Real-Time Syst (2007) 35:181–208
DOI 10.1007/s11241-006-9007-9

Modeling distributed real-time applications with specification PEARL

Roman Gumzej · Shourong Lu

Published online: 6 December 2006

Abstract The methodology of hardware/software co-design of embedded control systems with Specification PEARL is presented. Hardware and software are modeled with the language Specification PEARL, which has its origins in standard Multiprocessor PEARL. Its usefulness is enhanced for modeling hierarchical and asymmetrical multiprocessor systems, and by additional parameters for schedulability analysis. Graphical symbols are introduced for its constructs to enable graphical modeling while maintaining the semantical background. It is meant to be a superlayer for programs, based on the PEARL programming model. To model program tasks, Timed State Transition Diagrams have been defined. The model of a co-designed system is verified for feasibility with co-simulation. The resulting information should be used when considering changes in a current design with the goal of producing a temporally feasible model. To support dynamic re-configurations, configuration management is introduced into the models. Since UML is becoming a de facto standard also for designing embedded control systems, and since Timed State Transition Diagrams and State Chart Diagrams share great similarity, an interface of the methodology to UML 2 is defined, using UML's extension mechanisms.

Keywords Specification languages · Application modeling · Co-design · Co-simulation · Configuration management · PEARL · UML

R. Gumzej (✉)
Faculty of Electrical Engineering and Computer Science, University of Maribor, 2000 Maribor, Slovenia
e-mail: roman.gumzej@uni-mb.si

S. Lu
Chair of Computer Engineering, Fernuniversität in Hagen, 58084 Hagen, Germany
e-mail: shourong.lu@fernuni-hagen.de

1 Introduction

For real-time systems timeliness and safety issues are just as important as functional correctness. Hence, they should be designed holistically, considering all their components and functional properties, as well as their subsequent verification.

To enable verification, often formal languages and/or mathematical notations are used, for which subsequently a proof can be worked out (e.g., formal languages and timed automata (Agha, 1991), graphical techniques with the same expressive power as their formal language counterparts (Dietz, 1997), and combinations of conventional CASE methods and statecharts (Traore et al., 1997)). While enabling formal verification, most of these methods lack the versatility of basic constructs and user friendliness. Therefore, graphical formalisms with richer sets of basic constructs have been defined (e.g., CSR/CCSR by Lee et al., 1991, and GCSR by Abdallah, 1996, TTM/RTTL by Ostroff, 1997), while keeping enough "strictness" to enable verification. Dedicated state transition automata like CRSM (Shaw, 1992) are often used as basic internal computation models (e.g., POLIS (Balarin et al., 1997)).

An example for the wide variety of verification methods, which can be used for embedded real-time systems, is the VHDL language, for which verification methods have been devised, ranging from formal methods to simulation with fault insertion and combinations thereof (e.g., Khalil et al., 1998). For pragmatic reasons and to avoid combinatorial explosion with formal verification or exhaustive testing, simulation is often used to check the correctness of a designed system or parts thereof. Co-designing systems with time limitations led to the introduction of real-time scheduling strategies into their co-design and co-simulation (e.g., Mooney, 1998).

Some novel object-oriented design techniques have been devised for different aspects of real-time systems design (cp. Henzinger et al., 2003; Hylands et al., 2003; Licht, 2004; Gausemeier et al., 2004; Schröter et al., 2003; Bitsch et al., 2005), which are partly based on UML and its real-time profiles RT-UML (Douglass, 1999) and UML-RT (Selic and Rumbaugh, 1998). To explicitly address real-time issues in UML 2, the *Profile for Schedulability, Performance and Time* (OMG, 2005) has been defined by OMG.

In this article, the features of the Specification PEARL (S-PEARL) hardware/software co-design methodology are presented, namely, the specification language and notation, which represent hardware/software architectures, timed state transition diagrams, which represent the program tasks of a real-time application, configuration management for dynamic system (re-) configuration, and co-simulation to check the temporal feasibility of designs.

After the description of the S-PEARL modeling approach, it is shown how an interface of the S-PEARL methodology to UML (OMG, 2004) can be built. Since UML, being a prominent methodology for designing information systems, is also used to design embedded systems, an interface from Specification PEARL is defined to enable S-PEARL-like design in UML. Combining both methodologies could enable larger-scale S-PEARL-oriented design of real-time systems in combination with UML's versatile diagrammatic features.

```
ARCHITECTURE;
STATIONS;
 NAMES: KP;
   PROCTYPE: MC68370  AT 20 MHz;
   WORKSTORE: SIZE 65536  SPACE 0 - 'FFFF'B4
     READ/WRITE  WAITCYCLES 1;
   WORKSTORE: SIZE 32768  SPACE 0 - '7FFF'B4
     READONLY  WAITCYCLES 1;
   INTERFACE: KP_IO (DRIVER: KPINOUT;
     DIRECTION: INPUT; SPEED:20971520 BPS;
     UNIT:FIXED);
   STATEID: (NORMAL, CRITICAL);
   STATIONTYPE: KERNEL;
   SCHEDULING: EDF;
   MAXTASKS: 20;
   MAXSEMA: 5;
   MAXEVENT: 15;
   MAXEVENTQ: 5;
   MAXSCHED:30;
   TICK: 1E-3 SEC;...

   SYSTEM;
    NAMES: KP;
       KP.KP_IO INOUT;
      NAMES: Sensor 1;
        Sensor1.S1 OUT;
      NAMES: Sensor 2; ...
      NAMES: TP1;
        TP1.S1 IN;
        TP1.TP1_IO INOUT;
      NAMES:TP2; ...
   SYSEND;
```

```
CONFIGURATION;
  COLLECTION KP_WS;
    PORTS KP_TP1-lin, KP_TP2-lin;
    CONNECT KP_WS.KP-TP1_lin INOUT TP1-WS.TP1_KP_lin
      VIA KP.KP_IO;
  CONNECT KP_WS.KP_TP2_lin INOUT TP2_WS.TP2_KP_lin
      VIA KP.KP_IO;
  COLEND;

COLLECTION TP_WS;
  PORTS S1, TP1_KP_lin;
  CONNECT TP1_WS.S1 IN VIA TP1.S1;
  CONNCET-TP1_WS.TP1_KP_lin INOUT
    KP_WS.KP_TP1_lin VIA TP1.TP1_IO;

  MODULES TP1_WS_M1;
    EXPORTS(Side 1);
    TASK Side1
      TRIGGER PORT S1;
      DEADLINE 100;
    TASKEND;
  MODEND;
COLEND;...

CONFEND;
ARCHEND;

  NET;
   KP.KP_IO <-> TP1.TP1_IO;
   KP.KP_IO <-> TP2.TP2_IO;
   TP1.TP1_IO<-> Sensor1.S1;
   TP2.TP2_IO<-> Sensor2.S2;
  NETEND;
```

Fig. 1 Example of an S-PEARL textual architecture description based on Multiprocessor PEARL (note additional parameters)

2 Specification PEARL methodology

The Specification PEARL co-design methodology is based on the notation of standard Multiprocessor PEARL (DIN 66253, Part 3 Multiprocessor PEARL, 1989), where a textual (e.g., Fig. 1) system architecture description consists of divisions, which describe different aspects of a system's design. It enables the construction of conceptual system models, whereby the hardware and software architectures may be described in parallel. A system model is built and can be checked for coherence with the Specification PEARL methodology by running the associated CAD tools (cp. Gumzej, 2004). For more information on the constructs, their properties and notation of the Specification PEARL language cp. (Gumzej, 1999). Benefits of using the S-PEARL methodology are the abilities to

- reason on system integration at an early stage,
- introduce timing constraints wherever applicable into a design, and
- check the temporal feasibility of a design before implementation.

2.1 Modeling of software/hardware architecture

In the mentioned methodology, hardware and software architectures are described conjunctly.

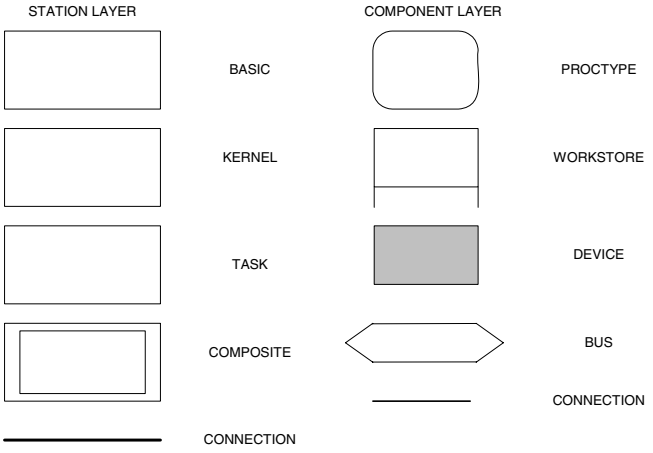


Fig. 2 Hardware architecture constructs of S-PEARL

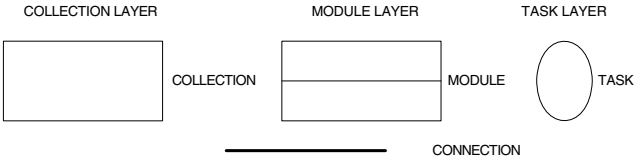


Fig. 3 Software architecture constructs of S-PEARL

A hardware model consists of STATIONs, being the processing nodes of the system. Their components (see Fig. 2) are chosen from a list of general components such as processors, memories, or interfaces. These components determine the structure of stations and, on the other hand, they also represent their resources and provide the necessary timing information for schedulability analysis.

