PEARL for Distributed Embedded Systems
"Object Oriented Perspective"

Roman Gumzej &
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI-2000 Maribor
Slovenia
roman.gumzej@uni-mb.si

Wolfgang A. Halang
FernUniversität Hagen
Faculty of Electrical Engineering
58084 Hagen
Germany
wolfgang.halang@fernuni-hagen.de

Abstract: This article is meant to highlight Specification PEARL in an object-oriented perspective. Specification PEARL is a specification and description language, which originates from Multiprocessor PEARL (DIN 66253, Part 3), also named PEARL for distributed systems. It extends the standard by allowing the description of asymmetrical distributed architectures as well as by additional parameters for the parameterisation of the RTOS and later feasibility analysis.

PEARL itself in its latest implementation still is a procedural language although it supports features like tasking and synchronisation, being supported only by some object oriented languages. Due to the nature of its applications, transferring PEARL into an object-oriented language was not an easy nor straightforward process; hence, there are several implementations of object-oriented PEARL. For Multiprocessor PEARL there was no attempt in this direction so far. In Specification PEARL HW/SW co-design methodology we are striving to use the Specification PEARL language as a specification language with the current release of PEARL (PEARL90).

The aim of this article is to give Specification PEARL and its components an object oriented perspective - to structure them in a way, which would lead to natural generalisation-specialisation and whole-part relationships and define their interfaces. It will also show why it would be convenient for Specification PEARL to support classes of objects.

Key words: real-time systems, co-design, object-orientation, PEARL.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35599-3_29

1. INTRODUCTION

Since the complexity of most automation and real-time processing applications requires the programming of distributed, fault-tolerant multiprocessor systems, PEARL has been extended with constructs for the programming of multiprocessors. In PEARL for distributed systems [6] the language is enhanced with constructs, which enable the abstract description of hardware and software. These enable the real-time embedded systems to be co-designed in order to increase their dependability and quality. They are not translated into machine code; instead, they are treated as directives for system programs (e.g.: configuration management programs, loaders, etc.).

The features of the specification language are:

- constructs for the description of hardware configurations,
- constructs for the description of software configurations,
- constructs for the specification of communication and its characteristics (peripheral and process connections, physical and logical connections, transmission protocols), as well as
- constructs for specifying both conditions and the method of carrying out dynamic reconfigurations in cases of failure.

The latest revision of PEARL carries the name PEARL90 and is still a procedural language. Some of the research on enhancing the predictability, safety and introducing object orientation into the PEARL language is described in [1, 3, 7]. This process was difficult because it was hard to sustain and improve the timely deterministic properties of PEARL together with the introduction of objects since by their dynamic allocation they are imposing non-determinism into the execution of PEARL programs. On the other hand the introduction of processes is not similar to C++ or Java threads, because PEARL TASKS have a different role in the program structure and additional scheduling parameters, requiring different status and handling of TASKs.

PEARL for distributed systems [6] has not been extended in an object-oriented fashion so far although it has the appropriate structure. It was addressed and in some extent further developed in the research of [1,3] with the intent to improve the safety of PEARL program execution.

In Specification PEARL [2] the standard has been extended in the foreseen manner in order to support the description of asymmetrical architectures as well as the parameterisation of the RTOS and schedulability analyser. It was meant to be used with the latest revision of PEARL. It was extended in its textual version and a corresponding graphical notation has been defined, used to build a CASE tool for the visual creation of specifications.

The description in Specification PEARL syntax consists of divisions, which describe different associated layers of the system design in considerable detail: station division, configuration division, net division and system division. The constructs from these layers may seem very disjoint on the first glance, but have common properties and can be structured in an object-oriented manner. In the following sections the structure of these layers and the properties of their constructs is given together with the idea on how to implement and use them in an object-oriented manner.

The article concludes with the description of the expected benefits from utilising the described HW/SW co-specification classes together with an idea on how to integrate them with current advances in real-time object-oriented languages and applications.

2. STATION DIVISION

Stations, being the processing nodes of the system are introduced here, stating their role in the distributed system and their structure. They are treated as black boxes with connections for their information exchange.

Several types of stations have been defined. Default is the "BASIC" station, which represents a general-purpose processing node. To be able to describe asymmetrical architectures, two additional types of processing nodes have been defined: "TASK" for application processors and "KERNEL" for operating system (kernel) processors. Since in embedded systems design (intelligent) peripheral devices are very important the "PERIPHERAL" station type was defined. A multiprocessor node is introduced by the "PART OF" attribute of the constituent processing nodes.

