Vertical Reuse in the Development of Distributed Systems with FDTs[1]

Reinhard Gotzhein

Computer Science Department, University of Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern, Germany
gotzhein@informatik.uni-kl.de

Abstract. The development and maintenance of large distributed software systems is intrinsically difficult and continues to worry generations of software engineers in academia and industry. Several key approaches to mastering these difficulties have been identified, including *structuring* and *reuse*. System structuring is essential to controlling complexity, and is a prerequisite for the extraction of reuse artifacts. Reuse of solutions is crucial to controlling quality and productivity. Previous work has addressed *horizontal reuse*, i.e., reuse within a single development phase. In this paper, *vertical reuse*, i.e., reuse across development phases, is introduced, focusing on the early development phases. Formal description techniques (FDTs) are applied to define reuse artifacts. Examples are drawn from the building automation domain.

1 Introduction

Reuse of solutions and experience for recurring software system development problems plays a key role for quality improvements and productivity increases. As a prerequisite, the problems and their solutions have to be in some sense "similar". These similarities should not be understood as purely syntactical, rather, semantical and conceptual similarities should be considered as well, which requires precise domain knowledge and conceptual thinking. Reuse has been studied thoroughly in software engineering, which has led to the distinction of three main reuse concepts [15]:

- *Components* are often characterized as self-contained, ready-to-use building blocks, which are selected from a component library and composed.
- A *framework* is the skeleton of a system, to be adapted by the system developer.
- *Patterns* describe generic solutions for recurring problems, which are to be customized for a particular, embedding context.

It should be emphasized that these reuse concepts can be applied together, for instance, by defining a component framework such as CORBA and adding components, or by using patterns to define components used in a component framework.

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

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 31–47, 2003.
© IFIP International Federation for Information Processing 2003

Each reuse concept is typically associated with a particular development phase. For instance, components are usually applied in the implementation phase, and patterns are related to the design phase. This, however, results from practical experience rather than from existing limitations. There is, for instance, no reason why the pattern idea can not be applied during requirements analysis. In [19], we have introduced the notion of *requirement patterns*, and have applied them successfully in the area of building automation systems [8,17]. Similar observations hold for frameworks and components.

In previous work, we have addressed *horizontal reuse*, i.e., reuse within a single development phase [4,19,10]. In this paper, *vertical reuse*, i.e., reuse across development phases, is introduced. To achieve maximum benefits, we focus on the early development phases and address vertical reuse from the requirements phase to the design phase. More specifically, we exploit the pattern idea: starting from a set of FoReST requirement patterns, we develop corresponding SDL design patterns defining generic design solutions. Formal description techniques are applied to define reuse artifacts. Examples are drawn from the building automation domain.

The paper is structured in the following way. In Section 2, we elaborate on structuring large software systems in general, which is a prerequisite for achieving a significant degree of reusability. Furthermore, we survey two reuse approaches that will be used to establish vertical reuse. In Section 3, we describe vertical reuse of system architecture specifications. Section 4 introduces vertical reuse of system behaviour specifications. Section 5 presents conclusions.

2 Reuse in Distributed Systems Engineering

In this section, we first describe different ways of structuring distributed systems. Structuring is important for controlling complexity on the one hand, and a prerequisite for extracting reuse artifacts on the other hand. We then survey two specific reuse approaches that will be integrated and extended to provide vertical reuse, namely FoReST requirement patterns and SDL design patterns.

2.1 Structuring of Distributed Systems

Large software systems exhibit a variety of structures, depending on the type of system, the degree of abstraction, the development paradigm, and the developers' viewpoints. We can distinguish between structuring in the large, which focuses on system architecture, and structuring in the small, where behaviour and data of system parts are decomposed. Structuring principles include module structuring (e.g., agent modules, object modules, collaboration modules, functional modules), hierarchical structuring (e.g., agent hierarchies, class hierarchies, state hierarchies), conceptual structuring (e.g., reference architectures), dynamic structuring (e.g., creation and termination of process modules, interaction relationships), and physical structuring (e.g., nodes, resources, topology).

In general, a software system can be structured from different perspectives and in many different ways. To structure *distributed systems*, a particular type of software systems that includes distributed applications and communication protocols, the following structuring principles are of specific interest:

- *Agent modules.* An agent is a system unit that exhibits a behaviour and interacts with other systems agents. Agent modules are typically described in a self-contained way, and can be composed by adding interaction channels.
- *Functional modules.* A functionality is a single aspect of internal system behavior that may be distributed among a set of system agents, with causality relationships between single events.
- *Collaboration modules.* A collaboration is a system unit that captures a distributed functionality together with the required interaction behaviour. Collaboration modules can be composed by adding synchronisation and causality relationships.
- *Hierarchical structuring*. Large systems are often decomposed in subsequent steps, leading to a hierarchical system structure. The external appearance of a system unit is then obtained from the composition of its parts. On each level, a different module structure may be chosen. For instance, once a system is decomposed into agents, their behaviour may in turn be decomposed using state hierarchies as well as compound statements.

