety point of view, but also from a security point of view. **Maintainability** is the ability to undergo modifications and repairs [2]. Maintainability can be achieved by additional interfaces for updates (of the whole software or components), by a maintainable structure of the software itself (e.g., documentation, appropriate architectures, comments in the source code), and by maintenance plans (e.g., restart the software once a week to reduce memory fragmentation). Maintainability is not considered in this paper.

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 features, as shown in earlier papers [7, 8]. In Section 2,

[1] Availability, in contrast to reliability, does not require correct service.

[2] If the system can for example be safely deactivated, it is sufficient to define the integrity requirement and the actions to be performed in case of an integrity error.

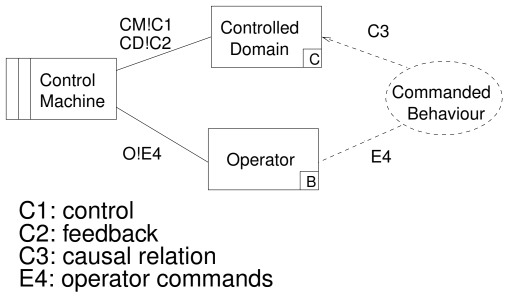

Fig. 1. *Commanded Behaviour* problem frame

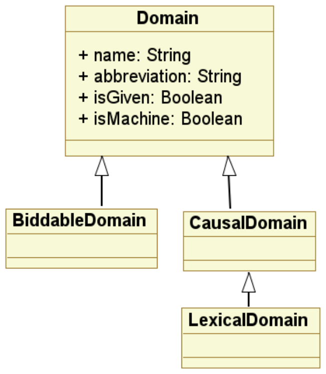

Fig. 2. Inheritance structure of different domain types

we present problem frames and the parts of the problem frames meta-model [10] used for the formalization of dependability features. In Section 3, we define a set of patterns that can be used to describe and analyze dependability requirements. Section 4 describes how to integrate the use of the dependability patterns into a system development process. The case study in Section 5 applies that process to a cooperative adaptive cruise control system. Section 6 discusses related work, and the paper closes with a summary and perspectives in Section 7.

2 Problem Frames

Problem frames are a means to describe software development problems. They were invented 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."* Problem frames are described by *frame diagrams*, which consist of rectangles, a dashed oval, and links between these (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.

Plain rectangles denote *problem domains* (that already exist in the application environment), a rectangle with a double vertical stripe denotes the *machine* (i.e., the software) that shall be developed, and *requirements* are denoted with a dashed oval. The connecting lines between domains represent interfaces that consist of *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***.

A dashed line represents a requirements reference. It means that the domain is referred to in the requirements description. An arrow at the end of such a dashed line indicates that the requirements constrain the problem domain. 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. In Fig. 1, the ***Controlled-Domain*** domain is constrained, because the ***ControlMachine*** has the role to change it on behalf of user commands for achieving the required ***Commanded Behaviour***.

Jackson distinguishes the domain types ***CausalDomain***s that comply with some physical laws, ***LexicalDomain***s that are data representations, and ***BiddableDomain***s that are usually people. In Fig. 1, the C indicates that the corresponding domain is a ***CausalDomain***, and B indicates that it is a ***BiddableDomain***. In our formal meta-model of problem frames [10] (see Fig. 2), ***Domains*** have ***names*** and ***abbreviations***, which are used to define interfaces. According to Jackson, domains are either ***designed***, ***given***, or ***machine*** domains. These facts are modeled by the Boolean attributes ***isGiven*** and ***isMachine*** in Fig. 2. The domain types are modeled by the subclasses ***BiddableDomain***, ***CausalDomain***, and ***LexicalDomain*** of the class ***Domain***. A lexical domain is a special case of a causal domain. This kind of modeling allows to add further domain types, such as ***DisplayDomain***s as introduced in [3].

Problem frames support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing the problem in depth. Other problem frames besides the commanded behavior frame are *required behaviour*, *simple workpieces*, *information display*, and *transformation*.

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, and it is not shown which domain is in control of the shared phenomena (see Fig. 3 for an example). Then, the problem is decomposed into subproblems. If ever 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.

To describe the problem context, a ***ConnectionDomain*** between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. Typical connection domains are ***CausalDomain***s, e.g., video cameras, sensors, or networks.

Since the requirements refer to the *environment* in which the machine must operate, the next step consists in deriving a *specification* for the machine (see [13] for details). The specification describes the machine and is the starting point for its construction.

3 Patterns for Dependability Requirements

We developed a set of patterns for expressing and analyzing dependability requirements. An important advantage of these 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.