A software model is composed of COLLECTIONs, which are mapped to the STATIONs of the hardware model, depending on their state information (see Fig. 3). They consist of MODULEs of TASKs. 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.

To administer collections, a Configuration Manager (CM) is required, which forms a layer between the real-time operating system (RTOS), if any, and the applications. Its rôle is to function as (1) a hardware abstraction layer, (2) a hardware/software interface, and (3) as an "Inter-Collection" co-operation agent. In order to form executable units, the CM is joined with the application model and compiled for the target platform, or additionally joined with the co-simulation environment for temporal feasibility checking (as shown in Fig. 4).

2.2 Task modeling

According to (Mok, 1991), the computational model of most applications running in the real-time mode can be written in the form of the following "equation":

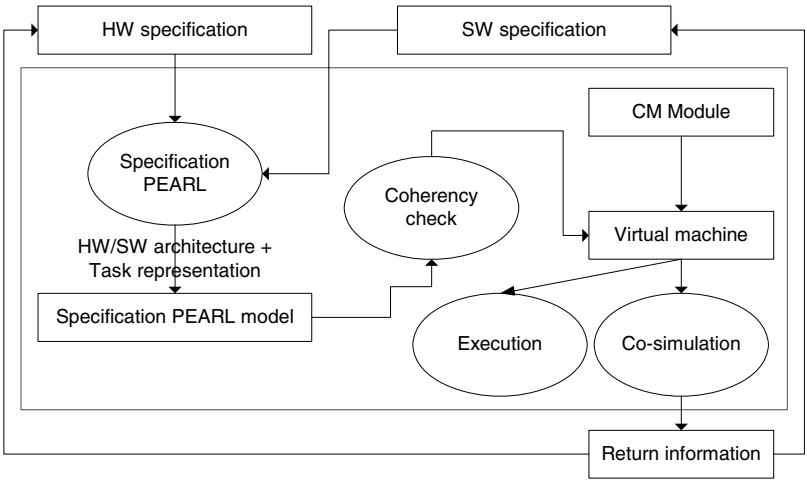


Fig. 4 Interrelations between hardware/software models

$$\begin{aligned} &\textit{Real} - \textit{time program model} \\ &= \textit{Dataflow model} + \textit{State automaton} + \textit{Timing limitations} \end{aligned}$$

The tasks of an application represent the processes in a running system. They are mainly characterised by activation conditions and timing limitations as well as by being part of certain collections. This information is sufficient to build a coarse program model, but it is not enough to determine its feasibility. Therefore, timed state transition diagrams have been introduced to represent them (cp. Gumzej and Colnarič, 2003). Their synchronisation and inter-communication are realised by calls to the configuration manager and/or the real-time operating system of the station executing the task.

Task State Transition Diagrams (TSTD) are hierarchical finite state automata consisting of

start states: task activation conditions and initialisation actions,
working states: atomic activities with possibly predictable duration,
super states: non-atomic activities – hierarchical decomposition of working states, and
final states: finalisation actions.

Every state contains the following data:

state type: start-, working-/super- and final state,
pre-condition for the state's execution: activation condition in case of a start state,
time frame: shortest and maximum execution times,
timeout action: the action executed in case a state exceeds its time frame,
connection(s) to the next state(s) in case the execution continues successfully, and
activities carried out within the execution of this state: program code with PEARL system calls.

The connections between states represent the progress of tasks in time. All connections are local (i.e., bound to the states of one task). Inter-task co-operation is modeled by the state actions, i.e., system calls to the operating system, through the configuration manager. These also trigger the continuation pre-conditions of the states. Operating system and configuration manager are visible to the designer through their system calls (API) and system configuration, which is defined by setting the parameters of the corresponding STATION, only.

Trigger conditions differ slightly depending on the state type. Only start states have the possibility of explicit (on-demand) activation. Other state types rely on the following types of pre-conditions:

external events (int(no)), representing interrupts,
internal events (sig(no)), representing signals,
timers (timer(at,every,during), representing timer signals,
general conditions (cd(expression)), i.e., expressions returning Boolean results from the evaluation of internal system/program states or data structures of the operating system.

Only final states may progress automatically (without a pre-condition; upon successful completion control is returned to (an) initial/start state(s)). Upon fulfilling the pre-condition of a superstate, control is automatically transferred to the start state of its subchart. If the condition to proceed to the next state is fulfilled by the maximum time frame (*maxT*) of the current state, the corresponding connection for successful continuation is followed. If a minimum time frame is foreseen (*minT*), it is not checked whether the continuation conditions are fulfilled, before the specified time has elapsed. The timeout condition is set to the maximum time frame at the beginning of each state's execution. In case a timeout occurs before the requested resources are available (the next state's pre-condition is fulfilled), the appropriate on-timeout action is executed. If it is not specified, an error has occurred (and is logged in the co-simulation). The activities within a state are a set of actions, which are carried out while a task is in this state. It is assumed that the actions form a single block of program statements including system service calls to the operating system and/or configuration manager, around which the control structure is formed by transforming the chart to program code. Their execution times are estimated by the designer and used in setting the time frames for each state.

2.2.1 Guidelines for task modeling

The rôle of a "task" is the same in the Specification PEARL methodology as it is in the programming language PEARL (DIN 66253 Parts 1, 2; 1981, 1982)—any procedure, which needs to be carried out within a given time frame, is a task. The problem in trying to break down the operations of tasks into states is that simple tasks have only three states: start, working and final. New states are only introduced (1) if a time-limited atomic (sub) operation is identified, (2) if synchronisation or communication between tasks is necessary, or (3) to define branching into different continuation paths depending on the pre-conditions of successor states. The following criteria were selected to form task states:

- a state represents a single logical activity, which is only dependent on its pre-conditions and whose execution time can be determined or predicted,
- any task must have at least one start-state, one or more working/superstates, and final states,
- in order to ease good decomposition, a complex operation shall be broken down into individual states by introducing a superstate on the current level and defining its operations in a subchart.

2.2.2 Translation from TST-diagrams to program tasks

TSTD task models are translated to program tasks in two forms:

1. target platform oriented, as it can be compiled by a corresponding compiler and executed on a specified hardware architecture, and
2. simulation oriented, as it is used and interpreted by co-simulation in a simulation environment.

The main difference between the two forms is the way external events are handled. In the first case, they are generated by the environment and handled as hardware interrupts (by the interface device drivers), whereas in the second one, they are generated in the co-simulation environment and handled as software signals (by the stub device drivers) through the RTOS.

The forms of diagram representation and storage as well as the mechanism to translate TST diagrams to program task prototypes are discussed and illustrated throughout the following example. The general form of task prototypes, obtained from TSTDs, is shown in Fig. 5.

2.2.3 Example of a translation from TSTD to task

This example is to illustrate the use of TSTDs on a simple task model to control a traffic light at a pedestrian crossing. The traffic light is controlled by a simple logic with the output to set the lights and a relay switch to request pedestrian crossing. The execution according to Table 1 is cyclic and can be represented with the series of steps as depicted in Fig. 6. While starting at Step 1 initially, the series starts upon pressing the button for pedestrian crossing, or after 3 min after the last cycle. The following demands and restrictions need to be observed: Step 1 is delayed 180 sec (unless the pedestrian relay switch is pressed beforehand), Step 2 is delayed 10 sec, Step 3 is delayed 30 sec, Step 4 is delayed 20 sec, and the execution continues at Step 1. The initial conditions are: red light is turned on for pedestrians, green light is turned on for cars (Step 1), and the relay switch is reset.