Agent modules and collaboration modules can be seen as orthogonal structuring principles, capturing different system views, possibly on the same level of abstraction. In addition, hierarchical structuring can be applied. Agent modules and their composition can be specified, e.g., with UML statecharts [1] or SDL [13]. MSC [14], UML sequence diagrams [1], and UML collaboration diagrams [1] support the description of collaboration modules.

2.2 The FoReST Requirement Pattern Approach

The earlier in the development process reuse is achieved, the larger its positive impact on the project. Following this observation, we have introduced *FoReST*, the *Formal Requirement Specification Technique*, a component and pattern approach for horizontal reuse in the requirements phase [17]. With FoReST, system requirements are specified in an object- and property-oriented style, covering both the architecture and the behaviour of a system.

The FoReST approach consists of a *pattern-based requirements analysis process*, a *template* and *rules* for the definition of FoReST requirement patterns, a *requirement pattern pool*, and *language support*. The approach has been successfully applied to the formalization of problem descriptions in the building automation domain, e.g., a light control [17] and a heating control with requirement specifications of 60 and 90 pages, respectively, and in the SILICON case study [9].

FoReST requirement patterns describe generic formalizations for recurring requirements and capture experience gained in the requirements analysis of previous system developments. In [8], a complete pattern discovery process for a non-trivial requirement pattern is documented. In [17], the degree of reuse is increased by incor-

porating object-oriented concepts (class definitions, specialization) and parameterization. The FoReST-approach has been formalized in [18].

Application of FoReST requirement patterns means that they are selected from the pattern pool, adapted and composed into a context specification. The pattern pool can be seen as a repository of experience from previous projects that has been analyzed and packaged. The FoReST requirement patterns we have identified so far can be classified into three categories:

- *Architecture patterns* capture generic architectures and their refinements.
  Example: COMPOSITION (Section 3.1). This pattern captures hierarchical architectures, consisting of a composite and its constituents.
- *Behaviour patterns* capture the causal relationships between phenomena of active system components.
  Example: WEAKDELAYEDIMPLICATION [19]. This pattern captures timing constraints between two phenomena stating that a causal relationship only exists with a specified delay, and that the defining phenomenon has to hold during this time span.
- *Phenomena patterns* capture the results of refining predicates and functions.
  Example: LAZYREACTION [8]. This pattern captures the result of refining a timed predicate such that the satisfaction of the predicate depends on a number of temporal constraints.

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.

2.3 The SDL Design Pattern Approach

Design patterns [3] are a well-known approach for the reuse of design decisions. In [4], another specialization of the design pattern concept for the development of distributed systems and communication protocols, called SDL design patterns, has been introduced. *SDL design patterns* combine the traditional advantages of design patterns – reduced development effort, quality improvements, and orthogonal documentation – with the precision of a formal design language for pattern definition and pattern application.

The SDL design pattern approach [7,10] consists of a *pattern-based d*esign process, a notation for the description of generic SDL fragments called *PA-SDL* (Pattern Annotated SDL), a *template* and *rules* for the definition of SDL design patterns, and an *SDL design pattern pool*. The approach has been applied successfully to the engineering and reengineering of several distributed applications and communication protocols, including the SILICON case study [9], the Internet Stream Protocol ST2+, and a quality-of-service management and application functionality for CAN (Controller Area Network) [5]. Applications in industry, e.g., in UMTS Radio Network Controller call processing development, are in progress [11].

An *SDL design pattern* [4,7] is a reusable software artifact that represents a generic solution for a recurring design problem with *SDL* [13] as design language. Over a period of more than 25 years, SDL (System Design Language) has matured from a sim-

ple graphical notation for describing a set of asynchronously communicating finite state machines to a sophisticated specification technique with graphical syntax, data type constructs, structuring mechanisms, object-oriented features, support for reuse, companion notations, tool environments, and a formal semantics. These language features and the availability of excellent commercial tool environments are the primary reasons why SDL is one of the few FDTs that are widely used in industry.