Our dependability requirements patterns are expressed as logical predicates. They are separated from functional requirements. On the one hand, this limits the number of patterns; on the other hand, it 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.

For each dependability requirement, a textual description pattern and a corresponding predicate pattern are given. The textual description helps to state dependability requirements more precisely. The 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 logical predicate patterns have several parameters. The first parameter of a predicate is the domain that is constrained by the requirement, whereas the other parameters are only referred to. The predicate patterns are expressed using the domain types of the meta-model described in Figure 2, i.e., ***Domain***, ***BiddableDomain***, ***CausalDomain***, and ***LexicalDomain***. From these classes in the meta-model, subclasses with special properties are derived:

– An ***Attacker*** is a ***BiddableDomain*** that describes all subjects (with their equipment) who want to attack the machine.
– A ***User*** is a ***BiddableDomain*** that describes subjects who have an interface to the machine.
– A ***Stakeholder*** is a ***BiddableDomain*** (and in some special cases also a ***CausalDomain***) with some relation to stored or transmitted data. It is not necessary that a stakeholder has an interface to the machine.
– A ***ConstrainedDomain*** is a ***CausalDomain*** that is constrained by a functional or dependability requirement.
– An ***InfluencedDomain*** is a ***CausalDomain*** that is influenced by the machine to fulfill the dependability requirement (it can be the same domain as the ***ConstrainedDomain***, but also another domain).
– A ***Display*** is a ***CausalDomain*** used to inform the user of the machine.
– ***StoredData*** is a ***CausalDomain*** or ***LexicalDomain*** used to store some data as defined by the functional requirement. Also the machine domain may include some (transient) stored data that must be considered.
– ***TransmittedData*** is a ***CausalDomain*** or ***LexicalDomain*** used to transmit data (e.g., a network).
– A ***Secret*** is a ***StoredData*** or ***TransmittedData*** that is used to implement a set of security requirements.

To use the predicate patterns for describing the dependability requirements of a concrete problem, the domains of the problem diagram (and in the context diagram) must be derived from the domains given in the dependability patterns. They must be described in such a way that it is possible to demonstrate that the dependability predicate holds for all objects of this class. The parts of the pattern's textual description printed in ***bold and italics*** should be refined according to the concrete problem.

The instantiated predicates are helpful to analyze conflicting requirements and the interaction of different dependability requirements, as well as for finding missing dependability requirements.

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 [17]. 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.

3.1 Confidentiality

A typical confidentiality requirement is to

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

The security requirement pattern can be expressed by the confidentiality predicate $conf_{att} : CausalDomain \times BiddableDomain \times BiddableDomain \rightarrow Bool$. The suffix "att" indicates that this predicate describes a requirement considering a certain *att*acker.

To apply the confidentiality requirement pattern, subclasses of ***StoredData*** or ***TransmittedData***, ***Stakeholder***, and ***Attacker*** must be derived and described in detail. For example, a special ***TransmittedData*** may be the ***PIN of a bank account***, a special ***Stakeholder*** may be the ***bank account owner***, and a special ***Attacker*** may be the class of all ***persons with no permission, who want to withdraw money and have access to all external interfaces of the machine***. The instances of ***Stakeholder*** and ***Attacker*** must be disjoint. The ***Stakeholder*** is referred to, because we want to allow the access only to ***Stakeholder***s with legitimate interest [5]. The reference to an ***Attacker*** is necessary, because we can only ensure confidentiality with respect to an ***Attacker*** with given properties.

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. A ***LexicalDomain*** is a special ***CausalDomain***. The following patterns can be used to define confidentiality requirements:

$\forall\, sd : StoredData;\ s : Stakeholder;\ a : Attacker \bullet conf_{att}(sd, s, a)$
$\forall\, td : TransmittedData;\ s : Stakeholder;\ a : Attacker \bullet conf_{att}(td, s, a)$

They express the informal requirement given above as a logical formula. The confidentiality predicate is often used together with functional requirements for data transmission and data storage.

3.2 Integrity - Random Faults

Typical integrity requirements considering random faults are that

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

The specific action could be, e.g.:

- write a log entry into ***InfluencedDomain***
- switch off the actuator ***InfluencedDomain***
- do **not** influence ***ConstrainedDomain***
- perform the same action as defined in the functional requirement on ***ConstrainedDomain***.
- inform ***User***

For this requirement it is important to distinguish ***ConstrainedDomain*** and ***InfluencedDomain***. The ***ConstrainedDomain*** is the domain that should work correctly or should be influenced correctly as described in the functional requirement. The ***InfluencedDomain*** is the domain that should react as described in the dependability requirement. The ***InfluencedDomain*** could be, e.g., an actuator or a log file. The last specific action directly refers to the ***User***. The ***User*** must be informed by some technical means, 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.

