Reuse in Requirements Engineering: Discovery and Application of a Real-Time Requirement Pattern*

R. Gotzhein, M. Kronenburg, and C. Peper

University of Kaiserslautern, Germany
{gotzhein|kronburg|peper}@informatik.uni-kl.de

Abstract. We apply a reuse approach to formally specify system requirements. The approach is based on *requirement patterns*, which are related to *design patterns* well-known in object-oriented software development. The focus of this paper is on the application-driven discovery of an interesting, non-trivial real-time requirement pattern, and its subsequent application. In particular, we demonstrate how a given natural language requirement for a real-time system has been transformed into a precise and concise formal specification using real-time temporal logic. Finally, we motivate and illustrate the reuse potential of requirement specifications by a diverse collection of examples.

Keywords: requirements analysis, reuse, pattern discovery, formal specification, temporal logic, real-time, formal methods, case study

1 Introduction

Requirements analysis and specification are among the first steps of system development. In order to avoid later disagreements between customer and system developer, and to obtain a sound basis for subsequent development steps, it is essential that requirements be stated completely and precisely, while still being intelligible. In practice, requirements are often stated unprecisely – due to the use of natural language – and incompletely – due to the inherent difficulty to perceive all essential aspects of the problem to be solved. This could lead to disagreements during subsequent development stages including the acceptance of the final product by the customer. In order to avoid these problems, the use of formal description techniques (FDTs) for the specification of requirements is being advocated.

Especially for large and complex systems, the investment to obtain a "good" requirement specification is considerable. To reduce this effort, it is possible to benefit from earlier system developments by reusing developer experience as well as parts of already developed products. Recently, *design patterns* [2] have

* This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Sonderforschungsbereich (SFB) 501, "Development of Large Systems with Generic Methods".

been advocated as a promising reuse concept for the design phase, which is related to other well-known approaches such as *frameworks*, or *toolkits*. We have adopted this idea in [3,4], where we have introduced and applied so-called SDL patterns for the development of communication protocols. Different from most reuse activities so far, which are informal, *SDL patterns* define building blocks that – when selected, adapted and composed properly – yield a formal protocol specification in SDL, an FDT that is internationally standardized and widely applied in industry.

It has been observed that the earlier the reuse, the larger the impact on a project [11]. It follows from this observation that reuse in the requirements phase may have a positive impact on subsequent development stages by an increased reuse of designs and implementations. While reuse has been well studied for systems *design*, less research is available on how to apply this principle to specify system *requirements*. In [10], we have introduced an approach that increases the reuse potential already in this early development stage. Our approach is based on so-called *requirement patterns*, which describe generic solutions for recurring problems and represent experience gained during previous system development activities.

In Section 2, we outline our reuse approach with emphasis on those aspects that are addressed in detail afterwards. In Section 3, we list an excerpt of a description of a real-time problem from the requirements document of a customer, which is stated in natural language augmented by a diagram as often found in this kind of document. In Section 4, we present our final result, consisting of a surprisingly concise formal specification using a real-time temporal logic with tailored operators. In Section 5, we extract the underlying requirement pattern, add it to our requirement pattern pool, and apply it to diverse requirements. We conclude with an outlook (Section 6).

2 Outline of Our Reuse Approach

In this section, we outline our reuse approach with emphasis on those aspects that are illustrated afterwards. The complete approach and additional details can be found in [10].

2.1 A Requirement Specification Development Model

System development usually starts with a *natural language requirement specification* (NLRS) consisting of an initial set of requirements that is supplied by the customer ("Customer NLRS" for short, see Figure 1a)). As already mentioned, natural language has no unique semantics. Therefore, these requirements should be formalized by the system developer, yielding a *formal requirement specification* (FRS).