When SDL patterns are applied, they are selected from a pattern pool, adapted and composed into an embedding context. The pattern pool can be seen as a repository of experience from previous projects that has been analyzed and packaged. The SDL patterns we have identified so far can be classified into five categories:

- *Architecture patterns* capture generic architectures and their refinements.
  Example: ClientServer [11]. This pattern captures a client/server architecture of a distributed system.
- *Interaction patterns* capture the interaction among peers, e.g., a set of application agents or service users.
  Example: SynchronousInquiry [10]. This pattern introduces a confirmed interaction between two peers. After a trigger from the embedding context, an agent sends an inquiry and is blocked until receiving a response from the second agent.
- *Control patterns* deal with the detection and handling of errors that may result from loss, delay, or corruption of messages, or from agent failures.
  Example: LossControl [10]. This pattern provides a generic solution for the detection and handling of message loss in the case of confirmed interactions, such as synchronous inquiries. If a response does not arrive before the expiry of a timer, the message is repeated (Positive Acknowledgement with Retransmission).
- *Management patterns* deal with local management issues, such as buffer creation or message addressing[2].
  Example: BufferManagement [12]. When a signal is passed between two local processes, the signal parameters are stored into a buffer, and a buffer reference is sent. This technique has an impact on implementation efficiency, it reduces memory consumption and copying overhead.
- *Interfacing patterns* replace the interaction between peers by interaction through a basic service provider. This may include segmentation and reassembly, lower layer connection management, and routing.
  Example: CoDex [2]. This pattern provides a generic solution for encoding service data units (SDUs) and interface control information into protocol data units, the exchange of PDUs among specific protocol entities, and the decoding of SDUs.

The definition of SDL design patterns supports their selection during the protocol design. As the result of the object-oriented analysis of requirements, an analysis model consisting of a UML object diagram and MSC message scenarios are built. Comparing the structure and the message scenarios of SDL design patterns against this analy-

[2] It can be argued that these management patterns are rather low-level, as compared to the other examples. However, they have been discovered in an industrial cooperation, and capture realistic design decisions that lead to the generation of more efficient code. Furthermore, application of these patterns signficantly reduces the number of design errors [12].

sis model strongly supports the selection of suitable patterns [10]. As the number of patterns in a typical pattern pool (see also [3]) is relatively small (10-30 patterns[3]), and with additional information contained in the pattern pool, for instance, on cooperative usage, this should be sufficient for a proper selection.

3 Vertical Reuse of System Architecture

In this section, we introduce the pattern description templates used to define FoReST requirement patterns and SDL design patterns, and instantiate them with architecture patterns, i.e., patterns that capture generic architectures and their refinements. Furthermore, we argue that the SDL design pattern solves the problem stated by the corresponding FoReST requirement pattern, which enables vertical reuse. The choice of patterns in this section will be complemented by behaviour patterns in Section 4 such that architecture and behaviour patterns can be applied to form a chain of related pattern applications on different levels of abstraction. All patterns have been obtained from analysing and packaging project experience [9].

3.1 FoReST Architecture Patterns

To define FoReST requirement patterns, we use the tabular format shown in Table 1, called *FoReST requirement pattern description template*. Instantiations of this template are termed *FoReST requirement patterns*, which, itself instantiated, form fragments of a requirement specification. The entries of the template are explained in Table 1.

In Table 2, the FoReST requirement pattern COMPOSITION is defined. COMPOSITION classifies as an architectural pattern, i.e., a pattern that captures a generic architecture and/or its refinement. In this particular case, hierarchical architectures, consisting of a composite and its constituents, are described in a generic way. Though this pattern is very simple, it fits nicely with the subsequent, more complex patterns in that they can be applied to form a chain of related pattern applications on different levels of abstraction. The pattern definition is given in tabular format, following the FoReST requirement pattern description template of Table 1.

The syntactical part of the solution is shown in entry `Definition`, represented in the syntax of FoReST class definitions. With FoReST, classes are defined in a tabular format by specifying a unique *class name*, a *signature*, and a *behaviour*, using appropriate keywords to distinguish specification items:

---

[3] These figures result from practical experience. They differ substantially from the size of typical component repositories with 100s of elements. The relatively small number can be explained by the generic nature of patterns. Also, as the definition of “good” patterns is a substantial investment, only those patterns that are frequently applied should be included in the pattern pool.

Table 1. FoReST requirement pattern description template

FoReST Requirement Pattern <PatternName>		
Intention	:	An informal description of the kind of problems addressed by this pattern.
Definition	:	A formal definition of the generic solution. Based on the formal definition and accompanying information, the pattern is selected, adapted, and composed into a context specification.
Natural Language	:	A uniform translation of the formal definition to natural language.
Illustration	:	An illustration of the generic solution, supporting its intelligibility.
Example	:	An example from the application area illustrating the purpose and the usage 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 2. FoReST requirement pattern Composition (excerpt)

FoReST Requirement Pattern Composition		
Intention	:	Composition is defined in a generic way.
Definition	:	**Class Composite** **Signature** (**Object** c: Component)$^+$
Natural Language	:	Elements of class Composite are defined to consist of objects $c_{1..n}$ of classes $Component_{1..n}$, respectively.
Illustration	:	Composite $Component_1$ $Component_2$ ... $Component_n$
Example	:	**Class Room** **Signature** **Object** md: MotionDetector **Object** la: LightActuator **Object** ts: TemperatureSensor