The requirement can be expressed by the integrity predicate $int_{rnd} : CausalDomain \times Domain \times Probability \rightarrow Bool$. The suffix “rnd” indicates that this predicate describes a requirement considering random faults.

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 $1 - 10^{-4}$.

The following patterns can be used to define the integrity requirements for a given probability P_i:

$\forall\, cd : ConstrainedDomain;\ u : User \bullet int_{rnd}(cd, u, P_i)$
$\forall\, cd : ConstrainedDomain;\ id : InfluencedDomain \bullet int_{rnd}(cd, id, P_i)$
$\forall\, cd : ConstrainedDomain \bullet int_{rnd}(cd, cd, P_i)$

The predicate $int_{rnd}(cd, cd, P_i)$ expresses that the specific action is either that the ***ConstrainedDomain*** is **not** influenced any longer, or that it is influenced as described in the functional requirement (same action).

3.3 Integrity - Security

A typical security integrity requirement is that

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

The specific action may be the same as described for random faults in Section 3.2. In contrast to the dependability requirement considering random faults, this requirement can refer to the data of a domain (instead of the functionality), because security engineering usually focuses on data. For security the ***ConstrainedDomain*** in the functional requirement is usually a display or some plain data. The security requirement pattern can be expressed by the integrity predicate $int_{att} : CausalDomain \times Domain \times BiddableDomain \rightarrow Bool$.

Similarly to Section 3.2 the following patterns can be used to define integrity requirements:

$\forall\, cd : ConstrainedDomain;\ u : User;\ a : Attacker \bullet int_{att}(cd, u, a)$
$\forall\, cd : ConstrainedDomain;\ id : InfluencedDomain;\ a : Attacker \bullet int_{att}(cd, id, a)$
$\forall\, cd : ConstrainedDomain;\ a : Attacker \bullet int_{att}(cd, cd, a)$

3.4 Availability - Random Faults

A typical availability requirement considering random faults is that

> The service (described in the functional requirement) ***with influence on / of*** the ***ConstrainedDomain*** must be available for ***User*** with a probability of P_a.[3]

The requirement can be expressed by the availability predicate $avail_{rnd\_user}$: $CausalDomain \times BiddableDomain \times Probability \rightarrow Bool$.

P_a is the probability that the service (i.e., the influence on the ***ConstrainedDomain***) is accessible for defined users. A probability P_a of $1 - 10^{-5}$ means that the service (influence on the ***ConstrainedDomain***) may be unavailable on average for 315 seconds in one year. The following pattern can be used to define the availability requirements for a given probability P_a:

$\forall\, cd : ConstrainedDomain;\; u : User \bullet avail_{rnd\_user}(cd, u, P_a)$

3.5 Availability - Security

When we talk about availability in the context of security it is not possible to provide the service to everyone due to limited resources and possible denial-of-service attacks. Availability can be expressed with the predicate $avail_{att\_user}(cd, u, a)$ similar to the availability requirement considering random faults.

3.6 Reliability, Authentication, Management, and Secret Distribution

Reliability is defined in the same way as availability with the predicates rel_{rnd_user}, rel_{rnd}, and rel_{att_user}. The same failure rates as for integrity (see Section 3.2) can be used. Other important security requirements are authentication ($auth_{att}(cd, sh, a)$) to permit access for ***Stakeholder*** (*sh*) and deny access for ***Attacker*** (*a*) on ***ConstrainedDomain*** (*cd*), security management ($man_{att}(sd, sh, a)$) to manage security-relevant ***StoredData*** (*sd*) (e.g., configure an access rule), and ***Secret*** (*s*) distribution ($dist_{att}(s, sh, a)$) that additionally keeps the managed secret *s* confidential.

4 Working with Dependability Requirement Patterns

This section describes how to work with the modular construction system built up on the predicates defined in Section 3. It can be used to find possible interactions with other dependability requirements and helps to complete the dependability requirements by a set of defined necessary conditions for each mechanism that can be used to solve dependability problem. To apply the dependability patterns, we assume that **hazards and threats are identified and a risk analysis** is 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** using patterns for this intended environment (see Section 2). The requirements describe how the environment should behave when the machine is in action. To describe the requirements, domains and phenomena of the environment description should be used. From hazards and threats an **initial set of dependability requirements can be identified**. These requirements are usually linked to a previously described functional requirement.

[3] In [6], a variant that does not refer to users is presented ($avail_{rnd}$).

