ISBN 978-83-60810-14-9
ISSN 1896-7094

A Safety Shell for UML-RT Projects

Roman Gumzej
University of Maribor
Faculty of Electrical Engineering and Computer Science
2000 Maribor, Slovenia
roman.gumzej@uni-mb.si

Wolfgang A. Halang
Fernuniversität
Chair of Computer Engineering and Real-time Systems
58084 Hagen, Germany
wolfgang.halang@fernuni-hagen.de

***Abstract*—A safety shell pattern was defined based on a reconfiguration management pattern, and inspired by the architectural specifications in Specification PEARL. It is meant to be used for real-time applications to be developed with UML-RT as described. The implementation of the safety shell features as defined by in [8], namely its timing and state guards as well as I/O protection and exception handling mechanisms, is explained. The pattern is parameterised by defining the properties of its components as well as by defining the mapping between software and hardware architectures. Initial and alternative execution scenarios as well as the method for switching between them are defined. The goal pursued with the safety shell is to obtain clearly specified operation scenarios with well defined transitions between them. To achieve safe and timely operation, the pattern must provide safety shell mechanisms for an application designed, i.e., enable its predictable deterministic and temporally predictable operation now and in the future.**

I. Introduction

PREDICTABILITY, dependability and timeliness are major pre-conditions for real-time applications to be safe. Hence, in order to be able to effectively address safety, it should be sustained throughout the entire life-cycle of an application – from design via implementation to upgrades and maintenance. Therefore, it appeared sensible to build a design pattern that would enable designers to address most safety issues and build safe and persistent applications. Kornecky and Zalewski [8] have defined a "safety shell" for real-time applications to be composed of several "guards", each one protecting a certain part of an application. Thus, the input/output needs to be protected from tampering as well as by range checking to sustain the environmental parameters of the application. Then, exception-handling mechanisms should protect the application from malicious consequences of unforeseen situations, by offering mechanisms that bring it back to normal operation. Finally, the operation should be monitored and safeguarded in its state and time spaces in order to prevent the application from leaving its specified execution and temporal frameworks. Since all mechanisms mentioned foresee different scenarios for initialisation, phases of normal operation and of various exception modes, enabling dynamic re-configurations on the application level is crucial to enable these features.

In the development of embedded real-time systems, management of dynamic (re-) configuration has systematically been addressed by hardware/software co-design methods (cp., e.g., [9, 11, 13]). Besides defining diverse (dynamic) operation scenarios, two main goals have been targeted by this approach:

1) achieving fault tolerance by system design (cp. [3, 6]),
2) fast scenario switching (e.g., in industrial automation and telecommunication systems [1, 4, 5, 11]).

II. Design for Safety

When designing a real-time system, generally three viewpoints must be considered:

1) the external (functional) one, which represents the inputs/outputs and usage scenarios,
2) the internal (behavioural) one, which deals with the definition of usage scenarios, and
3) the definition of system structure – hardware and software architectures together with the mapping of software components onto hardware components and the definition of configurations and re-configuration scenarios.

During re-configuration, application data must remain consistent and real-time constraints must be satisfied. In order to be able to achieve this, these issues must be addressed at multiple levels. At the lowest level, the hardware must be reconfigurable. Software-programmable hardware components support this inherently, since their functions can be changed by their memory contents. Internal hardware structures are designed to restrict dangerous conditions that could damage hardware. At the next higher level, the internal states of the software must be managed under changing tasking. Operating systems support flexible implementations of multiple tasks on single processors in form of time-sharing and/or multitasking. On the top level, one wants to define operation scenarios – configurations – for an application, which enable it to adapt to varying conditions in the environment on one hand, and to respond to changing operational modes by switching between operation scenarios in a safe and predictable manner on the other. Typically, these configurations cannot be managed by operating systems, since groups of processes and possibly also hardware components are involved. Hence, their management is usually placed on the application or middleware level, since it requires the observation of and actions based on the system state. Generally, by this approach, low-level efficiency and hard real-time properties are difficult to achieve. Because of this, we chose to distribute re-configuration management to all three levels – hardware, middleware, and software. With this in mind, the hardware/software co-design profile and pattern for real-time application design in UML based on the specification language Specification PEARL (S-PEARL) (cp. [2]) have been developed. While in the profile the constructs of S-PEARL are introduced with their properties, behaviour and interconnections, the configuration management pattern provides the mapping of software to hardware components, and a foundation on which to build custom real-time applications. The latter's approach is followed in extending and parameterising the configuration management pattern with safety features. The pattern and its safety-oriented use are presented throughout this article with the goal to construct a safety shell (cp. [7, 8]) for a designed application.