The TSTD textual representation shown in Fig. 6 consists of fragments of initialisation code for each of the TSTD states. A “Pre-condition” represents the trigger condition for its state. As mentioned previously, there are four types of pre-conditions, which can be used here. The *maxT* and *minT* parameters have the purpose to determine the time frame for a state’s execution. The “Action” and “OnTimeout” parts are represented by program statements, PEARL system calls and comments, which are delimited by the “END;” keyword.

```
MODULE module_name;
SYSTEM;
! interrupts, signals and system variables definitions
PROBLEM;

task_name : PROCEDURE (state_id REF INT);
  DCL timeout BIT;
                                  /* initialisation of all global structures
  timeout:='0'B1; ! timeout indicator
  WHILE '1'B1 REPEAT
    CASE state_id
      ALT (0) ! START state:
        IF timeout EQ '1'B1 THEN
                                      /* perform OnTimeout=action(s);
          state_id:=0; timeout:='0'B1; NEXT state_id;
        ELSE
                                /* RESUME task after fulfillment of the
          timeout:='1'B1;
          DELAY maxT;
                                  /* perform the appropriate start states
                /* check if any of the next working / super / end states'
                            /* if they are fulfilled, set timeout to false and
          NEXT state_id;
        FIN;
      ALT (1) ! for a WORKING state:
        IF timeout EQ '1'B1 THEN
                                      /* perform OnTimeout=action(s);
          state_id:=0; timeout:='0'B1; NEXT state_id;
        ELSE
          DELAY minT;
          timeout:='1'B1;
          DELAY maxT;
                                        /* perform Action=statements;
                /* check if any of the next working / super / end states'
                            /* if they are fulfilled, set timeout to false and
          NEXT state_id;
        FIN;
      ALT (2) ! for a SUPER state:
        IF timeout EQ '1'B1 THEN
                                      /* perform OnTimeout=action(s);
          state_id:=0; timeout:='0'B1; NEXT state_id;
        ELSE
                            /* set the state_id variable to the start state of
          NEXT state_id;
        FIN;
      ALT (3) ! for a SUPER state (as addition) - "return to" state:
                                  /* set the state_id variable to the next state
                /* for a SUPER state (the sub-diagram states numbered
          NEXT state_id;
      ALT (n) ! END state:
          DELAY minT;
                                        /* perform Action=statements;
          state_id:=0;
                              /* reset task state (state_id = super_state + 1)
          NEXT state_id;
          timeout:='0'B1; ! reset timeout indicator
    FIN;
  END;
END;

MODEND;
```

```
#include "module_name.h"

/* interrupts, signals and system variables definitions */

void task_name(int &state_id;) {
  bool timeout;
*/
  timeout=false; /* timeout indicator */
  while (1) {
    switch (state_id) {
      case 0: { /* START state: */
        if (timeout) {
*/
          state_id=0; timeout=false; Next(state_id);}
        else {
trigger conditions; */
          timeout=true;
          Delay(maxT);
Action=statements; */
pre-conditions are fulfilled; */
set the state_id variable accordingly; */
          Next(state_id);
        } }
      case 1: { /* for a WORKING state: */
        if (timeout) {
*/
          state_id=0; timeout=false; Next(state_id);}
        else {
          Delay(minT);
          timeout=true;
          Delay(maxT);
*/
pre-conditions are fulfilled; */
set the state_id variable accordingly; */
          Next(state_id);
        } }
      case 2: { /* for a SUPER state: */
        if (timeout) {
*/
          state_id=0; timeout=false; Next(state_id);}
        else {
the super state's sub-diagram; */
          Next(state_id);
        } }
      case 3: { /* for a SUPER state (as addition) - "return to" state: */
*/
consecutively) */
          Next(state_id); }
      case n: { /* END state: */
        Delay(minT);
*/
          state_id=0;
in case of a return from a sub-diagram */
          Next(state_id);
          timeout=false; /* reset timeout indicator */ }
    }
  }
}
```

* DELAY and NEXT are there for simulator control. DELAY represents a "busy wait'" while NEXT instructs the simulator to dispatch ("preemption point").

Fig. 5 Representations of a TSTD in PEARL and annotated C

While the graphical TSTD representation is mainly meant to be used in CASE design tools, the textual representation is meant for storage and automatic translation to a program (source code) prototype (cp. Fig. 5).

For easier interpretation/compilation of program tasks in/to the simulation/target platform environments, in the current implementation of the Specification PEARL co-design environment (Gumzej, 2004). the annotated C programming language is used. The algorithmic form of task prototypes remained the same, merely the commands,

Table 1 Logical execution table for traffic light control

	Traffic light pedestrians		Traffic light cars		
Step	Red	Green	Red	Yellow	Green
1	1	0	0	0	1
2	1	0	0	1	0
3	0	1	1	0	0
4	1	0	1	1	0

representing system calls, were replaced by appropriate calls to the CM/RTOS of the Specification PEARL environment.

2.3 Development environment, configuration manager and operating system

Most co-design methodologies do not consider the target operating systems. Those which do, and produce executable code, use off-the-shelf operating systems, in which case the tools are strictly bound to the target environments, from which they also inherit their limitations in expressing the real-time properties of applications.

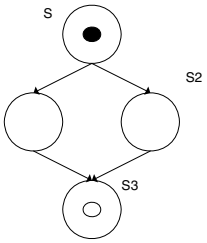
From this point of view, it appeared appropriate to use as orientation a real-time operating system with a rich set of system calls supporting real-time operation such as the PEARL RTOS (http://www.irt.uni-hannover.de/rtos/rtosfble.html) in order to support programmers and designers in expressing real-time behaviour of applications. Since our RTOS (cp. Gumzej, 1999) is not off-the-shelf, but rather a part of the designed system model, its source code can be compiled for any hardware platform.

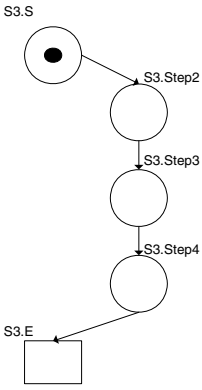
The S-PEARL development environment, which encompasses modeling and co-simulation tools, runs under Microsoft Windows 2000 and XP (cp. Gumzej, 2004). It enables cross-development for the defined hardware architectures through cross-compilation of its target platform models.

The Configuration Manager (CM) represents the hardware abstraction layer which is, as configured by a hardware architecture model, mainly used to define the structure, timing properties and interfaces of each STATION. The considered RTOS, being an optional part of the CM, supports the tasking model and system calls of the programming language PEARL as well as the deadline-driven scheduling strategy (later enhancements for other strategies are foreseen). Its resources are pre-determined (e.g., number of tasks, synchronisers, signals, events, or queued events) by setting the parameters of a KERNEL STATION.

2.3.1 Configuration manager's functionality

The execution at each processing node (station) starts with initiating the Configuration Manager (CM) object. Initially, it loads the collections of task objects, and activates the initial collection by triggering the latter's initialisation-task objects. In stations without a real-time operating system, the main task of the collection is started and delegated control to by the CM, whereas otherwise the CM acts as a front-end to






```
[State types]
START=0
TRANSIENT=1
SUPER=2
END=3

[States]
S=START
S1=WORK
S2=WORK
S3=SUPER
E=END

[S]
Precondition=
minT=0
maxT=0
Next=S1; S2;
Action= ! Step1 lights setting
END;

[S1]
Precondition=tm(180 SEC)
minT=0
maxT=0
Next=S3;
OnTimeout=END;
Action= END;

[S2]
Precondition=ev(button : INT)
minT=0
maxT=0
Next=S3;
OnTimeout= END;
Action= ! reset button
END;

[S3]
Precondition=
minT=0
maxT=0
Next=S;
OnTimeout= END;
Action=S3;
END;
```

```
[State types]
START=0
TRANSIENT=1
SUPER=2
END=3

[States]
S3.S=START
S3.Step2=WORK
S3.Step3=WORK
S3.Step4=WORK
S3.E=END

[S3.S]
Precondition=
minT=0
maxT=0
Next= S3.Step2;
OnTimeout= END;
Action=DISABLE button;
END;

[S3.Step2]
Precondition=
minT=10 SEC
maxT=
Next= S2.Step3;
OnTimeout= END;
Action= ! Step 2 lights setting
END;

[S3.Step3]
Precondition=
minT=30 SEC
maxT=
Next= S2.Step4;
OnTimeout= END;
Action= ! Step3 lights setting
END;

[S3.Step4]
Precondition=
minT=20 SEC
maxT=
Next= S3.E;
OnTimeout=END;
Action= ! Step4 lights setting
END;

[S3.E]
Precondition=
minT=
maxT=
Next=
OnTimeout= END;
Action=ENABLE button;
END;
```

Fig. 6 Example graphical and textual TSTD representations

the operating system functions, and uses appropriate system calls and system ports to transfer system requests to/from RTOS-enabled nodes to schedule the collection's tasks.

Besides local execution, the CM is also responsible for communication with other stations, and for co-operation among the tasks of the same collection. Hence, it must establish port-to-port connections through the interfaces of the station. Synchronisation and system service requests are serviced on the same station, in case the station is configured to run a real-time operating system. Otherwise, these requests are forwarded to the appropriate station through a proprietary port.

The application programming interface of the CM has the following functions:

(Re-) Configuration:

Cm_Init(*S*) – to initialise the station S and load the initial software configuration, and

Cm_Reset(*S*) – to restart the station with the initial software/hardware configuration.

Station state monitoring:

Cm_Getstate(*S*) – to retrieve the current state of station S, and

Cm_Setstate(*S*, *state*) – to change the current state of station S to "state".

