Refinement and Verification Applied to an In-Flight Data Acquisition Unit *

Wan Fokkink[1], Natalia Ioustinova[1], Ernst Kesseler[2], Jaco van de Pol[1], Yaroslav S. Usenko[1], Yuri A. Yushtein[2]

[1] Centre for Mathematics and Computer Science (CWI)
Department of Software Engineering
PO Box 94079, 1090 GB Amsterdam, The Netherlands
wan@cwi.nl, ustin@cwi.nl, vdpol@cwi.nl, ysu@cwi.nl

[2] National Aerospace Laboratory (NLR)
Department of Embedded Systems
PO Box 90502, 1006 MB Amsterdam, The Netherlands
kesseler@nlr.nl, yushtein@nlr.nl

Abstract. In order to optimise maintenance and increase safety, the Royal Netherlands Navy initiated the development of a multi-channel on-board data acquisition system for its Lynx helicopters. This AIDA (Automatic In-flight Data Acquisition) system records usage and loads data on main rotor, engines and airframe. We used refinement in combination with model checking to arrive at a formally verified prototype implementation of the AIDA system, starting from the functional requirements.
Keywords: refinement, verification, B-method, model checking, μCRL.

1 Introduction

A good method for developing (safety-critical) software is by means of a stepwise refinement, starting from the original user requirements. Furthermore, formal methods can be applied to guarantee correctness at the different stages of refinement.

B method [1] provides a notation and a toolset for requirements modelling, software interface specification, software design, implementation and maintenance. It targets software development from specification through refinement, down to implementation and automatic code generation, with verification at each stage. Refinement via incremental construction of layered software is the guiding principle of B.

Refinement verification is a methodology for verifying that the functionality of an abstract system model is correctly implemented by a low-level implementation. By breaking a large verification problem into small, manageable parts, the refinement methodology makes it possible to verify designs that are much

* This research was carried out in the framework of the KTV-FM project funded by the Dutch Ministry of Defense under the order #726/1/00301.

too large to be handled directly. This decomposition of the verification problem is enabled by specifying refinement maps that translate the behaviour of the abstract model into the behaviour of given interfaces and structures in the low-level design.

Refinement verification targets safety properties, which guarantee that a bad thing will never happen. However, in general one also wants to verify progress properties, which guarantee that a good thing will eventually happen. Special purpose theorem provers and model checkers have been developed which can check progress properties. Since these tools use their own languages, they cannot be applied to B specifications directly.

μCRL [3, 9] provides a notation and a toolset for the specification and verification of distributed systems in an algebraic fashion. It targets the specification of system behaviour in a process-algebraic style and of data elements in the form of abstract data types. The μCRL toolset, together with the CADP toolset [6], which acts as a back-end for the μCRL toolset, features visualisation, simulation, state space generation, model checking, theorem proving and state bit hashing capabilities. It has been successfully applied in the analysis of a wide range of protocols and distributed systems.

In this paper we combine the B refinement paradigm based on imperative programming with the μCRL verification support based on algebraic specification. The idea is that the models that are produced during the subsequent refinement stages in B can be quite easily transformed into μCRL specifications, where excellent tool support is available for the verification of these models.

In order to reduce maintenance costs and increase safety, the Royal Netherlands Navy initiated the development of a multi-channel on-board data acquisition system for its Lynx helicopters [19]. This AIDA (Automatic In-flight Data Acquisition) system records usage and loads data on main rotor, engines and airframe, thus making it possible to optimise the maintenance of Lynx helicopters. In a project funded by the Royal Netherlands Navy, the National Aerospace Laboratory (NLR) in collaboration with the Centre for Mathematics and Computer Science (CWI), made an effort to arrive at a formally verified implementation of the AIDA system, starting from the functional requirements, using refinement.

We built B models of the AIDA system, including a number of its monitoring tasks. These models are based on the functional requirements document for the AIDA system [5]. We started with a high level abstract description in the form of abstract machines; on this level we performed animation, and a number of internal consistency proof obligations were generated and discharged. Next, some of the machines were refined, and once more internal consistency proof obligations were generated and discharged. As a final refinement step, all abstract machines were implemented, and the resulting executable specification was tested. We also built a μCRL model of the AIDA system, including the same monitoring tasks, based on the B model. Moreover, correctness criteria for the AIDA system were formalised in modal logic. Using the μCRL toolset, we verified the system does not contain deadlocks. Furthermore, using a model checker within the CADP verification toolbox we verified that all usage and loads data are recorded, and that no recordings are made without reason. For the validity of these properties it turned out to be essential that some of the requirements in the original requirements document were strengthened.

Concluding, we found that the refinement paradigm together with formal verification methods can be successfully applied in the development of naval equipment.

This paper is set up as follows. Section 2 contains a description of the AIDA system and of the formal model that we designed. Section 3 sets forth the methodology behind our approach. Section 4 explains the basics of refinement in general and B in particular, and describes how refinement was applied to the formal model of the AIDA system. Section 5 deals with the μCRL model of the system and with the model checking analysis. Section 6 gives references to related work. Finally, Section 7 sums up our conclusions.

2 System overview

2.1 AIDA

The Lynx helicopter is in service with the Royal Netherlands Navy since the late 70's. In September 1994, a fatal accident with a UK Lynx helicopter occurred in Germany as a result of failure of a tie-bar, which connects the rotor hub with the blade. In response, at the end of 1996, the Royal Netherlands Navy initiated the development of an Automatic In-flight Data Acquisition Unit (AIDA) for its Lynx helicopters, in order to optimise maintenance and increase safety.

As inputs, the AIDA system gathers data from 15 analog and 2 discrete signals produced by several measurement and control devices. 39 different tasks of the AIDA system are responsible for data storage, conditioning and processing. The AIDA system performs several logging tasks simultaneously. Each of the tasks checks values of one or more input signals and depending on them performs some logging activities, such as writing to a data acquisition file, or producing a signal via an audio or video channel. Some of the tasks use timers to check whether a particular situation persists for some period of time, and only after that period the logging is performed.

In Figure 1, an overview of a part of the system is presented. The environment represents the measurement and control devices of a helicopter. *TASK2* receives *NR* (main rotor rotational speed) and *WOW* (weight-on-wheels) signals from the environment and uses timer *TIMER2*. The task can write data to the data administration file (*DAF*) and can set video warnings for the crew. *TASK27* performs activities similar to the activities of *TASK2*, but according to *NR* and *LHMY* (strain in left-hand side of rear spar sponson) signals. *TASK3* produces an audio warning depending on the values of *NR* and *WOW* signals. *TASK4* also sets an audio warning according to the value of *NR* signal.

With respect to the functional requirements of the AIDA system [5], there are the following groups of tasks:

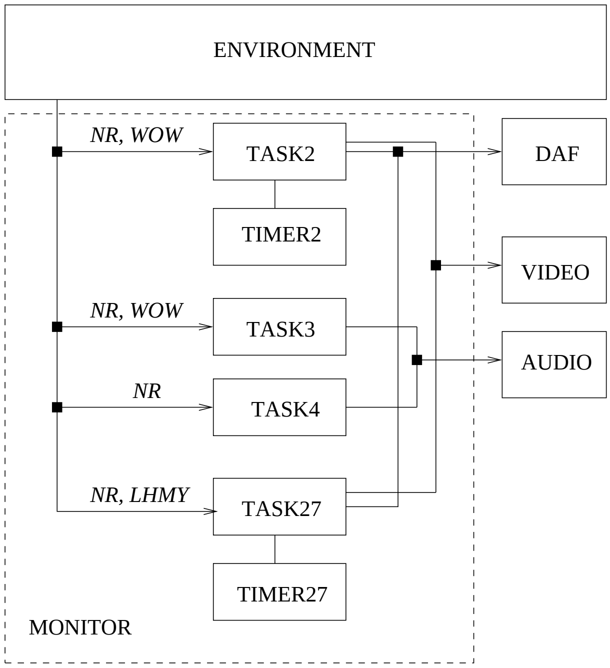

Fig. 1. System overview

- *Data reduction tasks* monitor and process signals using standard algorithms, with subsequent storage of data into memory. (For example, the so-called SPTTS algorithm searches the load trace for successive peaks and troughs and stores them with time stamps and momentary values of any slaved signals.)
- *Level crossing detection tasks* monitor signals to check whether a predefined level is crossed upwards or downwards. If so, audio or video warnings are given and relevant information is stored into AIDA memory. (A typical example is main rotor overspeed or underspeed.) Such tasks are usually a safety issue for the helicopter and its crew.
- *Event count and event duration tasks* provide signal monitoring, count the number of times that a signal reaches a certain predefined level, and determine the time span that the signal is at this pre-defined level. (A typical example is the weight-on-wheels task, which determines whether or not a helicopter is at the ground.) Such tasks provide direct compact event information without a further need for processing.
- *System integrity tasks* verify the status of the AIDA recorder and monitor various signals during the AIDA recorder operation to check whether a predefined malfunctioning condition is met. If so, integrity of the signals and/or the total system is considered questionable.

Since the AIDA system handles critical flight data, it should not change the data content or influence timing in any way. On the other hand, the monitoring system should not miss or discard the data on any of the monitored channels, as this could lead to incorrect calculations of the estimated helicopter operational status and required maintenance cycle.

2.2 Formal specification

Our goal was to build a formal specification of the AIDA system and to verify the specification with respect to functional requirements for an on-board loads and usage monitoring system for the Lynx helicopter. Considering the functional requirements we identified the following key entities of the system: monitor, tasks, data administration file, input and output channels. The monitor schedules the tasks, and also plays the role of system clock. The data administration file is used to store the data and the results of data processing.

Since AIDA is a real-time system, some primitives are needed to represent time aspects of the system. We employed a concept of timers, where a timer can be either active or deactivated, and time progression is discretised. An active timer is represented by a variable that is set to the delay left until expiration of the timer. System time elapses by counting down all the active timers present in the system. We refer to a segment of time separated by the transitions decrementing active timers as a time slice. A timer with zero delay expires, and in the next time slice is deactivated.

Since the specification of the system should abstract from details that depend on the hardware implementation of the system, we assume that operations performed by tasks are atomic and can be handled within a time slice.

All tasks are triggered by some kind of condition (either a safety-critical condition, a level crossing condition, or a condition checking whether a predefined level is reached). Considering the functional requirements, we divided tasks into three groups with respect to the activities that the tasks should perform:

- Tasks that use a timer, can write data to the data acquisition file and can produce a warning signal for the crew.
- Tasks that only produce a warning signal for the crew.
- Tasks that only write to the data acquisition file.

Tasks of the first type are usually triggered by a safety-critical condition. Being triggered, the task sets a timer and waits until the timer expires. After the timer is expired, the task checks whether the safety critical condition is still satisfied. If the condition is still true, then the data and time when the condition was recognised are stored into the data acquisition file and a warning signal is provided for the crew.

Tasks of the second type are triggered when a predefined level is crossed. Being triggered, the task provides a warning signal for the crew. The third kind of tasks mostly detect when a signal reaches a predefined level, process data with respect to some standard algorithm, and store the data. Since tasks of the same group show similar observable behaviour, we have decided to concentrate our attention on the refinement and verification of a part of the system.

3 Combining refinement and model checking

3.1 Refinement

A good method for developing safety-critical software is by means of a stepwise refinement, starting from the original user requirements. Formal methods can be applied to guarantee correctness at the different stages of refinement.

Refinement verification is a methodology for verifying that the functionality of an abstract system model is correctly implemented by a low-level implementation. By breaking a large verification problem into small, manageable parts, the refinement methodology makes it possible to verify designs that are too large to be handled directly. This decomposition of the verification problem is enabled by specifying refinement maps that translate the behaviour of the abstract model into the behaviour of given interfaces and structures in the low-level design. This makes it possible to verify small parts of the low-level design in the context of the abstract model. Thus, proof obligations can be reduced to a small enough scale.

B [1] provides a notation and a toolset for requirements modelling, software interface specification, software design, implementation and maintenance. It targets software development from specification through refinement, down to implementation and automatic code generation, with verification at each stage. Refinement via incremental construction of layered software is the guiding principle of B. Development with B is based on the concept of an abstract machine, a refinement and an implementation of the machine. The B Abstract Machine Notation [1] is used with many alternative development processes and with a number of existing development methods [18].

3.2 Model checking

Model checking is a completely different approach. Here we have one system model, described in some high-level programming language. Separately, we formulate a number of user requirements in some temporal logic. The model checker generates all reachable program states exhaustively (in principle) in order to check that the user requirements hold in all possible runs.

Several efficient algorithms exist, in order to verify that all states in a state space satisfy a certain formula from temporal logic. The main bottleneck of model checking is the state explosion problem: the size of the state space of a system tends to grow exponentially with respect to the number of concurrent components of the system. However, owing to recent advances such as model checking of symbolic representations of state spaces and state bit hashing, model checking is by now a mature technique for the analysis of real-life hardware and software systems.