All stations have common properties of the "BASIC" station. Depending on their role in the distributed system they may have additional properties which leads to a good opportunity to derive a station from a similar station using the generalisation-specialisation concept. Because of multiprocessor nodes and station components there are also whole-part relations here. The class structure of the constructs from the station division is depicted in Figure 1 and their properties are given in the framed boxes below.

Each station in a system is associated with the state information for reconfiguration purposes. The basic components of a station are its processors (PROCTYPES), WORKSTORES and DEVICES.

The station division also carries information about the intelligent peripheral devices, attached to the system, which also have the role of stations, although they represent the inputs/outputs of the modelled system. Their connections to the devices in the system are described by the attributes of the peripheral station's components (e.g.: the direction of data flow, the protocol used and any additional signals which may be necessary for the communication). To support schedulability analysis every signal may be assigned its minimum inter-arrival time.

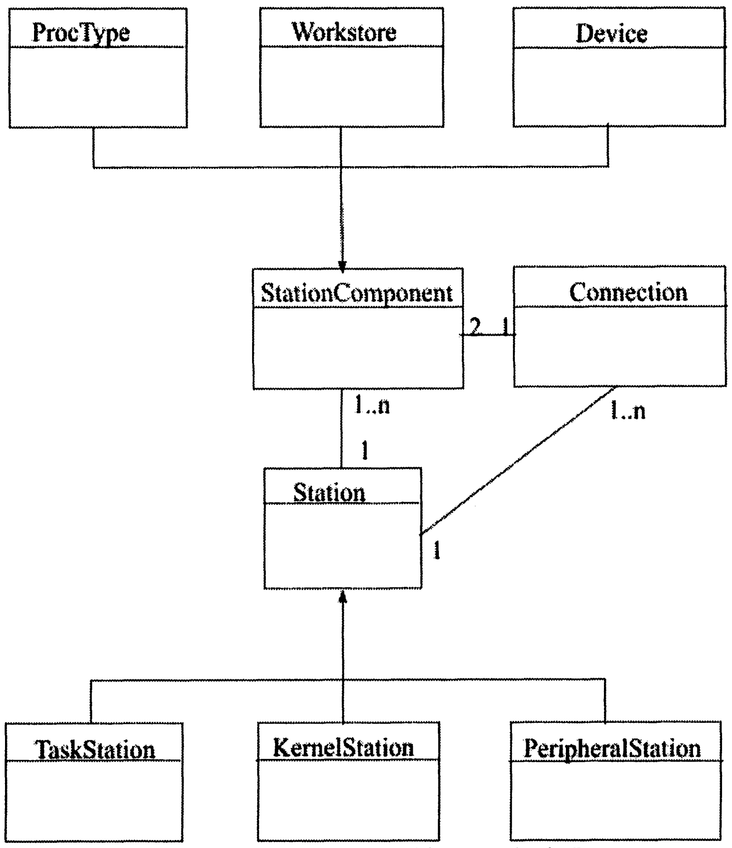

Figure 1: UML Station division class structure with properties in Java syntax

```
public class Station {
    StringBuffer name;
    String type={"BASIC", "KERNEL", "TASK", "PERIPHERAL"};
    boolean partOf;
    StringBuffer superStation;
    StateType states[];
    StateType stateRegister;
    StationComponent components;
    Connection connections;
}
public class PeripheralStation extends Station {
    int minStimuliPeriod;
}
```

```
public class TaskStation extends Station {
    StringBuffer supervisorName;
    Station supervisor;
}
public class KernelStation extends Station {
    int minTimeResolution;
    String schedulingPolicy={"RR","EDF","MLF","RM"};
    int minKernelRAM, minProcessRAM, minTaskRAM;
    int minIsrRAM, minQueueRAM;
    int maxTask, maxSema;
    int maxTaskSwitchTime, maxIntLatency;
    int maxQEvent, maxSchedEvent;
}
```

The station component's properties are limited to basic and timing information. They are uniquely identified by their IDs, which may be their HW device identifiers or logical names. Insignificant implementation details are omitted.

Station processors (PROCTYPEs) have speed descriptors, which tell their clock generator's frequencies. It is in general possible to generate multiprocessor stations and mix processors with different clock rates within the same station.

WORKSTORES are described by sizes and memory maps (they show the purpose of different areas of memory). The access time the different memory areas may also be specified (on-chip, RAM or ROM memories typically have different access times). This information is used by the compiler to determine the maximum execution times of tasks, which are loaded in these memory areas or access them during their execution.