Inter-station communication:

Cm_Transmit(*TCBi*, *portID*, *msg_buff*[]) – message transmission through a connection,

Cm_Reply(*TCBi*, *portID*, *msg_buff*[]) – response message transmission through a connection, and

Cm_Receive(*TCBi*, *portID*, *msg_buff*[]) – message receipt through a connection, where TCBi denotes the index of the task's control block (TCB), portID the name of the port, and msg_buff[] the buffer for the message.

The connections are established through ports of the software architecture and associated devices of the hardware architecture. The attributes of ports represent the communication parameters (smallest package, protocol, etc.) and routing parameters (VIA/PREFER). Routing affects the way the hardware communication devices are used. The attribute VIA determines the exact line to be used, while PREFER is usually assigned to the most trusted line in a list. Lines represent connections between hardware architecture devices (e.g., interfaces).

In asymmetrical architectures, direct calls to real-time operating system functions are not always possible. Hence, substitute RTOS API functions are called, to generate appropriate system request messages to the CM of the RTOS's processing (supervisor) node. The parameters of such system requests are extracted from the transferred messages in concordance with a pre-defined coding scheme also used in the construction of the parameter set.

To enable uniform handling of system requests, the RTOS API has been designed in a way enabling the transformation of system calls to parameter strings, which can be routed to the RTOS interface procedure directly, or sent to the KERNEL station for handling. Two additional internal functions have been introduced in the CM interface for this reason:

Cm_SysRequest(*S*, *sys_par*[]) – send system call parameters for processing to the RTOS, and

Cm_SysResult(*S*) – store result from the RTOS (result of the system call and possible context switch request for the local dispatcher routine of the CM (*CM_System*(*S*)).

3 Verification of specification PEARL system models

Verification of temporal feasibility of S-PEARL models is based on co-simulation with earliest deadline first (EDF) scheduling and time boundaries. It is primarily meant to check the timing properties of models designed for feasibility. A design is transformed into an internal representation for simulation, whose primary result is a successful execution or failure, whereas the secondary result is an execution trace, from which additional information can be extracted. This information is then used to discover bottlenecks and unreachable states, and to fine-tune the timing parameters of model and specifications. For a successful verification, after having carried out intermediate

checks of the system architecture, the model is subjected to the following coherency checks:

completeness check: all components are present and fully described,
range and compatibility check: parameter compatibility among components, and
software to hardware mapping check: complete coverage and consideration of resource limits.

These checks are in preparation to verify temporal feasibility, which is described in the forthcoming sections.

3.1 System model

The system model used is an internal representation of the system designed in form of "architecture data" and simulation nodes referring to parts of these "data". The hardware model consists of STATION nodes being the top-level simulation nodes with resources having specified properties. The software model consists of COLLECTION nodes, which are mapped to the STATION nodes based on their (initial) state. They are composed of TASK nodes, having the semantics of TSTD program representations. Co-simulation is based on the following presumptions.

- There is only one global simulation clock in the system, and all STATION real-time clocks (timers) relate to it (by perfect synchronisation).
- The time events relate to the corresponding station's real-time clock.
- Tasks are assigned deadlines for their execution (the only exception are short initialisation tasks).
- Task states (TSTD) are assigned time frames (minimum and maximum time) for the activities being performed within the states (in real-time clock time units).
- All simulation nodes are derived from a common simulation unit type.

3.1.1 Hardware model

There are three possible basic types of STATION simulation nodes: BASIC (general purpose – program and RTOS), TASK (program), and KERNEL (RTOS). They may have one or more communication lines attached to them for information exchange with other STATION nodes. The attributes of the stations' internal devices provide the parameters for the parameterisation of the stations' configuration manager (CM) and RTOS. A COMPOSITE STATION would merely represent a supersimulation node, composed of two or more STATION simulation nodes, hence only their constituent nodes take part in the co-simulation.

3.1.2 Software model

COLLECTION simulation nodes are linked as subnodes to associated STATIONs, whereas TASK simulation nodes are linked to their COLLECTIONs. For communi-

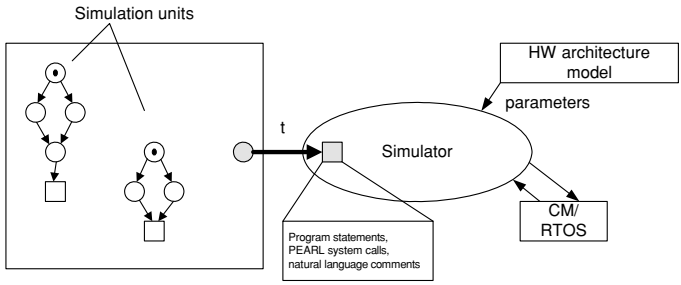


Fig. 7 The course of simulation

cation between tasks at different STATIONs the appropriate COLLECTION's PORTS are used. A STATION's CM determines when a certain COLLECTION is active and dispatches its messages accordingly.

3.1.3 Task (TSTD) model

TASKs are represented by Timed State Transition Diagrams (TSTD) (Gumzej, 1999, 2001), whose program representations are used to "drive" TASK simulation nodes. During simulation their execution is responsible for advances in time and state spaces.

3.1.4 Configuration manager and operating system model

The CM represents an inter-station/collection co-operation agent. It has information on the system software and hardware architectures, which originate from the S-PEARL system architecture specification.

Functionally, the CM/RTOS has the same rôle in co-simulation (Fig. 7) asin execution on the target platform. The main differences lie in the global real-time clock, which is maintained by the simulation environment, and the context switches, which are performed on the higher level only in the case of simulation (the context refers to task states – not processor registers).

Each RTOS processing node maintains a real-time clock. In a simulation environment, all these clocks are perfectly synchronised with the global simulation time, which in an execution environment should be implemented by an independent global time source and predictable time dissemination.

Pre-emption points in the simulation and in the target platform implementation are task state transfers. The resource access functions and interface device drivers of a STATION perform virtual functions in case of simulation, and concrete functions in case of target platform implementation.

The time required to execute the operating system itself (schedule and dispatch cycle) is assumed to be constant. The time needed to service the system calls is considered as being included in the time frame of the calling TASK's state. Their sole function is to change the state of internal data structures of the (operating) system and to trigger tasks and states, whose trigger conditions relate to them.

3.2 Verification of temporal feasibility

3.2.1 Criteria function

Any verification method requires the definition of a criteria function expressing when a system fails, i.e., what the limits of the "normal" execution of a system being checked are. The concept of correctness for the verification method, described here, is defined as follows: "A system fails if during co-simulation it reaches an undefined state, or if its pre-defined time frame is violated and no timeout-action is defined."

By trying the shortest and taking the longest transition times through the TSTD states, it is assumed that the time domain can be covered sufficiently to be able to generalise the results to an arbitrary transition instant within the *(minT, maxT)* interval for each state and, herewith, also the TASK as a whole.

3.2.2 Co-simulation with EDF scheduling

In simulation, the station clock rate is translated into the relative speed of the station, and may be used when the next event time is calculated (e.g., $rs_i = max\{s_i\}/s_i$ and $t_n = t_i \cdot rs_i$).

For verification, next critical event simulation and Earliest-deadline-first (EDF) scheduling are used. The next critical instant is always determined by the simulation unit whose activation time is the closest. This time is forwarded to all its parent units and, finally, becomes the next global critical instant. For each step it is checked, whether timing errors occurred. A "timeout" represents a controlled program fault, which is handled by the "timeout action", and by transition into the initial state. If this action is not defined for the current state, the system fails. Co-simulation EDF next-event scheduling is based on the following timing information (see Fig. 8):

A: task activation time,
R: accumulated task run time (updated with the next critical event),
E: task end time (the time when the normal task end is expected based on its maximum run time; upon a context switch the current time t_1 needs to be remembered, because to re-run the task this parameter needs to be reset based on the current time t_2 and the formula $E' = E + (t_2 - t_1)$),
D: task deadline (set, when A is known).

A task is re-scheduled when it is activated due to a scheduled event or on request. The task with the earliest deadline is chosen for execution, and its current state determines the next critical moment based on the current time t. The states of the tasks are executed atomically—a context switch is not performed before the task state is worked out. The

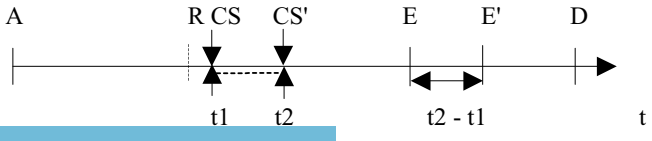


Fig. 8 Task run with a single context switch (see text for abbreviations)

RTOS scheduler is responsible for task scheduling (determining the most urgent task), while the simulator is responsible for determining the next critical moment for the current operation (task state or external event).

While re-scheduling, the following criteria (failure conditions) need to be checked for all active tasks:

$t < Z = D - (E - (A + R))$, where Z represents the latest time when the task needs to start/continue in order to meet its deadline;
$t < E \leq D$ must be true for all active tasks, since otherwise they have missed their deadlines.