- 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 (see Table 2). Furthermore, predicates and functions may be static or dynamic (timed), stating whether their values may vary over time. Also, specialization and inheritance are supported [17].
- The behaviour of a class is specified by a set of properties. According to the product reference model in [16], we distinguish between different kinds of behaviour statements, namely *domain*, *requirement* and *machine statements*. A domain statement describes pre-installed devices and/or existing environment behaviour. A requirement statement addresses the desired system behaviour. Finally, a machine statement characterizes behaviour of the machine, i.e., the part that is to be combined with the environment to achieve the desired system behaviour. 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* [8].

Composition is directly supported by the concept of composite classes: in FoReST, a composite class is defined by specifying, for each constituent, an attribute that is classified as *object*, and is associated with a class (see Table 2). Creation of a composite object always implies the instantiation of its constituents. This situation is graphically captured using UML notation in entry `Illustration`. Finally, an excerpt of the class definition `Room`, where the pattern has been applied, is shown.

3.2 SDL Architecture Patterns

To define SDL design patterns, we use the format shown in Table 3, called *SDL design pattern description template*. Instantiations of this template are termed *SDL design patterns*, which, itself instantiated, form fragments of a design specification. The entries of the template are explained in Table 3.

In Figure 1, the *SDL fragment* of the SDL design pattern BUILDINGCOMPOSITION is shown. BUILDINGCOMPOSITION classifies as an architectural pattern, i.e., a pattern that captures a generic architecture and/or its refinement. In this particular case, a building domain specific hierarchical architecture, consisting of a component, its constituents, a control cell, interaction points between the control cell and all constituents, and the required connection structure, is described in a generic way.

To define the SDL fragment, the pattern definition language PA-SDL (Pattern Annotated SDL), a pattern-specific extension of SDL, is used. With PA-SDL, the context where the pattern may be applied, the permitted adaptations, and the embedding into the context specification can be described. For instance, the pattern BUILDINGCOMPOSITION introduces new design elements that are added to the context specification as the result of the pattern application. SBU (Structural Block Unit) denotes a structural SDL unit, a system or a block, SU (Structural Unit) allows for processes and services. These choices are further constrained by the syntax of SDL.

Table 3. SDL design pattern description template

<PatternName> [*Each pattern is identified by a pattern name, which serves as a handle to describe a design problem, its solution, and its consequences.*]
Intention [*provides an informal description of the design problem and its solution.*] Motivation [*gives an example for the pattern usage without relying on the pattern definition.*] Structure [*is a graphical representation of the involved design components and their relations. Structural aspects before and after the application of the pattern are covered.*] Message Scenarios [*illustrate typical behaviour related to this pattern and thus complement the structural aspects.*] SDL Fragment [*describes the syntactical part of the design solution, which is adapted and composed when the pattern is applied. The notation used here is called PA-SDL (Pattern Annotated SDL). It defines the context in which the pattern is applicable, the permitted adaptations, and the embedding into the context specification.*] Syntactical Embedding Rules [*constrain the application of the pattern such that certain desirable properties are added or preserved.*] Semantic Properties [*resulting from the correct application of the pattern.*] Refinement [*states rules for further redefining an applied pattern.*] Cooperative usage [*describes the usage together with other patterns of the pool.*] Known Uses [*documents where the pattern has been applied so far.*]

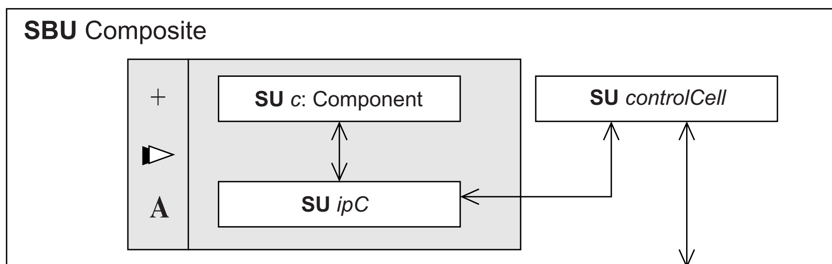
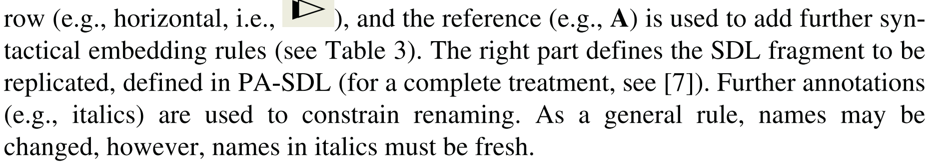


Fig. 1. SDL design pattern BuildingComposition (SDL fragment, excerpt)

The shaded part called *border symbol* is an annotation denoting replications and consists of two parts. The left part defines replication parameters: the number of replications is specified by the multiplicity (e.g., +), the direction of is given by the ar-



row (e.g., horizontal, i.e., ▷), and the reference (e.g., **A**) is used to add further syntactical embedding rules (see Table 3). The right part defines the SDL fragment to be replicated, defined in PA-SDL (for a complete treatment, see [7]). Further annotations (e.g., italics) are used to constrain renaming. As a general rule, names may be changed, however, names in italics must be fresh.

