Developing Reliable Systems with SDL Design Patterns and Design Components

Christian Webel, Ingmar Fliege, Alexander Geraldy, Reinhard Gotzhein

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

Abstract. SDL is a system design language that is being promoted for the development of reliable systems. In this paper, we apply SDL to capture design solutions to well-known mechanisms found in reliable systems - a watchdog and a heartbeat - for reuse. In particular, we present a methodology to augment system reliability step-by-step, and define and apply generic design solutions for reliable systems expressed as SDL design patterns and design components. These solutions can be integrated into an existing system design, to protect against certain types of system failures. We illustrate the approach by an application to a remote airship flight control over WLAN.

Keywords: distributed systems engineering, SDL, reuse, design pattern, reliability

1 Introduction

Reuse has been thoroughly studied in software engineering, and has led to the distinction of three main reuse concepts: components, frameworks, and patterns. Components can be 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, to be customized for a particular context.

To identify and extract reuse artifacts, i.e. components, frameworks, and patterns, the structuring of a software system plays a key role. Software systems may have a variety of structures, depending, for instance, on the type of system, the degree of abstraction, the development paradigm, and the developers' view points.

Design patterns are a well-known approach for the reuse of design decisions. In [1], a specialization of the design pattern concept for the development of communication protocols, called SDL patterns, has been introduced. SDL patterns combine the traditional advantages of design patterns - reduced development effort, quality improvement, and orthogonal documentation - with the precision of a formal design language for pattern definition and pattern application.

In previous work, we have identified and applied the structuring unit micro protocol, i.e. a communication protocol with a single (distributed) functionality and the required protocol collaboration [2][3]. A functionality (e.g., flow control) is a single aspect of internal system behavior that may be distributed among a set of system agents, with causality relationships between single events. By collaboration, we refer to the interaction behavior of a distributed functionality. From a reuse viewpoint, micro protocols classify as components, they may be selected from a micro

protocol library, and composed.

In this paper, we present an approach to augment reliability of an existing system by applying SDL design patterns and using SDL design components (micro protocols). Of special interest are distributed control systems in unreliable environments. Here, failures and interruptions of communication links need to be detected and handled.

2 Reuse and Reliability

Reliability is needed if one part of a system (A) relies on the data input of another part in the system (B). A system failure may result in a catastrophe for the application, which can only be avoided by moving the system into a safe state. In this case, B has to be monitored in order to detect failure and to respond adequately. To provide a *fail-safe* or *fail-operational* state [15], a system must have the ability to detect system failures, which in this case can be done by using a *Watchdog*. This is a special component monitoring the operation of a system. The observed system has to send a periodic life-sign called *Heartbeat* to the *Watchdog*. If this life-sign fails to arrive at the *Watchdog* within a certain period, the *Watchdog* assumes a system failure and therefore moves the controlled system into a *fail-safe* or *fail-operational* state.

Therefore new functionality must be added to realize those new requirements. We have identified two different solutions. The first solution introduces the new functionality by refining the given components using existing design patterns WATCHDOG and HEARTBEAT, the second by adding available reliability design components to the system.

Augmentation using SDL design patterns: The extension of the system using design patterns is more flexible than using ready-to-use components. Figure 1 shows a generic application of the methodology:

A component (*comp A*) of subsystem A communicates virtually with *comp B* of subsystem B. *Comp B* provides necessary data for *comp A* and must therefore be monitored. C*omp A* is refined by applying the WATCHDOG pattern and *comp B* is refined by applying the HEARTBEAT pattern. There are several advantages of this approach: no change in the structure of the system is necessary, it is fast to realize, it has the ability to detect system failures within subsystem B and within the communication link. The disadvantage is that we can not detect failures within the component containing the new watchdog functionality.

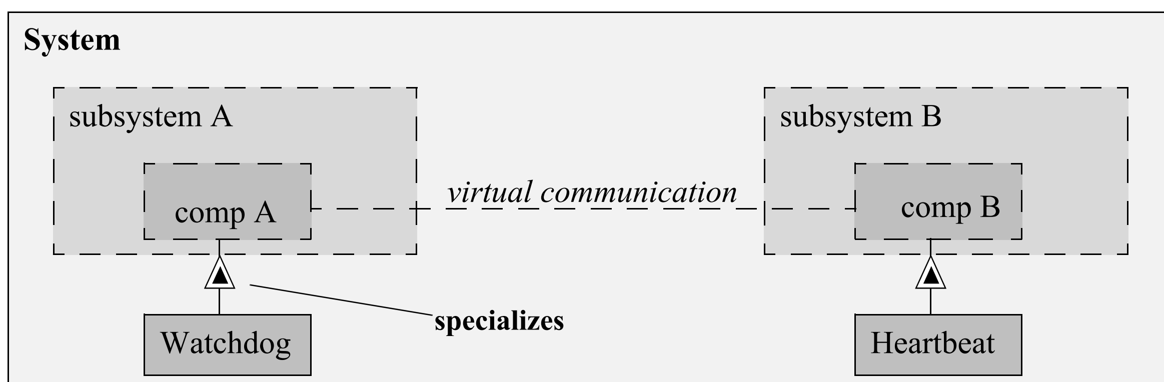


Figure 1: Using SDL design patterns

Using SDL design components: The extension of the system with available design components is necessary either if we don't want to change an existing component or we are not able to, e.g., if the system contains 3rd party components. Figure 2 shows a generic application of the methodology.

The design component *Heartbeat* generates a periodic alive signal and can easily be integrated into an existing system. The component *Watchdog* contains the watchdog functionality and is triggered by *Heartbeat*. The disadvantage of this solution is that the structure of the system must be changed, and that it is only possible to monitor the communication link between both subsystems.

