Reuse-Oriented Requirements Engineering with FoReST [1]

Reinhard Gotzhein[a], Jürgen Avenhaus[a], Martin Kronenburg[b],
Klaus Madlener[a], Christian Peper[c]

[a]Computer Science Department, University of Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern, Germany
{gotzhein,avenhaus,madlener}@informatik.uni-kl.de
[b]sd&m AG, Thomas-Dehler-Str. 27, D-81737 München, Germany
martin.kronenburg@sdm.de
[c]Fraunhofer Institute Experimental Software Engineering,
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
Christian.Peper@iese.fraunhofer.de

Abstract. Reuse of experience and solutions for recurring development problems is of the utmost importance for a pragmatic software engineering discipline. The earlier in the development process reuse is achieved, the larger its positive impact on the project. This paper reports on reuse approaches that are applicable during the requirements analysis phase.

1 Introduction

Especially for large and complex systems, the investment to obtain a "good" requirement specification is considerable. To reduce this effort, it is of the utmost importance to benefit from earlier system developments by reusing developer experience as well as parts of already developed products. This insight has led to basically three different reuse concepts [3]:

- *Components* are often characterized as self-contained, ready-to-use building blocks, which are selected from a component library and composed. As a component is usually defined by a piece of code, this reuse concept is perceived as being implementation-oriented. Being a syntactically complete code fragment, a component has to be used as it comes. Therefore, design for independence, for instance, by adequate parameterization, is an important issue here, as well as the definition of suitable "glue" to combine components.
- A *framework* is the skeleton of a system, to be adapted by the system developer. As compared to components, there are several major differences. Firstly, a framework anticipates certain design decisions that are common for a particular project family, while leaving open others that are specific to a given system; therefore, only semi-complete code is provided. Secondly, a framework, being a skeleton of

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

an entire system, has a coarser granularity than components, which makes selection and composition steps obsolete.

- *Patterns* describe generic solutions for recurring problems, which are to be customized for a particular, embedding context. Application of a particular pattern means that the pattern has to be selected from a pool, that it is adapted, and finally composed with other description fragments. The pattern pool can be seen as a repository of experience from previous projects, which has been analyzed and packaged. In comparison to frameworks and components, patterns are fine-grained and more flexible.

In [5], it has been observed that the earlier the reuse, the larger the (positive) impact on a project. While reuse has been well studied for the design and implementation phases, less results are available on how to achieve reuse in the requirements analysis phase. In [10], we have introduced a pattern-based approach for specifying system requirements in a property-oriented style, consisting of a pattern-based development process, a pattern pool, and language support. This approach is based on so-called *FoReST requirement patterns*, which describe generic solutions for recurring problems and capture experience gained in the requirements analysis of previous system developments. In [4], a complete pattern discovery process for a non-trivial requirement pattern is documented. In [8], the degree of reuse is increased by incorporating object-oriented concepts (class definitions, specialization) and parameterization. The FoReST-approach has been formalized in [9]. Meanwhile, the approach has been improved and consolidated.

The paper is structured as follows. In Section 2, we will outline the *Formal Requirement Specification Technique* (*FoReST*) for the object- and property-oriented specification of system requirements. In Section 3, we illustrate FoReST requirement patterns and report on our experiences so far. We draw conclusions in Section 4.

2 Object-oriented requirements specification with FoReST

Building automation systems are often *large systems*, controlling hospitals, airport terminals, or institutes, with thousands of sensors and actuators involved. Therefore, the problem documentation can be of substantial size and complexity. This observation already holds for the medium-sized problem documentations treated in our projects so far, which are in the order of 100 pages of text.

In order to manage both size and complexity and to enhance reusability, it is good practice to analyze and decompose the original problem description. This results in a more precise description of the *structural aspects* of the document. Methods such as OMT [11] and UML [1] aim at the definition of a model that reflects the structure of the system environment. Their application leads, among other things, to structure diagrams, such as class diagrams consisting of classes, associations, and class features. In Figure 1, an excerpt of the class diagram from one of our projects is shown, using UML notation. This class diagram provides an overview of the FoReST descriptions below by identifying classes and associations. It is not part of the FoReST requirement specification and serves only illustrative purposes here. However, it can be generated from the FoReST specification.