The SDL fragment shown in Figure 1 defines SDL structures consisting of a single *controlCell* and one or more components *c* with an associated interaction point *ipC*. Furthermore, SDL channels are introduced as shown. Components can exchange messages with *controlCell* via their interaction points. Furthermore, *controlCell* can interact with the environment of structuring unit Composite.

3.3 Vertical Reuse

The architectural patterns defined in Sections 3.1 and 3.2 support *horizontal reuse*, i.e., reuse within a single development phase. For instance, if a hierarchical architecture is derived from the problem description, application of the Composition requirement pattern yields a suitable formalization. In a similar way, if a hierarchical building topology is designed, application of the BuildingComposition pattern leads to a suitable SDL design. This means that FoReST requirement patterns as well as SDL design patterns can be used “stand alone”.

In order to further enhance the benefits of pattern-based reuse, both approaches can be coupled, leading to vertical reuse, i.e., reuse across development phases. More specifically, starting from a FoReST requirement pattern, we can develop corresponding SDL design patterns defining generic design solutions. Obviously, the BuildingComposition pattern has been defined with this objective in mind: it defines one domain-specific design solution for the Composition requirement pattern. This means that for each application of the Composition pattern in the building automation domain, a suitable design solution can directly be derived from its instantiation and its specification context.

The architecture defined by the BuildingComposition pattern prepares the distributed implementation of properties associated with instances of the class `Composite`. While all components introduced by Composition are represented as structural SDL units (with identical names to enhance traceability), further design components are added. In particular, a control cell that coordinates the behaviour of the components such that the properties of `Composite` are satisfied is added. Furthermore, interaction points between the control cell and all constituents and the required connection structure including a channel to the context of `Composite` (the pattern may be applied recursively) are introduced. Thus, while being on a lower level of abstraction, the design preserves the structural properties of the requirement level, which supports traceability.

Note that the BuildingComposition pattern respects the hierarchical structure established by Composition. This is a deliberate choice at this stage that needs reconsideration when the implementation design is derived. In fact, the structure is later transformed into a layered architecture, consisting of an application layer, a communi-

cation middleware and basic technology. Interestingly, this transformation can be achieved without modifying the behaviour of active system components.

The BUILDINGCOMPOSITION pattern is the result of analysing and packaging experience gained in the SILICON case study [9], where a distributed interactive light control for a building model has been developed from scratch. Application of this pattern in conjunction with the COMPOSITION pattern improves traceability between development phases, and documents design decisions.

4 Vertical Reuse of System Behaviour

In this section, we introduce patterns capturing the behaviour of system components. Again, we argue that the SDL design pattern solves the problem stated by the corresponding FoReST requirement pattern, which enables vertical reuse.

4.1 FoReST Behaviour Patterns

In Table 4, the FoReST requirement pattern WEAKDELAYEDIMPLICATION that classifies as a behaviour pattern is defined. The pattern definition is given in tabular format, following the FoReST requirement pattern description template of Table 1. WEAKDELAYEDIMPLICATION addresses situations where a causal relationship between phenomena 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 [8].

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 continues to hold. In the example in Table 4, for instance, whenever a room is used for at least 2 seconds, the light is switched on within this time span and remains on at least as long the room is used. This avoids “fluttering” of the controlled phenomenon, as illustrated in the pattern definition. Also note that the phenomena may be associated with different system components. In the example, `roomUsed` and `on` are attributes of components `md` and `la`, respectively (cf. `Example` in Table 2).

Interestingly, the WEAKDELAYEDIMPLICATION 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 are used for configuring an underlying communication system.