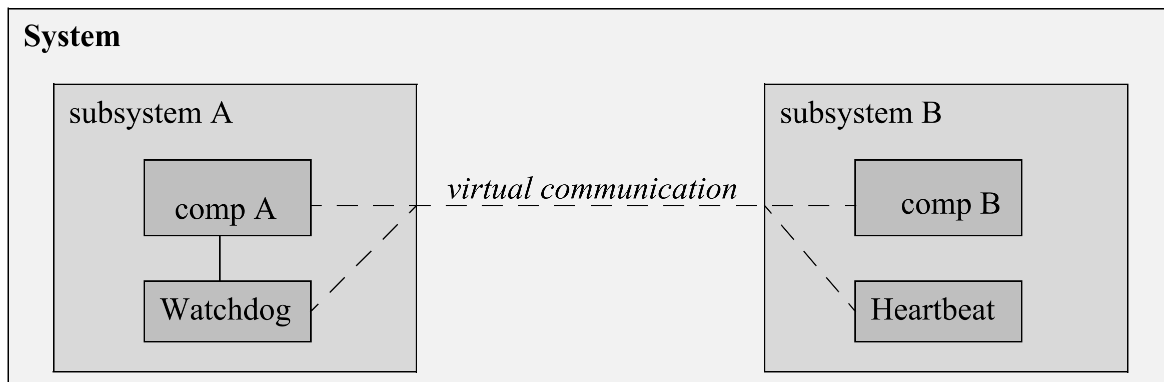


Figure 2: Using design components

Of course, both approaches can be combined, for example by using the design component *Watchdog* and the HEARTBEAT design pattern to avoid the addressed disadvantage. In the following, both solutions will be presented in more detail.

3 SDL design patterns for reliable systems

3.1. SDL design patterns

Design patterns [5] are a well-known approach for the reuse of design decisions. In [1], 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 [7,9] consists of a *pattern-based design 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 [8], the Internet Stream Protocol ST2+ [14], and a quality-of-service management and application functionality for CAN (Controller Area Network) [6]. Applications in industry, e.g., in UMTS Radio Network Controller call processing development, are in progress [10].

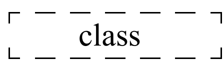
An *SDL design pattern* [5,7] is a reusable software artifact that represents a generic solution for a recurring design problem with *SDL* [12] as design language. Over a period of more than 25

years, SDL (System Design Language) has matured from a simple 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.

The following two paragraphs describe the notation used to specify design patterns.

3.2. Pattern Annotated UML

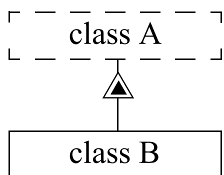
Pattern Annotated UML (PA-UML) extends the UML class diagrams and is used to describe the structure of a design pattern.



A class of the context in which the adapted pattern has to be embedded.

New class that results from applying the pattern.

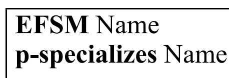
Assoziation that is still valid after applying the pattern. It can be a new or an existing one.



Class B **p-specializes** class A, i.e class B describes a specialization by applying the pattern.
This notation can also be used between diagrams

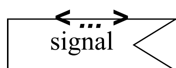
3.3. Pattern Annotated SDL

Pattern Annotated SDL (PA-SDL) extends SDL with meta-symbols and is used to describe the SDL fragments introduced by the patterns solution.

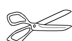


Extended Finite State Machine: The EFSM block describes a process (type), a service (type) or a procedure. It can also be a specialization of an other EFSM.

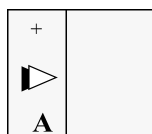
Context: The parts of the SDL system that belong to the context are dashed.



Generic trigger: This meta-symbol can be replaced with any input trigger, e.g., input, priority input, continous signal, etc.



Scissor symbol: The scissor symbol indicates a point where a transition can be refined by the context.



Border symbol: Part of an SDL-system, which can be copied either vertically or horizontally. The direction of replication is given by the arrow. In the upper left corner you can specify the multiplicity.

3.4. The design pattern *Watchdog*

The *Watchdog* pattern realizes the safety functionality described in Section 2 and belongs to the category of *Interaction patterns.* It describes a behaviour, that extends a given system.

The MSC in Figure 3 shows an example where the described design problem arises, which is solved by the suggested solution:

In an automatic safety device on trains (dead man's control), an operator has to press a button periodically within a prior well defined time interval. When the operator desists from pressing the button, the automatic safety device assumes the operator is dead and stops the train, which leads the system to a *fail-safe* state in order to prevent a catastrophe, e.g., a crash at an unmanned crossing.

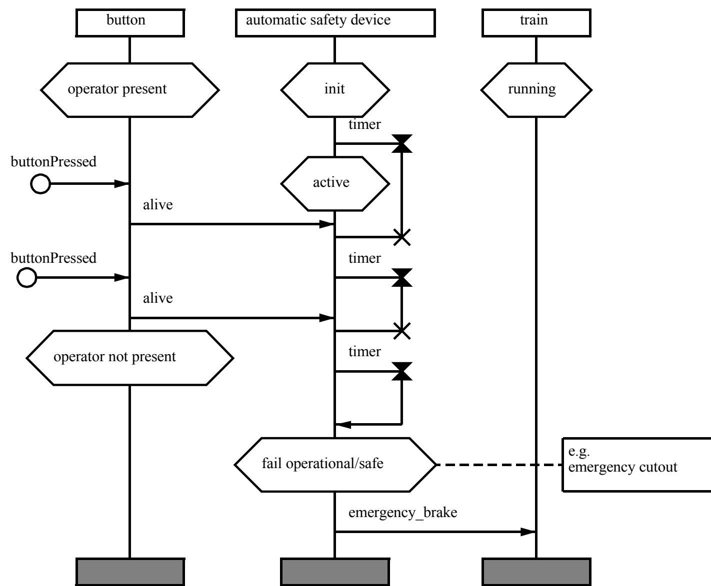


Figure 3: MSC automatic safety device

Figure 4 shows the graphical representation of the structural aspects of the pattern's solution. Note that *Watchdog* either refines a component from the context or is added as a new component to the structure:

- *Trigger* is a component of the context, which provides an alive signal periodically.
- *Controller* is a component where watchdog functionality is to be added. The WATCHDOG pattern can either refine a component of the system or augment the system with a new component.
- *Controlled System* is the part of the system which is moved into a safe state, when life-sign fails to arrive.