The use of FoReST to formally capture requirements is explained in several steps. To start with, the specification of classes including class features is introduced. Then, composition, specialization and inheritance, and finally parameterization are added.

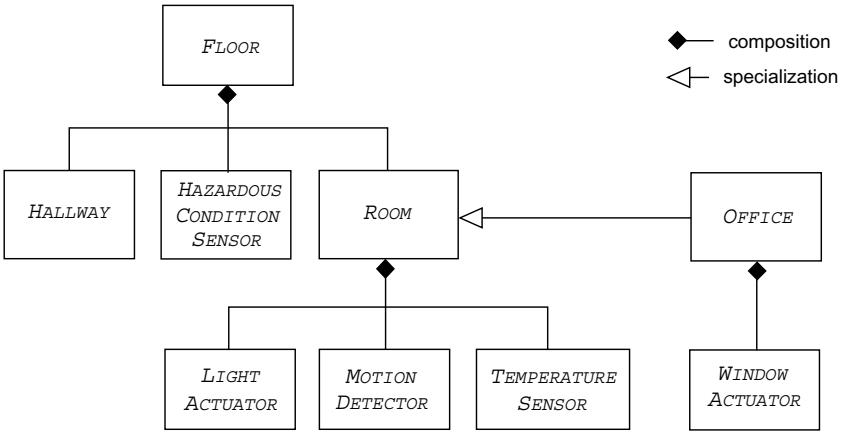


Fig. 1. UML class diagram of a floor (excerpt)

2.1 Specifying classes with FoReST

The application of object-oriented analysis methods leads to an architectural model that reflects the structure of the system environment. For each set of objects sharing the same features, i.e., attributes and behavior, a class is introduced. With FoReST, classes are defined in a tabular format by specifying a unique *class name*, a *signature*, and a *behavior*, using appropriate keywords to distinguish specification items.

The *signature* of a class is a sequence of attribute declarations, where each declaration consists of the attribute name, a classification and an intention. Attributes may, for instance, be classified as predicates, functions, or objects. Furthermore, predicates and functions may be static or dynamic (timed), stating whether their values may vary over time. The intention is stated informally, and serves documentation purposes.

In Table 1, the class `MOTIONDETECTOR` is specified, which is specified with the intention to abstract from pre-installed sensors that can detect motion. All instances of this class have the attributes `roomUsed` and `roomOccupied`, which are classified to be timed predicates. In addition to the signature, the behavior of objects is formally specified in a class definition. Unlike notations such as UML, which simply list operation signatures, we specify behavior by a list of properties. In our view, a property-oriented style is well-suited to capture requirements, since it omits operational, irrelevant details that can be left to the designer. Furthermore, we have found that informal problem descriptions are often written in this style, so using the same style in the formalization makes the documents better comparable. According to the product reference model in [6], we distinguish between different kinds of behavior statements, namely *domain*, *requirement* and *machine statements*. A domain statement describes pre-installed devices and/or existing environment behavior. A requirement statement

addresses the desired system behavior. Finally, a machine statement characterizes behavior of the machine, i.e., the part that is to be combined with the environment to achieve the desired system behavior. As a general rule, domain statements and machine statements, taken together, have to imply the requirement statements. All statements are specified using *tRTTL*, the *tailored Real Time Temporal Logic* [4], which is our particular choice of FDT; however, different FDTs may be used instead.

Class *MOTIONDETECTOR*	
Intention	: *MOTIONDETECTOR* is an abstract description of pre-installed sensors that can detect motion.
Signature	
Timed Predicate *roomUsed*	
Intention	: *roomUsed* is true iff there is a person in the room.
Timed Predicate *roomOccupied*	
Intention	: *roomOccupied* is true iff the motion detector has detected a person in the room.
Domain Statements	
Property *correctMotionDetection*	
Formal	: $\Box$ (*roomUsed* $\Rightarrow_{\leq 0.2\,s}$ *roomOccupied*)
NL	: Whenever the room is used for at least 0.2 seconds, the room is detected as occupied within this time span and remains occupied at least as long as the room is used.