As a result of this formalization, existing ambiguities of the natural language description are resolved in one particular way, which may differ from the original intentions of the customer. Therefore, the customer needs to check whether his

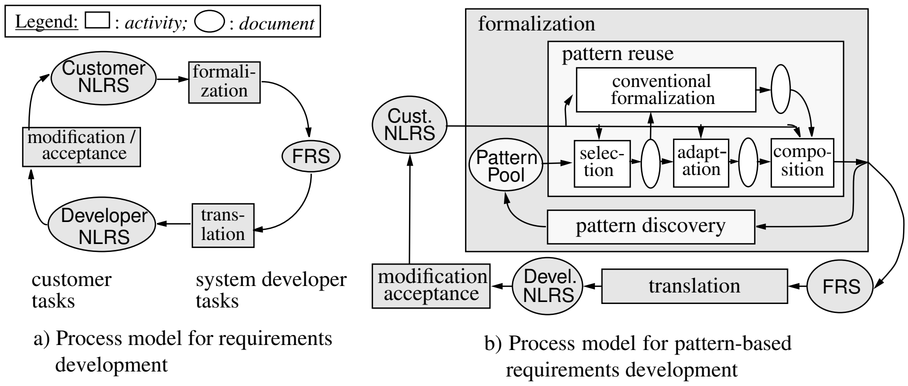

Fig. 1. Process models

intentions are correctly expressed in the FRS. Since the customer may not have a background in FDTs, we translate the FRS back to natural language resulting in a further document called "Developer NLRS" (see Figure 1a)). Since this natural language description is directly translated from a formal specification, we assume that it is more precise than the original Customer NLRS. The Developer NLRS may now serve as a basis for customer and developer to reach agreement on the system requirements.

If agreement is reached, the Developer NLRS replaces the previous Customer NLRS and serves as the basis for the acceptance of the final implementation. As another benefit, the new Customer NLRS already has a corresponding formalization, namely the FRS, which can be used as the starting point for subsequent development steps. If agreement is not reached, the customer supplies a modified Customer NLRS based on the Developer NLRS, and another cycle of the requirement specification development is started. Thus, the development of a requirement specification is an iterative process.

2.2 Reuse of Requirement Patterns

As we have argued, the described approach to the development of requirement specifications has a number of benefits. However, the effort to produce the FRS and its derived natural language description usually is considerable, especially if large and complex systems are to be characterized. To reduce this effort, we have proposed a *reuse approach* to the formalization of requirements. Our approach is based on a *pool of requirement patterns*. By *requirement pattern*, we refer to a generic description of a class of domain-specific requirements. Requirement patterns are related to *design patterns*, a well-known concept of object-oriented software development [2].

To describe requirement patterns, we use the format shown in Table 1, called *requirement pattern description template*. Instantiations of this template are termed *requirement patterns*, which, itself instantiated, form the constituents of a requirement specification. The contents of the template will depend on the application domain and the FDT used to specify patterns and their semantic properties.

Table 1. Requirement pattern description template

Name	The name of the requirement pattern.
Intention	An informal description of the kind of requirements addressed by this pattern.
Example	An example from the application domain illustrating the purpose of the requirement pattern.
Definition	The pattern is described both formally, using a suitable FDT, and in natural language. The formal description is the basis for subsequent development steps finally leading to the requirement specification (pattern selection, adaptation, and composition). The description in natural language will serve the translation of instantiated patterns of the FRS into informal requirements of the NLRS. Furthermore, the description provides some information about possible instantiations.
Semantic properties	Properties that have been formally proven from the pattern. By instantiating these properties in the same way as the requirement pattern, proofs can be reused, too.

Based on the pattern pool and the Customer NLRS, the formalization of requirements through pattern reuse consists of the following steps (see Figure 1b)):

Step 1. Requirement patterns are *selected* from the pattern pool. This selection is supported by information provided by pattern descriptions such as intention, definition, and semantic properties (see Table 1).

Step 2. The selected patterns are *adapted* by suitable instantiations. The same kind of adaptation is applied to the semantic properties. Already at this stage, it is possible to formally reason about single requirements.

Step 3. The adapted patterns are *composed* to yield the requirement specification. Composition can be supported by suitable operators.

The degree to which the formalization of requirements can be achieved through pattern reuse depends both on the Customer NLRS and the contents of the pattern pool. If the structure of an informal requirement follows a requirement pattern that is already contained in the pool, then its formalization can be achieved directly by instantiating the pattern. If the structure is different, then either transformations and/or modifications of the informal requirement (cf. Section 2.1) may lead to a structure that is already supported by the pattern pool, or the formalization must be done in the conventional way, i.e. without reuse (see Figure 1b)). The pool of requirement patterns should evolve over time. As a consequence, the portion of requirements that is developed from requirement patterns will increase, reducing the overall effort of requirement specification.