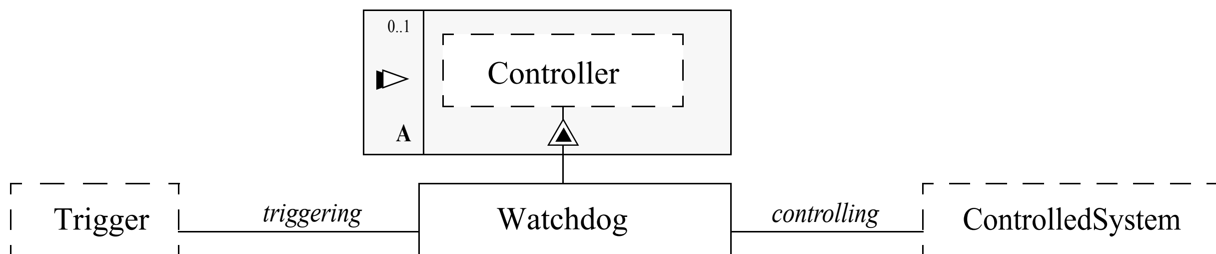


Figure 4: Structure of the design solution - WATCHDOG

The *SDL-Description* of the WATCHDOG pattern is shown in Figure 5. It describes the syntactical part of the suggested design solution, which is adapted and composed when the pattern is applied. The extended finite state machine *Watchdog* that optionally refines *Controller* describes the watchdog functionality. The timer *watchdogT* is set for a duration of *safeInterval* when triggered by a certain input from the context and restarted after a trigger (Figure 5 centre). The trigger

showing that the system is still alive can be one or more inputs or continuous signals. The duration *safeInterval* is the timeout interval after which the system changes to a *fail-safe/fail-operational* state. This is done by sending one or more control signals to the controlled system (Figure 5 right). Disabling the *watchdog* is also possible (Figure 5 left).

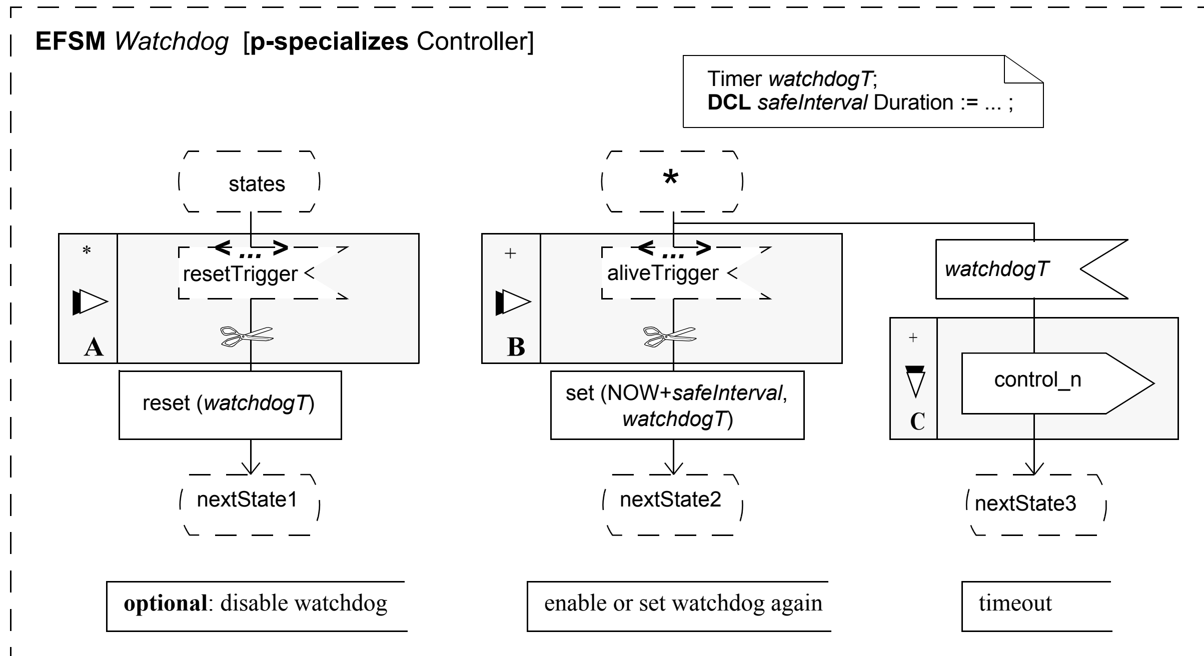


Figure 5: SDL design pattern WATCHDOG

3.5. The design pattern *Heartbeat*

The watchdog functionality described in Section 2 assumes a periodic trigger in order to prevent the watchdog from sending control signals. If the system does not provide a periodic communication with an adequate interval, the *Heartbeat* pattern can be applied. This augments the system behaviour with the *heartbeatSignal* that is periodically sent.

The following figure shows the graphical representation of the structural aspects of the patterns solution. Note that *Heartbeat* either refines a component from the context or is added as a new component to the structure:

- *Application* is refined by *Heartbeat* and describes the system that has to be monitored.
- *Watchdog* is a component realizing watchdog functionality as described in Section 3.4.