The WEAKDELAYEDIMPLICATION pattern has been applied many times in our projects in order to formalize statements of the problem description, and, among the patterns of our requirement pattern pool, has turned out to be the most useful one. Note that the fact that a pattern has been applied to formalize a statement is not obvious from

Table 4. FoReST requirement pattern WeakDelayedImplication (excerpt)

FoReST Requirement Pattern WeakDelayedImplication		
Intention	:	Phenomena may be in a causal relationship that only exists with a specified *delay*, where the the defining phenomenon has to hold during this time span (hence *weak*).
Definition	:	$\Box (\varphi \Rightarrow_{\leq t} \psi)$
Natural Language	:	Whenever φ holds for at least *t*, ψ is true within this *t* and then remains true at least as long as φ.
Illustration	:	The diagram below shows a possible scenario for phenomena φ and ψ. In the shaded areas, the value of ψ is constrained by φ.
Example	:	$\Box$ (*md.roomUsed* $\Rightarrow_{\leq 2s}$ *la.on*) Whenever a room is used for at least 2 seconds, the light is on within this time span and remains on at least as long as the room is used.
Semantic properties	:	$\Box (\varphi_1 \Rightarrow_{\leq t} \varphi_2) \leftrightarrow \Box (\varphi_1 \rightarrow \Diamond_{\leq t}(\varphi_2 \; \mathcal{W} \; \varphi_1))$ $\Box (\varphi_1 \Rightarrow_{\leq t} \varphi_2) \wedge \Box (\varphi_2 \Rightarrow_{\leq t'} \varphi_3) \rightarrow \Box (\varphi_1 \Rightarrow_{\leq t+t'} \varphi_3)$ $\Box (\varphi_1 \Rightarrow_{\leq t} \varphi_2) \wedge \Box (\varphi_1 \Rightarrow_{\leq t} \varphi_3) \leftrightarrow \Box (\varphi_1 \Rightarrow_{\leq t} (\varphi_2 \wedge \varphi_3))$

the specification itself. The knowledge that a particular pattern has been applied is a link to the additional documentation in the pattern definition.

4.2 SDL Behaviour Patterns

In Figure 2, an excerpt of the *SDL fragment* of the SDL design pattern DistributedControl is shown. DistributedControl classifies as an interaction pattern, i.e., a pattern capturing interaction between active system components. In this particular pattern, a specific behaviour establishing a delayed causal relationship between two phenomena in a distributed environment is defined.

The SDL fragment defines three active design elements, represented as extended finite state machines (EFSMs): phiSource, psiSink, and controlCell. They have to be part of the context specification before the pattern can be applied, which is expressed by the dashed frame symbols in the SDL fragment. In general, dashed symbols are annotations of PA-SDL denoting design elements (e.g., structural units, triggers, actions) that are part of the context, while solid symbols denote design elements that are added as a result of the pattern application. The effect of a pattern application can be