A. *Configuration Management Pattern*

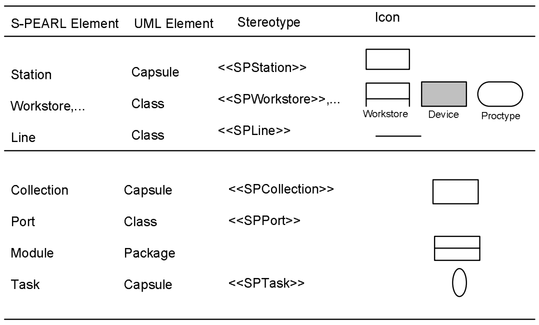
The configuration management pattern was constructed by combining a set of UML-RT [12] stereotypes [10], which represent S-PEARL constructs (see Fig. 1) as a coherent whole. They constitute building blocks at the three levels of architectural modeling: (1) hardware architecture, (2) software architecture, and (3) software-to-hardware mapping. A hardware architecture is composed of processing nodes, termed "stations", whose descriptions also mention their components with their properties. A software architecture is organised in the form of "collections" of "modules", comprising program "tasks", functions and procedures of the application software. The "collection" is the unit of software to be mapped onto "stations", i.e., at any time there is exactly one collection assigned to run on a station. Thus, the "collection" is also the unit of dynamic re-configuration. Each of these constructs has its specific attributes, which pertain to all objects of this type (such as properties, relations, and initialisation). They are layered on three levels of abstraction (see Fig. 2):

1) station level, where a mapping of collection configurations to stations is established;
2) configuration level, where collections grouped into scenarios, named "configurations", are managed by a "reconfiguration manager"; and
3) collection level, where the tasks, which may be grouped into modules (UML packages), are managed by their collections according to their scheduling parameters.

Each collection belongs to a configuration and is mapped to a station. Configuration management is responsible for the co-operation among collections and possible dynamical re-configurations, which depend on the state changes of the station they are residing on. A detailed description of its safety-oriented use is presented in the sequel.

III. Introducing a Safety Shell

A safety shell is responsible for guarding the main process termed "Primary control" (see Fig. 3) by providing it with additional functionality which keeps the possible sources of error to a minimum. In order to be effective, these functions have to be integrated into or built around an application. In our case, the second variant was chosen by implementing a pattern, which forms the "backbone" of an application, requiring it



S-PEARL Element	UML Element	Stereotype	Icon
Station	Capsule	<<SPStation>>	▭
Workstore,...	Class	<<SPWorkstore>>,...	Workstore Device Proctype
Line	Class	<<SPLine>>	—
Collection	Capsule	<<SPCollection>>	▭
Port	Class	<<SPPort>>	
Module	Package		▭
Task	Capsule	<<SPTask>>	⬯