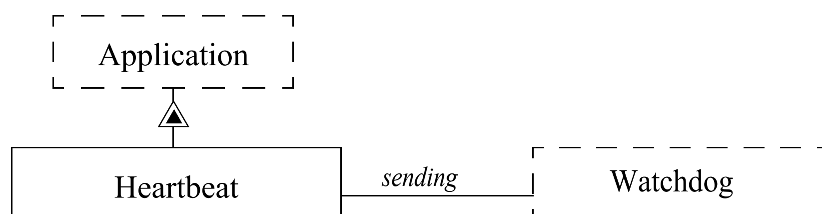


Figure 6: Structure of the design solution - HEARTBEAT

Figure 7 shows the SDL-description of the HEARTBEAT pattern. The EFSM *Heartbeat* describes the heartbeat functionality. After the initialization or an input signal (Figure 7, left), the timer *heartbeatT* is set to the duration of *hbInterval* and after the timeout a *heartbeatSignal* signal is generated and propagated. Therefore *Application* is refined by adding transitions to start and stop the heartbeat and one to handle the *heartbeatT* by sending the *heartbeatSignal*. The *heartbeat-Signal* has to be consumed by a corresponding component which realizes watchdog functionality. This signal can also be an existing signal in the system that can be used for a heartbeat.

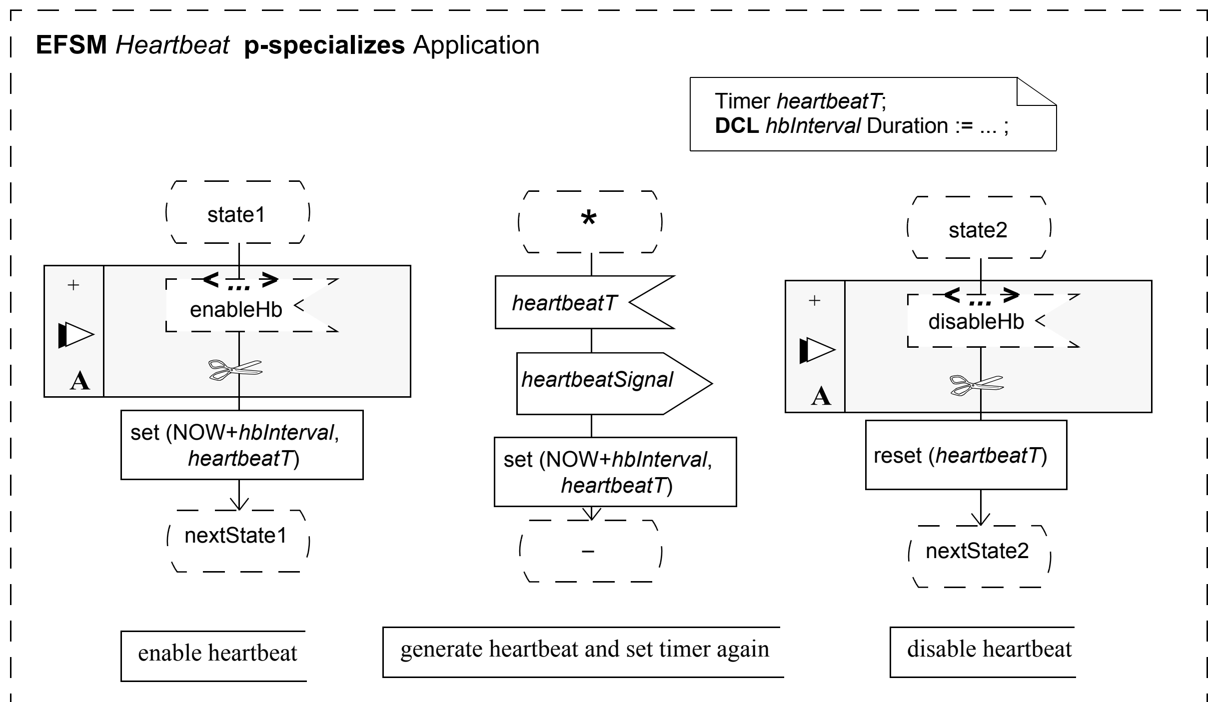


Figure 7: SDL design pattern HEARTBEAT

It is possibe to define a combined WATCHDOG and HEARTBEAT pattern, but not mandatory.

4 Micro protocols for reliable systems

4.1. Micro protocols

In [2][3], we have introduced and applied a new type of communication component, called *micro protocol*, i.e., a communication protocol with a single (distributed) protocol functionality and the required protocol collaboration. Here, a protocol functionality is a single aspect of internal protocol entity behavior (operational structuring), e.g., flow control, loss control, corruption control. It is realized by a particular protocol mechanism (e.g., sliding-window, sequence numbering, checksum), and generally distributed among a set of protocol entities. A protocol collaboration is a self-contained subset of synchronization and causality relationships of a set of protocol entities. Because a protocol functionality covers only one single aspect of protocol be-havior, a micro protocol is not decomposable into smaller protocol units. A micro protocol is a special kind of a design component.

Conceptually, we model protocol entities by asynchronously communicating extended Mealy machines. Obviously, there are several ways to represent them in SDL, for instance, by specifying SDL block types, SDL process types, SDL service types , or SDL procedures. Which one to use depends on the composition of micro protocols, which in turn depends on the protocol that is to be configured.

Micro protocol definitions are organized using SDL packages. An SDL package is a collection of type definitions, and is used here to encapsulate SDL types belonging to the same micro protocol. This way, a micro protocol library can be expressed as a set of SDL packages, i.e., ready-to-use components. Also, common parts of a set of micro protocols may be extracted into a package that is imported by each micro protocol definition. Alternatively, several related micro protocols may be grouped into one package.

In the context of reliability we have identified two different micro protocols.

4.2. The micro protocol *Watchdog*

The micro protocol *Watchdog* is encapsulated in one single process type (Figure 8) and may be specialized to match the requirements of the embedding context. A timer *watchdogT* is used to monitor receipt of an *alive* signal from the context within a well-defined interval. This *safeInterval* is initially defined, but can be modified by redefining the virtual start transition. When *Watchdog* does not receive an *alive* signal within this given period, the timer *watchdogT* triggers a transition to send a signal to the context (controlled system). Again, this signal must be specified by redefining a virtual transition. Optionally, a signal may be send when the watchdog assumes the observed system to be dead and an *alive* signal reappears.