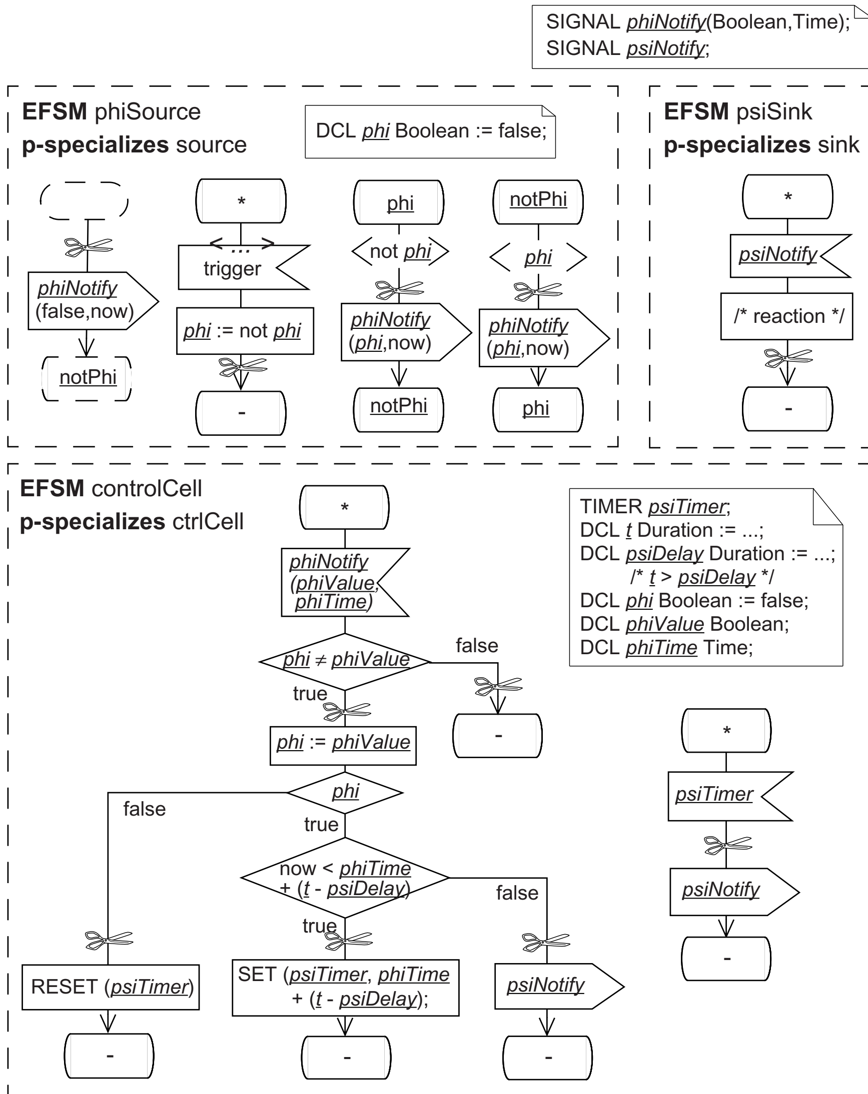


Fig. 2. SDL design pattern DistributedControl (SDL fragment, excerpt)

characterized as a pattern-specific specialization (p-specialization, see [6]), where the context specification is extended and/or redefined.

Further annotations apply to names. As a general rule, names may be changed. However, names in italics must be fresh, and if underlined, renamed in a unique way when adapting the pattern. For instance, *phiNotify* declared in the text symbol and then used in several input and output symbols is a new signal.

In the pattern definition, some further annotations of PA-SDL are used. The generic trigger symbol may be adapted to an input, priority input, or spontaneous input

symbol, a continuous signal, or an enabling condition. Scissor symbols indicate the possibility of refinements, for instance, by adding further actions to a transition, without disrupting the control flow.

By applying the DISTRIBUTEDCONTROL pattern, the behaviour of the active design elements is extended such that phiSource communicates state changes of a phenomenon phi to controlCell, which in turn decides whether and when to notify psiSink and thus to trigger a certain reaction:

- When phiSource detects a state change, it sends a signal phiNotify that carries the current value of phi as well as a time stamp.
- On receipt of this message, controlCell determines whether appropriate action has to be taken.
- Depending on the urgency of the action, which is determined on the basis of the time stamp and a maximum delay psiDelay, a timer psiTimer may be set such that after its expiry, there is sufficient time for exchanging a signal psiNotify with psiSink, and for the following reaction.
- Depending on the state changes of phi, the timer may also be reset before expiry.

4.3 Vertical Reuse

The behaviour patterns defined in Sections 4.1 and 4.2 support *horizontal reuse*, i.e., reuse within a single development phase. As already stated, this means that FoReST requirement patterns as well as SDL design patterns can be used “stand alone”. However, both approaches can be coupled, leading to vertical reuse. Here, starting from a FoReST requirement pattern, we can develop corresponding SDL design patterns defining generic design solutions. Following this idea, the DISTRIBUTEDCONTROL design pattern defines a domain-specific design solution for the WEAKDELAYEDIMPLICATION requirement pattern. This means that for each application of the WEAKDELAYEDIMPLICATION pattern in the building automation domain, a suitable design solution can directly be derived from its instantiation and its specification context.

As observed in Section 4.1, the WEAKDELAYEDIMPLICATION pattern supports distributed implementations. In particular, a reaction on state changes of the phenomenon φ is not required to be immediate, but may occur with a specified delay, and may depend on the “continuity” of φ. This observation is exploited in the generic design solution[4]:

- The generic design solution identifies cooperating active components and adds local functionality and collaborations such that the tRTTL property is satisfied.
- The controlling component receives all updates about the phenomenon phi and decides about further measures. A reaction concerning the phenomenon phi is delayed until the latest possible point in time, to avoid “fluttering”.
- The components phiSource and psiSink, which can be viewed as a sensor (e.g., a motion detector) and an actuator (e.g., a light group), remain independent, which is a good design choice in general.

[4] Strictly speaking, a generic design solution is intended to define a generic model that satisfies the requirement pattern.

The generic design solution introduced by DistributedControl is based on the architecture defined by BuildingComposition: phiSource and psiSink are among the set of components *c* and coordinated through *controlCell*. Thus, the chain of pattern applications in the requirements analysis has a counterpart in the design phase.

The reader may have noted that the real time expressiveness of FoReST and SDL is different. While it is possible to state maximum reaction times in tRTTL, only bounded omissions can be expressed with SDL. Strictly speaking, the design solution is therefore not precise. We therefore require the SDL timer mechanism be used with an expiry time that is derived from a worst-case estimate.

The DistributedControl pattern is the result of analysing and packaging experience gained in the SILICON case study, where a distributed interactive light control for a building model has been developed from scratch [9]. Application of this pattern in conjunction with the WeakDelayedImplication pattern improves traceability between development phases, and documents design decisions.

5 Conclusions

We have presented pattern-based reuse approaches for the requirements and the design phase, and have shown how they can be integrated to support *vertical reuse*, i.e., reuse across development phases. To enable vertical reuse, for each FoReST requirement pattern, one or more domain-specific SDL design patterns representing generic design solutions are specified. This way, for each application of a FoReST requirement pattern, a suitable design solution can be directly obtained by applying the corresponding SDL design pattern. We have exemplified these ideas by two pairs of related patterns to capture system architecture and system behaviour, respectively.