μCRL [9] is a language for specifying and verifying distributed systems in an algebraic fashion. It targets the specification of system behaviour in a process-algebraic style and of data elements in the form of abstract data types. The μCRL toolset [3, 20] supports efficient state space generation of μCRL specifications.

The CADP toolset [6] acts as a back-end for the μCRL toolset, so that model checking can be applied with respect to the state spaces generated from μCRL specifications.

3.3 The combination

We note that in a refinement step from an abstract to a concrete model, new details are added. In refinement tools it is only checked that the concrete model is internally consistent (by invariants and preconditions) and consistent with the abstract model. Except consistency, nothing is checked about the initial most abstract model, nor about the added details. So refinement verification is limited to safety properties, which guarantee that a bad thing will never happen.

However, in general one wants to verify that the initial model and the refined model satisfy certain user requirements. In particular, one also wants to verify progress properties, which guarantee that a good thing will eventually happen. Model checking can be applied for the verification of progress properties [13].

Another reason to apply model checking could be that the automated proof search capabilities of B turn out to be relatively limited; model checking can be used as a debugging device for proof obligations [16].

For these reasons we combined the refinement paradigm and verification of safety properties using B with the verification of progress properties using the model checking capabilities of μCRL and CADP. We applied this combination with respect to the functional requirements of the AIDA system. In the next two sections, refinement using B and model checking using μCRL are explained, respectively.

4 Refinement using B

4.1 The B method

A typical development process using B [14] might cover requirement analysis, specification development, design, and coding, integration and test phases of software development. For all these phases, some tool support exists. The B-Toolkit of the company B-Core in the UK supports the incremental construction of software, where validation is supported by static analysis such as type checking, by dynamic analysis using simulation, and by proof of correctness using an integrated theorem prover. An alternative commercial toolset that supports the B method is offered by Atelier B in France.

Informal structured models of the problem domain are created during requirement analysis. Specification development results in formalisation of analysis models in terms of abstract machines. The specification can be validated by animation in B-Toolkit. Internal consistency obligations can be generated and proved to check whether all operations of the machines respect their invariants. During the design phase, the decomposition of the system is identified and selected components of the system are refined. The proof obligations generated by B-Toolkit during this phase are used to prove whether refinements are correct with respect to the specification. At the end, a code generator can be applied to the model. Generated code can be tested using test cases based on the requirements.

4.2 Specification design

To get a more concrete picture, we focus on one of the tasks within AIDA, viz. *TASK2* (for more details see appendix C). Some tasks of the system, including *TASK2*, use timers. For this reason we specified timer machines; the timer for *TASK2* is called `Timer2`. The timer can be either active (`on`) or deactivated (`off`). The state of the timer is represented by two variables `tstat2` and `tval2`. The first one shows whether the timer is active or deactivated, the second one represents the delay left until expiration of the active timer.

Each timer machine contains the following operations: `reset` deactivates the timer, `set` activates and sets up the timer, `expire` returns `true` if and only if the timer is active and expired, and `tick` decreases the value of the timer if it is active and carries a value greater than zero (see Figure 2). The invariant of the machine imposes a restriction on the possible value of the delay as well as typing constraints for variables `tval2` and `tstat2`. The precondition of operation `set` states that the delay should not exceed the maximum prescribed by the system requirements.

```
MACHINE  Timer2
SEES     Bool_TYPE, CommDefs
VARIABLES tval2, tstat2
INVARIANT tval2:0..max & tstat2:TISTATE
INITIALISATION tval2:=0 || tstat2:=off
OPERATIONS
reset2   = BEGIN tstat2 := off END ;
set2(del) = PRE del:NAT & del <= max
           THEN tval2 := del || tstat2:=on END;
ok <-- expire2 = IF tval2 = 0 & tstat2=on
                THEN ok := TRUE ELSE ok := FALSE END;
tick2 = IF tstat2=on & tval2>0
        THEN tval2 := tval2-1 ELSE tval2:=tval2 END
END
```

Fig. 2. High-level machine description for `Timer2`

Figure 3 shows a high-level machine specification of *TASK2*. It has two operations: initialisation `tinit` and "do some work" `work`. Doing some work depends on the state, which is one of `idle`, `wait` or `check`:

- in the state `idle`, the condition is watched. If it holds, the timer is set and we go to state `wait`.

- in the state `wait`, the timer has been set. If it expires we go to state `check`.
- in the state `check`, the condition is checked again, and we go to state `idle`.

On this abstract level, it is only defined which transitions are possible, but not when they are taken. Machines `Bool_TYPE` and `CommDefs` are seen by `Task2`. `Bool_TYPE` contains a specification of the Boolean operators. `CommDefs` defines sets, constants and their properties reused by several components of the specification. The invariant defined for `Task2` provides a typing constraint for the variable of the machine. Operation `ttick2` calls operation `tick2` of the timer. This operation is invoked itself by the monitor to decrease values of all active timers at the same time.

```
MACHINE   Task2
SEES      Bool_TYPE, CommDefs
INCLUDES  Timer2
VARIABLES tstate2
INVARIANT tstate2:TSTATE
INITIALISATION tstate2:=idle
OPERATIONS
tinit2 = BEGIN reset2 || tstate2:=idle END;
work2 = CASE tstate2 OF
        EITHER idle THEN CHOICE tstate2:=wait || set2(5)
                         OR tstate2:=idle END
            OR wait THEN CHOICE tstate2:=check OR tstate2:=wait END
            OR check THEN tstate2:=idle END
        END;
ss <-- getstate2 = ss:=tstate2;
ttick2 = tick2
END
```

Fig. 3. High-level machine description for `Task2`

To make the definition of task behaviour independent on particular values of signals triggering the task, we defined a condition machine for each task. We can easily change the condition without modifying the specification of the task. Initially, the condition machine (see Figure 4) has only one operation `issatisfied2`, which returns either `TRUE` or `FALSE`.

To mimic the input channels, we developed sensor machines. We consider the operations of a sensor machine using the weight-on-wheels (*WOW*) sensor as an illustrative example. Since the *WOW* signal can be either `high` or `low`, the operation `refreshWOW` of `SensorWOW` is defined as a non-deterministic choice of the machine state between `high` or `low` (see Figure 5). Operation `getstateWOW` returns the current state of the sensor. This approach to specification of the input channels allows us to cover the possible inputs of the system.

```
MACHINE         Condition2
SEES            Bool_TYPE, CommDefs
OPERATIONS
xx <-- issatisfied2 = CHOICE xx := FALSE OR xx := TRUE END
END
```

Fig. 4. High-level machine description for *WOW* sensor

```
MACHINE         SensorWOW
SEES            Bool_TYPE, CommDefs
VARIABLES       sstateWOW
INVARIANT       sstateWOW:LHLEVELS
INITIALISATION  sstateWOW:=low
OPERATIONS
refreshWOW = CHOICE sstateWOW := high OR sstateWOW := low END;
xx <-- getstateWOW = xx := sstateWOW
END
```

Fig. 5. High-level machine description for *WOW* sensor

Audio and video output channels and the data administration file are shared by several tasks. Therefore we specified a controller for each output channel and for the data administration file.

4.3 Refinement and implementation

A B development consists of a set of components defined by abstract specification machines and a path of refinement steps down to an executable description, called an implementation. An implementation of the component is decomposed via IMPORTS and SEES clauses. A specification of the component may itself be constructed from a set of machines using INCLUDES, USES, SEES and EXTENDS mechanisms.

Usage of IMPORTS and SEES constructs implies a layered approach to system development whereby the internal details of implementation of one layer are hidden from the next layer. Such a structure improves the maintainability of the system because higher layers are independent of the internal details of lower layers and rely only on the specification of these layers.

The implementation of `Monitor` relies only on the specifications of tasks, sensors and controllers of output channels and data administration file. Since we have no information about scheduling the tasks, we have chosen one of the possible scenarios. To simulate fresh inputs, the monitor refreshes values of the sensors at the beginning. Then it allows each task to do some work and finally it decreases the values of all active timers, by which the system time elapses (Figure 6).

```
IMPLEMENTATION  MonitorI
REFINES         Monitor
SEES            CommDefs, Bool_TYPE, basic_io, String_TYPE
IMPORTS         Monitor_Vvar(MSTATE),
                Task4, Task2, Task27, Task3,
                SensorLHMY, SensorWOW, SensorNR,
                OutChController, DAFController, AudioChController
INVARIANT       (mstate=Monitor_Vvar)
INITIALISATION  Monitor_STO_VAR(start)
OPERATIONS
minit = BEGIN
        Monitor_STO_VAR(monitor);
        tinit1; tinit2; tinit3; tinit27;
        resetanalog;resetaudio
        END;
monitortask = VAR curmstate IN
                     curmstate<--Monitor_VAL_VAR;
                     IF curmstate=monitor THEN
                        refreshWOW; refreshNR; refreshLHMY;
                        work2; work3; work4; work27;
                        ttick2; ttick27
                     ELSE PUT_STR("Initialisation error"); END
              END
END
```

Fig. 6. Implementation of `Monitor` machine

The implementation of `Condition2` refines operation `issatisfied2` as is shown on the Figure 8. Now `issatisfied2` returns `TRUE` only if the condition triggering `Task2` is satisfied, i.e. the value of the *NR* sensor is below 50%RPM and the *WOW* signal is low.

The refinement of `Task2` (Figure 7) defines INCLUDES and SEES relations for `Task2` and the other components of the system. A task sees the condition `Condition2` that acts as its trigger, the data acquisition file controller `DAFController`, the video channel controller `OutChController` and includes the `Timer2` machine. The controllers of the data administration file and the video channel regulate access to these entities by the tasks of the system.

Figure 7 contains a refinement of `Task2`. The refinement not only specifies possible transitions but also defines enabling conditions for those transitions. For example, after refinement, the operation `work2` changes the state of the task from `wait` to `check` only if the timer is expired. The timer is deactivated by

```
REFINEMENT      Task2R
REFINES         Task2
SEES            Bool_TYPE, CommDefs, Condition2, OutChController,
                DAFController
INCLUDES        Timer2
VARIABLES       tstate2
INVARIANT       tstate2:TSTATE
INITIALISATION  tstate2:=idle
OPERATIONS
tinit2 = BEGIN reset2 END;
work2 = VAR exp IN
        CASE tstate2 OF
        EITHER idle THEN exp<--issatisfied2;
                     IF exp=TRUE THEN tstate2:=wait; set2(5)
                     ELSE tstate2:=idle END
        OR wait THEN exp <--expire2;
                     IF exp=TRUE THEN tstate2:=check ; reset2
                     ELSE  tstate2:=wait END
        OR check THEN exp <--issatisfied2;
                     IF exp=TRUE THEN tstate2:=idle;
                                     setanalogon; writetodaf
                     ELSE tstate2:=idle END
                 END
         END
        END;
ss <-- getstate2 = ss:=tstate2;
ttick2 = tick2
END
```

Fig. 7. Refinement of `Task2`

`reset`, after the task state is changed to `check`. If the condition triggering the task is still satisfied after timer expiration, the task writes data to the data acquisition file (by means of the `writetodaf` operation of the data acquisition file controller), sets up an output signal (by means of the `setanalogon` operation of the output channel controller) and becomes `idle`. Otherwise, the task just goes back to state `idle`.

4.4 Proof and validation

B-Toolkit generates and assists to prove a number of proof obligations. In our development we have proved that all operations of the machines respect their invariants and that the invariants are established by the initialisation. We proved that preconditions for any invoked machine operations are satisfied. For example, it was shown that the specification, the refinement and the implementation of

```
xx <-- issatisfied2 = VAR currNR, currWOW IN
                      currNR <-- getstateNR;
                      currWOW <-- getstateWOW;
                      IF (currNR=l10 or currNR=meq10l50) & currWOW=low
                      THEN xx:=TRUE ELSE xx:=FALSE END
                     END
```

Fig. 8. Refinement of `issatisfied2` operation of `Condition2` machine

`Task2` do not violate the precondition imposed on the input parameter of the operation `set2` of the machine `Timer2`.

At the specification development phase, we also used animation supported by B-Toolkit. The animation can be really helpful for debugging the specification. It allows to test the specification against various scenarios. At the design phase we used B-Toolkit's theorem prover to check whether the refinement and implementation machines satisfy the refinement constraints and whether the implementations are correct refinements of machines.

Since the development of test cases for this kind of the systems is not a trivial task, we used the code, generated by B-Toolkit, to test the system against a "chaotic" environment. In this case "chaotic" means that we abstracted from real values of analog signals NR and $LHMY$ and implemented the `refresh` operation of the sensors as a random choice (provided by B-Toolkit) on the set of abstract values. So we have specified an environment that covers all combinations of input signals.