Tasks can be scheduled to be executed on external events. For simulation purposes, they are assigned occurrence times. They are represented as native station unit events, whose next critical time instants are taken from an occurrence table, which lists interrupt numbers with their corresponding occurrence times. When they occur, they are handled by the station's RTOS (being a part of the CM), waking up appropriate tasks.

During co-simulation, the time of progression to the next state is calculated in two variants for each state:

1. $RTC + minT$ to check the pre-conditions, and
2. $RTC + maxT$ for transition to a new state.

If in critical instant (2) the pre-condition for the transition to any further state is not fulfilled, the system fails and the on-timeout action is executed. If it is not defined, the system fails. During simulation, the parameters E and D are set for each task when it is activated (the parameter A is set). When a critical instant is reached, it is checked if herewith the time frame, given for the task, has been violated, which results in the following consequences: (1) subtraction of the overhead from the task's slack time, or (2) the system fails as the task deadline is being missed.

The simulation results are logged during the execution of each simulation unit, and every step is accounted for within all parent simulation units, too. This means that every task state logs its action into the TASK log, whereas a task logs its state changes into the COLLECTION log. A collection logs the time when it was first allocated to a station, possible subsequent re-loads, and the changes of states which triggered them into the STATION log. The stations and collections also log the times when they were communicating among each other. All exceptions are logged, where they are discovered.

The temporal feasibility, determined by co-simulation, retains its validity if the execution times foreseen do not change when the software model is extended to fully functional programs.

4 UML profile for specification PEARL

The Unified Modeling Language (UML) provides constructs to deal with varying levels of modeling abstraction to visualise and specify both the static and dynamic aspects of systems (OMG, 2004). Its notation defines the semantics of an object metamodel to capture and communicate object structure and behaviour. In its metamodel

architecture, UML supports the following extension mechanisms: *stereotypes*, *tagged values* and *constraints*, which allow to tailor it to fit the needs of specific domains.

A UML profile is a pre-defined set of extension mechanisms for a particular domain, technology, or methodology, which provides a connection of how to apply and specialise UML to this domain. A *stereotype* provides a way to define virtual subclasses of UML metaclasses with additional semantics. It can set additional constraints over its base metamodel class as well as tags to define additional properties. A *constraint* is a semantic restriction represented as a text expression, which is usually formulated in the Object Constraint Language (OCL). Constraints are attached to one or more model elements. Tag definitions specify new kinds of properties as part of a stereotype definition. The actual properties of individual model elements are specified using *tagged values*.

The process of defining a general UML profile for a given platform or application domain can be summarised as follows:

- First, we need to define a set of elements that will comprise the platform or system, and the relationships between them, which can be expressed in terms of a metamodel, i.e., the metamodel includes the definition of the domain entities, the relationships between them, and the constraints that govern both structure and behaviour of these entities.
- Once the domain metamodel is built, we are ready to define the UML profile, in which a set of stereotypes are defined for each relevant element of the metamodel.
- Tagged values should be defined as attributes that appear in the metamodel. They include the corresponding types and initial values. The domain restrictions are expressed with constraints.

Here, we build a UML profile to describe the constructs and capture the essential semantic concepts of S-PEARL. In S-PEARL, STATION and COLLECTION are defined as basic entities in terms of the corresponding UML stereotypes, as shown in Fig. 9. The entities (stereotypes) will be used to define the class diagrams that specify the compositional model of S-PEARL, i.e., the structure and relationships between the model entities. The compositional model can be used for application frameworks built with S-PEARL elements.

4.1 Mapping S-PEARL architecture constructs to UML

As described earlier, S-PEARL includes elements to describe hardware and software configurations of distributed systems. To map the S-PEARL constructs onto UML elements, it is indispensable to compare UML and S-PEARL constructs, to be able to choose appropriate base elements and define UML stereotypes for S-PEARL elements. The essential point of this mapping is the S-PEARL architecture, its real-time features, and its runtime constraints. Figure 10 showsthe main defined stereotypes for S-PEARL.

4.1.1 Station

In S-PEARL, hardware and its deployment is introduced on the station layer. The processing nodes (stations) of a system are treated as black boxes with connections for information exchange. On their subordinate layers, stations are described by the

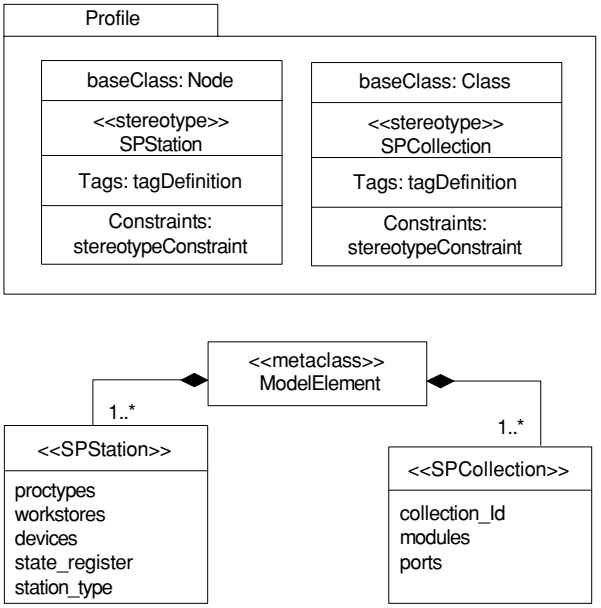


Fig. 9 S-PEARL profile and compositional model

S-PEARL Element	UML Element	Stereotype	Icon
Station	Node	<<SPStation>>	
Component	Class	<<SPComponent>>	Workstore, Device, Proctype
Line	Connector	<<SPLine>>	
Collection	Class	<<SPCollection>>	
Port	Class	<<SPPort>>	
Module	Class	<<SPModule>>	
Task	Class	<<SPTask>>	

Fig. 10 UML stereotypes for S-PEARL constructs

properties of their components, such as Proctypes, Workstores, or Devices. There may be many stations in a system, each one being uniquely identified, and equipped with an abstract state register variable for re-configuration purposes. Stations communicate among each other through the connections established, which are defined on the component layer by devices of type *interface* and referenced through *ports* from the software architecture.

A node in UML is a run-time physical element that represents a computational resource, which may be instantiated and stereotyped to be distinguished among dif-

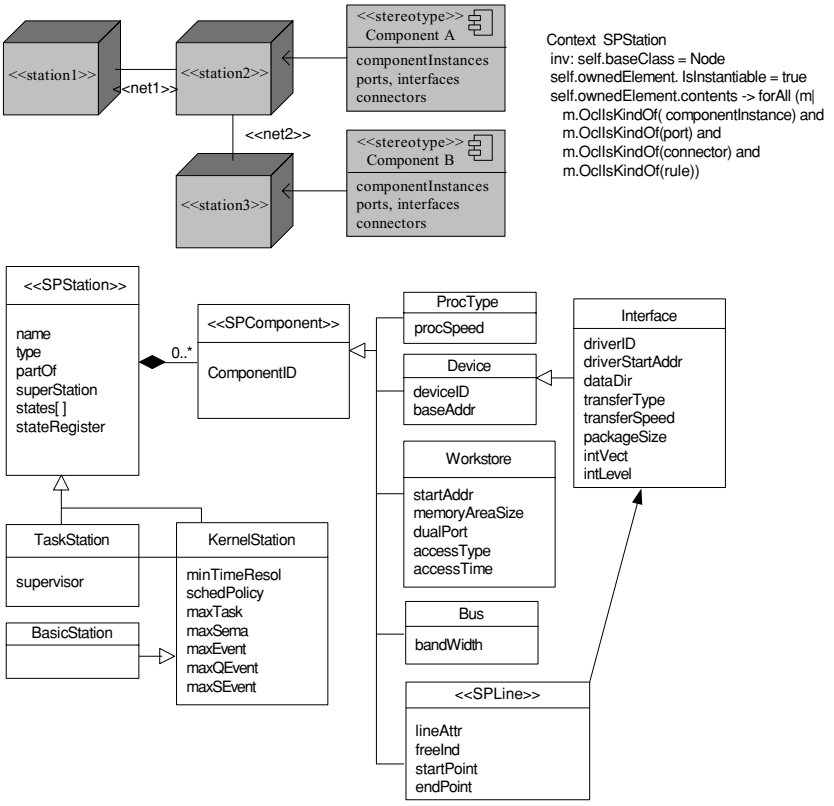


Fig. 11 Station stereotype (deployment diagram) and structure of the hardware part of "Architecture Data" (class diagram)

ferent kinds of resources. Associations among nodes represent their communication paths. They can be stereotyped to distinguish between different (types of) paths. Nodes have unique names. They may hold objects and component instances and represent the physical deployment of components. Therefore, it is natural to describe stations and net-connections from S-PEARL with nodes and their associations in UML, and define corresponding stereotypes for various types thereof. Figure 11 depicts a station in S-PEARL and defines it in terms of UML deployment and class diagrams.