All patterns shown in this paper have been obtained from analysing and packaging project experience. They have been chosen from different categories in order to illustrate both architectural and behavioural aspects. Also, they have been chosen to form a chain of related pattern applications on different levels of abstraction.

To define patterns, we have applied formal description techniques, which has the advantage of making the pattern selection, adaptation, and composition more precise, and of improving traceability between documents of different development phases. However, although both FoReST and SDL have a formal semantics, there is no formalized relationship between corresponding requirement and design patterns. While it may be feasible to establish such a relationship between complete FoReST and SDL specifications, it is extremely difficult to define it between incomplete specification fragments, which in fact is the situation for related patterns. We are not aware of any research in this direction.

Our choice of FDTs - FoReST and SDL - has been influenced by the structure and style of the problem description as well as by our intention to develop distributed solutions. While FoReST is very close to the customer requirements, SDL is an appropriate language for distributed systems design and widely used in industry. Also, the SDL application design has turned out to be a good starting point for the development

of a customized communication system. Of course, the principles of horizontal and vertical reuse are not restricted to these languages.

References

1. G. Booch, J. Rumbaugh, I. Jacobsen: The Unified Modelling Language User Guide, Addison-Wesley, 1999
2. Computer Networks Group: The SDL Pattern Pool, Online document, University of Kaiserslautern, Kaiserslautern, Germany, 2002 (available on request)
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, 1995
4. B. Geppert, R. Gotzhein, F. Rößler: Configuring Communication Protocols Using SDL Patterns, in: A. Cavalli, A. Sarma (eds.), SDL'97 - Time for Testing, Proceedings of the 8th SDL Forum, Elsevier, Amsterdam, 1997, pp. 523-538
5. B. Geppert, A. Kühlmeyer, F. Rößler, M. Schneider: SDL-Pattern based Development of a Communication Subsystem for CAN, in : S. Budkowski, A. Cavalli, E. Najm (eds.), Formal Description Techniques and Protocol Specification, Testing, adn Verification, Proceedings of FORTE/PSTV'99, Kluwer Academic Publishers, Boston, 1998, pp. 197-212
6. B. Geppert, F. Rößler, R. Gotzhein: Pattern Application vs. Inheritance in SDL, 3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), Florenz, Italien, Kluwer Academic Publishers, 1999
7. B. Geppert: The SDL-Pattern Approach - A Reuse-Driven SDL Methodology for Designing Communication Software Systems, PhD Thesis, Kaiserslautern Univ., Germany, 2000
8. R. Gotzhein, M. Kronenburg, C. Peper: Reuse in Requirements Engineering: Discovery and Application of a Real-Time Requirements Pattern, 5th Intern. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98), Lyngby, Denmark, 1998
9. R. Gotzhein, C. Peper, P. Schaible, J. Thees: SILICON - System Development for an Interactive LIght CONtrol, URL: http://rn.informatik.uni-kl.de/projects/sfb501/b4/activities/casestud/silicon/, 2001
10. R. Gotzhein: Consolidating and Applying the SDL-Pattern Approach: A Detailed Case Study, Information and Software Technology, Elsevier Sciences (in print)
11. R. Grammes, R. Gotzhein, C. Mahr, P. Schaible, H. Schleiffer: Industrial Application of the SDL- Pattern Approach in UMTS Call Processing Development - Experience and Quantitative Assessment, 11th SDL Forum (SDL'2003), Stuttgart/Germany, July 1-4, 2003
12. R. Grammes: Evaluation and Application of the SDL Pattern Approach, Master Thesis, Computer Networks Group, Univ. of Kaiserslautern, Germany, February 2003
13. ITU-T Recommendation Z.100 (11/99) - Specification and Description Language (SDL), International Telecommunication Union (ITU), 2000
14. ITU-T Recommendation Z.120 (11/99) - Message Sequence Chart (MSC), Intern. Telecommunication Union (ITU), 2000
15. R. E. Johnson: Frameworks = (Components + Patterns), in: Object-Oriented Application Frameworks (Special Issue), Comm. of the ACM, Vol. 40, No. 10, 1997, pp. 39-42
16. 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

17. 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
18. M. Kronenburg: An Approach to the Creation of Precise, Intelligible Problem Specifications of Large Reactive Systems, PhD thesis, Shaker Verlag, Aachen, 2001
19. C. Peper, R. Gotzhein, M. Kronenburg: A Generic Approach to the Formal Specification of Requirements, First IEEE International Conference on Formal Engineering Methods (ICFEM'97), Hiroshima, Japan, 1997