Fig. 1. S-PEARL constructs and their UML (-RT) stereotypes

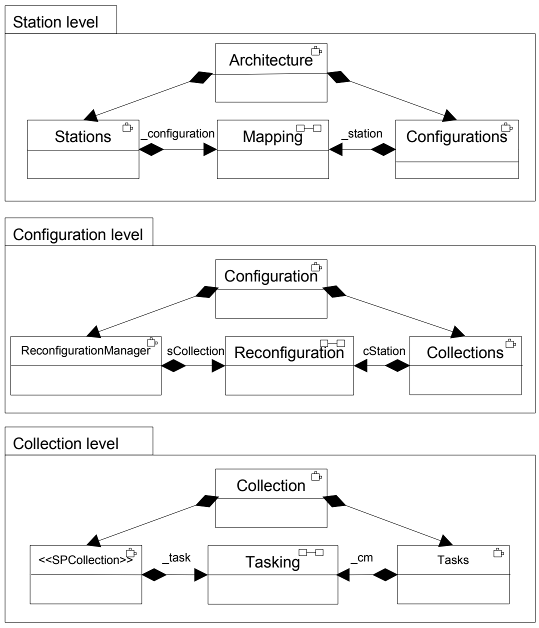


Fig. 2. Levels of configuration management

to be formed in a specific manner in order to function in the safety shell's environment. The configuration management pattern has the structure and functions needed to fulfill the rôle of a safety shell in terms of guarding a system in such a way that it always remains in a foreseen state and time frame of operation. In the sequel, the functions of the four protective mechanisms are described, and it is explained in which manner and to which degree they safeguard an application's execution.

A. *Protected Input/Output*

Protected input/output refers to well defined interfaces with the environment. By well defined we mean stable physical connections and sound protocols with integrated error checking and correction techniques. Usually, the possible problems originate from data overruns or malicious data. By themselves, the device drivers of interfaces can only correct a part of these problems associated with data formats and protocols. They could, however, also detect overruns or out-of-scope data, prevent recurring corruption of data, and signal possible errors. In our implementation of I/O *ports* (see Fig. 4), an important

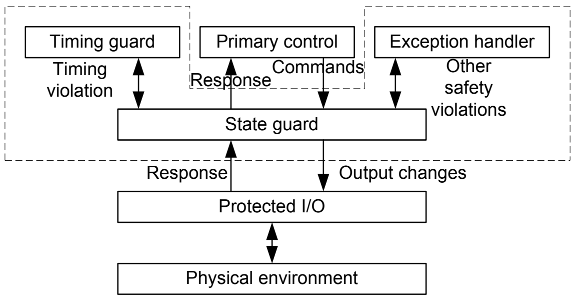


Fig. 3. Safety shell scheme

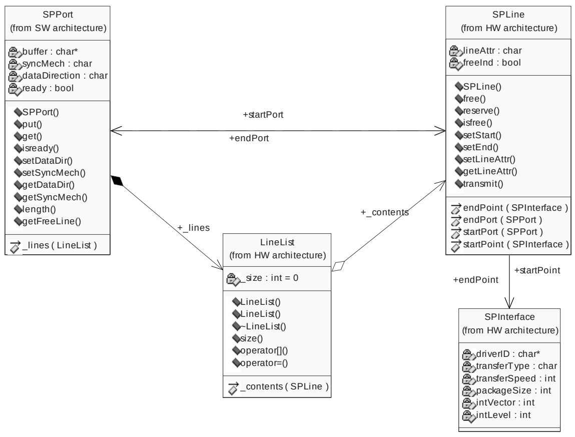


Fig. 4. Safeguarded port-to-port communications

safety-related feature is present, namely, routing parameters. Where stable line connections are of utmost importance, they are usually designed redundantly, being doubled, tripled, or with one of the *lines* representing a slower yet reliable (e.g., wireless) connection. In our routing parameters we can determine the lines which can/must be used, and/or assign a preferred line, being the fastest or most trusted one. If a line is not or becomes unavailable, the protocol automatically searches for the next available line. On the application level, it is important to have uniform interfaces between software components possibly executing at different processing nodes, which is also achieved by “port-to-port” communication. The lower levels are suppressed on the application level, however the parameterisation of the connection lines between ports and device drivers is made transparent by the S-PEARL profile, and enables complete oversight down to the physical level.

B. State Guard

There should be a predefined scenario for each possible state a system can find itself in (“state guard”). The problem decomposition enables considering loosely coupled interdependent processing nodes, which ensure local predictability, and have well defined global interconnections that form an integral part of each scenario. Due to a possible state explosion during execution, it is impossible to (pre-) determine all global states a system can assume and define scenarios for them. Since scenarios are defined for each station separately (e.g., see Fig. 5),

```
// "initial_load_statement"
void init() {
   reconfigure(0); // '0' is the initial state
}

// "reconfiguration_statement"
void reconfigure(char s) {
   if (sreg!=s) { // sreg is the station state register
      switch (sreg) {
         case -1: // initially it is undefined
            break;
         default: // upon state change the current collection is unloaded
            sCollection.unload().sendAt(sreg);
            break;
      }
      switch (s) {
         case -1: // if there is no valid state, nothing is loaded
            break;
         default: // the collection associated with the new state is loaded
            sCollection.load().sendAt(s);
            break;
      }
   }
   sreg=s;
}
```