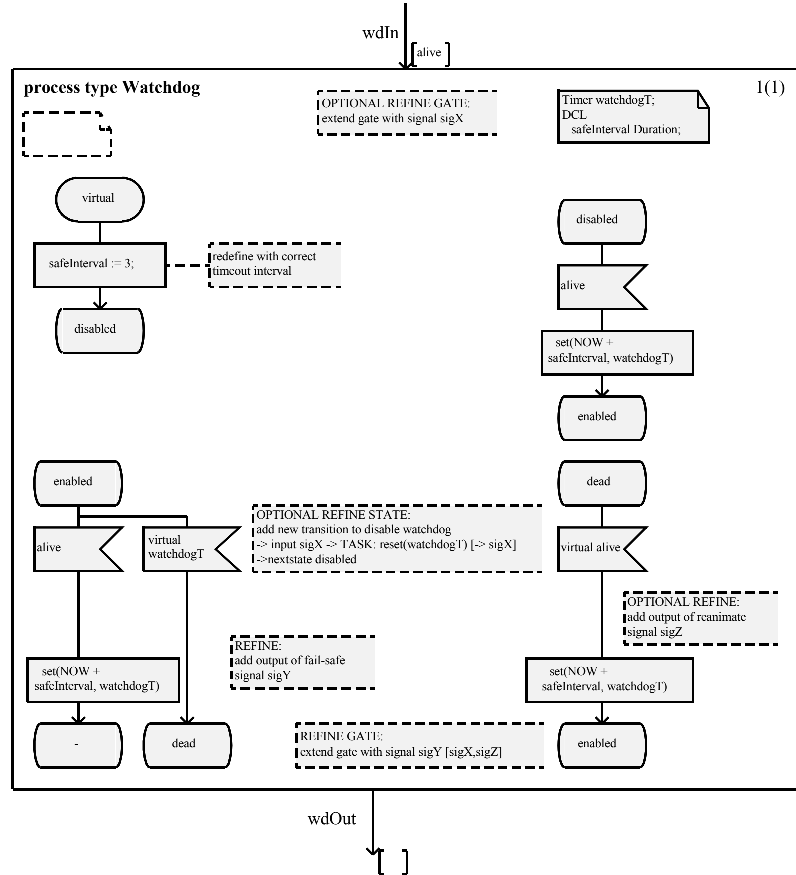


Figure 8: Micro protocol *Watchdog*

In order to provide a periodic *alive* trigger, another micro protocol *Heartbeat* can be used.

4.3. The micro protocol *Heartbeat*

The micro protocol *Heartbeat* is used to provide a system with a periodic sending of an *alive* signal. This signal is used to trigger the micro protocol *Watchdog* showing that the observed system is still alive.

The behaviour is encapsulated in one single process type shown in Figure 9. The predefined *heartbeatInterval* in which signals are sent should be adapted to fit the requirements of the watchdog observing this system. This is done by refining the start transition.

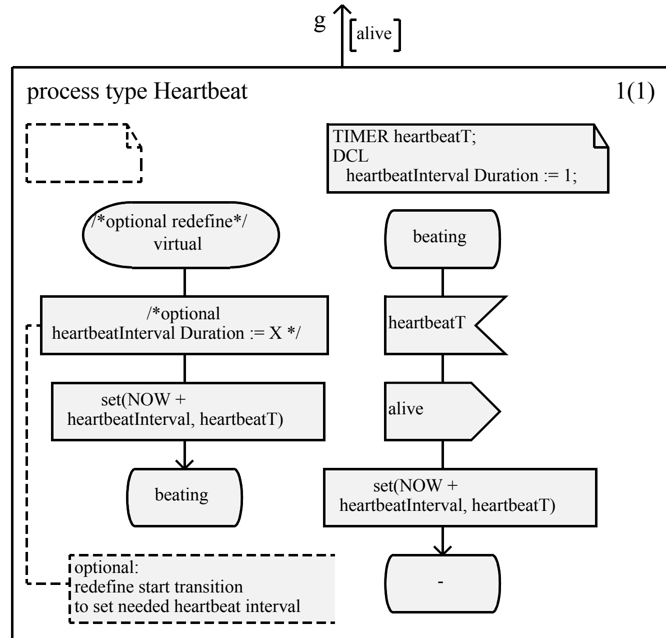


Figure 9: Micro protocol *Heartbeat*

5 System „Airship-Control“

The sample system is an excerpt from the airship system in [16]. It consists of a control application to control the airship via an external controller. The control application is divided into two parts, an *airshipClient* and an *airshipServer*. The *airshipClient* transmits the processed control values generated by the external controller, the *airshipServer* receives the values and controls the airship hardware. Because of the properties of both parts of the application, we have a classical Client-Server-Architecture. Figure 10 shows an overview of the available architecture and the structure of the client and the server. The applications are distributed and communicate via WLAN.