2.3 Discovery of Requirement Patterns

So far, we have assumed the existence of a pattern pool containing a set of already known domain specific patterns, where each entry follows the template defined in Table 1. The main difficulty here is that it is by no means obvious a priori what patterns will be useful later on, as this depends on the application domain as well as on the requirements to be specified. Therefore, building up the pattern pool will be an iterative process itself. This *pattern discovery* task can be modeled as a sub-cycle in the specification development model. Typically, each external specification development cycle triggers one or more internal pattern discovery/reuse cycles affecting both pattern pool and FRS, since each FRS modification can lead to new patterns or the improvement of existing patterns.

The discovery of new patterns is a difficult and time-consuming process. In general, many requirements have similarities in the way they restrict time bounds, delays, and dependences between system states or other domain specific properties. These similarities can be exploited in order to extract the underlying patterns. Below, we report on the discovery of a particular real-time requirement pattern in detail. The discovery of another real-time requirement pattern is presented in [10]. From these examples, it becomes evident that pattern discovery indeed is a considerable investment that will only pay off through extensive pattern reuse.

3 The Original Problem Description

The requirement discussed in the following is taken from the domain of building-automation. Building-automation systems usually exhibit reactivity, real-time behaviour, scalability, and distribution, and therefore qualify as instances of large software systems. Thus, requirements of such systems as well as methods and techniques to specify and analyse them can be evaluated in that domain.

Among the tasks of a building-automation system are the control of temperature, humidity, light, air flow, security, and safety. Depending on the type of a building (e.g. house, research lab, hospital, airport terminal), hundreds or thousands of sensors and actuators can be involved in maintaining required conditions in a building. Our approach in the requirements phase in general has been to use the initial informal problem description as delivered by the customer (initial Customer NLRS) in order to work out a precise and formal specification in real-time temporal logic (FRS) [10]. These formulae were then translated back to natural language, so that the resulting description (Developer NLRS) could be compared with the initial one and be discussed with the customer. When, after necessary revisions, full agreement with the customer was reached, the temporal formulae formed the basis for subsequent design activities.

In the following, we focus on the formalization of a real-time requirement, the extraction of the underlying requirement pattern, and its application. Our starting point is an excerpt of the original problem description, stated in natural language augmented by a diagram, which forms part of a larger document:

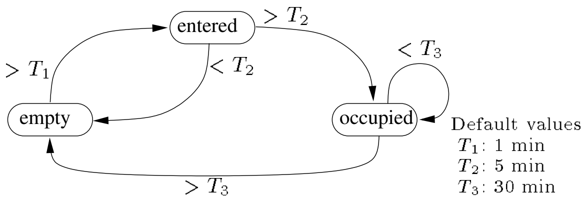

Fig. 2. State diagram

The comfort temperature shall be maintained during preset comfort times or reached as fast as possible, when the room is occupied for a longer period of time. This state is called "occupied" and is explained in the state diagram. This requirement shall be fulfilled during the heating period, and, if possible, also during the rest of the year. During the states "empty" and "entered", appropriate action has to be taken. We distinguish three time spans for detecting occupancy. If a person is in a room shorter than T_1 minutes, the control system does not try to reach the comfort temperature (state "empty", stand-by or off-time temperature, depending on the time period). If the person leaves the room before T_2 minutes, the system returns to the previous temperature. If the person occupies the room longer than T_2, the system enters the state "occupied", where the comfort temperature should be held, until this person leaves the room for at least T_3 minutes. Different settings for (T_1, T_2, T_3) may exist for each room, depending on the expected usage (day, night, vacation, etc.).

4 The Final Formal Specification of the Real-Time Requirement

The final formal specification (FRS) is written in a propositional real-time temporal logic with tailored operators (for details, see [5] and [9]). Here we only give the informal semantics of the temporal operators used in this paper. Our basic model is a set of timed state sequences, as defined in [1]. Note that time is dense.