Table 1. Class *MOTIONDETECTOR* (excerpt)

The definition of the class *MOTIONDETECTOR* (Table 1) contains the single statement *correctMotionDetection*, which establishes a causal relationship between two phenomena, namely *roomUsed* and *roomOccupied*. This constraint states that the causal relationship only exists with a specified delay, and that *roomUsed* has to hold continuously during this time span. To formally specify this domain statement, we use a special tRTTL operator called *weak delayed implication* (see [4]).

2.2 Specifying composition with FoReST

High-level objects are often modeled to consist of tightly-bound parts, and are therefore called *composite objects*. The parts of a composite object can be modeled as objects themselves, which may contain further parts, resulting in an *is-part-of*-hierarchy. A composite object is an instance of a *composite class*. In FoReST, a composite class is defined by specifying, for each part of its instances, an attribute that is classified as *object*, and is associated with a class. Instantiation of a composite object always implies the instantiation of its parts.

In Table 2, an excerpt of the FoReST specification of the composite class *ROOM* is shown. Each instance of this class is a composition of a light actuator, a motion detector, and a temperature sensor, which is specified by appropriate *object* attributes. The requirement statement *safeLightCondition* expresses a delayed causal relationship between an attribute of one of the room's parts, namely *roomUsed* of the motion detector *md*, and the attribute *safeLight*. Please note that the statement refers to an attribute of an object that is part of a room and to an attribute of the room itself. Therefore, the class *ROOM* is the right level in the is-part-of-hierarchy to place this statement.

Class *ROOM*	
Intention	: *ROOM* is an abstraction of rooms in a building.
Signature	
Object *la: LightActuator*	
Object *md : MotionDetector*	
Object *ts : TemperatureSensor*	
Dynamic Predicate *safeLight*	
Intention	: *safeLight* is true iff there is enough light in the room to move safely.
Requirement Statements	
Property *safeLightCondition*	
Formal	: $\Box(\mathit{md.roomUsed} \Rightarrow_{\leq 2s} \mathit{safeLight})$
NL	: Whenever the room is used for at least 2 seconds, there is safe light within this time span, and the light remains safe at least as long as the room is used.
Property *noFreezeCondition*	
Formal	: $\Box(\mathit{ts.temperature} > 4^{o}C)$
NL	: The room temperature is always above $4^{o}C$.

Table 2. Composition: class *ROOM* (excerpt)

2.3 Specialization and inheritance

Inheritance is one of the key mechanisms of object-oriented system development to support reusability of components. It is based on specialization, a (formal) relation between classes. Though being a semantic concept, specialization is mostly captured syntactically in existing language approaches including formal description techniques. This has the advantage that it is straightforward to use it as a basis for defining inheritance, and thus to apply this concept in practice.

Previous work [2] distinguishes two kinds of inheritance. *Incremental inheritance* is the process of adding or redefining features to/of an existing class definition, yielding derived class definitions. However, there is no guarantee that the objects of the newly-derived class can replace objects of the base class. *Subtyping inheritance*, on the other hand, ensures that objects of a derived class are also members of their base class. Thus, the substitutability constraint is met.

Class *OFFICE*
Intention : *OFFICE* is an abstraction of a certain room type.
Base Class
Class *ROOM*
Requirement Statements
Property *comfortTemperature*
Formal : ❑ (21° C < *ts.temperature* < 23° C)
NL : The room temperature is always between 21° C and 23° C.

Table 3. Specialization and inheritance: class *OFFICE* (excerpt)

Though having a semantic foundation, subtyping inheritance does not by itself support reusability during software development. On the other hand, incremental inheritance, while boosting reusability, does not meet the substitutability constraint. Object-oriented languages usually support incremental inheritance without taking subtyping inheritance into account, because the demand for substitutability limits the applicability of incremental inheritance and thus reusability. Unlike these languages, FoReST offers incremental inheritance that meets the substitutability constraint. By adding or redefining features to/of base classes, new classes may be derived. By convention, only the added or redefined features have to be explicitly stated, while all features of the base class are inherited or superseded. Note that in order to guarantee substitutability, redefinition of features is restricted. As any class definition, definitions of derived classes are required to be consistent.