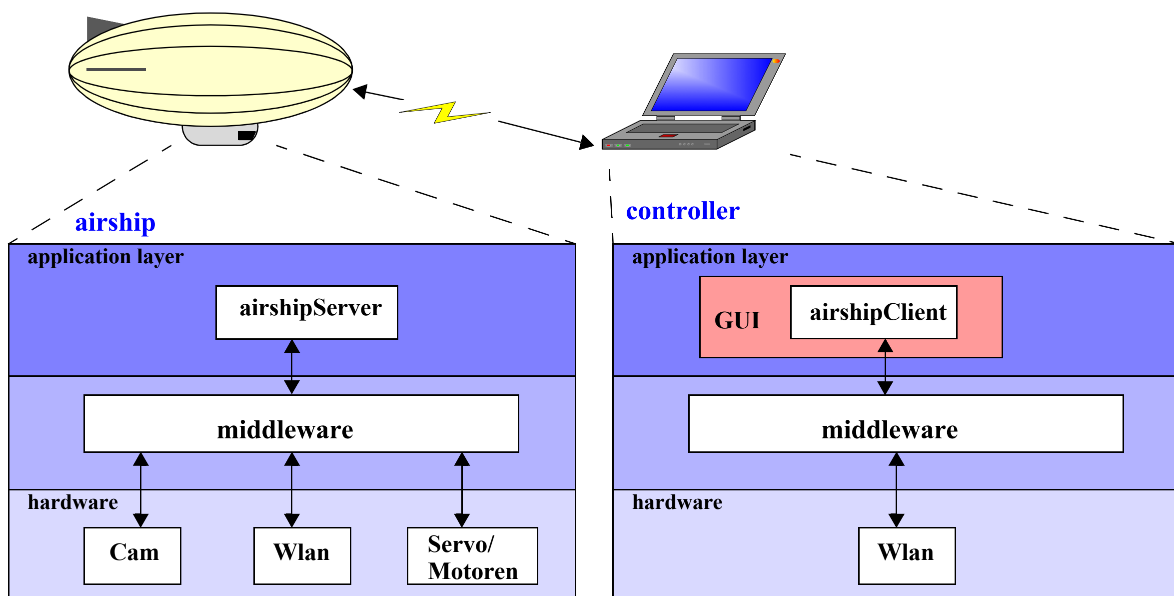


Figure 10: Architecture and structure of the application

The SDL process in Figure 11 shows the simplified functional behaviour of the *airshipClient*. When triggered by a *value* from the environment (generated by an external controller) with the parameter *id* of type *StartStop, airshipClient* sends a signal *startAirship* to the *airshipServer* and enters state *enabled.* This indicates the begin of the flight. On the next occurence of *value* with the same id (*StartStop*), an output *stopAirship* is generated and the state *disabled* is re-entered. The variables *val1, val2* and *val3* contain the current values to control the airship. These values are calculated from the data received by the external controller and propagated at every change (*newCtrlValues*). This calculation is done by the procedure *processValue.*

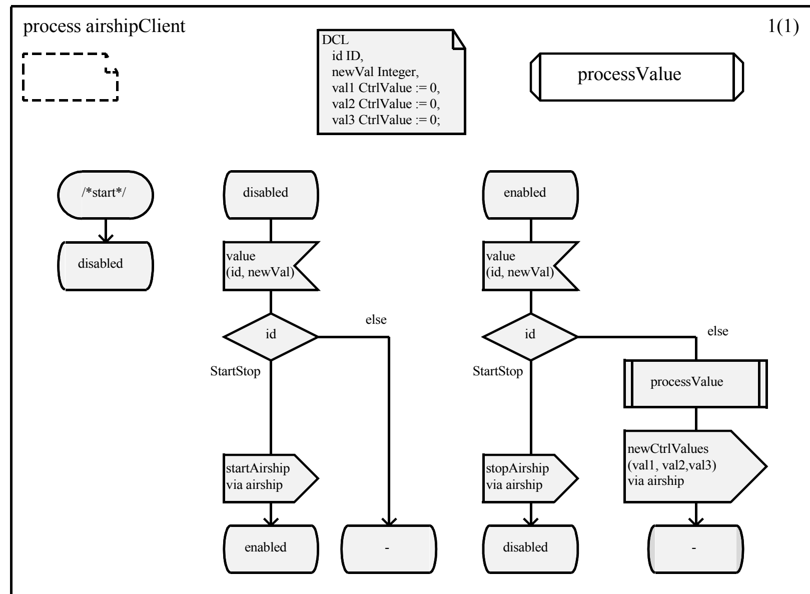


Figure 11: Process *airshipClient*

Figure 12 shows the behaviour of *airshipServer*. On reception of *startAirship*, it sets an SDL timer *t* with a default interval of 50 milliseconds and enters state *enabled*. The signal *newCtrlValues* updates the current settings of the airship and triggers the output of the signals *ctrlValue* controlling the hardware. The timer *t* is used to prevent the hardware to fall back into initial state. On input of the signal *stopAirship,* the timer is switched off and the new state is *disabled*.

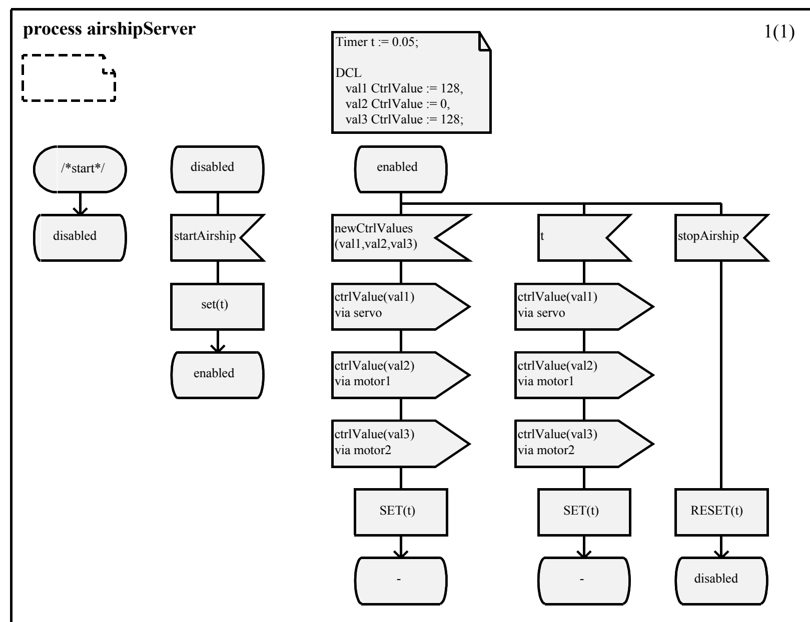


Figure 12: Process *airshipServer*

In the following we shortly present the application of the introduced design patterns to augment this system with reliability.