- $\Box\varphi$ means that φ is true now and will be true at all future points in time.
- $\blacksquare\varphi$ means that φ is true now and has always been true at all points in time of the current context in the past. The current context in the past usually covers the entire past, i.e. from the beginning of the timed state sequence. However, it may be limited using interval operators (see below).
- $\blacksquare_{\leq T}\varphi$ means that φ is true now and has always been true in the preceeding time span T, limited by the current context in the past. I.e., if the current context in the past is shorter than T, φ is only required to be true during that context.
- $\blacklozenge\varphi \leftrightarrow_{Df} \neg\blacksquare\neg\varphi$; $\blacklozenge_{\leq T}\varphi \leftrightarrow_{Df} \neg\blacksquare_{\leq T}\neg\varphi$
- $[\varphi]$ (read "action of type φ") means that φ has just become true, i.e. φ is true now and has been false in the preceeding state (note that time is dense, however, the state sequence is discrete).
- $\langle[\varphi_1] \Rightarrow\rangle\varphi_2$ (read "φ_2 restricted on the context $[\varphi_1] \Rightarrow$") means that φ_2 holds in the context limited backwards by the last action of type φ_1; the formula is vacuously true if the context does not exist.
- $\langle *[\varphi_1] \Rightarrow\rangle\varphi_2 \leftrightarrow_{Df} \langle[\varphi_1] \Rightarrow\rangle\varphi_2 \wedge \blacklozenge[\varphi_1]$

Note that the final formal specification listed below has been done conventionally, without the use of requirement patterns (see Figure 1b)). It uses several predicates considered atomic in the sense that they are not refined at this stage, with the following intuitive meaning:

roomEmpty	no person present (not to be confused with the state "empty"!)
comfortTime	current time within a time period where presence of persons is expected (e.g. during regular working hours)
comfortTemperature	valves of the heating system are currently controlled such that the comfort temperature is either reached within certain time bounds and maintained, or approximated, depending on outside temperatures
standByTemperature	as before, w.r.t. stand-by temperature
offTimeTemperature	as before, w.r.t. off-time temperature

Based on *roomEmpty*, the non-atomic (auxiliary) predicate *roomUsed* is defined:

D1. $$roomUsed \leftrightarrow_{Df} \blacksquare_{\leq T_1} \neg roomEmpty \vee \langle * [\blacksquare_{\leq T_1 + T_2} \neg roomEmpty] \Rightarrow \rangle \blacksquare \blacklozenge_{\leq T_3} \neg roomEmpty$$

Informally, a room is currently in use (written *roomUsed*) iff one of the following statements holds:

- during the past T_1 time units, a person has continuously been in the room;
- there has already been a time span of length $T_1 + T_2$ such that a person has continuously been in the room, and since this has been the case for the last time, periods where the room was empty were not longer than or equal to T_3.

The real-time requirement can then be formalized as follows:

R1. $\Box(roomUsed \rightarrow comfortTemperature)$
R2. $\Box(\neg roomUsed \wedge comfortTime \rightarrow standByTemperature)$
R3. $\Box(\neg roomUsed \wedge \neg comfortTime \rightarrow offTimeTemperature)$

Informally, the comfort temperature has to be reached and/or maintained whenever the room is used as defined by *roomUsed* (R1). During the remaining time periods, the stand-by or off-time temperatures apply, depending on the setting of *comfortTime* (R2, R3). Note that this requirement is stated on a high abstraction level. It will, for instance, be necessary to refine the predicate *roomEmpty* by introducing a collection of sensors and by specifying how these sensors determine its value. Also, the predicates *comfortTime*, *comfortTemperature*, *standByTemperature*, and *offTimeTemperature* have to be refined.

It is not obvious that the formal specification indeed captures the original problem description stated in Section 3. Actually, it doesn't, since natural language does not possess a formal semantics, and since the original description exhibits some ambiguities. So the goal must be to argue why the formal specification captures the intentions of the informal description. The detailed justification and a proof of the soundness and completeness of the final formalization are presented in [7].

5 Extraction and Application of the Requirement Pattern

While working on application requirements in the domain of building-automation, we have found that most of them obey a small number of syntactic patterns [10]. One of the patterns we found has been extracted from the definition of roomUsed and is shown in Table 2. We have termed this pattern "LazyReaction", because it is applicable in cases where a reaction is not immediate, but is delayed by a specified time period.