5 Model checking using μCRL

5.1 The μCRL toolset

μCRL [9] is a language for specifying and verifying distributed systems in an algebraic fashion. It targets the specification of system behaviour in a process-algebraic style and of data elements in the form of abstract data types. System specification takes place in the form of recursive equations that include recursion variables, the basic process algebraic constructors (atomic actions, alternative and sequential composition), parallelism, synchronous communication, encapsulation of actions, and hiding of internal activity. Furthermore, atomic actions and recursion variables can carry data parameters, data elements can influence the course of a process via an if-then-else construct , and a summation operator allows to take the alternative composition with respect to all possible values of some data parameter.

The μCRL toolset [3, 20] supports the analysis and manipulation of μCRL specifications, based on term rewriting and linearisation techniques. It supports efficient state space generation, deadlock detection, interactive simulation, and theorem proving. The CADP toolset [6] acts as a back-end for the μCRL toolset, so that state spaces can be visualised, analysed and minimised, and model checking and state bit hashing are available.

The toolset is constructed around a restricted linear form of μCRL specifications, which does not include parallelism, encapsulation and hiding. The tool `mcrl` checks whether a certain specification is well-formed and brings it into linear form, which is stored as a binary file. All other tools use this linear form as their starting point. These tools come in four kinds:

1. (`msim`) steps through a process described in μCRL;
2. (`instantiator`) generates a state space that can serve as input to the CADP toolset;
3. several tools optimise linearised specifications:
   (a) `rewr` applies the equations of the data types as rewrite rules;
   (b) `constelm` removes data parameters that are constant throughout any run of the process;
   (c) `parelm` removes data and process parameters that do not influence the behaviour of the system;
   (d) `structelm`, expands variables of compound data types;
4. a tool (`pp`) to print the linearised specification.

An overview of the relations between the tools in the μCRL toolset is sketched in Figure 9. Some other tools rely on a recently developed automated theorem prover. These tools implement reachability and confluence analysis, and some sophisticated control flow analysis methods.

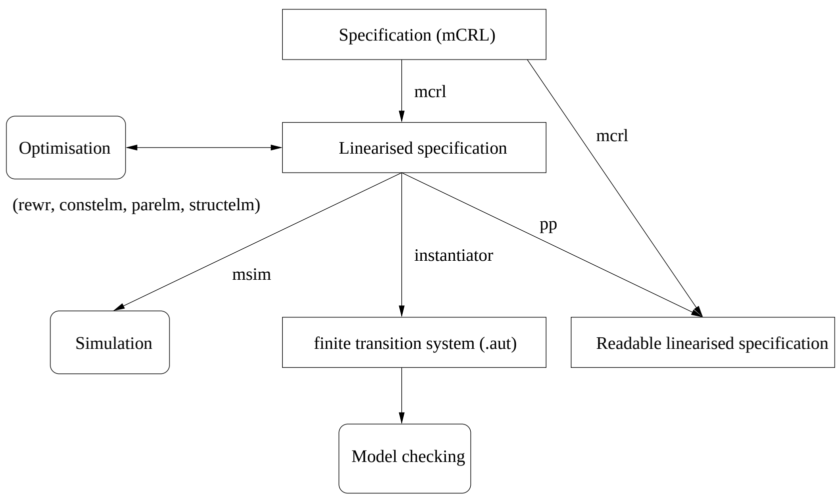

Fig. 9. The main relations between the tools in the μCRL toolset

μCRL was successfully applied in the analysis of a wide range of protocols and distributed systems. Recently it was used to support the optimised redesign of the Transactions Capabilities Procedures in the SS No. 7 protocol stack for telephone exchanges [2], to detect a number of mistakes in an industrial protocol over the CAN bus for lifting trucks [8], and to analyse the coordination languages SPLICE [4, 11] and JavaSpaces [17].

5.2 AIDA model in μCRL

We specified the different components of the AIDA system (tasks, sensors, buttons, monitor, data acquisition file, output channels, environment) as independent μCRL processes. These components communicate synchronously by performing communication actions. During these communications data can be transferred between the components.

The μCRL specification of the AIDA system consists of several parts. The first part specifies the data types used by means of equations. Some of them are general data structures, like booleans, natural numbers and lists. Others are specific to AIDA, like states of tasks, sensors, buttons, data acquisition file, output channels, etc. Next, the atomic actions are specified, and it is declared which actions can communicate. The different processes are defined by means of recursive equations. Finally, the initial state of the whole system is given, being a parallel composition of components where some actions are encapsulated or hidden. Figure 10 shows the various processes in the μCRL specification

Figure 11 shows the definition of Task 2 as a μCRL process. According to this definition *TASK2* has one state variable (for more details see appendix A). It can synchronise on init, run and tick actions (+ is alternative choice). After an init action the timer is reset ($\cdot$ denotes sequential composition), and the state is set to idle. If a run action occurs the process *TASK2RUN* is executed, *provided* the state is either idle or wait ($x \lhd b \rhd y$ means do x if b, else do y). A tick action results in ticking the accompanying timer, provided the task is in waiting state. *TASK2RUN* first checks the state. If the state is wait, the timer value is obtained (sum denotes alternative choice over a data sort). If no timeout occurs, *TASK2RUN* finishes. Otherwise, the timer is reset, the sensors are inspected, and depending on the safety condition a logging action is performed. If the state is not wait, the sensors are inspected, and depending on the safety condition, either the timer is set and the state is changed to wait, or we stay in idle. Eventually, in all cases, control is returned to the *TASK2* process.

The `mcrl` tool was used to check the syntax and static semantics of the specification, and also to transpose it to linear form. The specification was thoroughly simulated using the `msim` tool in a way that all parts of the code were reached. Several specification errors were discovered and corrected. The linear form of the ultimate (after corrections of section 5.3) μCRL specification contained 47 data parameters and 132 summands. Using the tool `constelm`, the number of parameters was reduced to 32, and the number of summands to 85.

State space generation took approximately 1 minute and 10 seconds on an AMD Athlon 1.4GHz machine, and resulted in a state space with approximately

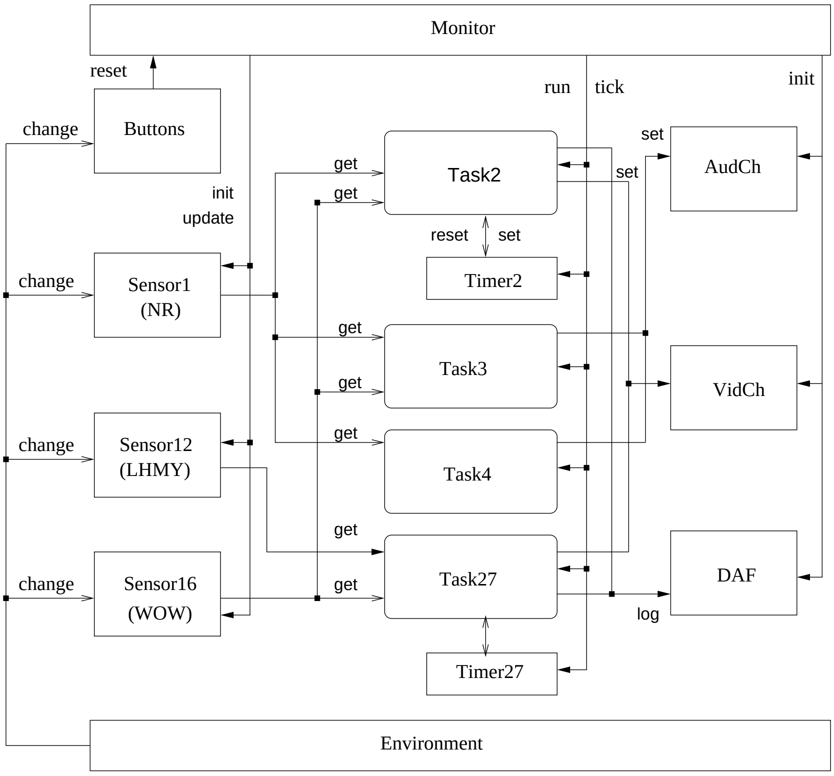

Fig. 10. System decomposition in the μCRL model

250,000 states and 900,000 transitions. The state space did not contain deadlocks (checked with the μCRL toolset) and livelocks (checked with the CADP toolset).

Reduction of the state space (modulo branching bisimulation equivalence [7]) took approximately 47 seconds on the same machine and produced 1,072 states and 5,855 transitions.

5.3 Validation of user requirements

The informal description of the AIDA system contains a rather technical specification. It provides several kinds of operational details, for instance when timers must be set and reset. This made it difficult to validate the specification. We took the following approach: we invented "reasonable" user requirements for the system, formalised them in temporal logic, and checked automatically whether the μCRL model of the AIDA system satisfies these requirements.

We formulated the following user requirements, that seemed plausible to us. For each task, it should hold that:

```
TASK2(st:TaskSt)=
  init_task(2)._reset_timer(2).TASK2(TST_IDLE)
 +
  run_task(2).
    (  TASK2RUN(st)
    <|or(eq(st,TST_IDLE),eq(st,TST_WAIT))|>
       TASK2(st)
    )
 +
  tick_task(2).
    (  _tick_timer(2).TASK2(st)
    <|eq(st,TST_WAIT)|>
       TASK2(st)
    )

TASK2RUN(st:TaskSt)=
       sum(t:Nat,_get_timer(2,t).
           (TASK2(st)
          <|gt(TIMEOUT2,t)|>
            _reset_timer(2).
            sum(sst:SensorSt,_get_state_sensor(1,sst).
             sum(sst1:SensorSt,_get_state_sensor(16,sst1).
                 ( TASK2LOG.TASK2(TST_IDLE)
               <|and(eq(sst,sst1(1)),eq(sst1,sst16(F)))|>
                   TASK2(TST_IDLE)
          ))   )))
       <|eq(st,TST_WAIT)|>
            sum(sst:SensorSt,_get_state_sensor(1,sst).
             sum(sst1:SensorSt,_get_state_sensor(16,sst1).
                 ( _set_timer(2).TASK2(TST_WAIT)
               <|and(eq(sst,sst1(1)),eq(sst1,sst16(F)))|>
                   TASK2(TST_IDLE)
               )))

TASK2LOG=_log(LogEntryTask(2))._set_vidch(VidchSt1)
```

Fig. 11. Specification of `Task2`

Property 1. if the condition checked by some task is satisfied during a certain period of time, then this task will perform its logging activity.

Property 2. the task only performs a logging activity if the condition checked by it was satisfied during a certain period of time.

The first requirement indicates that all strains and loads are logged, and the second one indicates that no false alarms are logged.

The properties were formulated in the regular alternation-free μ-calculus [15]. We show this for `Task2`, which watches the sensors *NR* and *WOW* (see Figure 1). The following abbreviations are used (in CADP these are defined as macros): *NR* – indicates the event that the *NR*-sensor is below 50%RPM. *WOW* – indicates the event that the weight is not on the wheels (so the combination *NR* and *WOW* indicates that the helicopter is probably falling down). OTHER – any sequence of actions that does not contain logging actions, reset buttons, or changes to *NR*-sensor and *WOW*-sensor.

From now on we assume that the timeout value `TIMEOUT2` is equal to 20 time units. The first property can be expressed as follows: There is no execution trace of the AIDA system with a subsequence consisting of 23 (*NR*, *WOW*) or (*WOW*,*NR*) pairs, interspersed only with OTHER sequences (see Figure 12).

```
[true*.(  (NR.OTHER.WOW) | (WOW.OTHER.NR) )
  .OTHER.( (NR.OTHER.WOW) | (WOW.OTHER.NR) )

... dropped 19 similar lines

  .OTHER.( (NR.OTHER.WOW) | (WOW.OTHER.NR) )
  .OTHER.( (NR.OTHER.WOW) | (WOW.OTHER.NR) )
] false
```

Fig. 12. Modal formula expressing Property 1

The second property can be expressed as follows: there is no execution trace of the AIDA system ending in a logging activity, preceded by at most 20 time slices where WOW = low (see Figure 13). A similar formula is needed for the *NR*-sensor. The full source code of the properties will be available in a technical report.