In Table 3, the class *OFFICE* is derived from the base class *ROOM*. In addition to the attributes and statements defined in *ROOM*, *OFFICE* introduces a requirement statement *comfortTemperature*. It is straightforward to see that this statement is consistent with *noFreezeCondition*, where another constraint on *temperature* is defined, and which is inherited from *ROOM*.

2.4 Parameterization

It is common to define initial attribute values when objects are instantiated. If the initial value of an attribute is the same for all objects of a class, this value should be defined in the class definition. If the initial value depends on the context of the object, it

can be defined using a parameter mechanism, yielding more generic class definitions. For instance, a floor may have a certain number of hallways and offices depending on the building to which the floor belongs. This can be expressed formally in FoReST by associating *formal attributes* with a class definition, and by using them in statements.

2.5 Experiences

In several projects, with documentation in the order of one hundred pages of text, we have successfully applied the FoReST approach [8,7]. Here are some of the observations that we have made:

- During the formalization of the initial problem description, a large number of unclarities and ambiguities have been revealed and resolved. Without the obligation to specify requirements formally, most of these would have probably gone undetected in the requirements phase, leading to additional costs in later development stages.
- The application of a property-oriented technique to specify behavioral constraints has led to a first problem specification that has been quite close to the initial, informal problem description. This had a positive impact on traceability.
- The effort to modify the formal problem specification after a customer review has been small. The main reason for this can be seen in the clear relationship between informal and formal problem specification parts. This has made it nearly trivial to localize the specification parts affected by the customer review.
- In general, the class structure does not impose any restrictions on the design of the machine under development. However, it may well be chosen as the machine architecture that is further refined in subsequent phases. This choice has the advantage of good vertical traceability of design decisions.
- The classification of attributes and the distinction between domain, requirement, and machine statements has led to a practical criterion for the completeness of the problem specification.

3 Pattern-based Requirements Specification with FoReST

So far, reuse is based on components, i.e. class definitions that may be collected in packages. Also, classes may be specialized and parameterized, adding further to the reuse potential. In the following, we will extend the reuse potential of the requirements analysis phase and outline a pattern-based approach.

The degree to which the formalization of requirements can be achieved through pattern application depends both on the problem description and the contents of the pattern pool. If the structure of an informal specification follows a specification pattern that is already contained in the pool, then its formalization can be achieved directly by applying the pattern. If the structure is different, then either transformations and/or modifications of the informal specification may lead to a structure that is supported by the pattern pool, or the formalization has to be done conventionally. The pool of requirement patterns is evolving over time. As a consequence, the portion of

requirements that is developed from patterns will increase, reducing the overall effort of requirement specification and improving its quality.

In Section 2, we have used an object- and property-oriented style for the specification of system requirements. Please recall that we state properties as class features instead of listing operation signatures, as found in other approaches such as UML. System specifications are formally described using FoReST. In this section, we survey our pattern approach and combine it with the object-oriented problem specification of Section 2.

3.1 FoReST requirement patterns

To describe requirement patterns, we use the tabular format shown in Table 4, called *FoReST requirement pattern description template*. Instantiations of this template are termed *FoReST requirement patterns*, which, itself instantiated, form constituents of a requirement specification. To distinguish requirement patterns from other description elements such as class definitions, the keyword Requirement Pattern precedes the pattern name. The choice of a good pattern name is crucial, because it extends the vocabulary of the requirement engineer, raising it to a higher level. The entries of the template are explained in Table 4.

Requirement Pattern <*PATTERNNAME*>		
Intention	:	An informal description of the kind of problems addressed by this pattern.
Definition	:	A formal definition of the generic solution using a suitable FDT. Based on the formal definition and accompanying information, the pattern is selected, adapted, and composed into a context specification.
NL	:	A translation of the formal definition to natural language. This translation may serve as a starting point for deriving natural language statements from instantiated patterns.
Illustration	:	An illustration of the generic solution, supporting its intelligibility.
Example		An example from the application area illustrating the purpose of the pattern.
Semantic properties	:	Properties that have been formally proven from the formal definition. By instantiating these properties in the same way as the formal definition, proofs can be reused.