Requirement	**Generic mechanism**	**Possible interaction**	**Introduced / considered domains**	**Necessary conditions**	**Conditions to be established before**	**Related**
$int_{rnd}(c_1, u, P_i)$	checksums	$avail_*(c_1, *)$[4]	$di : Display$	$int_{rnd}(m, u, P_i) \wedge$ $int_{rnd}(di, u, P_i) \wedge$ user sees display message	-	int_{att}
$int_{rnd}(c_1, c_2, P_i)$	checksums	$avail_*(c_1, *)$[4]	-	$int_{rnd}(m, c_2, P_i) \wedge rel_{rnd}(c_2, P_i)$	-	int_{att}
$int_{att}(d, u, a)$	MAC	$avail_*(d, u, *)$[4]	$s_{Snd}, s_{Rcv} : Secret$ $di : Display$	$conf_{att}(m, u, a) \wedge int_{att}(m, u, a) \wedge$ $int_{att}(di, u, a) \wedge$ $conf_{att}(s_{Snd}, u, a) \wedge$ $int_{att}(s_{Snd}, u, a) \wedge$ $conf_{att}(s_{Rcv}, u, a) \wedge int_{att}(s_{Rcv}, u, a)$	$dist_{att}(s_{Snd}, u, a) \wedge$ $dist_{att}(s_{Rcv}, u, a)$	$conf_{att}$
	cryptographic signature	$avail_*(d, u, *)$[4]	$s_{Snd} : SenderSecret$ $s_{Rcv} : ReceiverSecret$ $di : Display$	$conf_{att}(m, u, a) \wedge int_{att}(m, u, a) \wedge$ $int_{att}(di, u, a) \wedge int_{att}(s_{Snd}, u, a) \wedge$ $conf_{att}(s_{Rcv}, u, a) \wedge int_{att}(s_{Rcv}, u, a)$	$dist_{att}(s_{Snd}, u, a) \wedge$ $dist_{att}(s_{Rcv}, u, a)$	$conf_{att}$
$avail_{rnd\_user}(c_1, u, P_a)$	reliable hardware and software		-	$rel_{rnd\_user}(m, u, P_r)$	-	rel_{att}
$rel_{rnd}(c_1, P_r)$	reliable hardware and software		-	$rel_{rnd}(m, P_r)$	-	rel_{att}
$auth_{att}(d, u, a)$	dynamic authentication using random numbers (symmetric)	$avail_*(d, *)$	$s_{Mchn}, s_{Ext} : Secret$	$conf_{att}(m, u, a) \wedge int_{att}(m, u, a) \wedge$ $conf_{att}(s_{Mchn}, u, a) \wedge$ $int_{att}(s_{Mchn}, u, a) \wedge$ $conf_{att}(s_{Ext}, u, a) \wedge int_{att}(s_{Ext}, u, a)$	$dist_{att}(s_{Mchn}, u, a) \wedge$ $dist_{att}(s_{Ext}, u, a)$	
$dist_{att}(d, u, a)$	see *dynamic authentication*	...	...	...	...	...

Table 1: Selected dependability dependencies

[4] Availability may be descreased if modified data is just deleted.

For each dependability requirement, a pattern from our pattern catalog should be selected, using the informal description of the dependability requirements given in Section 3. After an appropriate pattern is determined, is must be "instantiated" with the concrete domains from the environment description. To instantiate the domains that represent potential attackers, a certain level of skill, equipment, and determination of the potential attacker must be specified. 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. For each dependability requirement stated as a predicate, we **select a generic mechanism** that solves the problem; for example, to achieve integrity (int_{att_bidd}) message authentication codes (MACs) can be used. Table 1 lists for each dependability requirement pattern a set of possible mechanisms. The dependability requirement predicates in the table refer to all instances d of ***TransmittedData*** or ***StoredData***, the ***ConstrainedDomain***s c_1 and c_2, the ***user***s u, the ***Attacker***s a, and the ***Machine*** with all relevant connection domains m.

Table 1 supports the analysis of conflicts between the dependability patterns. For some of the mechanisms, **possible interactions** with other dependability requirements are given in the third column. These possible conflicts must be analyzed, and it must be determined if they are relevant for the application domain. In case they are relevant, conflicts can be resolved by modifying or prioritizing the requirements. For example, if the MAC protection mechanism is applied and the specific action is to delete modified data, we may have a contradiction with the availability of that data.

For many mechanisms, additional **domains must be introduced or considered**. MAC protection, e.g., requires a ***Secret*** s_{Snd} used to calculate the MAC and another ***Secret*** s_{Rcv} used to verify the MAC. For asymmetric mechanisms, ***SenderSecret*** and ***ReceiverSecret*** need to be introduced. They are special ***StoredData***. For dynamic authentication, the ***Secret*** s_{Mchn} (stored in the machine) and the ***Secret*** s_{Ext} (known by the subject) are necessary. Such introduced domains must be **added to the description of the environment**.