The next step is to verify that these properties hold for the AIDA system. To this end we applied the CADP model checker to the reduced transition system generated by the μCRL toolset. The first formula holds straight away, but surprisingly, the second formula failed. In this case, the model checker provided a counter-example in the form of a wrong execution trace. This trace was carefully inspected, and appeared to correspond to the following situation (see Figure 14): The condition on *WOW* and *NR* holds in the first time slice, and the timer is

```
[                  (not(WOW)*.
        (true|nil).(not WOW)*.

... dropped 17 similar lines ...

        (true|nil).(not WOW)*.
        (true|nil).(not WOW)*
     ).
  "__log(LogEntryTask(x2p2(0)))"
]false
```

Fig. 13. Modal formula expressing Property 2

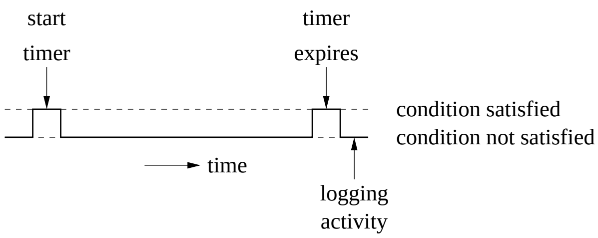

Fig. 14. Spurious logging activity

correctly set. After some time slices, the timer expires, and the condition on *WOW* and *NR* holds again. However, *it was not checked by the system whether the condition holds between setting and expiration of the timer.*

We could now proceed in two ways: either adapt the model, or the user requirements. As our properties seem quite natural we adapted the μCRL model. In the modified version, a task goes from idle to wait if the condition checked by it is satisfied. In the wait phase, it continuously checks the condition, and if it fails it returns to idle and resets the timer. If the condition remains true until the timer expires a logging action is performed. This is implemented in the μCRL specification by modifying the *TASK2RUN* process according to Figure 15 (cf. Figure 11).

In this new model, both properties appear to hold. It is interesting to remark that the mentioned time slices (20 and 23) are strict boundaries. This means that if the condition holds during exactly 21 or 22 time slices, then the system may or may not log this event, depending on the exact order in which the *NR*- and *WOW*-sensors are changed by the environment and read by the *TASK2* process.

We conclude that the informal specification was too operational, in the sense that the user requirements were missing, only the solution in terms of timers was mentioned. At the same time, they were ambiguous, because it was not indicated what the system should do between starting the timer and expiration

```
TASK2RUN(st:TaskSt)=
  sum(sst:SensorSt,_get_state_sensor(1,sst).
    sum(sst1:SensorSt,_get_state_sensor(16,sst1).
         (  (  _set_timer(2).TASK2(TST_WAIT)
            <|eq(st,TST_IDLE)|>
               sum(t:Nat,_get_timer(2,t).
                    (  TASK2(st)
                    <|gt(TIMEOUT2,t)|>
                       _reset_timer(2).TASK2LOG.TASK2(TST_IDLE)
            )      ) )
         <|and(eq(sst,sst1(1)),eq(sst1,sst16(F)))|>
            (  _reset_timer(2).TASK2(TST_IDLE)
            <|eq(st,TST_WAIT)|>
               TASK2(TST_IDLE)
     ) ) ) )
```

Fig. 15. Modified specification of `Task2`

of the timer. In the formal specification this ambiguity must be resolved in some way. We showed two ways to resolve this ambiguity, and proved that one of the solutions is preferable, because it meets certain plausible user requirements.

6 Related work

Combination of different formal techniques to support the development of verifiable correct systems is an active line of research. Our work is closely related in spirit and techniques to [13] and [16].

Julliand, Legeard, Machicoane, Parreaux and Tatibouët [13] used B Atelier in combination with the model checker Spin [10] to analyse a protocol within the Integrated Circuit Card (European Standard EN 27816). First a B model of the protocol was constructed, which was manually translated to a PROMELA specification (the input language for Spin). Similar to our approach, safety properties were verified within B, while progress properties were verified using temporal logic.

Mikhailov and Butler [16] used the B Method in combination with the state-based model checker Alloy Constraint Analyser [12] to derive the proof obligations generated by B. Their approach is motivated by the fact that a formal proof using a theorem prover of all proof obligations generated by B is often practically unfeasible. Sometimes, proof obligations are actually false, due to underspecification or to a specification error. Detecting such flaws by means of theorem proving is very difficult indeed. Therefore, in [16] model checking was used as a debugging device. The B specification of a student grades data base was manually translated to an Alloy specification. In cases where proof obligations were left unproved, these were transposed to the Alloy language, and the Alloy constraint analyser was applied. The counter-examples that the Aloy constraint analyser can generate are usually suggestive, so that the developer may realise how and why a certain property is invalidated. This leads to a debugging process with a shorter life-cycle than when only an interactive theorem prover is used.

7 Conclusion

We have shown that one can perform a whole cycle of software development from an informal task description to a specification close to an executable prototype implementation within the B-Method. By identifying the key entities in the informal description, a designer can easily develop the needed data structures. We captured the functional requirements of the system in a stepwise manner. The layered design method advocated with B allowed us to obtain an elegant and succinct implementation.

The relationship between the initial formal requirements and the final implementation is given by formal refinement maps, whose correctness is expressed by proof obligations. For the developed system, 127 nontrivial proof obligations of internal consistency of the system were generated and proved. Also invariants led to proof obligations and were proved. Generation of these proof obligations was sometimes time consuming. The automated proof search capabilities of the B-Toolkit are rather limited. It is time consuming for the user to decide whether a failed proof indicates an error in the specification. In some cases, we adapted the specification in order to assist the theorem prover.

In a refinement step many details are added. Even formally correct refinement steps can introduce new errors, so there is a need to validate the models at various levels. To this end we used a model checker. This also makes it possible to check a wider set of properties. In B only safety properties can be checked (a bad thing will never happen). Using model checking, we can also check progress properties (something good will eventually happen). Therefore, the core part of AIDA system was formally specified in μCRL, and absence of deadlocks and some temporal properties were established using the model checker CADP. For the validity of these temporal properties, the original model had to be modified.

Although the μCRL toolset is not directly applicable to the B model, the translation from B to μCRL turned out to be relatively straightforward. The specification language of μCRL is sufficiently expressive for this kind of applications. Also the specification of the user requirements was rather straightforward owing to the use of regular expressions. The combination of the μCRL and CADP toolsets turned out to be very effective for a completely automated verification of properties with respect to the AIDA model.

Compared to the informal requirements we started with, the formal description is much better structured. Also, we concluded that the informal specification was not abstract enough, in the sense that operational details were given instead of user requirements. On the other hand, many ambiguities were detected, and had to be resolved in the formal specification. The analysis carried out on the formal models (e.g. exact conditions on when safety conditions lead to logging activity) cannot be performed on the informal models. Also, formal analysis was used to choose how to resolve certain types of ambiguity.

The industrial partners could use the B specification, extend it to contain descriptions of the other tasks, refine it to the special purposes of the implementation, and use the μCRL toolset to verify additional properties. For these activities, however, some expert knowledge is still needed in order to understand the specification languages and make effective use of the tools.

Acknowledgements: This research was funded by the Dutch Ministry of Defense. We would like to thank Wim Pelt and Jan Friso Groote for their support and valuable input, and Radu Mateescu for his advice on using the regular alternation-free μ-calculus.

References

1. J.R. Abrial. *The B-Book.* Cambridge University Press, 1996
2. Th. Arts and I.A. van Langevelde. Correct performance of transaction capabilities. In *Proceedings 2nd Conference on Applications of Concurrency to System Design (ICACSD'2001)*, Newcastle upon Tyne, UK, pp. 35–42. IEEE Computer Society Press, 2001.
3. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, eds, *Proceedings 13th Conference on Computer Aided Verification (CAV'01)*, Paris, France, LNCS 2102, pp. 250–254. Springer-Verlag, July 2001.
4. P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In A. Porto and G.-C. Roman, *Proceedings 4th Conference on Coordination Languages and Models (COORDINATION'2000)*, Limmasol, Cyprus, LNCS 1906, pp. 335–340. Springer-Verlag, 2000.
5. J.A.J.A. Dominicus, A.A. ten Have, M.C. Buitelaar, P.R. Hoek, and F.J. Carati. Functional requirements for an on-board loads and usage monitoring system for the WHL Lynx SH-14D helicopter. Report CR 97568, National Aerospace Laboratory, November 1997.
6. J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, eds, *Proceedings 8th Conference on Computer-Aided Verification (CAV'96)*, New Brunswick, New Jersey, LNCS 1102, pp. 437–440. Springer-Verlag, 1996.
7. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
8. J.F. Groote, J. Pang, and A.G. Wouters. A balancing act: Analyzing a distributed lift system. In S. Gnesi and U. Ultes-Nitsche, eds, *Proceedings 6th Workshop on Formal Methods for Industrial Critical Systems (FMICS'2001)*, Paris, France, pp. 1–12, 2001.
9. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes 1994*, pages 26–62. Workshop in Computing Series, Springer-Verlag, 1995.
10. G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

11. J. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings 17th Symposium on Applied Computing (SAC'2002) – Coordination Models, Languages and Applications*, Madrid, Spain, pp. 351–358. ACM Press, 2002.
12. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings 22nd Conference on Software Engineering (ICSE'2000)*, Limerick, Ireland, pp. 730–733. ACM Press, 2000.
13. J. Julliand, B. Legeard, T. Machicoane, B. Parreaux, and B. Tatibouët. Specification of an integrated circuit card protocol application using the B method and linear temporal logic. In D. Bert, ed., *Proceedings 2nd B Conference (B'98) – Recent Advances in the Development and Use of the B Method*, Montpellier, France, pp. 273–292, LNCS 1393. Springer-Verlag, 1998.
14. K. Lano and H. Haughton. *Specification in B: An Introduction Using the B Toolkit.* World Scientific, 1996.
15. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. Technical Report 3899, INRIA, March 2000. To appear in *Science of Computer Programming.*
16. L. Mikhailov and M. Butler. An approach to combining B and Alloy. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, eds, *Proceedings 2nd Conference of B and Z Users (ZB'2002) – Formal Specification and Development in Z and B*, Grenoble, France, pp. 140–161, LNCS 2272. Springer-Verlag, 2002.
17. J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces architecture using μCRL. In F. Arbab and C.L. Talcott, eds, *Proceedings 5th Conference on Coordination Languages and Models (COORDINATION'2002)*, York, UK, LNCS 2315, pp. 274–290. Springer-Verlag, 2002.
18. E. Sekerinski and K. Sere (eds). *Program Development by Refinement.* Springer-Verlag, 1999.
19. A.L. Vergroesen, P.R. Hoek, F.J. Carati, J.A.J.A. Dominicus, A.A. ten Have, and D. Schütz. An automatic in-flight data acquisition system for the RNLN Lynx helicopter. In *Proceedings 19th International Symposium on Aircraft Integrated Monitoring Systems (AIMS'98)*, Garmisch Partenkirchen, Germany, May 1998.
20. A.G. Wouters. Manual for the μCRL tool set (version 2.8.2). Report SEN-R0130, CWI, December 2001.

A μCRL Specification of AIDA

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% AIDA (Project HELI)                                        %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Parameters
rew
  TIMEOUT2=20                 % was x2p2(x2p1(x2p1(x2p2(36)))) = 600
  TIMEOUT27=TIMEOUT2

% No need to change
map
  TIMEOUT2,TIMEOUT27:->Nat
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% Sorts                                                %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort Bool (Booleans)                                  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Bool
func
  T,F: -> Bool
map
  and: Bool#Bool    -> Bool
  or:  Bool#Bool    -> Bool
  not: Bool         -> Bool
  if:  Bool#Bool#Bool -> Bool
  eq:  Bool#Bool    -> Bool
  gt:  Bool#Bool    -> Bool
var
  b,b1,b2: Bool
rew
  and(T,b)=b                and(b,T)=b
  and(b,F)=F                and(F,b)=F
  and(b,b)=b
  and(b,not(b))=F           and(not(b),b)=F
  and(or(b,b1),b2)=or(and(b,b2),and(b1,b2))
  and(b,or(b1,b2))=or(and(b,b1),and(b,b2))

  or(T,b)=T                 or(b,T)=T
  or(b,F)=b                 or(F,b)=b
  or(b,b)=b
  or(b,not(b))=T            or(not(b),b)=T

  not(F)=T                  not(T)=F
  not(not(b))=b
  not(or(b,b1))=and(not(b),not(b1))
  not(and(b,b1))=or(not(b),not(b1))

  if(T,b1,b2)=b1            if(F,b1,b2)=b2
  if(b,b1,b1)=b1            if(not(b),b1,b2)=if(b,b2,b1)
  if(b,T,b2)=or(b,b2)       if(b,F,b2)=and(not(b),b2)
  if(b,b1,T)=or(not(b),b1)  if(b,b1,F)=and(b,b1)
  if(b,b1,b2)=or(or(and(b,b1),and(not(b),b2)),and(b1,b2))

  eq(b,b)=T eq(b,not(b))=F eq(not(b),b)=F eq(not(b),not(b1))=eq(b,b1)
  eq(F,b)=not(b) eq(b,F)=not(b) eq(T,b)=b eq(b,T)=b
  eq(b,b1)=or(and(b,b1),and(not(b),not(b1)))

  gt(b,b)=F gt(T,F)=T gt(b,T)=F
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort  Nat (Natural numbers with binary representations) %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Nat
func
  0:               -> Nat
  x2p1:        Nat -> Nat     % 2n+1
  x2p2:        Nat -> Nat     % 2n+2
map
  eq:          Nat#Nat       -> Bool