4.1.2 Collection

In S-PEARL, collections are introduced as the largest separately loadable software components, which are assigned a state when they are active at the station, to which they are loaded. Collections are composed of modules of tasks. Communication between collections is performed by message exchange only, on the basis of the port concept. Upon station state change another collection is activated and the connections are re-connected. The collections, which are loaded to the same station, are grouped into "configurations". They are managed by the configuration manager (CM), which

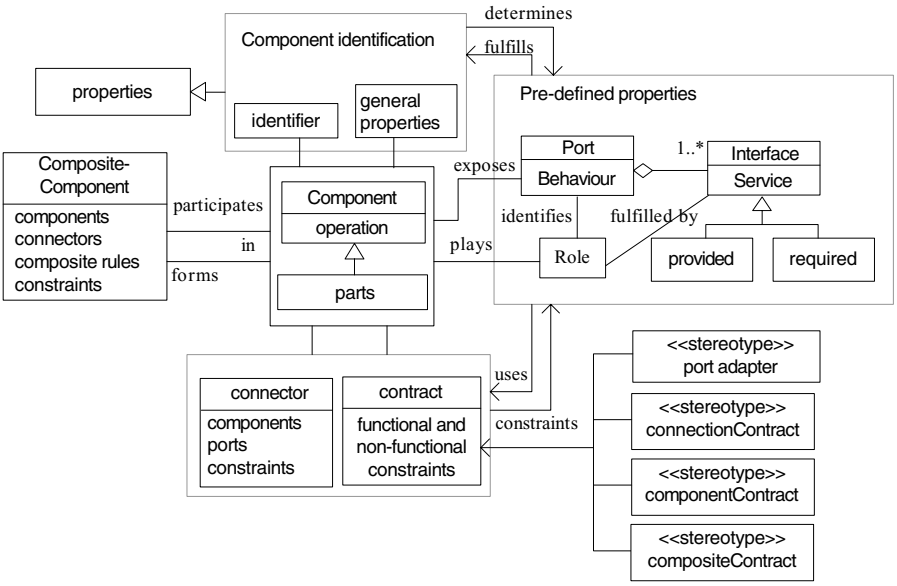


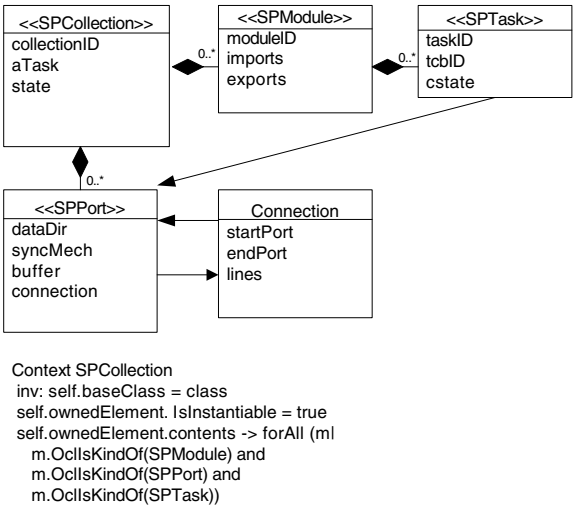
Fig. 12 UML component metamodel

chooses the active collection and dispatches messages among collections (also at different stations) through their ports.

Components are treated in UML as software components instead of merely pieces of software for deployment purposes. This means, a component can be a modular, replaceable, and deployable piece of software that is available at specification time, at deployment time, and at runtime. A component's internal structure also shows how it interacts with its environment—exclusively through interfaces or, more often, through ports. Therefore, a component can be replaced by another one which offers at least the same provided and required interfaces or ports, as these are the only parts of a component which are accessible by its environment. Physical instances of software components can be deployed on nodes.

In UML, components are composed of parts, connectors, ports and interfaces. They exchange data with each other through ports. Viewed from the outside, a component is a set of provided and required interfaces, which may be exposed via ports. Internally, it is a set of class instances or parts that collaborate to implement the services exposed by the component's interfaces. Parts represent subcomponents. A component metamodel is defined in Fig. 12, which is an extension of UML components by adding the non-functional aspects contract and general properties. This figure illustrates the component concepts and reflects both external and internal views. A component owns a unique identifer and a set of properties, and defines a set of communication ports which provide interfaces. Components can exchange data with each other through ports and connectors, only. A component may be composite—containing other component(s).

With its behaviour and elements (modules and ports) as shown in Fig. 13, the configuration of collections in S-PEARL shares greatest similarity with a component in UML, as both of them represent primary computational elements, both have ports,



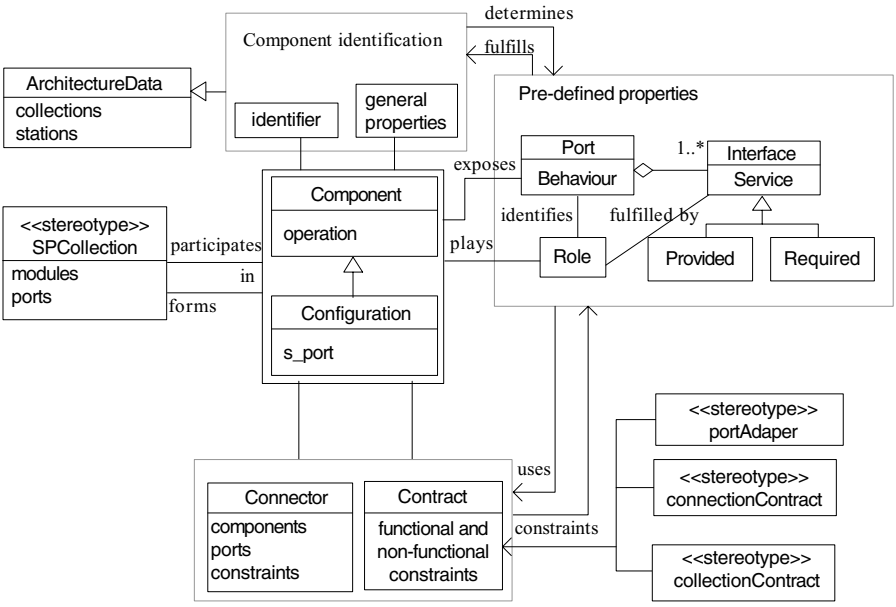


Fig. 13 Structure of the software part of "Architecture Data" with stereotypes for Collection, Module, Task and Port

Fig. 14 Stereotype for a configuration of collections in the context of a UML component

both may be decomposed hierarchically, and both are replaceable. Thus, it is natural to associate a configuration of collections with a component (see Fig. 14), and a collection, as its part, with a class as shown in Fig. 13:

Connector is a link in the component metamodel that may be of kind *delegation* or *assembly*. A *delegation connector* either links a provided port of a component to a part of the component's realisation, signifying that requests, received through the port, are forwarded to the part, or it links a realisation part to a required port,

signifying that requests sent through the port originate in the part. Several connections may exist between a single port and different realisation parts. An *assembly connector* links a required interface or port of a component to a matching provided interface or port of another component.

A connection in S-PEARL represents a link between ports of collections. For the same purpose in UML, the connector is used to link components, or subcomponents through port-to-port connections. Thus, connections can be mapped to UML connectors.

Communication among collections in S-PEARL is performed through port-to-port message exchange, which avoids direct references to communication objects in other collections, and decouples the communication infrastructure from the logic of message passing. One-to-many and many-to-one communication structures are allowed. A message may be sent using either an asynchronous "no-wait-send," a synchronous "blocking-send," or a synchronous "send-reply" protocol. Synchronous sends and receives may be specified with a timeout clause. The main purpose of protocols in S-PEARL is the definition of communication patterns, i.e., patterns of messages sent from one collection to another. In UML, protocols represent the behavioural aspects of connectors, which are similar to the communication patterns in S-PEARL. Thus, we can define constraints and tagged values for the communication patterns, and assign them to ports and connections in order to achieve similar effects as in S-PEARL.

Port in the component metamodel is a named and typed interaction point of a component. A provided port is characterised by a provided interface, a required port by a required interface, and a complex port by an arbitrary set of provided and required interfaces. Complex ports enable the localisation of complex interaction patterns where calls may occur in both directions. Unlike interfaces, a port may be associated with a behaviour, specifying the externally observable behaviour of the component when interacting through the port. This allows the specification of semantic contracts. A component may have multiple ports typed by the same interface, and is able to distinguish between calls received through different ports. In S-PEARL, there are in-, out-, and in-out ports which could directly be mapped to ports in the component metamodel, since both serve as interfaces that define points of interaction between the computational elements and their environments. However, we have defined a Port stereotype for inter-collection communication with S-PEARL port properties and functionality. We use a dedicated component port "s_port" for transferring system call parameters in asymmetrical systems which are serviced through the CM object (see Fig. 16).