Table 4. FoReST requirement pattern description template

In Table 5, a requirement pattern called *WEAKDELAYEDIMPLICATION* is defined. It addresses situations where a causal relationship that is subject to certain timing constraints is given. These timing constraints are formally expressed in the definition,

which uses the operator $\Rightarrow_{\leq t}$ (*delayed implication*), a tailored operator of tRTTL. In effect, the discovery of this pattern, which can be syntactically defined as shown in the list of semantic properties, has caused us to define this operator, which is specific for tRTTL. The generic translation can be used for generating requirement documents for the customer, and has been used, for instance, to provide a translation for the example that forms part of the pattern definition. The pattern is applicable in all cases where a controlled phenomenon is required to hold only after a precondition holds for a certain amount of time, and thereafter only as long as the precondition holds. In the example, for instance, the windows have to be closed after the hazardous condition has continued to hold for at least 30 seconds. This avoids "fluttering" of the controlled phenomenon. Interestingly, the pattern also supports distributed implementations in several ways. Firstly, it is sufficient to sample the phenomenon at discrete points in time, so continuous observation is not necessary. Secondly, there is time for reaction concerning the controlled phenomenon, which may be exploited in associating priorities to phenomena that is used for configuring an underlying communication system.

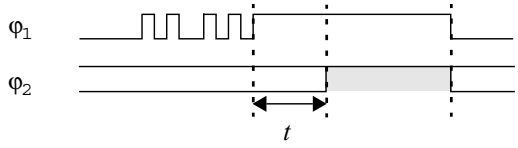
Requirement Pattern *WEAKDELAYEDIMPLICATION*	
Intention	: Phenomena may be in a causal relationship that is subject to certain timing constraints. The constraints state that the causal relationship only exists with a specified *delay*, and that the defining phenomenon has to hold during this time span (hence *weak*).
Definition	: $\Box\,(\varphi_1 \Rightarrow_{\leq t} \varphi_2)$
NL	: Whenever φ_1 holds for at least t, φ_2 is true within this t and then remains true at least as long as φ_1.
Illustration	: The diagram below shows a possible scenario for phenomena φ_1 and φ_2. In the shaded areas, the value of φ_2 is constrained by φ_1. 
Example	: $\Box$ (*hazardousCondition* $\Rightarrow_{\leq 30s}$ *windowsClosed*) Whenever a hazardous condition holds for at least *30s*, the windows are closed within these *30s* and remain closed at least as long as the hazardous condition is true.
Semantic properties	: $\Box\,(\varphi_1 \Rightarrow_{\leq t} \varphi_2) \leftrightarrow \Box\,(\varphi_1 \rightarrow \Diamond_{\leq t}(\varphi_2\ \boldsymbol{W}\ \varphi_1))$ $\Box\,(\varphi_1 \Rightarrow_{\leq t} \varphi_2) \wedge \Box\,(\varphi_2 \Rightarrow_{\leq t'} \varphi_3) \rightarrow \Box\,(\varphi_1 \Rightarrow_{\leq t+t'} \varphi_3)$ $\Box\,(\varphi_1 \Rightarrow_{\leq t} \varphi_2) \wedge \Box\,(\varphi_1 \Rightarrow_{\leq t} \varphi_3) \leftrightarrow \Box\,(\varphi_1 \Rightarrow_{\leq t} (\varphi_2 \wedge \varphi_3))$

Table 5. Pattern definition: *WEAKDELAYEDIMPLICATION*

The *WeakDelayedImplication* pattern has been applied many times in our projects in order to formalize statements following this pattern, and, among the patterns of our requirement pattern pool, has turned out to be the most useful one. In Tables 1 and 2, two applications of this pattern are shown. Note that the fact that a pattern has been applied to formalize these statements is not obvious from the specification itself. The knowledge that a particular pattern has been applied is a link to the additional documentation in the pattern definition.