5.1. Application of the HEARTBEAT-Pattern

Figure 13 shows the result of applying the HEARTBEAT pattern to the process *airshipClient*. Two transitions were extended as described in the pattern to enable and disable the heartbeat and a new transition generates a heartbeat signal every *hbInterval* seconds.

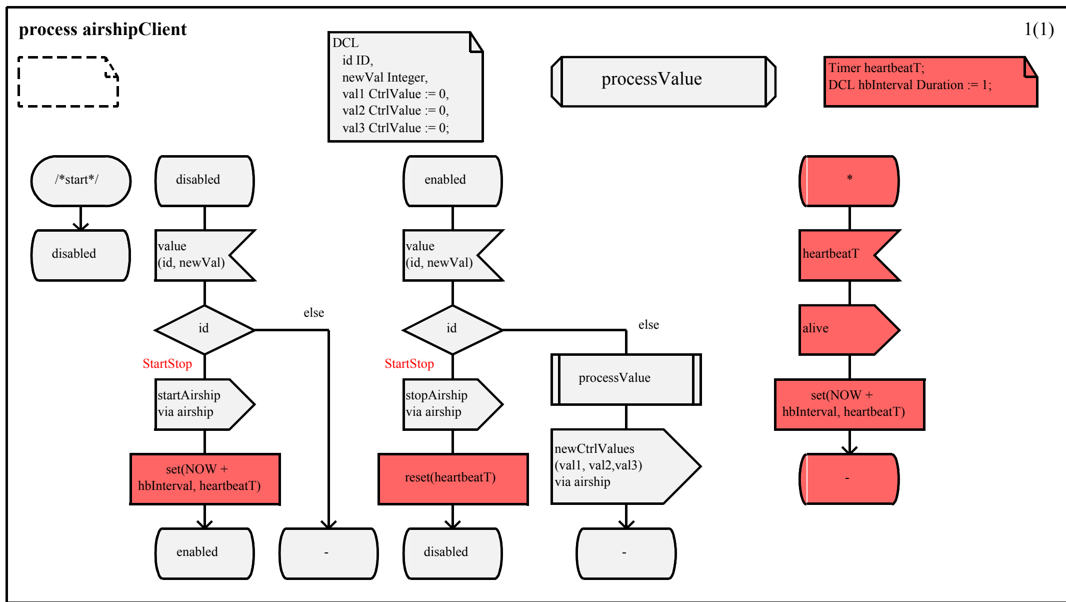


Figure 13: Application of the HEARTBEAT pattern

5.2. Application of the WATCHDOG-Pattern

The result of applying the WATCHDOG pattern to the process *airshipServer* is shown in Figure 14. We modified one existing transition and added two new transitions as described in the pattern description. The signal *stopAirship* is used to disable the watchdog and therefore the existing transition is extended by a new task resetting the watchdog timer *watchdogT*. The heartbeat *alive* starts and resets the watchdog. After a timeout of the watchdog timer the fail-safe signals are sent

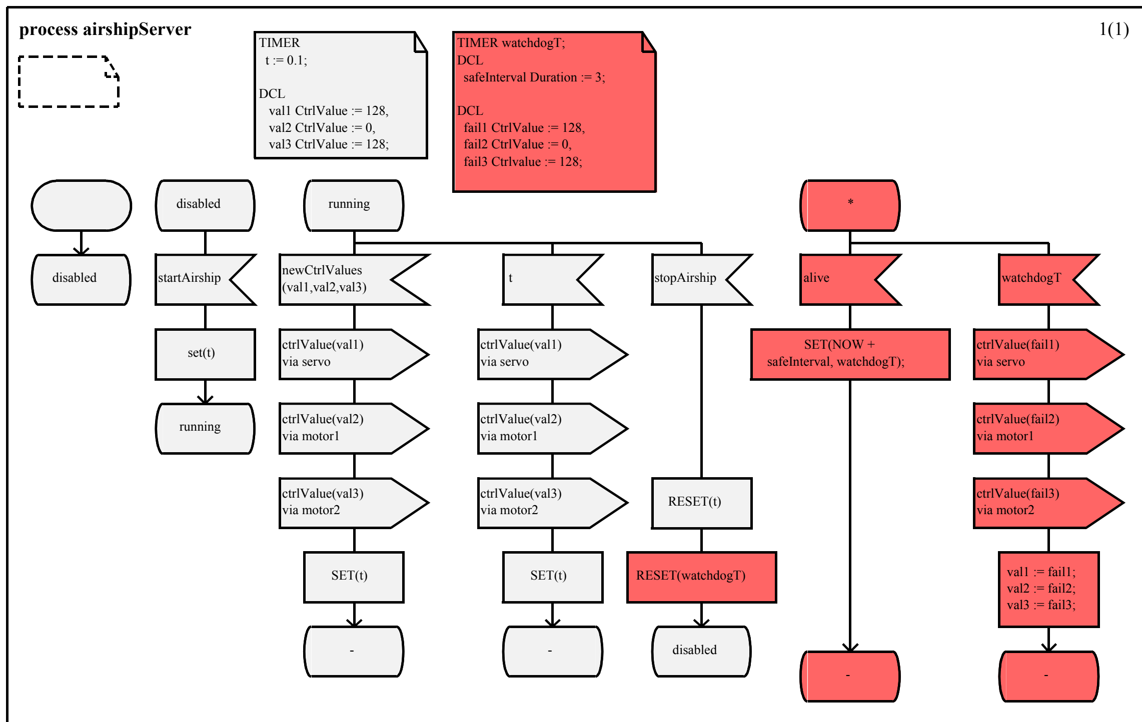


Figure 14: Application of the WATCHDOG pattern

to the airship hardware moving the airship into a safe state.

If it is not possible to use SDL design patterns, existing micro protocols can instead easily be integrated to your system.