Fig. 5. Safeguarded state transitions within a station (configuration)

this problem becomes easier to tackle. There exists a limited number of states, only, and as global interconnections are (re-) connected during re-configuration, the global implications are unproblematic. Hence, besides *fault containment*, this makes designing distributed real-time applications easier, and the systems designed more robust. A rigorously designed application structure, in which the execution of a collection of tasks is associated with an exactly pre-determined state, and a simple and well defined station state changing mechanism prevent the transition to an undefined state, herewith implementing the state guard function. The same holds for tasks, where an exactly defined activity structure does not only prevent transitions to an undefined state by exception handling, but also supports safe-guarded execution of temporally sensitive operations (see Fig. 6).

C. Timing Guard

Timeliness, being a critical property of real-time systems, is of utmost importance for applications and, hence, it is vital that its “backbone” does not introduce any significant delays into execution. Due to this and to ensure observability, the service algorithms of the pattern have been kept simple. They introduce no unbounded delays into scenario switching. This was one of the pre-dispositions while designing them, and is as important for safety as for timeliness of operation. Since some operations such as scheduling, message transmission, or task activation still require some time, however, the operating system overhead shall not introduce any unbounded delays either and, moreover, the service times of operating system calls have to be incorporated into task/operation execution times. Hence, the execution of an underlying real-time operating system has also to be temporally predictable in order to enable timeliness. In our implementation this was accomplished by a small custom real-time operating system (RTOS) kernel, which could be substituted with a fully functional off-the-shelf RTOS, keeping in mind the restriction on bounded and

```
void taskmain()
{
        if (_start_state) {
                aState=choose_start();
                _start_state=false;
                _timeout=false;
        }
        switch (aState) {
                case i^th_state: {
                        if (_timeout) {
                                /* timeout action */
                        }
                        else {
                                _timeout=true; /* watchdog start*/
                                maxT.informIn(RTTimespec(timeout,0));
                                /* activity */
                                aState= i^th_next();
                                maxT.cancelTimer();
                                _timeout=false; /* watchdog end*/
                        }
                        break;
                }
                case end_state: {
                        /* activity*/
                        _start_state=true;
                        _timeout=false;
                        break;
                }
                otherwise:
                        break;
        }
}
```

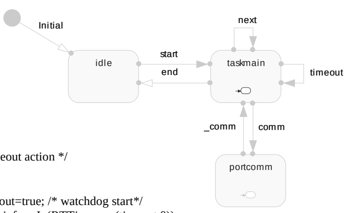


Fig. 6. Temporally safeguarded execution of task activities

predictable system call service times. Temporal monitoring of atomic activities such as executions of task operations is possible by introducing timers into them and, in this sense, prohibit any unreasonably long executions of atomic activities (e.g., by using a watchdog mechanism – "time guard" – see Fig. 6). Here it is also possible (for any activity – task state) to define a time-out action, which is executed in response to a possible time-out condition.

D. Exception Handler

Since tasks can be re-scheduled at pre-emption points, only, and tasking operations are defined for active tasks, it is sensible to limit their duration. In case of violating an execution time frame, a pre-defined scenario could be activated representing, for instance, *graceful degradation*. To further support *fault-tolerant* operation, however, it would be desirable to additionally check the correctness of other vital operation parameters, too, and by doing so implementing other features of the – "exception handler" – as well. This may introduce additional overhead, but as long as the execution times remain predictable and in the foreseen time frames, this is not a problem. Exception handling is, in part, already present in the safeguarded I/O operations. As already mentioned, the port-to-port communication protocol also enables line replication, thus allowing for "no single point of failure" planning. Range and other error checking mechanisms could be implemented to ensure *fail-safe* operation using well known mechanisms in the same manner as we implemented the time-out handling by introducing, e.g., pre-/post-condition checking of the (critical) tasks' activities. To support *reversion modes*, several collections with the same functionality may be defined within the same configuration to be activated depending on the different operational modes. The "context" of a configuration could be maintained as a list of "collection contexts" or – as in our case – in the form of procedures to (re-) establish collections. They can be (re-) established while (re-) loading and (re-) connecting their ports when desired/needed. The choice to (re-) start or continue a collection execution depends on the nature of the application and is, hence, left to the designer. Although typically one would continue from a state-switching condition, it is not always desirable or even dangerous to do so. In most cases, a collection is only re-loaded if its initial pre-conditions and environment state have been re-established. The "collection context" itself consists of its task control block (TCB) table as well as the lists of currently active tasks and ports. All of these would need to be re-established when re-loading a collection for execution continuation.