The next step is to **inspect the necessary conditions and the conditions to be established beforehand**. The generic mechanisms usually have a set of *necessary conditions* to be fulfilled. These necessary conditions describe conditions necessary to establish the dependability requirement when a certain mechanism is selected. For example, the introduced secrets for the MAC protection must be kept confidential, and their integrity must be preserved. Before the mechanism is applied, some other activities are necessary, e.g., a secret must be distributed before it can be used for MAC calculation (*conditions to be established beforehand*). Two alternatives are possible to guarantee that the necessary conditions hold: either, they can be *assumed* to hold, or they have to be *established* by instantiating a further dependability requirement pattern, that matches the necessary condition. What assumptions are reasonable depends on the hazards to be avoided and the threats the system should be protected against. Assumptions cannot be avoided completely, because otherwise it may be impossible to achieve a dependability requirement. For example, we must assume that the user sees a warning messages on a display or keeps a password confidential. Only in the case that necessary conditions *cannot* be assumed to hold, one must instantiate further appropriate dependability patterns, and the procedure is repeated until all necessary conditions of all applied mechanisms can be proved or assumed to hold. The dependencies expressed as necessary condition are used to develop a consolidated set of dependability require-

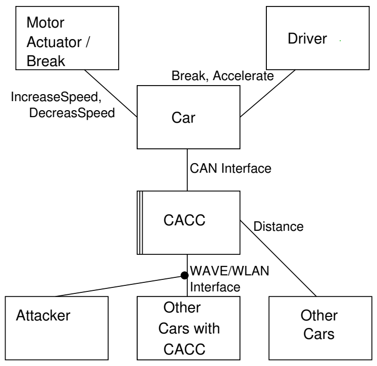

Fig. 3. CACC context diagram

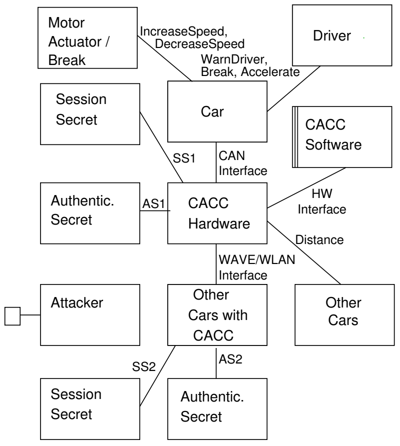

Fig. 4. CACC context diagram after mechanisms have been selected

ments and solution approaches that additionally cover all dependent requirements and corresponding solution approaches, some of which may not have been known initially.

The next step is to check the **Related** column. There, dependability requirements that are commonly used in combination with the described dependability pattern are mentioned. This information helps to find missing dependability requirements right at the beginning of the requirements engineering process. The dependencies for security requirements are based on previous work [9].

Table 1 only shows some important dependencies used in Section 5. A comprehensive version can be found in our technical report [6]. The next step in the software development life-cycle is to **derive a specification**, which describes the machine and is the starting point for its development.

5 Case Study

The approach is illustrated by 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[5]. 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.[6] Examples for domain knowledge of the CACC in the **described environment** are physical properties about acceleration, breaking, 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 *WAVE/WLAN interface*. The context diagram for the CACC is shown in Fig. 3. The **functional requirement** for the CACC is to maintain string stability.

[5]cf. United States Patent 20070083318

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

R1 The CACC should control the speed of a ***Car*** using the ***MotorActuator_Break*** according to the desired speed given by the ***Driver*** and the measured *Distance* to the car ahead (commanded behaviour, see Section 2).

R2 The CACC should also consider information about speed and acceleration of ***Other Cars with CACC*** ahead which is sent using a wireless network (*WAVE/WLAN interface*) (required behaviour).

The next step is to **identify an initial set of dependability requirements**. For the functional requirement **R2**, the following security requirement can be stated using the textual pattern from Section 3.3:

> The ***influence (as described in the functional requirement) on*** the ***MotorActuator_Break*** must be either correct, or in case of any modifications by some ***Attacker the ConstrainedDomain should <u>not</u> be influenced <u>and</u> the Driver must be informed***.

These requirements can be expressed using the integrity predicates

$$\begin{aligned}\forall\, mab :\ & MotorActuator\_Break;\ dr : Driver;\ a : Attacker\ \bullet \\ & int_{att}(mab, mab, a) \wedge int_{att}(mab, dr, a)\end{aligned} \tag{1}$$

The first occurrence of the variable *mab* in Equation 1 refers to the influenced domain as described in the functional requirement, and the second occurrence of *mab* expresses that this domain is not influenced in case of an attack.