5.3. Using design components

Starting point for the system design activity is a requirements specification, resulting from a thorough requirements analysis. According to these requirements, the corresponding micro protocols are selected from a micro protocol library and glued together. This results in a self-contained SDL design specification. Exisiting system designs may be augmented by additional micro protocols when new requirements are demanded.

In this example, new reliability functionality is added to an existing system. Therefore we select the micro protocol *Watchdog* from the micro protocol library and refine this micro protocol in order to integrate it into the system (Figure 15). Therefore, we have to redefine the *watchdogT* timeout transition to handle the failure of the monitored system. To be able to stop the watchdog a new transition is added.

There is no need to refine *Heartbeat.* We can simply integrate this component to our system.

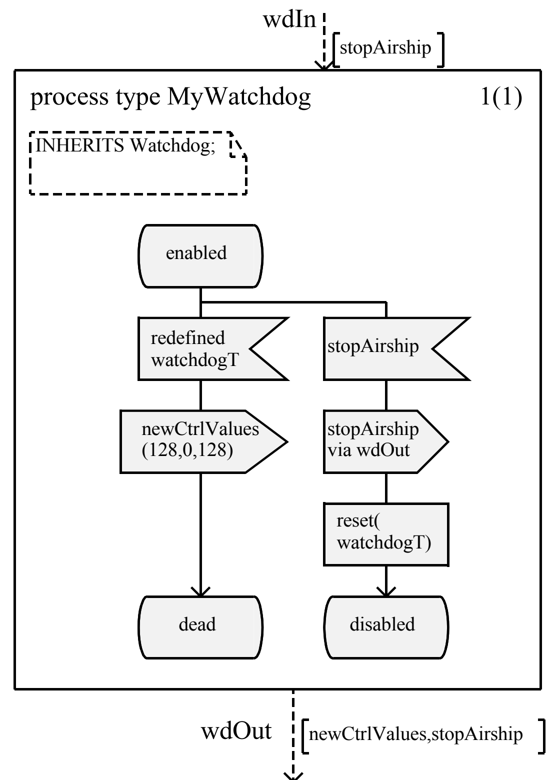


Figure 15: Refinement of *Watchdog*

6 Conclusions

In this paper, we have shown how to use the systematic approach of SDL design patterns and design components to augment existing systems with reliability aspects. To illustrate this approach, we have given two examples of design patterns that can be used to specify and to document the behavior of a heartbeat/watchdog-system. By selecting these patterns from the pool, and by adapting them to the context, these patterns may support a wide variety of distributed systems. The system designer profits from their genericity and the early application of these design patterns.

Second, we have presented two ready-to-use micro protocols as design components that have been used in the development of the remote airship flight control over WLAN. With parametrization and the ability to use object oriented inheritance, this solution offers rapid development profiting from advanced reuse.

We are using a micro protocol-based development process, which starts with a functionality analysis, followed by an abstract design using a micro protocol framework, and completed by a concrete SDL design. The design activities are supported by a micro protocol library, from which micro protocols are selected and composed. This may pave the way to compositional testing. Each of these components can be tested using well-proven techniques. However, when these

components are put together, the resulting system must only be tested for composition faults, which may increase the reliability of systems.

References

[1] B. Geppert, R. Gotzhein, F. Rößler: *Configuring Communication Protocols Using SDL Patterns,* In Cavalli, A., Sar,a, Q., eds.: SDL'97 - Time For Testing, Proceedings of the 8th SDL Forum, Amsterdam, Elsevier (1997) pp. 523-538

[2] R. Gotzhein, F. Khendek: *Conception avec Micro-Protocoles,* Colloque Francophone sur l'Ingénierie des Protocoles, Montreal, Canada, May 27-30, 2002

[3] R. Gotzhein, F. Khendek, P. Schaible: *Micro Protocol Design: The SNMP Case Study,* in: Telecommunications and beyond: The Broader Applicability of SDL and MSC, E. Sherratt (Ed.), LNCS 2599, Springer, 2003, pp. 61-73

[4] Computer Networks Group: *The SDL Pattern Pool*, Online document, University of Kaiserslautern, Kaiserslautern, Germany, 2002 (available on request)

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995

[6] 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

[7] B. Geppert: *The SDL-Pattern Approach - A Reuse-Driven SDL Methodology for Designing Communication Software Systems*, Ph.D. Thesis, University of Kaiserslautern, Germany, 2000

[8] R. Gotzhein, C. Peper, P. Schaible, J. Thees: *SILICON - System Development for an Interactive LIght CONtrol*, URL: http://vs.informatik.uni-kl.de/activities/silicon/, 2001

[9] R. Gotzhein: *Consolidating and Applying the SDL-Pattern Approach: A Detailed Case Study*, Information and Software Technology, Elsevier Sciences

[10] 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

[11] R. Grammes: *Evaluation and Application of the SDL Pattern Approach*, Master Thesis, Computer Networks Group, University of Kaiserslautern, Kaiserslautern, Germany, February 2003

[12] ITU-T Recommendation Z.100 (11/99) - *Specification and Description Language (SDL)*, International Telecommunication Union (ITU), 2000

[13] ITU-T Recommendation Z.120 (11/99) - *Message Sequence Chart (MSC)*, Intern. Telecommunication Union (ITU), 2000

[14] F. Rößler, B. Geppert, P. Schaible: *Re-Engineering of the Internet Stream Protocol ST2+ with Formalized Design Patterns*, Proceedings of the 5th International Conference on Software Reuse (ICSR5), Victoria, Canada, 1998

[15] H. Kopetz: *Real-Time Systems - Design Principles for Distributed Embedded Applications,* Kluwer Academic Publisher, 1997

[16] C. Webel: *Development and Integration of QoS Micro Protocols for Controlling an Airship via WLAN,* Master Thesis, Computer Networks Group, University of Kaiserslautern, Kaiserslautern, Germany, June 2004 (in german)