  1,2,3,4,5,6,7,8,9,10,11,12,13,14,
  15,16,17,18,19,20,21,22,23,24,25,
  26,27,28,29,30,31,32,33,34,35,36,
  37,38:                     -> Nat % useful abbreviations

  x2p0:        Nat           -> Nat  % 2n
  succ:        Nat           -> Nat % n+1
  pred:        Nat           -> Nat % n-1 (partial)
  gt:          Nat#Nat       -> Bool % greater than
  if:          Bool#Nat#Nat  -> Nat
var
  n,m: Nat b:Bool
rew
  gt(n,n)=F gt(0,n)=F gt(x2p1(n),0)=T gt(x2p2(n),0)=T

  gt(x2p1(n),x2p2(m))=gt(n,m)
  gt(x2p2(n),x2p1(m))=not(gt(m,n))

  gt(x2p1(n),x2p1(m))=gt(n,m)
  gt(x2p2(n),x2p2(m))=gt(n,m)

  % eq(n,m)=not(or(gt(n,m),gt(m,n))) % sane, but inefficient

  eq(n,n)=T
  eq(x2p1(n),0)=F
  eq(0,x2p1(n))=F
  eq(x2p2(n),0)=F
  eq(0,x2p2(n))=F
  eq(x2p1(n),x2p2(m))=F
  eq(x2p2(n),x2p1(m))=F
  eq(x2p2(n),x2p2(m))=eq(n,m)
  eq(x2p1(n),x2p1(m))=eq(n,m)

   1=x2p1(0)   2=x2p2(0)                  % 1=2*0+1 2=2*0+2
   3=x2p1(1)   4=x2p2(1)                  % 3=2*1+1 4=2*1+2
   5=x2p1(2)   6=x2p2(2)                  % 5=2*2+1 6=2*2+2
   7=x2p1(3)   8=x2p2(3)
   9=x2p1(4)  10=x2p2(4)
  11=x2p1(5)  12=x2p2(5)
```

```
  13=x2p1(6)  14=x2p2(6)
  15=x2p1(7)  16=x2p2(7)
  17=x2p1(8)  18=x2p2(8)
  19=x2p1(9)  20=x2p2(9)
  21=x2p1(10) 22=x2p2(10)
  23=x2p1(11) 24=x2p2(11)
  25=x2p1(12) 26=x2p2(12)
  27=x2p1(13) 28=x2p2(13)
  29=x2p1(14) 30=x2p2(14)
  31=x2p1(15) 32=x2p2(15)
  33=x2p1(16) 34=x2p2(16)
  35=x2p1(17) 36=x2p2(17)
  37=x2p1(18) 38=x2p2(18)

  x2p0(0)=0                          % 2*0=0
  x2p0(x2p1(n))=x2p2(x2p0(n))        % 2(2n+1)=2(2n)+2
  x2p0(x2p2(n))=x2p2(x2p1(n))        % 2(2n+2)=2((2n+1)+1)=2(2n+1)+2

  succ(0)=x2p1(0)                    % 0+1=2*0+1
  succ(x2p1(n))=x2p2(n)              % (2n+1)+1=2n+2
  succ(x2p2(n))=x2p1(succ(n))        % (2n+2)+1=2(n+1)+1

  pred(x2p1(n))=x2p0(n)              % (2n+1)-1=2n+2
  pred(x2p2(n))=x2p1(n)              % (2n+2)-1=2n+1

  if(T,n,m)=n if(F,n,m)=m if(b,n,n)=n if(not(b),n,m)=if(b,m,n)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort LNat (list of Naturals)                               %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LNat
func
  LNat0:->LNat
  add:Nat#LNat->LNat
map
  eq:LNat#LNat->Bool
  if:Bool#LNat#LNat->LNat
  len:LNat->Nat
  cat:LNat#LNat->LNat
  head:LNat->Nat             % return the head of the list
  tail:LNat->LNat            % return the tail of the list
  in:Nat#LNat->Bool
var
  lnat,lnat1:LNat n,m:Nat b:Bool
rew
  eq(lnat,lnat)=T eq(LNat0,add(n,lnat))=F
  eq(add(n,lnat),LNat0)=F
  eq(add(n,lnat),add(m,lnat1))=and(eq(n,m),eq(lnat,lnat1))
  if(T,lnat,lnat1)=lnat if(F,lnat,lnat1)=lnat1
  if(b,lnat,lnat)=lnat if(not(b),lnat,lnat1)=if(b,lnat1,lnat)
```

```
  len(LNat0)=0
  len(add(n,lnat))=succ(len(lnat))
  cat(LNat0,lnat)=lnat cat(lnat,LNat0)=lnat
  cat(add(n,lnat),lnat1)=add(n,cat(lnat,lnat1))
  head(add(n,lnat))=n
  tail(add(n,lnat))=lnat
  in(n,LNat0)=F
  in(n,add(m,lnat))=or(eq(n,m),in(n,lnat))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort  SensorSt (State of a Sensor)                         %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort SensorSt
func
  SensorSt0:->SensorSt % default state
  sst1:Nat->SensorSt  % state of sensor 1 representing a period
                       % R1-R6 by the number 1-6
  sst12:Bool->SensorSt % state of sensor 12 representing
                       % epsilon_lhmy<LevelP27 by False
  sst16:Bool->SensorSt % state of sensor 16 representing
                       % WOW=0Vdc by False and WOW=28Vdc by True
map
  eq:SensorSt#SensorSt->Bool
var sst:SensorSt n,m:Nat b,b1:Bool
rew
  eq(sst,sst)=T
  eq(sst1(n),sst1(m))=eq(n,m)
  eq(sst12(b),sst12(b1))=eq(b,b1)
  eq(sst16(b),sst16(b1))=eq(b,b1)
  eq(sst1(n),sst12(b))=F eq(sst12(b),sst1(n))=F
  eq(sst1(n),sst16(b))=F eq(sst16(b),sst1(n))=F
  eq(sst12(b),sst16(b1))=F eq(sst16(b),sst12(b1))=F
  eq(SensorSt0,sst1(n))=F eq(sst1(n),SensorSt0)=F
  eq(SensorSt0,sst12(b))=F eq(sst12(b),SensorSt0)=F
  eq(SensorSt0,sst16(b))=F eq(sst16(b),SensorSt0)=F

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort  TaskSt (State of a Task)                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort TaskSt
func
  TST_IDLE:->TaskSt
  TST_WAIT:->TaskSt
map
  eq:TaskSt#TaskSt->Bool
var t:TaskSt
rew
  eq(t,t)=T eq(TST_IDLE,TST_WAIT)=F eq(TST_WAIT,TST_IDLE)=F

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%% sort  ButtonsSt (State of the Buttons)                %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ButtonsSt
func
  ButtonsSt0:->ButtonsSt     % default state
  BST_RESET:->ButtonsSt      % reset button has been pressed
map
  eq:ButtonsSt#ButtonsSt->Bool
  is_reset:ButtonsSt->Bool
var s:ButtonsSt
rew
  eq(s,s)=T eq(ButtonsSt0,BST_RESET)=F eq(BST_RESET,ButtonsSt0)=F
  is_reset(ButtonsSt0)=F is_reset(BST_RESET)=T

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort  AudchSt (State of the Audio channel             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort AudchSt
func
  AudchSt0:->AudchSt  % default state (silent)
  AudchSt1:->AudchSt  %  600Hz, 250ms, 50% (2Hz)
  AudchSt2:->AudchSt   % 1600Hz, 62.5ms, 50% (8Hz)
map
  eq:AudchSt#AudchSt->Bool
var s:AudchSt
rew
  eq(s,s)=T eq(AudchSt0,AudchSt1)=F eq(AudchSt0,AudchSt2)=F
           eq(AudchSt1,AudchSt0)=F eq(AudchSt1,AudchSt2)=F
           eq(AudchSt2,AudchSt0)=F eq(AudchSt2,AudchSt1)=F

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort  VidchSt (State of the Video channel             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort VidchSt
func
  VidchSt0:->VidchSt  % default state (no visual signal)
  VidchSt1:->VidchSt  % visual signal to the pilot is "on"
map
  eq:VidchSt#VidchSt->Bool
var s:VidchSt
rew
  eq(s,s)=T eq(VidchSt0,VidchSt1)=F eq(VidchSt1,VidchSt0)=F

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort  DAFSt (State of the Data Admin File             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort DAFSt
func
  DAFSt0:->DAFSt             % default state (no records)
  add:LogEntry#DAFSt->DAFSt  % add something to the file
```

```
map
  update:DAFSt#LogEntry->DAFSt % add something to the file
  eq:DAFSt#DAFSt->Bool
var dst,dst1:DAFSt l,l1:LogEntry
rew
  eq(dst,dst)=T eq(DAFSt0,add(l,dst))=F eq(add(l,dst),DAFSt0)=F
  eq(add(l,dst),add(l1,dst1))=and(eq(l,l1),eq(dst,dst1))
  update(dst,l)=dst

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort LogEntry (Entry in DAF)                            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LogEntry
func
  LogEntryTask:Nat->LogEntry
map
  eq:LogEntry#LogEntry->Bool
var l:LogEntry n,m:Nat
rew
  eq(l,l)=T eq(LogEntryTask(n),LogEntryTask(m))=eq(n,m)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% Actions                                                  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act
  init_sensor,_init_sensor,__init_sensor:Nat
  init_task,_init_task,__init_task:Nat
  init_buttons _init_buttons __init_buttons
  init_audch _init_audch __init_audch
  init_vidch _init_vidch __init_vidch
  init_daf _init_daf __init_daf
  update_sensor,_update_sensor,__update_sensor:Nat
  update_buttons _update_buttons __update_buttons
  run_task,_run_task,__run_task:Nat
  tick_task,_tick_task,__tick_task:Nat

  get_resetstatus,_get_resetstatus,__get_resetstatus:Bool
  get_state_sensor,_get_state_sensor,__get_state_sensor:Nat#SensorSt

  log,_log,__log:LogEntry
  set_audch,_set_audch,__set_audch:AudchSt
  set_vidch,_set_vidch,__set_vidch:VidchSt

  get_timer,_get_timer,__get_timer:Nat#Nat
  tick_timer,_tick_timer,__tick_timer:Nat
  reset_timer,_reset_timer,__reset_timer:Nat
  set_timer,_set_timer,__set_timer:Nat

  change_sensor,_change_sensor,__change_sensor:Nat#SensorSt
  change_buttons,_change_buttons,__change_buttons:ButtonsSt
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% Communicaion Function                                      %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
comm
  init_sensor|_init_sensor=__init_sensor
  init_task|_init_task=__init_task
  init_buttons|_init_buttons=__init_buttons
  init_audch|_init_audch=__init_audch
  init_vidch|_init_vidch=__init_vidch
  init_daf|_init_daf=__init_daf
  update_sensor|_update_sensor=__update_sensor
  update_buttons|_update_buttons=__update_buttons
  run_task|_run_task=__run_task
  tick_task|_tick_task=__tick_task

  get_resetstatus|_get_resetstatus=__get_resetstatus
  get_state_sensor|_get_state_sensor=__get_state_sensor

  log|_log=__log
  set_audch|_set_audch=__set_audch
  set_vidch|_set_vidch=__set_vidch

  get_timer|_get_timer=__get_timer
  tick_timer|_tick_timer=__tick_timer
  reset_timer|_reset_timer=__reset_timer
  set_timer|_set_timer=__set_timer

  change_sensor|_change_sensor=__change_sensor
  change_buttons|_change_buttons=__change_buttons

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% Processes                                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process Monitor                                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Main monitor process
MONITOR(task_list:LNat, sensor_list:LNat)
       =MONITOR_INIT(task_list, sensor_list)

% High level behavior of Monitor
MONITOR_INIT(task_list:LNat, sensor_list:LNat)=
  (  MONITOR_INIT_SENSORS(sensor_list).MONITOR_INIT_BUTTONS
  <|not(eq(sensor_list,LNat0))|>
     MONITOR_INIT_BUTTONS
  ).
  MONITOR_INIT_AUDCH.MONITOR_INIT_VIDCH.MONITOR_INIT_DAF.
```

```
  (  MONITOR_INIT_TASKS(task_list).MONITOR_RUN(task_list, sensor_list)
  <|not(eq(task_list,LNat0))|>
     MONITOR_RUN(task_list, sensor_list)
  )

MONITOR_RUN(task_list:LNat, sensor_list:LNat)=
  (  MONITOR_UPDATE_SENSORS(sensor_list).MONITOR_UPDATE_BUTTONS
  <|not(eq(sensor_list,LNat0))|>
     MONITOR_UPDATE_BUTTONS
  ).