DEVICES are identified by IDs (like stations, but they may be assigned a logical name for easier reference). The device types may vary and have different attributes assigned depending on their nature. Currently, interfaces, timers and shared variables are supported. The use of specific standard devices is supported through the generic device specification.

```
public class StationComponent {
    StringBuffer componentID;
    StationComponent NextComponent;
}
class Proctype extends StationComponent {
    int processorSpeed;
}
class Workstore extends StationComponent {
    int startAddress;
    int memoryAreaSize;
    boolean dualPort;
    String accessType={"RAM", "ROM", "XOM"};
    int accessTime;
}
class Device extends StationComponent {
    StringBuffer DeviceID;
    int baseAddress;
}
class Interface extends Device {
    StringBuffer driverID;
    int driverStartAddress;
    String dataDirection={"IN", "OUT", "INOUT"};
    String transferType={"DMA", "PACKAGE"};
    int transferSpeed;
    int packageSize;
    int intVector;
    int intLevel;
}
class Timer extends Device {
    int timerActivation;
    int timerPeriod;
    int timerDuration;
}
class SharedVariable extends Device {
    StringBuffer name;
    String signalTriggerCondition={"CHANGE", "EQUAL",
" GREATER", "LESS"};
    int referenceValue;
    int comparisonRegister;
}
public class Connection {
    StringBuffer connectionID;
    int connectionSpeed;
    int bandWidth;
    StationComponent endPoint1, endPoint2;
    Connection nextConnection;
}
```

Standard devices are identified only by their identifiers. Their behaviour is assumed to be known. Like in Full PEARL [4, 5] it is assumed that we have a database of standard devices with their relevant properties.

2.1 Configuration division

In the configuration division the software architecture is described. The largest executable program, which can be loaded to a station depending on its state is a "COLLECTION" of "MODULEs". It is also possible to specify under which conditions certain collections are removed from a station and which collections are loaded instead. These conditions are station state dependent.

Modules consist of "TASKs", which may communicate through "PORTS". Each program part has its unique name for reference. Modules are further described by their import and export definitions, in which it is stated, which data structures and tasks are shared with other modules.

Tasks are described by their trigger conditions and response times. Task alternatives, which serve the purpose of increasing fault-tolerance and enhance the feasibility of task scheduling, are given (during scheduling an alternative task with shorter run time or longer requested response time can be scheduled in order to maintain the feasibility of the schedule).

The connections between the ports are described by their directions and line attributes. Line attributes state which connections from the station division are used for the communication. It is stated which thereof are always followed ("VIA" attribute) and which can be chosen from a list based on the "PREFER" attribute.

The relations of configuration division components are depicted in Figure 2 and their properties are given in the framed box below. The software configuration constructs are given here with their mapping and references to the constructs from the station division.

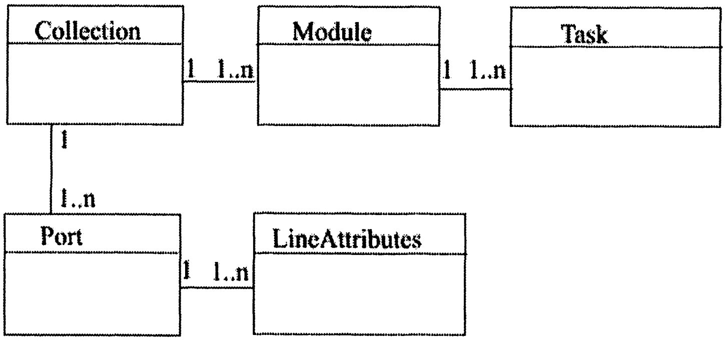

Figure 2: UML Configuration division class structure with properties in Java syntax

```
public class Collection {
    StringBuffer collectionID;
    Station station;
    StringBuffer state;
    Port ports;
}
class Module {
    StringBuffer moduleID;
    Collection collection;
    StringBuffer import;
    StringBuffer export;
}
class Task {
    StringBuffer taskID;
    Module module;
    String triggerCondition={"ON_DEMAND", "TIMER", "INT",
"SIGNAL", "SEMA"};
    int deadline;
    StringBuffer altTaskID;
}
class LineAttributes {
    Connection line;
    String attr[]={"VIA","PREFERED"};
    LineAttributes nextLine;
}
class Port {
    StringBuffer portID;
    String dataFlow={"IN", "OUT", "INOUT"};
    String transferType={PACKAGE, DMA};
    int bytesInPackage;
    String syncMechanism={"SEND-REPLY", "NOWAIT-SEND",
"BLOCKING-SEND"};
    LineAttributes lines;
    Port nextPort;
}
```

2.2 Net division

Any net topology of a distributed system can be described by point-to-point connections. Net division describes the physical connections between the station's components of the system by their logical names and directions.