Table 2. LazyReaction pattern

Name	LazyReaction (p, q, T_1, T_2, T_3)
Intention	Applicable if a reaction is not immediate, but contains some delay.
Example	$roomUsed \leftrightarrow_{Df} (\blacksquare_{\leq T_1} \neg roomEmpty \vee \langle * [\blacksquare_{\leq T_1 + T_2} \neg roomEmpty] \Rightarrow \rangle \blacksquare \blacklozenge_{\leq T_3} roomEmpty)$
	A room is currently in use (*roomUsed*) iff one of the following holds: – during the past T_1 time units, a person has continuously been in the room; – there has already been a time span of length $T_1 + T_2$ such that a person has continuously been in the room, and since this has been the case for the last time, periods where the room was empty were not longer than or equal to T_3.
Definition	$p \leftrightarrow_{Df} \blacksquare_{\leq T_1} q \vee \langle * [\blacksquare_{\leq T_1 + T_2} q] \Rightarrow \rangle \blacksquare \blacklozenge_{\leq T_3} q$
	The predicate p holds iff one of the following statements holds: – during the past T_1 time units, q has continuously been true; – there has already been a time span of length $T_1 + T_2$ such that q has continuously been in the room, and since this has been the case for the last time, periods where q was false were not longer than or equal to T_3.
Semantic properties	$T_1 = T_2 = T_3 = 0 \rightarrow p \leftrightarrow q$; $T_2 = T_3 = 0 \rightarrow p \leftrightarrow \blacksquare_{\leq T_1} q$ $T_2 = 0 \rightarrow p \leftrightarrow \langle * [\blacksquare_{\leq T_1} q] \Rightarrow \rangle \blacksquare \blacklozenge_{\leq T_3} q$; $T_1 = T_2 = 0 \rightarrow p \leftrightarrow \blacklozenge_{\leq T_3} q$

The pattern has the parameters p, q, T_1, T_2, and T_3, which have to be instantiated in order to obtain a particular requirement. By instantiating p and q with formulae, a more specific pattern is obtained. Therefore, pattern instantiation can be understood as an incremental process. The time parameters T_1, T_2, and T_3 can be replaced by specific values, and/or restricted by adding constraints (note that a replacement by a value is a special case of an added constraint). The entry "semantic properties" contains special cases of pattern instantiations, where based on a particular selection of time parameters, it is possible to transform the resulting formula into an even more concise expression. These expressions have been proven equivalent to the original pattern.

Below, we list some requirements occurring in air flow control, motion detection, and safety control. These requirements include applications of the *LazyReaction* pattern. Further applications of this pattern can be found in [7].

Motion detection
The specification of a motion detector on sale in a hardware store was obtained by instantiating p, q, T_1, T_2 with *personPresent*, *motion*, 0, and 0, respectively, by restricting T_3 as shown in D5, and by adding R5:

motion	motion detected (implemented by an infrared sensor)
darkness	light intensity below a threshold value, adjustable from 2 to 2000 lux
lightOn	a light is switched on to illuminate the considered area

D4. $personPresent \leftrightarrow_{Df} \blacklozenge_{\leq T_3} motion$
D5. $T_3 \in [10 seconds, 15 minutes]$
R5. $\Box(darkness \wedge personPresent \leftrightarrow lightOn)$

Safety control
By instantiating p and q with *hazardousCondition* and *hazardousSituation*, respectively, and by adding R6, we obtain a safety control requirement (see also [6]). As already discussed, the time parameters can either be replaced by specific values, or restricted by additional properties.

hazardousSituation	holds in case of heavy winds, heavy rainfall, etc.
upperSashClosed	upper sash closed

D6. $hazardousCondition \leftrightarrow_{Df} \blacksquare_{\leq T_1} hazardousSituation \vee$
$\langle *[\blacksquare_{\leq T_1+T_2} hazardousSituation] \Rightarrow \rangle \blacksquare \blacklozenge_{\leq T_3} hazardousSituation$
R6. $\Box(hazardousCondition \mapsto_{\leq T} upperSashClosed)$

R6 uses a tailored operator, $\mapsto_{\leq}$, of the real-time temporal logic introduced in [9]. Informally, R6 states that each time *hazardousCondition* holds continuously for at least T time units, the upper sash must be closed within this time span and must then remain closed as long as *hazardousCondition* holds. R6 is an instantiation of another pattern occurring quite frequently in requirements on building-automation systems, namely $\Box(r \mapsto_{\leq T} s)$ (compare [10]).