A safety requirement is to keep a safe distance to the car ahead while being activated (see **R1**). For each safety requirement the integrity or the reliability must be defined. For the CACC only integrity is required, because it is safe to switch off the functionality and inform the driver in case of a failure. The risk analysis performed in the first step showed that a probability of at most 10^{-6} untreated random errors per hour (that may lead to an accident) can be accepted. Hence, for **R1** it can be stated that

> With a probability of $1 - 10^{-6}$ ***per hour***, one of the following things should happen: service (described in the functional requirement) ***with influence on*** the ***MotorActuator_Break*** must be correct, or ***the ConstrainedDomain should <u>not</u> be influenced <u>and</u> the Driver must be informed***.

The corresponding predicates are:

$$\begin{aligned}\forall\, mab :\ & MotorActuator\_Break;\ dr : Driver\ \bullet \\ & int_{rnd}(mab, mab, 1 - 10^{-6}) \wedge int_{rnd}(mab, dr, 1 - 10^{-6})\end{aligned} \tag{2}$$

Additionally, to satisfy the drivers buying the CACC:

> The service (described in the functional requirement) ***with influence on*** the ***MotorActuator_Break*** must be available with a probability of $1 - 10^{-4}$.

This requirement can be expressed with the predicate

$$\forall\, mab : MotorActuator\_Break\ \bullet\ avail_{rnd}(mab, 1 - 10^{-4}) \tag{3}$$

For availability, we only consider random faults, because for the corresponding security requirement we have to limit the group of users (the service is provided for) as described in Section 3.5, and this is not possible in the described environment.

The next step is to **Select appropriate generic mechanisms** for each dependability requirement expressed as a predicate. Depending on the generic mechanism, additional **domains must be introduced or considered**.

To establish Equation 1 messages authentication codes (MACs) can be used to check integrity and authenticity of the messages (position, acceleration and speed data) from other cars with trusted CACCs. According to Table 1, ***Secret***s for sender and receiver are necessary to calculate and verify the MAC. We decide to use ***SessionSecret***s for Sender (ss_2) and Receiver (ss_1). A ***SessionSecret*** has the advantage that it has a short life-time: even if the attacker is able to obtain this secret, it can only be used for a short time period. The "**necessary conditions**" column of Table 1 shows for the MAC mechanism that the secrets (ss_1 and ss_2) and the machine processing the secrets (*cacc*) must be protected from modification and disclosure. In case of any modification by the ***attacker***, the ***driver*** is informed, and there will be no influence on ***MotorActuator_Break*** (Equation 4). The "**conditions to be established beforehand**" column of Table 1 shows that the secrets must be distributed beforehand (Equation 7), as stated with the following predicates:

$$\begin{aligned}
\forall\, cacc :\ & CACC;\ ss_1, ss_2 : SessionSecret;\\
mab :\ & MotorActuator\_Break;\ dr : Driver, a : Attacker\ \bullet\\
& conf_{att}(cacc, dr, a) \wedge int_{att}(cacc, dr, a) \wedge int_{att}(cacc, mab, a) \wedge && (4)\\
& conf_{att}(ss_1, dr, a) \wedge int_{att}(ss_1, dr, a) \wedge int_{att}(ss_1, mab, a) \wedge && (5)\\
& conf_{att}(ss_2, dr, a) \wedge int_{att}(ss_2, dr, a) \wedge int_{att}(ss_2, mab, a) \wedge && (6)\\
& dist_{att}(ss_1, cacc, a) \wedge dist_{att}(ss_2, cacc, a) && (7)
\end{aligned}$$

The integrity and confidentiality of the ***CACC*** with its data, in particular the ***SessionSecret*** ss_1 (required by Equations 4 and 5), can be established by some physical protection. The ***SessionSecret*** ss_2 is stored in the ***OtherCarsWithCACC***. Its confidentiality and integrity (Equation 6) are also established by physical protection. To establish Equation 7, a dynamic authentication mechanism with random numbers can be used. With this authentication mechanism additionally a session key can be generated. Since replay attacks cannot be avoided in the described context, random numbers are used for authentication (cf. *CSPF Dynamic Authentication* in [7]). The necessary conditions for this mechanism are similar to those for MAC protection. Integrity and confidentiality of the machines and secrets are established in the same way as for the MAC protection. Secure distribution of the ***AuthenticationSecret***s is assumed to be done in the production environment of the ***CACC***.