IV. Conclusion

With the help of the safety shell features of the S-PEARL configuration management pattern presented, distributed real-time application programs designed with UML-RT can run with safe, predictable behaviour and re-configuration support. Besides the structure of the applications, the configuration management pattern also defines uniform interfaces and protocols for intra- and inter-component/-node communication using pre-defined port/interface definitions. In (hard) real-time systems, it shall provide the necessary support for deterministic and dependable dynamic system re-configuration. The safety shell features are enabled by the pattern, but the selection and usage of the mentioned mechanisms remain the responsibility of a real-time application's designer, since no two safety critical real-time applications are equal.

References

[1] Eisenring M., Platzner M. and Thiele L.: Communication Synthesis for Reconfigurable Embedded Systems. P. Lysaght, J. Irvine, R. W. Hartenstein (Eds.): *Field-Programmable Logic and Applications, Proc.* (1999) 205–214, Berlin: Springer-Verlag.

[2] Gumzej R., Colnarič M. and Halang W. A.: A Reconfiguration Pattern for Distributed Embedded Systems. *Software and Systems Modeling* (2007) Springer-Verlag. http://dx.doi.org/10.1007/s10270-007-0075-7

[3] Hofmeister C. R.: Dynamic Reconfiguration of Distributed Applications. *PhD thesis* (1993) University of Maryland.

[4] Hutchings B. L. and Wirthlin M. J.: Implementation Approaches for Reconfigurable Logic Applications. *Field-Programmable Logic and Applications, Proc.* (1995) 419–428, Berlin: Springer-Verlag.

[5] Jean J., Tomko K., Yavgal V., Cook R. and Shah J.: Dynamic Reconfiguration to Support Concurrent Applications. *IEEE Symposium on FPGAs for Custom Computing Machines, Proc.* (1998) 302–303, Los Alamitos: IEEE Computer Society Press.

[6] Kalbarczyk Z. T., Iyer R. K., Bagchi S. and Whisnant K.: Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Trans. Parallel and Distributed Systems 10(6)* (1999).

[7] Katwijk J. van, Toetenel H., Sahraoui A., Anderson E. and Zalewski J.: Specification and Verification of a Safety Shell with Statecharts and Extended Timed Graphs. *Computer Safety, Reliability and Security* (2000) 37–52, LNCS 1943, Berlin: Springer-Verlag.

[8] Kornecki A. J. and Zalewski J.: Software Development for Real-Time Safety – Critical Applications. *Software Engineering Workshop – Tutorial Notes, 29th Annual IEEE/NASA 03* (2005) 1–95.

[9] Kramer J. and Magee J.: Dynamic Configuration for Distributed Systems. *IEEE Trans. Software Engineering 11(4)* (1985).

[10] Object Management Group: Unified Modeling Language: Superstructure. Version 2.0. *OMG document formal/2005-07-04* (2005).

[11] Rust C., Stappert F. and Bernhardi-Grisson R.: Petri Net Design of Reconfigurable Embedded Real-Time Systems. *IFIP 17th World Computer Congress – Design and Analysis of Distributed Embedded Systems, Proc.* (2002) 41–50, Dordrecht: Kluwer.

[12] Selić B. and Rumbaugh J.: Using UML for Modeling Complex Real-Time Systems. *Rational Software Corporation, White Paper* (1998) http://www.rational.com/media/whitepapers/umlrt.pdf

[13] Wolf W.: A Decade of Hardware/Software Codesign. *IEEE Computer 36(4)* (2003).