6 Conclusions and Outlook

The focus of this paper has been on the discovery and the application of an interesting, non-trivial real-time requirement pattern called *LazyReaction* pattern. Starting with the description of a real-time problem in natural language augmented by a diagram, we have developed a precise and concise formal requirement specification in a step-by-step fashion.

It has become evident that pattern discovery is a time-consuming task and a major investment. *Good* requirement patterns are not just a by-product of specifying system requirements, but the result of rigorous development and continuous improvement. We have motivated and illustrated the reuse potential by a diverse collection of requirements in the domain of building-automation systems that we have formalized by applying the *LazyReaction* pattern. Thus, the original investment in developing a formal and rigorous specification pays off.

At the moment, only very few patterns have been reported, which keeps the reuse potential within narrow bounds. We expect that in the near future and with increasing experience, additional patterns will be discovered and added to the pattern pool. In fact, we perceive pattern discovery as a fertile field of future research.

Another benefit of pattern-based requirements engineering is that generic solutions that can be adapted to particular machine environments can be developed. We are currently developing distributed solutions for several real-time patterns, based on message passing. We expect that by a pattern-based development of systems and the resulting reuse, customized solutions can be developed with significantly less effort.

Acknowledgements

We are grateful for the stimulating environment provided by the SFB 501 and the discussions with other project members, in particular with Gerhard Zimmermann.

References

1. R. Alur, and T. A. Henzinger: Logics and Models of Real Time: A Survey. In: J. W. de Bakker, C. Huizing, W. P. de Roever, G. Rozenberg (Eds.), Real-Time: Theory and Practice, LNCS 600
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
3. B. Geppert, R. Gotzhein, F. Rößler: Configuring Communication Protocols Using SDL Patterns. in: A. Cavalli, A. Sarma (Eds.), SDL'97 - Time for Testing, Elsevier Science Publishers, Proceedings of the 8th SDL-Forum, Evry, France, Sept. 1997
4. B. Geppert, F. Rößler: Generic Engineering of Communication Protocols - Current Experience and Future Issues. Proceedings of the 1st IEEE International Conference on Formal Engineering Methods (ICFEM'97), Hiroshima, Japan, 1997
5. R. Gotzhein: Open Distributed Systems - On Concepts, Methods and Design from a Logical Point of View. Vieweg Wiesbaden, 1993
6. R. Gotzhein, B. Geppert, C. Peper, and F. Rößler: Generic Layout of Communication Subsystems - A Case Study. SFB 501 Report 14/96, University of Kaiserslautern, Germany, 1996
7. R. Gotzhein, M. Kronenburg, and C. Peper: Reuse in Requirements Engineering: Discovery and Application of a Real-Time Requirement Pattern. SFB 501 Report 8/98, University of Kaiserslautern, Germany, 1998
8. T. A. Henzinger, Z. Manna und A. Pnueli: Timed Transition Systems. In: J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg (Hrsg.), Real-Time: Theory in Practice, LNCS 600, Springer 1991, 226-251
9. M. Kronenburg, R. Gotzhein, and C. Peper: A Tailored Real-Time Temporal Logic for Specifying Requirements of Building-Automation Systems. SFB 501 Report 16/96, University of Kaiserslautern, Germany, 1996
10. C. Peper, R. Gotzhein, M. Kronenburg: A Generic Approach to the Formal Specification of Requirements. Proceedings of the 1st IEEE International Conference on Formal Engineering Methods (ICFEM'97), Hiroshima, Japan, 1997, pp. 252-261
11. R. E. Johnson: Frameworks = (Components + Patterns), Communications of the ACM, Vol. 40, No. 10 (October 1997), pp. 39-42