The net division should become obsolete with the use of direct references to connections between the station division components and configuration division PORT mappings.

2.3 System division

System division encapsulates the hardware description and the assignment of symbolic names to hardware devices for their easier reference from the program or the net division. The described components from the station division are used.

By the object naming scheme this division would also become obsolete.

3. THE EXPECTED BENEFITS OF USING OO TECHNIQUES WITH SPECIFICATION PEARL

The dual representations (textual and graphical) of the same specification are difficult to manage and they introduce superfluous work for the designer who wants to transform the design from the desktop drawing into code. Hence it would mean a simplification if the constructs would represent target objects. Both net and system divisions are mainly present for backward compatibility to enable the output of textual specification code if needed. Otherwise the CASE tool would probably generate (references to) specification classes and objects in the target implementation language syntax.

The possible incompatibility of parameters is checked during the design process or while creating the architecture description in the Specification PEARL syntax. The modelled system is checked for completeness and parameter compatibility, since for subsequent design steps (e.g. SW/HW mapping), schedulability analysis or target language implementation a coherent, unambiguous description is needed. These checks would be simplified by the verification of the connections between the specified component objects during their creation.

Specification PEARL as a specification language needs a configuration manager or loader to be able to influence the loading scheme and execution of programs in a distributed system. Recently the specification for RT-Java and its reference implementation of the JVM have been issued [8]. In addition to previous releases they also incorporate the classes and attributes for RT operation. JVM is an ideal example of a configuration manager. Hence and since Java is an object oriented language it would be sensible to define Java classes for Specification PEARL constructs and use them to access resources of a distributed system through the JVM.

4. CONCLUSION

The method for real-time system's HW/SW co-specification Specification PEARL enables parallel design of the hardware and software parts and offers textual as well as graphical notations. For PEARL programmers it provides a good way to design the SYSTEM part of their program. On the other hand it is not strictly bound to PEARL and its output can be used as input of a configuration manager or loader. As mentioned in the pervious section the object-oriented Specification PEARL could be used with an object-oriented real-time language like RT Java. Its use in a CASE tool would be more straightforward than currently with the dual graphical and textual representations, although for compatibility reasons the possibility of the textual output should be retained.

Specification PEARL enables early reasoning about the system integration, but at the same time its hierarchical structure also enables top-down stepwise refinement in design. The designer first sets up the logical structure of the system, which is being detailed with time as well as implementation parameters. When at least the logical hardware architecture is set up, software units (COLLECTIONs) may be associated with it. The shell of the hardware architecture and the interconnections are sufficient to logically map the software onto hardware. The design can also be started from the software point of view and the mapping can then be done subsequently, when the stations are present.

Currently only the implementation of Specification PEARL for use with PEARL90 already exists, however with the advent of the RT Java specification with its reference implementation and the described specification language implementation, it seems to be a good idea to implement Java classes for Specification PEARL components. This would also ease the implementation of specifications and provide them with the appropriate loader and configuration manager.

REFERENCES

[1] A. H. Frigeri, W. A. Halang. Eine objektorientierte Erweiterung von PEARL 90. PEARL 97 - Workshop ueber Realzeitsysteme, Boppard, Germany, November 1997.

[2] R. Gumzej. Embedded System Architecture Co-Design and its Validation. Doctoral thesis, University of Maribor, Slovenia, 1999.

[3] W.A. Halang, C.E. Pereira and A.H. Frigeri: Safe Object Oriented Programming of Distributed Real Time Systems in PEARL. Comput. Syst. Sci. & Eng. (2002) 2: 85-94.

[4] Basic PEARL, DIN 66253, Part 1.

[5] Full PEARL, DIN 66253, Part 2.

[6] Multiprocessor PEARL, DIN 66253, Part 3.

[7] D. Verber, Object Orientation in Hard Real-Time System Development. Doctoral thesis, University of Maribor, Slovenia, 1999.

[8] The Real-Time Specification for Java (rtsj-V1.0), Addisson-Wesley, 2000, http://www.rtj.org.