  sum(reset:Bool,_get_resetstatus(reset).
       (  MONITOR_INIT(task_list, sensor_list)
       <|reset|>
          (  MONITOR_RUN_TASKS(task_list).
             MONITOR_TICK_TASKS(task_list).
             MONITOR_RUN(task_list, sensor_list)
          <|not(eq(task_list,LNat0))|>
             MONITOR_RUN(task_list, sensor_list)
          )
       )
     )

% Behavioral details of Monitor
MONITOR_INIT_SENSORS(sensor_list:LNat)=
  (  MONITOR_INIT_SENSOR(head(sensor_list)).
       MONITOR_INIT_SENSORS(tail(sensor_list))
  <|gt(len(sensor_list),1)|>
     MONITOR_INIT_SENSOR(head(sensor_list))
  )

MONITOR_INIT_TASKS(task_list:LNat)=
  (  MONITOR_INIT_TASK(head(task_list)).
       MONITOR_INIT_TASKS(tail(task_list))
  <|gt(len(task_list),1)|>
     MONITOR_INIT_TASK(head(task_list))
  )

MONITOR_INIT_SENSOR(id:Nat)=_init_sensor(id)
MONITOR_INIT_BUTTONS=_init_buttons
MONITOR_INIT_AUDCH=_init_audch
MONITOR_INIT_VIDCH=_init_vidch
MONITOR_INIT_DAF=_init_daf
MONITOR_INIT_TASK(id:Nat)=_init_task(id)


MONITOR_UPDATE_SENSORS(sensor_list:LNat)=
   (  MONITOR_UPDATE_SENSOR(head(sensor_list)).
       MONITOR_UPDATE_SENSORS(tail(sensor_list))
```

```
   <|gt(len(sensor_list),1)|>
     MONITOR_UPDATE_SENSOR(head(sensor_list))
   )

MONITOR_RUN_TASKS(task_list:LNat)=
   (  MONITOR_RUN_TASK(head(task_list)).
       MONITOR_RUN_TASKS(tail(task_list))
   <|gt(len(task_list),1)|>
     MONITOR_RUN_TASK(head(task_list))
   )

MONITOR_TICK_TASKS(task_list:LNat)=
   (  MONITOR_TICK_TASK(head(task_list)).
       MONITOR_TICK_TASKS(tail(task_list))
   <|gt(len(task_list),1)|>
     MONITOR_TICK_TASK(head(task_list))
   )

MONITOR_UPDATE_SENSOR(id:Nat)=_update_sensor(id)
MONITOR_UPDATE_BUTTONS=_update_buttons
MONITOR_RUN_TASK(id:Nat)=_run_task(id)
MONITOR_TICK_TASK(id:Nat)=_tick_task(id)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process Sensor                                        %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SENSOR(id:Nat,st:SensorSt)=
  update_sensor(id).
       sum(st1:SensorSt,change_sensor(id,st1).SENSOR(id,st1))
 +
  get_state_sensor(id,st).SENSOR(id,st)
 +
  init_sensor(id).SENSOR(id,SensorSt0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process Buttons                                       %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
BUTTONS(st:ButtonsSt)=
  update_buttons.sum(st1:ButtonsSt,change_buttons(st1).BUTTONS(st1))
 +
  get_resetstatus(is_reset(st)).BUTTONS(st)
 +
  init_buttons.BUTTONS(ButtonsSt0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process AUDCH                                         %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
AUDCH(st:AudchSt)=
  sum(st:AudchSt,set_audch(st).AUDCH(st))
 +
```

```
  init_audch.AUDCH(AudchSt0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process VIDCH                                                       %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
VIDCH(st:VidchSt)=
  sum(st:VidchSt,set_vidch(st).VIDCH(st))
 +
  init_vidch.VIDCH(VidchSt0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process DAF                                                         %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
DAF(st:DAFSt)=
  sum(l:LogEntry,log(l).DAF(update(st,l)))
 +
  init_daf.DAF(DAFSt0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process TASK                                                        %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TASK2(st:TaskSt)=
  init_task(2)._reset_timer(2).TASK2(TST_IDLE)
 +
  run_task(2).
    (  TASK2RUN(st)
    <|or(eq(st,TST_IDLE),eq(st,TST_WAIT))|>
       TASK2(st)
    )
 +
  tick_task(2).
    (  _tick_timer(2).TASK2(st)
    <|eq(st,TST_WAIT)|>
       TASK2(st)
    )

TASK2RUN(st:TaskSt)=
  sum(sst:SensorSt,_get_state_sensor(1,sst).
    sum(sst1:SensorSt,_get_state_sensor(16,sst1).
         (  (  _set_timer(2).TASK2(TST_WAIT)
            <|eq(st,TST_IDLE)|>
               sum(t:Nat,_get_timer(2,t).
                    (  TASK2(st)
                    <|gt(TIMEOUT2,t)|>
                       _reset_timer(2).TASK2LOG.TASK2(TST_IDLE)
                    )
                  )
            )
         <|and(eq(sst,sst1(1)),eq(sst1,sst16(F)))|>
            (  _reset_timer(2).TASK2(TST_IDLE)
```

```
           <|eq(st,TST_WAIT)|>
              TASK2(TST_IDLE)
           )
        )
      )
     )

TASK2LOG=_log(LogEntryTask(2))._set_vidch(VidchSt1)

TASK3=
  init_task(3).TASK3
 +
  run_task(3).TASK3RUN
 +
  tick_task(3).TASK3

TASK3RUN=
  sum(sst:SensorSt,_get_state_sensor(1,sst).
    sum(sst1:SensorSt,_get_state_sensor(16,sst1).
         (  TASK3LOG.TASK3
         <|and(eq(sst,sst1(3)),eq(sst1,sst16(F)))|>
            TASK3
         )
      )
     )

TASK3LOG=_set_audch(AudchSt1)

TASK4=
  init_task(4).TASK4
 +
  run_task(4).TASK4RUN
 +
  tick_task(4).TASK4

TASK4RUN=
  sum(sst:SensorSt,_get_state_sensor(1,sst).
       (  TASK4LOG.TASK4
       <|or(eq(sst,sst1(5)),eq(sst,sst1(6)))|>
          TASK4
       )
     )

TASK4LOG=_set_audch(AudchSt2)


TASK27(st:TaskSt)=
  init_task(27)._reset_timer(27).TASK27(TST_IDLE)
 +
  run_task(27).
```

```
    (  TASK27RUN(st)
    <|or(eq(st,TST_IDLE),eq(st,TST_WAIT))|>
       TASK27(st)
    )
 +
  tick_task(27).
    (  _tick_timer(27).TASK27(st)
    <|eq(st,TST_WAIT)|>
       TASK27(st)
    )

TASK27RUN(st:TaskSt)=
  sum(sst:SensorSt,_get_state_sensor(12,sst).
    sum(sst1:SensorSt,_get_state_sensor(16,sst1).
         (  (  _set_timer(27).TASK27(TST_WAIT)
            <|eq(st,TST_IDLE)|>
               sum(t:Nat,_get_timer(27,t).
                    (  TASK27(st)
                    <|gt(TIMEOUT27,t)|>
                       _reset_timer(27).TASK27LOG.TASK27(TST_IDLE)
                    )
                  )
            )
         <|and(eq(sst,sst12(F)),eq(sst1,sst16(T)))|>
            (  _reset_timer(27).TASK27(TST_IDLE)
            <|eq(st,TST_WAIT)|>
               TASK27(TST_IDLE)
            )
         )
       )
     )

TASK27LOG=_log(LogEntryTask(27))._set_vidch(VidchSt1)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process TIMER                                            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
TIMER(id:Nat,t:Nat)=
  reset_timer(id).TIMER(id,0)
 +
  tick_timer(id).TIMER(id,succ(t))
 +
  get_timer(id,t).TIMER(id,t)
 +
  set_timer(id).TIMER(id,0)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process Environment                                      %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Random Environment
```

```
ENV=
  (sum(n:Nat,_change_sensor(1,sst1(n))<|and(gt(7,n),gt(n,0))|>delta)
  +
  sum(b:Bool,_change_sensor(12,sst12(b)))
  +
  sum(b:Bool,_change_sensor(16,sst16(b)))
  +
  _change_buttons(ButtonsSt0)
  +
  _change_buttons(BST_RESET)
  ).ENV

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% Complete System                                           %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init hide({__init_sensor,__init_task,__init_buttons,__init_audch,
          __init_vidch,__init_daf,__update_sensor,__update_buttons,
          __run_task,__tick_task,__get_resetstatus,__get_state_sensor},
      encap({init_sensor,_init_sensor,init_task,_init_task,
             init_buttons,_init_buttons,init_audch,_init_audch,
             init_vidch,_init_vidch,init_daf,_init_daf,
             update_sensor,_update_sensor,update_buttons,_update_buttons,
             run_task,_run_task,tick_task,_tick_task,
             get_resetstatus,_get_resetstatus,
             get_state_sensor,_get_state_sensor},
          MONITOR(add(2,add(3,add(4,add(27,LNat0)))),
                   add(1,add(12,add(16,LNat0))))
        ||%hide({__log,__set_audch,__set_vidch},
            encap({log,_log,set_audch,_set_audch,
                   set_vidch,_set_vidch},
                AUDCH(AudchSt0)||VIDCH(VidchSt0)||DAF(DAFSt0)
              ||hide({__get_timer,__tick_timer,
                      __reset_timer,__set_timer},
                  encap({get_timer,_get_timer,tick_timer,_tick_timer,
                         reset_timer,_reset_timer,set_timer,_set_timer},
                    TASK2(TST_IDLE)||TIMER(2,0)
                  )
                )
              ||TASK3||TASK4
              ||hide({__get_timer,__tick_timer,
                      __reset_timer,__set_timer},
                  encap({get_timer,_get_timer,tick_timer,_tick_timer,
                         reset_timer,_reset_timer,set_timer,_set_timer},
                    TASK27(TST_IDLE)||TIMER(27,0)
                  )
                )
              ||%hide({__change_sensor,__change_buttons},
                  encap({change_sensor,_change_sensor,
                         change_buttons,_change_buttons},
                      SENSOR(1,SensorSt0)
```

```
                    ||SENSOR(12,SensorSt0)
                    ||SENSOR(16,SensorSt0)
                    ||BUTTONS(ButtonsSt0)
                    ||ENV
                  )
                %)
              )
            %)
          )
        )
```

B μCRL Specification of properties of AIDA in RAFMC

B.1 Properties of Task 2

The following property says that every execution, where both Sensor 1 is in the range <50% and Sensor 16 is WoW=0 Vdc during at least TIMEOUT2+2 time intervals, and no reset occur during this time, leads to logging this event.

```
macro S() =
    "__change_sensor(x2p1(0),sst1(x2p1(0)))"
end_macro

macro T() =
    "__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),sst16(F))"
end_macro

macro NOT() =
    (not('__change_sensor(x2p1(0),.*)'
      or '__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),.*)'
      or "__change_buttons(BST_RESET)"
      or " __log (LogEntryTask(x2p2(0)))"))*
end_macro

[true*.( (S.NOT.T) | (T.NOT.S) )
  .NOT.( (S.NOT.T) | (T.NOT.S) )
    (*........................    TIMEOUT2 + 2 times    .....................*)
  .NOT.( (S.NOT.T) | (T.NOT.S) )
] false
```

The following property says that logging could not occur if the above mentioned condition had not been being valid for at least TIMEOUT2 time units.

```
macro S() =
    "__change_sensor(x2p1(0),sst1(x2p1(0)))"
end_macro

macro T() =
    "__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),sst16(F))"
end_macro

[  (not(S or T))*.
   (  nil
      |
      (
        (true|nil).(not T)*.
(*........................ TIMEOUT2 times .....................*)
        (true|nil).(not T)*
```

```
      )
      |
      (
         (true|nil).(not S)*.
(*........................ TIMEOUT2 times .....................*)
         (true|nil).(not S)*
      )
   ).
   "__log(LogEntryTask(x2p2(0)))"
]false
and
[  (
      (
         ('__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),.*)' and not T).
         (true|nil).(not T)*.
(*........................ TIMEOUT2 times .....................*)
         (true|nil).(not T)*
      )
      |
      (
         ('__change_sensor(x2p1(0),.*)' and not S).
         (true|nil).(not S)*.
(*........................ TIMEOUT2 times .....................*)
         (true|nil).(not S)*
      )
   )*.
   "__log(LogEntryTask(x2p2(0)))"
]false
```

B.2 Properties of Task 3

This macro definition will be used to express inevitability.

```
(* Inevitable reachability of an action *)

macro inevitable (A) =
    mu X . (< true > true and [ not (A) ] X)
end_macro
```

The following property says that every execution, where both Sensor 1 is in the range >60% and <95% and Sensor 16 is WoW=0 Vdc, leads to logging this event, unless a reset occurs or one of the above Sensors changes its value immediately.