Interface is the only part of a component that is visible to users. It should provide all the information that the users need in order to deploy the component, and contain specifications for its operations. It is a collection of operations that is used to specify a service of a class or a component. During execution they are used when invoking the component's functionality by the application.

In S-PEARL, the collections are called through uniform interfaces, and their only points of interaction are the ports mentioned before. Their execution and collaboration is organised by the configuration manager being the primary execution class of any component at any station.

Properties are used to characterise aspects of components. General properties can be expressed with respect to timing and resource usage such as deadline, time period, and worst-case execution time (WCET), or resource consumption. Predefined properties are used to express supercomponent, ports, or constraints.

Timing requirements could be expressed as *TaggedValues* attached to the "Task" stereotype of a "Collection". However, the task stereotype can also hold this information. When several tasks are ready to run, a priority-driven scheduler should select the task with the highest priority to run. Schedulers are timed systems that manage shared resources. Usually, schedulers apply scheduling policies to select among pending requests to allow for access to resources. The scheduler polices can be expressed by contract.

Similar properties also pertain to other S-PEARL constructs and may be assigned to them as properties. The assigned properties are meant for system programs, which have to know how to interpret them. Hence, these features are target-platform-dependent and have to be used with caution.

Contract is defined as a class in the component metamodel used to specify a component's operation constraints. It uses the theory and methods of the design by contract approach (Meyer, 1992) to specify functionality. It can be assigned a port, connector or component, and govern some functional or non-functional constraints. Furthermore, architecture constraints can be divided into component constraints, composition constraints, and connection constraints. In real-time systems, a component constraint may describe a property of time-criticality, which its environment expects from a component. A connection constraint describes time-criticality of message transmission across components which is, normally, a system-wide (or subsystem-wide) timing requirement. A composition constraint describes the time behaviour expected by a component from its environment. Also, a UML operation contract can be employed that identifies system state changes when an operation takes place. Effectively, it will define what each system operation does. All constraints can be specified by employing contracts and assigning them to corresponding participants.

Operation specifies an individual action that a component object will perform. It deals with input parameters which specify the information provided or passed to the component, output parameters which specify the information updated or returned by the component, any resulting change of the component's state, and any constraints that apply.

Port Adapter enables the connection of two incompatible ports. It defines the semantics associated with the ports and provides the operations, which are expected from the respective other port. The adaptation is realised at mapping time. A port adapter can also describe time-dependent, operational-behaviour constraints of components.

Composition Components specify how components are interconnected. They contain a number of component instances and define their configurations. In addition, a composition component also specifies how the ports of those instances are wired, i.e., which connector is used to connect which ports. It is defined for the purpose of configuration, and may occur in parts of components or in a main component such as system composition. A composition component contains a number of connected subcomponents, rules which specify compositional constraints, and component ports, which may form internal ports of the composite. A composite component

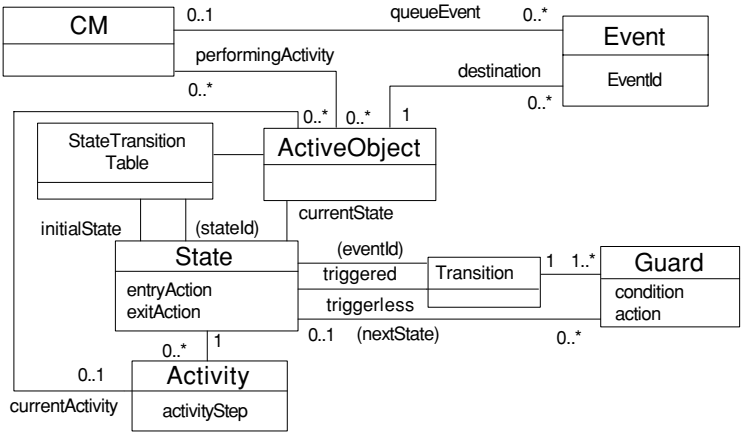


Fig. 15 CM in the context of the UML statechart mechanism

also has external ports, which are the only ones that are externally visible. The external ports are connected to appropriate internal ports and connectors.

4.1.3 Binding the S-PEARL TSTD to UML's statechart concept

For proper task representation and management some additional constructs still need to be defined for use in UML models. In UML, state machines are adopted to model the dynamic aspects of a system, which focuses on the event-order behaviour of an object and shows the event-triggered flow of control due to transitions leading from state to state. A state machine models the lifetime of a single object, whether it is an instance of a class, a use case, or even an entire system. An object may receive an event, respond with an action, then change its state, and it may also receive another event. Its response may be different, depending on its current state in response to the previous event.

A statechart representation is chosen to model adaptive operational behaviour. The modeling objects provided are states, events and transitions: (1) states represent the operational model, (2) events represent the causes of mode-shifts, and (3) transitions and transition rules define the pre-conditions and the consequences of mode-changes. States can contain states, events, and transitions, thus enabling the creation of hierarchical finite state machines. Therefore, the UML statechart formalism can be used to model S-PEARL's task concept by defining a translation to the task's "main()" method.

In Fig. 15, the UML statechart mechanism is shown. It includes CM, Event, ActiveObject, StateTransitionTable, State, Transition and Activity. The necessary adjustments for implementing S-PEARL TST diagrams are discussed below.

- The main flow of control is based on events rather than function calls. The CM object as local executive for each station controls the execution path at this (part of the) system. It also keeps a list of active objects (e.g., collections/tasks in S-PEARL) that are currently executing activities. Switching between dispatching events and

executing activities allows the other active objects in the system to process, and also allows the activities to be interrupted by incoming events.
- A state reflects a situation in the life of an object during which this object satisfies some condition, performs some activity, or waits for some event. An object remains in a state for a finite amount of time.
- A transition indicates a change from one state to another, indicating that an object in the first state will perform certain actions, and enter the second state when a specified event occurs and other specified conditions are satisfied. Each transition has a label that comes in three parts (Fowler, 2004): trigger-signature, guard and activity. All parts are optional. The trigger-signature is usually a single event that triggers a potential change of state. The guard, if present, is a Boolean condition that must be true for the transition to be taken. The activity is some behaviour that is executed during the transition. It may be any behavioural expression (e.g., C statements, commands, or PEARL system calls in our case). The full form or a trigger-signature may include multiple events and parameters.
- An event class defines the functions that dispatch an event to its destination active object. Each event carries the identifier of the active object that will receive the event.
- The StateTransitionTable consists of a set of states defined by ActiveObject. It also maintains the initial state to enter when a new instance ActiveObject is created.

In the S-PEARL methodology, an executable program is a collection of modules, being composed of a set of tasks that respond to events (see Fig. 13). Tasks represent the processes of a running system, i.e., active objects in the UML model. They are modeled in S-PEARL by TST diagrams. The translation of TST diagrams to task prototypes relies on state enumeration, which enables the execution and collaboration managing CM to switch among active tasks and states, and to return to the previous state upon resumption of the previously active task. Pre-emption points concur with task state transfers (i.e., a context switch shall occur when a task state is worked off) as modeled in its source TST diagram. If the mentioned enumeration is introduced in the translation of a UML statechart to task prototypes, these two formalisms may be used interchangeably. The states in TST diagrams can be assigned time/event trigger conditions and minimum/maximum times for their execution. These parameters are taken into account during the translation to task prototypes for the generation of appropriate system calls to the CM's inherent real-time operating system.

The CM object as local executive for each station controls the execution path at this (part of the) system (see Fig. 16). It also maintains a list of collections and a reference to the one currently executing activities. This collection is responsible for scheduling its associated tasks and for their communication. The main flow of control is based on state changes of the task's TSTD translations (e.g., see Fig. 5) and their system/CM requests.

Since the translation of UML statecharts depends on tools (and target platforms), the system calls and timing limitations should be coded into statechart actions and CM steering actions, but to be interpreted correctly CM and architecture data libraries need to be combined in the final compiled project.