3.2 Experiences

In several projects, we have successfully combined pattern-based requirements engineering with the FoReST approach [8,7]. Here are some of the pattern-related observations that we have made:

- Building a requirement pattern pool is a substantial investment that will only pay off through extensive use of the pool.
- Pattern application leads to an orthogonal structure of a specification that should be documented. Without this documentation and detailed knowledge of the pattern pool, it is very difficult to identify these parts afterwards.
- The knowledge that a particular pattern has been applied is a link to the additional documentation in the pattern definition.
- Requirement patterns provide a high-level vocabulary for the analysis team that contributes to making requirements specification an engineering task.
- Patterns support the uniform translation of formal statements to natural language, which is important if the customer has no mathematical background.
- The application of patterns during the requirements phase increases the reuse potential in subsequent development phases, called *vertical reuse*. For instance, design decisions can be coupled to the application of requirement patterns.

4 Conclusions

We have presented and applied several reuse approaches and their application to formally specify system requirements. In this paper, we have focused on reuse through object-oriented and pattern-based requirements engineering. To formally specify requirements, we have used the formal description technique FoReST, which supports an object- and property-oriented style. Examples have been drawn from the area of building automation systems, where we have conducted several projects to evaluate the feasibility of the approach. The complete problem specification of the light control system, for instance, contains a total of 44 class definitions with 162 predicates and functions, 87 domain statements, 65 requirement statements, and 22 machine statements, and covers a floor with 3 hallways and 22 rooms including offices and computer labs [8].

Requirement specifications have been engineered using a pattern-based approach, applying requirement patterns of a pattern pool. From the 174 statements of the final specification, 125 have directly resulted from the application of 5 different FoReST

requirement patterns [8]. The pattern-based approach has reduced the overall effort to formalize the user needs, and has improved the quality of the problem specification. In particular, experience of former projects has been made reusable, upgrading the work of the analysis team and contributing to making it an engineering approach. Finally, pattern-based requirements engineering prepares the ground for vertical reuse.

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. At the moment, the pool of requirement patterns only has a small number of elements, which keeps the reuse potential within certain bounds. We expect that with increasing experience and further projects, additional patterns will be discovered and added to the requirement pattern pool. In fact, we perceive pattern engineering as a fertile field of future research.

References

[1] G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999

[2] E. Cusack: *Refinement, Conformance and Inheritance*, Formal Aspects of Computing, 1991, No. 3, pp. 129-141

[3] M. E. Fayad, D. C. Schmidt: *Object-Oriented Application Frameworks* (Special Issue), Communications of the ACM, Vol. 40, No. 10, 1997, pp. 32-38

[4] R. Gotzhein, M. Kronenburg, C. Peper: *Reuse in Requirements Engineering*: *Discovery and Application of a Real-Time Requirement Pattern*, 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98), Lyngby, Denmark, 1998, pp. 65-74

[5] R. E. Johnson: *Frameworks = (Components + Patterns)*, Communications of the ACM, Vol. 40, No. 10 (October 1997), pp. 39-42

[6] M. Kronenburg, C. Peper: *Definition and Instantiation of a Reference Model for Problem Specifications*, 11th International Conference on Software Engineering and Knowledge Engineering (SEKE'99), Kaiserslautern, Germany, 1999

[7] M. Kronenburg, C. Peper: *Specification of the Case Study "Light Control System"*, Dagstuhl-Seminar on Requirements Capture, Documentation, and Validation, June 13-18, 1999, http:// rn.informatik.uni-kl.de/~forest/examples

[8] M. Kronenburg, C. Peper: *Application of the FoReST Approach to the Light Control Case Study*, in: Journal of Universal Computer Science, Special Issue on Requirements Engineering 6(7), pp. 679-703, Springer, 2000

[9] M. Kronenburg: *An Approach to the Creation of Precise, Intelligible Problem Specifications of Large Reactive Systems*, Ph.D. thesis, Shaker Verlag, Aachen, 2001

[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] J. Rumbaugh et. al.: *Object-Oriented Modeling and Design*, Prentice Hall, 1991