```
library macros.mcl end_library

macro S() =
    "__change_sensor(x2p1(0),sst1(x2p1(x2p1(0))))"
end_macro

macro T() =
```

```
    "__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),sst16(F))"
end_macro

[ true*.
  "__change_buttons(ButtonsSt0)".
  (not('__change_buttons.*'))*.

  S.
  (not('__change_sensor(x2p1(0),.*)'
       or "__change_buttons(BST_RESET)" )
    or S)*.
  T
]
  inevitable
    ("__set_audch(AudchSt1)"
     or "__change_buttons(BST_RESET)"
     or ('__change_sensor(x2p1(0),.*)' and not S))

and
[ true*.
  "__change_buttons(ButtonsSt0)".
  (not('__change_buttons.*'))*.

  T.
  (not('__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),.*)'
       or "__change_buttons(BST_RESET)" )
    or T)*.
  S
]
  inevitable
    ("__set_audch(AudchSt1)"
     or "__change_buttons(BST_RESET)"
     or ('__change_sensor(x2p1(0),.*)' and not T))
```

The following property says that logging could not occur, if the above mentioned condition had never been satisfied.

```
macro S() =
    "__change_sensor(x2p1(0),sst1(x2p1(x2p1(0))))"
end_macro

macro T() =
    "__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),sst16(F))"
end_macro

[ (not(S or T))*.
  ( ( S.(not T)* ) | (T.(not S)* ) ).
  "__set_audch(AudchSt1)"
] false
and
[
```

```
(
   ( S.
     (not T)*.
     ('__change_sensor(x2p1(0),.*)'
        and not("__change_sensor(x2p1(0),sst1(x2p1(x2p1(0))))"))
   )
   |
   ( T.
     (not S)*.
     ('__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),.*)'
       and not("__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),sst16(F))"))
   )
   |
   not(S or T)
 )*.
 "__set_audch(AudchSt1)"
] false
```

B.3 Properties of Task 4

The following property says that every execution, where Sensor 1 is in the range >111.5%, leads to logging this event, unless a reset occurs.

```
library macros.mcl end_library

[ true*.
  "__change_buttons(ButtonsSt0)".
  (not('__change_buttons.*'))*.
  ("__change_sensor(x2p1(0),sst1(x2p1(x2p2(0))))"
     or "__change_sensor(x2p1(0),sst1(x2p2(x2p2(0))))")
] inevitable ("__set_audch(AudchSt2)"
              or "__change_buttons(BST_RESET)")
```

The following property says that logging could not occur, if the above mentioned condition had never been satisfied.

```
[
  (not("__change_sensor(x2p1(0),sst1(x2p1(x2p2(0))))"
       or "__change_sensor(x2p1(0),sst1(x2p2(x2p2(0))))"))*.
  "__set_audch(AudchSt2)"
] false
```

B.4 Properties of Task 27

The following property says that every execution, where both Sensor 12 is in the range <LevelP27 and Sensor 16 is WoW=28 Vdc during at least TIMEOUT27+2 time intervals, and no reset occur during this time, leads to logging this event.

```
macro S() =
    "__change_sensor(x2p2(x2p1(x2p2(0))),sst12(F))"
```

```
end_macro

macro T() =
    "__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),sst16(T))"
end_macro

macro NOT() =
    (not('__change_sensor(x2p2(x2p1(x2p2(0))),.*)'
      or '__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),.*)'
      or "__change_buttons(BST_RESET)"
      or "__log(LogEntryTask(x2p1(x2p1(x2p2(x2p2(0))))))"))*
end_macro

[true*.( (S.NOT.T) | (T.NOT.S) )
  .NOT.( (S.NOT.T) | (T.NOT.S) )
(*........................ TIMEOUT27 + 2 times .....................*)
  .NOT.( (S.NOT.T) | (T.NOT.S) )
] false
```

The following property says that logging could not occur if the above mentioned condition had not been being valid for at least TIMEOUT27 time units.

```
macro S() =
    "__change_sensor(x2p2(x2p1(x2p2(0))),sst12(F))"
end_macro

macro T() =
    "__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),sst16(T))"
end_macro

[  (not(S or T))*.
   (  nil
      |
      (
         (true|nil).(not T)*.
(*........................ TIMEOUT2 times .....................*)
         (true|nil).(not T)*
      )
      |
      (
         (true|nil).(not S)*.
(*........................ TIMEOUT2 times .....................*)
         (true|nil).(not S)*
      )
   ).
   "__log(LogEntryTask(x2p1(x2p1(x2p2(x2p2(0))))))"
]false
and
[  (
      (
         ('__change_sensor(x2p2(x2p1(x2p1(x2p1(0)))),.*)' and not T).
```

```
        (true|nil).(not T)*.
(*........................ TIMEOUT2 times .....................*)
        (true|nil).(not T)*
      )
      |
      (
        ('__change_sensor(x2p2(x2p1(x2p2(0))),.*)' and not S).
        (true|nil).(not S)*.
(*........................ TIMEOUT2 times .....................*)
        (true|nil).(not S)*
      )
   )*.
   "__log(LogEntryTask(x2p1(x2p1(x2p2(x2p2(0))))))"
]false
```

C Development by refinement in B

The section contains the B-model of AIDA. We do not include files that are generated or proved by B-Toolkit automatically and files that are results of renaming (B-model can be downloaded from `http://www.cwi.nl/ ustin/b/AIDA.tar.gz`).

C.1 Abstract Machines

Tasks and timers: Machine `CommDefs` contains definitions of sets and constants used by other machines.

```
MACHINE          CommDefs
SEES String_TYPE
CONSTANTS max, timeout, EOL
PROPERTIES
       max=600 & timeout = 0 & EOL:STRING & EOL="\12"
SETS
       TISTATE ={on, off};
       TSTATE = {idle, active, check, wait};
       CSTATE = {son, soff};
       ConSTATE = {satisfied, notsatisfied};
       MSTATE = {start, stop, monitor};
       HMLEVELS = {peak, trough, npt, 127, 128};
       NRLEVELS = {l10, meq10l50, meq50l60, meq60leq95,
                  m95leq100, m100leq1115, m1115l1158,
                  m11158};
       LHLEVELS = {low, high}
END
```

The specification of `Task27` is analogical to `Task2` machine.

```
MACHINE         Task2
SEES Bool_TYPE, CommDefs
INCLUDES Timer2
```

```
VARIABLES tstate2
INVARIANT tstate2:TSTATE
INITIALISATION tstate2 := idle
OPERATIONS
tinit2 = BEGIN reset2 END;

work2 = CASE tstate2 OF
       EITHER idle THEN
           CHOICE tstate2:=wait || set2(5) OR tstate2:=idle END
       OR wait THEN
           CHOICE tstate2:=check || reset2 OR tstate2:=wait END
       OR check THEN tstate2:=idle
       END
      END;
ss <-- getstate2 = ss:=tstate2;
ttick2 = tick2

END
```

The specification of `Timer2` is analogical to `Timer2` machine.

```
MACHINE         Timer2
SEES Bool_TYPE, CommDefs
VARIABLES tval2, tstat2
INVARIANT tval2:0..max & tstat2:TISTATE
INITIALISATION tval2:=0 || tstat2:=off
OPERATIONS
       reset2 = BEGIN tstat2 := off END ;
       set2(del)= PRE del:NAT & del <= max
                 THEN
                     tval2 := del || tstat2:=on
                  END;
       ok <-- expire2 = IF tval2 = timeout & tstat2=on
                             THEN ok := TRUE
                             ELSE ok := FALSE
                             END;
       tick2 = IF tstat2=on & tval2>0
              THEN tval2 := tval2-1 ELSE tval2:=tval2 END
END

MACHINE         Task3
SEES Bool_TYPE, CommDefs
INCLUDES AudioChController
VARIABLES tstate3
INVARIANT  tstate3:TSTATE
INITIALISATION  tstate3:=idle
OPERATIONS
  tinit3 = tstate3:=idle;
  work3 = CHOICE setaudio OR skip END
END
```

```
MACHINE         Task4
SEES Bool_TYPE, CommDefs
INCLUDES AudioChController
VARIABLES tstate4
INVARIANT  tstate4:TSTATE
INITIALISATION  tstate4:=idle
OPERATIONS
  tinit4 = tstate4:=idle;
  work4 = CHOICE setaudio OR skip END
END
```

```
MACHINE         SensorLHMY
SEES Bool_TYPE, CommDefs
VARIABLES       sstateLHMY
INVARIANT       sstateLHMY:HMLEVELS
INITIALISATION  sstateLHMY:=npt
OPERATIONS
  refreshLHMY = CHOICE sstateLHMY:=npt
               OR sstateLHMY:=peak
               OR sstateLHMY:=trough
               OR sstateLHMY:=l27
               OR sstateLHMY:=l28
           END ;
  xx <-- getstateLHMY = xx := sstateLHMY
END
```

```
MACHINE         SensorWOW
SEES Bool_TYPE, CommDefs
VARIABLES       sstateWOW
INVARIANT       sstateWOW:LHLEVELS
INITIALISATION  sstateWOW:=low
OPERATIONS
  refreshWOW = CHOICE sstateWOW:=low
              OR sstateWOW:=high END;
  xx <-- getstateWOW = xx:=sstateWOW
END
```

```
MACHINE         SensorNR
SEES Bool_TYPE, CommDefs
VARIABLES       sstateNR
INVARIANT       sstateNR:NRLEVELS
INITIALISATION  sstateNR:=l10
OPERATIONS
  refreshNR = CHOICE sstateNR:=l10
               OR sstateNR:=meq10l50
               OR sstateNR:=meq50l60
               OR sstateNR:=meq60leq95
               OR sstateNR:=m95leq100
```

```
               OR sstateNR:=m100leq1115
               OR sstateNR:=m1115l1158
               OR sstateNR:=m11158
           END ;
  xx <-- getstateNR = xx := sstateNR
END

MACHINE         Condition2
SEES Bool_TYPE, CommDefs
OPERATIONS
xx <-- issatisfied2 =
       CHOICE xx := FALSE OR xx:=TRUE END
END
```

Sensors and conditions:

```
MACHINE         OutChannel
SEES CommDefs
VARIABLES      cstate
INVARIANT      cstate:CSTATE
INITIALISATION  cstate:=soff
OPERATIONS
creset = cstate := soff;
cset = cstate:=son
END

MACHINE         OutChController
OPERATIONS
  resetanalog = skip;
  setanalogon = skip
END

MACHINE         DAF
SEES CommDefs, Bool_TYPE
OPERATIONS
storedata = skip
END

MACHINE         DAFController
OPERATIONS
  writetodaf = skip
END

MACHINE         Monitor
SEES CommDefs, Bool_TYPE
VARIABLES   mstate
INVARIANT       mstate : MSTATE
INITIALISATION  mstate := start
OPERATIONS
```

```
minit = BEGIN mstate := monitor END;
monitortask = skip
END
```

Output channels, DAF, controllers and monitor:

C.2 Refinements and implementations

```
IMPLEMENTATION  CommDefsI
REFINES         CommDefs
SEES String_TYPE
END

REFINEMENT      Task2R
REFINES         Task2
SEES            Bool_TYPE, CommDefs, Condition2,
                OutChController, DAFController
INCLUDES        Timer2
VARIABLES       tstate2
INVARIANT       tstate2:TSTATE
INITIALISATION  tstate2:=idle
OPERATIONS
  tinit2 = BEGIN reset2 END;
  work2 =VAR exp
       IN
       CASE tstate2 OF
       EITHER idle THEN exp<--issatisfied2;
                    IF exp=TRUE THEN tstate2:=wait; set2(5)
                    ELSE tstate2:=idle
                    END
       OR wait THEN exp <--expire2;
                    IF exp=TRUE THEN tstate2:=check ; reset2
                    ELSE  tstate2:=wait
                    END
       OR check THEN exp <--issatisfied2;
                    IF exp=TRUE THEN tstate2:=idle;
                    setanalogon; writetodaf
                    ELSE tstate2:=idle
                    END
       END
       END
      END;
  ss <-- getstate2 = ss:=tstate2;
  ttick2 = tick2
END

IMPLEMENTATION  Task2RI
REFINES         Task2R
SEES Bool_TYPE, CommDefs, basic_io, String_TYPE,
```