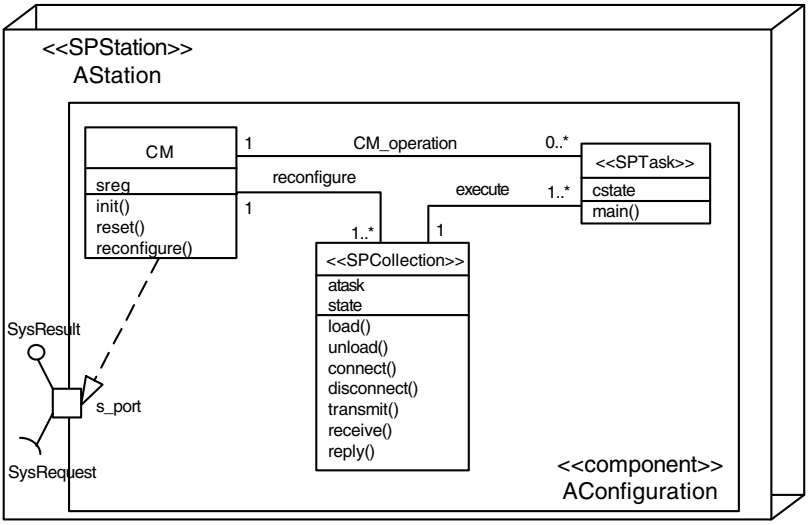


Fig. 16 S-PEARL oriented application architecture in UML

4.2 UML application architecture with S-PEARL stereotypes

An application architecture should consist of a set of station nodes and configuration components, and a set of static or dynamic links that may be established during the application's execution. As shown in Fig. 16, the CM is described as a global configuration object that performs the runtime re-configuration of stations and collections according to the application architecture. In this application architecture, the *ArchitectureData* package is defined as part of the configuration. It stores the relevant information about the system architecture which forms (a part of) the application. This information is represented in the UML model by the parameterised stereotype objects *SPStation* and *SPCollection*, respectively, representing *ArchitectureData*, whose structure is outlined in Figs. 9, 11, and 13. It also specifies dependencies that exist between station stereotypes and nodes for deployment.

The configuration manager (CM), being the base class of each station's *Configuration* component, is responsible for their activation and deactivation, as well as to connect and disconnect logical communication paths, based on the stations' states.

5 Conclusion

The Specification PEARL methodology to co-design embedded control systems was presented together with its textual and graphical modeling languages to describe hardware/software architectures and program tasks. The translation of Specification PEARL models to application prototypes for (1) execution on specified target architectures, or (2) co-simulation to verify temporal feasibility was shown.

Since UML, being a prominent design methodology also for embedded (real-time) systems, is still lacking some of the features of S-PEARL, a matching UML profile was

defined. Also the semantic translation of Timed State Transition Diagrams (TSTD), being used to model tasks in S-PEARL, into UML statecharts was presented. Hence, for UML-based design of embedded control systems oriented at S-PEARL the two methodologies can be combined in the framework of a stereotyped UML model, provided the corresponding configuration management (CM) classes and their associated architecture data structures are included in the application models.

Acknowledgments This article presents the main results of the research project "Holistic Embedded Control Systems Design" (Z2-3493), which was financed by the Slovenian Ministry of Education, Science and Sport. Ms. Lu's work was supported by a matching funds scholarship of the German Academic Exchange Service (DAAD) and Institut für Automation und Kommunikation e.V. Magdeburg.

References

Abdallah HB (1996) GCSR: a graphical language for the specification, refinement, and analysis of real-time systems. PhD Dissertation, Dept. of Computer and Information Science, University of Pennsylvania

Agha G (1991) The structure and semantics of actor languages. In: de Bakker JW, de Roever WP, Rozenberg G (eds), Foundations of object-oriented languages. Springer-Verlag, pp. 1–59

Balarin F, Chiodo M, Giusto P, Hsieh H, Jurecska A, Lavagno L, Passerone C, Sangiovanni-Vincentelli A, Sentovich E, Suzuki K, Tabarra B (1997) Hardware-software co-design of embedded systems: the POLIS approach. Kluwer Academic Publishers

Bitsch F, Göhner P, Gutbrodt F, Katzke U, Vogel-Heuser B. Specification of hard real-time industrial automation systems with UML-PA http://www.ias.uni-stuttgart.de/forschung/pub/indin2005_paper_gt.pdf

Dietz C (1998) Action diagrams. In: Maranzana M (ed), Real Time Programming 1997, Pergamon

Douglass BP (1999) Doing hard time: developing real-time systems with UML, objects, frameworks and patterns. Addison-Wesley Professional

Fowler M (2004) UML Distilled, 3th ed. A Brief Guide to the Standard Object Modeling Language, Addison-Wesley

Gausemeier J, Glässer U, Schäfer W, Eckes R, Kardos M, Wagner R. ISILEIT project.http://wwwcs.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/ISILEIT/index.html

Gumzej R (1999) Embedded system architecture co-design and its validation. Doctoral thesis, University of Maribor, Slovenia

Gumzej R (2004) Holistic embedded control systems design with specification PEARL, Overview of the project Z2-3493. http://www.rts.uni-mb.si/misc/projekti/SPEARL/

Gumzej R, Colnarič M (2001) An approach to modeling and verification of real-time systems. In: Proc. 4th IEEE Intl. Symp. on object-oriented real-time distributed computing, Magdeburg

Gumzej R, Colnarič M (2003) The representation of PEARL tasks as timed state transition diagrams. In Proceedings of 27th IFAC/IFIP/IEEE workshop on real time programming, WRTP'03, May 14–17 (2003), Lagów (Poland)

Henzinger TA, Kirsch CM, Sanvido MAA, Pree W (2003) From control models to real-time code using Giotto. IEEE Control System Mag 23(1):50–64

Hylands C, Lee E, Liu J, Liu X, Neuendorffer S, Xiong Y, Zhao Y, Zheng H (2003) Overview of the ptolemy project, technical memorandum UCB/ERL M03/25, http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/overview/overview03.pdf

Khalil M, Le Traon Y, Robach C (1998) Control-flow system diagnosis: an evolutive method. In: Proc. 24th EUROMICRO Conference. Västerås

Lee I, Davidson S, Gerber R (1991) Communicating shared resources: a paradigm for integrating real-time specification and implementation. Foundations of Real-Time Computing: Formal Specifications and Methods. Kluwer Academic Publishers

Licht T (2004) Ein verfahren zur zeitlichen analyse von UML-Modellen beim entwurf von automatisierungssystemen. PhD Thesis, Faculty of computer science and automation, Technical University of Ilmenau, Ilmenau

Meyer B (1992) Applying design by contract. IEEE Comput 25(10):40–51

Mok AK (1991) Towards mechanization of real-time system design. Foundations of real-time computing: Formal Specifications and Methods. Kluwer Academic Publishers

Mooney III VJ (1998) Hardware/Software Co-Design of Run-Time Sytems. PhD thesis, Stanford University
Multiprocessor PEARL (1989) DIN 66253, Part 3
OMG (2004) UML 2.0 superstructure specification. http://www.omg.org/cgi-bin/doc?ptc/04-10-02
OMG (2005) UML profile for schedulability, performance, and time specification 1.1 (formal/05-01-02). OMG
Ostroff JS (1997) A visual toolset for the design of real-time discrete event systems. IEEE Trans on Control Systems Technology
RTOS-UH. http://www.irt.uni-hannover.de/rtos/rtosfble.html
Schröter G, Braatz B, Klein M (2003) ODEMA–semantische konsistenz objekt-orientier viewpoint-spezifikationen, projekt IOSIP. kolloquium des DFG-SPP software-spezifikation, stuttgart, 20. http://www.bbraatz.eu/talks/2003-SBK-SPPKoll.pdf
Selic B, Rumbaugh J (1998) Using UML for modeling complex real-time systems. White Paper, Rational Software Corp
Shaw AC (1992) Communicating real-time state machines. IEEE Trans. Software Engineering, 18(9):805–816
Traore I, Sahraoui A-K (1998) A multiformalism specification framework with statecharts and VDM. In: Maranzana M (ed), Real Time Programming 1997, Pergamon

Roman Gumzej started his career as external co-worker of the University Computer Centre of the University of Maribor and Institute of Information Science in Maribor during his studies, where he took part in the university information system and COBISS (Co-operative Online Bibliographic System & Services) projects. After his graduation in 1993 he joined the Faculty of Electrical Engineering and Computer Science of University of Maribor, Slovenia, from which he received his Master of Science and Doctor of Science degrees in Computer and information science in 1997 and 1999, respectively. In 2004 he was elected assistant professor at the same institution. He has co-operated in several national and international projects on embedded real-time systems design, real-time operating systems, as well as verification and validation of real-time applications. From 2001 to 2004, he conducted a national basic research project entitled "Holistic Embedded Control Systems Design." In 2002 he co-operated in industrial communications research projects at IFAK e.V. in Magedburg, Germany, as guest scientist in the framework of a DAAD academics exchange programme. He has authored or co-authored 32 book chapters and journal and conference papers, mostly on embedded real-time systems, their operating systems and applications, safety, hardware/software co-design and co-simulation. He is a member of the IEEE Computer Society.

Shourong Lu received her M.Sc. degree in theory of automatic control and its applications from Xidian University, Xian, China in 1998. In 1999 she was promoted to associate professor at the Department of Electrical Engineering, Hubei Automotive Industries Institute, Shiyan, China, where she conducted basic and applied research on intelligent control. The areas of her research interests are distributed embedded real-time systems, and intelligent control and applications. She has authored or co-authored 27 journal articles, conference contributions and book chapters, mostly on industrial automation and design of real-time systems. Currently, she is pursuing her doctoral degree in real-time control systems at the Faculty of Electrical and Computer Engineering, FernUniversität in Hagen, Germany.