To establish Equation 2, we regard the machine ***CACC*** as consisting of two parts: the ***CACCSoftware*** $cacc_{SW}$ and the ***CACCHardware*** $cacc_{HW}$. For the hardware we use, a reliability of only $1 - 10^{-4}$ is guaranteed. For our software we assume (and try to achieve using several quality assurance activities, see ISO/IEC 61508 [11, Part 3]) a reliability of $1 - 10^{-6}$. Therefore, our software must check the hardware and initiate the required actions. Several checks on the hardware have to be performed as given, e.g., in the standard ISO/IEC 61508, Part 2, Tables A.1 to A.15 [11]. The first row of Table 1 shows for the checksum mechanism (as one example from [11]) that the integrity of the ***Machine*** and of the ***Display*** have to be ensured; i.e., if these domains are not able to forward the warning to the user, the user must be informed by other means. The ***Machine*** is here the ***CACC Hardware***, and the ***Display*** is here (to simplify the example) the ***Car*** used in the following predicates (Equation 8). The warning is given acoustically and visually to increase the probability that the ***Driver*** recognizes the warning. Additionally, it is necessary that in this case there is no automatic control of the speed of the car, i.e., no influence on the ***MotorActuator_Break*** (Equation 9).

$$\forall\, cacc_{HW} : \; CACCHardware; \; car : Car; \; dr : Driver; \; mab : MotorActuator\_Break \;\bullet$$

$$int_{rnd}(cacc_{HW}, dr, 1-10^{-6}) \wedge int_{rnd}(car, dr, 1-10^{-6}) \wedge \qquad (8)$$

$$int_{rnd}(cacc_{HW}, mab, 1-10^{-6}) \wedge int_{rnd}(car, mab, 1-10^{-6}) \wedge \qquad (9)$$

The first part of Equations 8 and 9 cannot be assumed, because of the reliability of the hardware is only $1-10^{-4}$. Therefore, our solution for this contradiction consists of two parts. The first one is the dependability requirements for the software:

> With a probability of $1-10^{-6}$, one of the following things should happen: service (described in the functional requirement) ***of*** the ***Hardware*** must be correct, or ***the CACC must be switched off using*** **SwitchOffPartsOfCACCHardware** $cacc_{HW\_OFF}$ (omitted in Fig. 4).

$$\forall\, cacc_{HW} : \; CACCHardware; \; cacc_{HW\_OFF} : SwitchOffPartsOfCACCHardware \;\bullet$$

$$int_{rnd}(cacc_{HW}, cacc_{HW\_OFF}, 1-10^{-6}) \qquad (10)$$

To establish Equation 10, the pre-requisites according to Table 1 can be fulfilled by a reliability of $1-10^{-6}$ for the ***CACCSoftware*** and the ***SwitchOffPartsOfCACCHardware***, which is also assumed.

The second part of the solution is that the ***Car*** has to detect a switched-off ***CACC***. In this case the ***Car*** should warn the driver, and the ***Car*** should not use the output of the ***CACC*** to control the ***MotorActuator_Break***. For this requirement (R_{car}), the following reliability (stated as a predicate) is necessary and must be assumed for the CACC development[7]:

$$\forall\, car : Car \;\bullet\; rel_{rnd}(car, 1-10^{-6}) \qquad (11)$$

To establish Equation 3, reliable hardware and software can be used, because $rel_{rnd}(c,P) \Rightarrow avail_{rnd}(c,P)$. The required reliabilities of the machine and all relevant connection domains are assumed as shown in [6]. The new context diagram for the CACC resulting from applying dependability requirements patterns is shown in Fig. 4. New domains were **added to the description of the environment**, and the connection of the attacker to the *WAVE/WLAN Interface* is replaced by the more generic "window to the world", because the new domains ***SessionSecret AuthenticationSecret***, and the ***CACC*** itself are of great interest for the ***Attacker***. Additionally, the machine ***CACC*** is split into ***CACCHardware*** and ***CACCSoftware***. Since some dependability requirements state that the ***Driver*** must be informed, the additional phenomenon *WarnDriver* is introduced.

By using the dependencies given in Section 4, we systematically developed more than 27 dependability requirements to be inspected from the 3 initial dependability requirements.

6 Related Work

We are not aware of any similar approach for modeling a wide range of dependability requirements. However, the Common Criteria [1], Part 2 define a large set of so-called *Security Functional Requirements (SFRs)* with explicitly given dependencies between these SFRs. 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* ***s****upport*

[7]Equation 11 expresses together with the functional requirement R_{car} the same requirements as Equations 8 and 9.