```
 Condition2, OutChController, DAFController
IMPORTS         Task2_Vvar(TSTATE), Timer2
INVARIANT tstate2=Task2_Vvar
INITIALISATION
       BEGIN
       Task2_STO_VAR(idle)
       END
OPERATIONS
tinit2 = BEGIN
       reset2;
       PUT_STR("TASK2: is idle "); PUT_STR(EOL)
       END;
work2 = VAR curtstate, exp
      IN
       curtstate <-- Task2_VAL_VAR;
       CASE curtstate OF
           EITHER idle THEN exp<--issatisfied2;
                            IF exp=TRUE THEN
                            Task2_STO_VAR(wait); set2(5);
                            PUT_STR("TASK2: set the timer"); PUT_STR(EOL)
                            ELSE
                            Task2_STO_VAR(idle)
                            END
           OR wait THEN exp<--expire2;
                      IF exp=TRUE
                            THEN Task2_STO_VAR(check);reset2;
                            PUT_STR("TASK2: the timer is expired"); PUT_STR(EOL)
                            ELSE Task2_STO_VAR(wait);
                            PUT_STR("TASK2: is waiting"); PUT_STR(EOL)
                      END
           OR check THEN exp<--issatisfied2;
                     IF exp=TRUE
                            THEN setanalogon; writetodaf;
                            Task2_STO_VAR(idle);
                            PUT_STR("TASK2: is done (satisf)"); PUT_STR(EOL)
                            ELSE Task2_STO_VAR(idle);
                            PUT_STR("TASK2: is done (not satisf.)"); PUT_STR(EOL)
                     END
           END
       END
     END;
ss <-- getstate2 = ss<-- Task2_VAL_VAR;
ttick2 = tick2
END

IMPLEMENTATION  TimerI2
REFINES         Timer2
SEES            basic_io, Bool_TYPE,
                CommDefs, String_TYPE
IMPORTS         Timer2Val_Vvar(0..max),
```

```
                 Timer2Stat_Vvar(TISTATE)
INVARIANT        tval2 = Timer2Val_Vvar
                 & tstat2=Timer2Stat_Vvar
INITIALISATION   Timer2Val_STO_VAR(0);
                 Timer2Stat_STO_VAR(off)
OPERATIONS
  reset2 = Timer2Stat_STO_VAR(off) ;
  set2(del) = VAR currtimer
              IN
               currtimer <-- Timer2Val_VAL_VAR;
               PUT_STR("TIMER2: was ");
               PUT_NBR(currtimer); PUT_STR(EOL);
               PUT_STR(EOL);
               IF del<=max THEN Timer2Val_STO_VAR(del) END;
               Timer2Stat_STO_VAR(on)
          END;
  ok <-- expire2 = VAR currtimer, currstat
                 IN
                 currtimer <-- Timer2Val_VAL_VAR;
                 currstat <-- Timer2Stat_VAL_VAR;
                         IF currtimer = timeout & currstat=on THEN
                               ok:=TRUE;
                               PUT_STR("TIMER2: is expired! ");
                               PUT_STR(EOL)
                             ELSE
                               ok:=FALSE;
                               PUT_STR("TIMER2: is not expired! ");
                               PUT_STR(EOL)
                               END
                 END;

  tick2 = VAR currtimer, currstat
               IN
               currtimer <-- Timer2Val_VAL_VAR;
               currstat <-- Timer2Stat_VAL_VAR;
                 PUT_NBR(currtimer); PUT_STR(EOL); */
                IF currtimer>0 & currstat=on
                 THEN
                 Timer2Val_STO_VAR(currtimer-1);
                 currtimer <-- Timer2Val_VAL_VAR;
                 PUT_STR("TIMER2: ticked ");
                 PUT_NBR(currtimer); PUT_STR(EOL)
                ELSE
                 Timer2Val_STO_VAR(currtimer);
                 PUT_STR("TIMER2: Not tickable "); PUT_STR(EOL)
                END
               END

END
```

```
IMPLEMENTATION  Task3I
REFINES         Task3
SEES            String_TYPE, Bool_TYPE, CommDefs,
                basic_io, Condition3,
                AudioChController
IMPORTS         Task3_Vvar(TSTATE)
INVARIANT       tstate3=Task3_Vvar
INITIALISATION  BEGIN
              Task3_STO_VAR(idle)
              END
OPERATIONS
  tinit3 = BEGIN
       Task3_STO_VAR(idle);
       PUT_STR("TASK3: is idle ");
       PUT_STR(EOL)
       END;
  work3 = VAR curstate, exp
       IN
        curstate<--Task3_VAL_VAR;
        exp<--issatisfied3;
        IF (curstate=idle)&(exp=TRUE)
              THEN setaudio;
              PUT_STR("TASK3: is done (satisf)");PUT_STR(EOL)
              ELSE
              PUT_STR("TASK3:is done (not satisf)");PUT_STR(EOL)
              END
       END
END

IMPLEMENTATION  Task4I
REFINES         Task4
SEES            String_TYPE, Bool_TYPE,
                CommDefs, basic_io, Condition4,
                AudioChController
IMPORTS         Task4_Vvar(TSTATE)
INVARIANT       tstate4=Task4_Vvar
INITIALISATION  BEGIN
              Task4_STO_VAR(idle)
              END
OPERATIONS
  tinit4 = BEGIN
       Task4_STO_VAR(idle);
       PUT_STR("TASK4: is idle "); PUT_STR(EOL)
       END;
  work4 = VAR curstate, exp
       IN
        curstate<--Task4_VAL_VAR;
        exp<--issatisfied4;
        IF (curstate=idle)&(exp=TRUE)
              THEN setaudio;
```

```
              PUT_STR("TASK4: is done (satisf)");PUT_STR(EOL)
              ELSE
              PUT_STR("TASK4:is done (not satisf)");PUT_STR(EOL)
              END
       END
END

IMPLEMENTATION  SensorLHMYI
REFINES         SensorLHMY
SEES            Bool_TYPE, CommDefs, String_TYPE,
                Scalar_TYPE, Scalar_TYPE_Ops,
                basic_io
IMPORTS         SStateLHMY_Vvar(HMLEVELS)
INVARIANT       sstateLHMY=SStateLHMY_Vvar
INITIALISATION  SStateLHMY_STO_VAR(npt)
OPERATIONS
  refreshLHMY = VAR rnd
              IN
               rnd<-- RND(4);
               CASE rnd OF
                     EITHER 0 THEN SStateLHMY_STO_VAR(npt)
                     OR 1 THEN SStateLHMY_STO_VAR(peak)
                     OR 2 THEN SStateLHMY_STO_VAR(trough)
                     OR 3 THEN SStateLHMY_STO_VAR(l27)
                     OR 4 THEN SStateLHMY_STO_VAR(l28)
                     END
               END
              END;
  xx <-- getstateLHMY = xx<--SStateLHMY_VAL_VAR
END

IMPLEMENTATION  SensorWOWI
REFINES         SensorWOW
SEES           String_TYPE, String_TYPE_Ops,
               Bool_TYPE, CommDefs, Scalar_TYPE,
               Scalar_TYPE_Ops, basic_io
IMPORTS          SStateWOW_Vvar(LHLEVELS)
INVARIANT     sstateWOW=SStateWOW_Vvar
INITIALISATION  SStateWOW_STO_VAR(low)
OPERATIONS
refreshWOW = VAR rnd
           IN
              rnd <-- RND(1);
                     IF rnd=0 THEN SStateWOW_STO_VAR(low)
                     IF rnd=1 THEN SStateWOW_STO_VAR(high)
           END;
xx <-- getstateWOW = xx <-- SStateWOW_VAL_VAR
END

IMPLEMENTATION  SensorNRI
```

```
REFINES         SensorNR
SEES            Bool_TYPE, CommDefs, String_TYPE,
                Scalar_TYPE, Scalar_TYPE_Ops,
                basic_io
IMPORTS         SStateNR_Vvar(NRLEVELS)
INVARIANT       sstateNR=SStateNR_Vvar
INITIALISATION  SStateNR_STO_VAR(l10)
OPERATIONS
  refreshNR = VAR rnd
          IN
              rnd<-- RND(7);
              CASE rnd OF
                     EITHER 0 THEN SStateNR_STO_VAR(l10)
                     OR 1 THEN SStateNR_STO_VAR(meq10l50)
                     OR 2 THEN SStateNR_STO_VAR(meq50l60)
                     OR 3 THEN SStateNR_STO_VAR(meq60leq95)
                     OR 4 THEN SStateNR_STO_VAR(m95leq100)
                     OR 5 THEN SStateNR_STO_VAR(m100leq1115)
                     OR 6 THEN SStateNR_STO_VAR(m1115l1158)
                     OR 7 THEN SStateNR_STO_VAR(m11158)
                     END
              END
          END;
  xx <-- getstateNR = xx <-- SStateNR_VAL_VAR
END

IMPLEMENTATION  Condition2I
REFINES         Condition2
SEES            Bool_TYPE, CommDefs, SensorNR, SensorWOW
OPERATIONS
  xx <-- issatisfied2 = VAR currNR, currWOW
                     IN
                     currNR <-- getstateNR;
                     currWOW <-- getstateWOW;
                      IF (currNR=l10 or currNR=meq10l50) & currWOW=low
                      THEN
                         xx:=TRUE
                      ELSE
                         xx:=FALSE
                      END
                      END
END

IMPLEMENTATION  Condition27I
REFINES         Condition27
SEES            Bool_TYPE, CommDefs,
                SensorLHMY, SensorWOW
OPERATIONS
  xx <-- issatisfied27 = VAR currLHMY, currWOW
                     IN
```

```
                    currLHMY <-- getstateLHMY;
                    currWOW <-- getstateWOW;
                     IF  currWOW=high & currLHMY=l27
                     THEN
                        xx:=TRUE
                     ELSE
                        xx:=FALSE
                     END
                     END
END

IMPLEMENTATION  Condition3I
REFINES         Condition3
SEES            Bool_TYPE, CommDefs, SensorNR, SensorWOW
OPERATIONS
  xx <-- issatisfied3 = VAR currNR, currWOW
                     IN
                            currNR <-- getstateNR;
                            currWOW <-- getstateWOW;
                            IF currNR=meq60leq95 & currWOW=low THEN
                            xx:=TRUE
                            ELSE
                            xx:=FALSE
                     END
              END
END

IMPLEMENTATION  Condition4I
REFINES         Condition4
SEES            Bool_TYPE, CommDefs, SensorNR
OPERATIONS
  xx <-- issatisfied4 = VAR currNR
                     IN
                            currNR <-- getstateNR;
                            IF currNR=m111511158 THEN
                            xx:=TRUE
                            ELSE
                            xx:=FALSE
                     END
                     END
END

IMPLEMENTATION  OutChannelI
REFINES         OutChannel
SEES            CommDefs, Bool_TYPE, String_TYPE,
                basic_io
IMPORTS        OutChannel_Vvar(CSTATE)
INVARIANT       cstate=OutChannel_Vvar
INITIALISATION  OutChannel_STO_VAR(soff)
OPERATIONS
```

```
  creset = BEGIN
              OutChannel_STO_VAR(soff);
              PUT_STR("ANALOGCH: Signal is OFF ");
              PUT_STR(EOL)
        END;
  cset = BEGIN
              OutChannel_STO_VAR(son);
              PUT_STR("ANALOGCH: Signal is ON ");
              PUT_STR(EOL)
        END
END

IMPLEMENTATION  OutChControllerI
REFINES         OutChController
IMPORTS         OutChannel
OPERATIONS
  setanalogon = cset;
  resetanalog = creset
END

IMPLEMENTATION  DAFI
REFINES         DAF
SEES            CommDefs, Bool_TYPE, String_TYPE, basic_io
OPERATIONS
  storedata = BEGIN PUT_STR("DAF:Write data to data administration file ");
                   PUT_STR(EOL)
              END
END

IMPLEMENTATION  DAFControllerI
REFINES         DAFController
IMPORTS         DAF
OPERATIONS
  writetodaf = storedata
END

IMPLEMENTATION  MonitorI
REFINES         Monitor
SEES            CommDefs, Bool_TYPE, basic_io, String_TYPE
IMPORTS         Monitor_Vvar(MSTATE), Task2, Task27,
                Task3, SensorLHMY, SensorWOW, SensorNR,
                OutChController, DAFController,
                AudioChController
INVARIANT     (mstate=Monitor_Vvar) &
 ((tstate2=check & mstate=monitor) => (tstat2=off)) &
 ((tstate2=idle & mstate=start) => (tstat2=off)) &
 ((tstate27=check & mstate=monitor) => (tstat27=off)) &
 ((tstate27=idle & mstate=start) => (tstat27=off))
INITIALISATION  Monitor_STO_VAR(start)
```

```
OPERATIONS
minit = BEGIN
          Monitor_STO_VAR(monitor);
          tinit2; tinit3; tinit27;
          resetanalog;resetaudio
          END;
monitortask = VAR curmstate
              IN
                     curmstate<--Monitor_VAL_VAR;
                     IF curmstate=monitor
                     THEN
                        refreshWOW; refreshNR;
                        refreshLHMY;
                        work2; work27; work3;
                        ttick2; ttick27
                     ELSE
                        PUT_STR("MONITOR:monitor is not initialised yet");
                        PUT_STR(EOL)
                     END
              END
END
```