(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. The dependencies given in the Common Criteria are re-used for our pattern system. Our dependability requirements can be regarded on the level of Security Objectives that have to be stated according to Common Criteria, Part 3, before suitable SFRs are selected. For example, for int_{att_d} the SFRs *Cryptographic operation (FCS_COP)*, *Cryptographic key management (FCS_CKM)*, and Stored data integrity (FDP_SDI) can be instantiated.

7 Conclusions and Future Work

In this paper, we have presented a set of patterns for expressing and analyzing dependability requirements. These patterns are separated from the functional requirements and expressed without anticipating solutions. They can be used to create re-usable dependability requirement descriptions for a wide range of problems.

This paper also describes a pattern system that can be used to identify missing requirements in a systematic way. The pattern system is based on the predicates used to express the requirements. The parameters of the predicates refer to domains of the environment descriptions and are used to describe the dependencies precisely. The pattern system may also show possible conflicts between dependability requirements in an early requirements engineering phase.

In summary, our pattern system has the following advantages:

- The dependability patterns are re-usable for different projects.
- A manageable number of patterns can be applied on a wide range of problems, because they are separated from the functional requirements.
- Requirements expressed by instantiated patterns only refer to the environment description and are independent from solutions. Hence, they can be easily re-used for new product versions.
- The patterns closely relate predicates and their textual descriptions. The textual description helps to state the dependability requirements more precisely.
- The patterns help to structure and classify the dependability requirements. For example, requirements considering integrity can be easily distinguished from availability requirements. It is also possible to trace all dependability requirements that refer to one domain.
- The predicates are the basis of a modular construction system used to identify dependencies and possible interactions with other dependability requirements.

In the future, we plan to elaborate more on the later phases of software development. For example, we want to apply our patterns to software components to show that a certain architecture is dependable enough for its intended usage. Additionally, we plan to systematically search for missing dependability requirements and dependencies using existing specifications (e.g., public Security Targets).

References

1. Common Criteria for Information Technology Security Evaluation, Version 3.1, September 2006. http://www.commoncriteriaportal.org/public/expert/.
2. A. Avizienis, J.-C. Laprie, B. Randall, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. http://se2c.uni.lu/tiki/se2c-bib_download.php?id=2433.

3. I. Côté, D. Hatebur, M. Heisel, H. Schmidt, and I. Wentzlaff. A systematic account of problem frames. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP 2007)*. Universitätsverlag Konstanz, 2008.
4. P.-J. Courtois. Safety, reliability and software based systems requirements. *Contribution to the UK ACSNI Report of the Study Group on the safety of Operational Computer Systems*, June 1997.
5. S. Gürses, J. H. Jahnke, C. Obry, A. Onabajo, T. Santen, and M. Price. Eliciting confidentiality requirements in practice. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 101–116. IBM Press, 2005.
6. D. Hatebur and M. Heisel. A foundation for requirements analysis of dependable software (technical report). Technical report, Universität Duisburg-Essen, 2009. http://swe.uni-due.de/techrep/founddep.pdf.
7. D. Hatebur, M. Heisel, and H. Schmidt. Security engineering using problem frames. In G. Müller, editor, *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS'06)*, LNCS 3995, pages 238–253. Springer-Verlag, 2006.
8. 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.
9. D. Hatebur, M. Heisel, and H. Schmidt. Analysis and component-based realization of security requirements. In *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*, IEEE Transactions, pages 195–203. IEEE, 2008.
10. D. Hatebur, M. Heisel, and H. Schmidt. A formal metamodel for problem frames. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 5301, pages 68–82. Springer Berlin / Heidelberg, 2008.
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. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.
14. J.-C. Laprie. Dependability computing and fault tolerance: Concepts and terminology. *Fault-Tolerant Computing – Highlights from Twenty-Five Years*, pages 2–13, June 1995. http://lion.ee.ntu.edu.tw/Class/FTDS_2008/Laprie-Definitions.pdf.
15. A. Pfitzmann and M. Hansen. Anonymity, unlinkability, unobservability, pseudonymity, and identity management - a consolidated proposal for terminology. Technical report, TU Dresden and ULD Kiel, 5 2006. http://dud.inf.tu-dresden.de/Anon_Terminology.shtml.
16. L. Røstad, I. A. Tøndel, M. B. Line, and O. Nordland. Safety vs. security. In M. G. Stamatelatos and H. S. Blackman, editors, *Safety Assessment and Management - PSAM 8, Eighth International Conference on Probabilistic*. ASME Press, New York, 2006.
17. T. Santen. Stepwise development of secure systems. In J. Górski, editor, *International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS 4166, pages 142–155. Springer-Verlag, 2006.