The Framework Profile
C1 Implementation
-FREQUENTLY ASKED QUESTIONS-

Alessandro Pasetti & Vaclav Cechticky

7 February 2018

Revision 1.2.0
PP-FQ-COR-0001

P&P Software GmbH
High Tech Center 1
8274 Tägerwilen
Switzerland

Web site: `www.pnp-software.com`
E-mail: `pnp-software@pnp-software.com`

Abstract

This document is the FAQ for the C1 Implementation of the FW Profile. The FW Profile is a specification-level modelling language defined as a restriction of UML. The core modelling constructs offered by the FW Profile are State Machines, Procedures (equivalent to UML's Activity Diagrams), and RT Containers (encapsulations of threads).

The FW Profile is implementation-independent. The C1 Implementation is a C language implementation of the modelling concepts of the FW Profile. The main features of the C1 Implementation are: small memory footprint, small CPU demands, scalability, and high reliability.

The C1 Implementation is provided with a Qualification Data Package which can be used to support the certification of applications built using its components.

Contents

1 General

1.1 What is the C1 Implementation?

The C1 Implementation is a C-language implementation of the modelling concepts of the FW Profile of reference [1]. It offers components to build State Machines, Procedures (activity diagrams) and RT Containers (encapsulations of threads). It is extensively documented and tested and is provided both as free software (Mozilla Public Licence v2) and on a commercial license. Its main features are:

- **Well-Defined Semantics**: clearly and unambiguously defined behaviour.
- **Minimal Memory Requirements**: core module footprint of a few kBytes.
- **Small CPU Demands**: one single level of indirection (due to actions and guards being implemented as function pointers).
- **Scalability**: memory footprint and CPU demands are independent of number and size of state machine, procedure, and RT Container instances.
- **High Reliability**: test suite with 100% code, branch, and condition coverage (excluding error branches for system calls).
- **Formal Specification**: user requirements formally specify the implementation.
- **Requirement Traceability**: all requirements are individually traced to their implementation and to verification evidence.
- **Formal Verification**: key requirements are formally verified using the Spin verifier on a Promela model.
- **Documented Code**: doxygen documentation for all the source code.
- **Demo Application**: complete application demonstrating capabilities and mode of use.
- **Support for Extensibility**: an inheritance-like mechanism is provided through which a *derived state machine* or a *derived procedure* is created from a *base state machine* or *base procedure* by overriding some of its actions or guards.

1.2 What is the FW Profile?

The FW Profile is a specification-level modelling language defined as a restriction of UML. It can be used to specify the behaviour of a software application in a manner that is independent of the application's implementation.

The core modelling constructs of the FW Profile are *state machines*, *procedures*, and *RT Containers*. State machines and procedures are used to model functional behaviour. RT Containers are used to model timing-related behaviour. Procedures are equivalent to UML's Activity Diagrams). The definition of state machines and activity diagrams in UML is complex and ambiguous. The FW Profile offers a simplified but clearer version of these concepts and fully and unambiguously specifies their semantics. The definition of the FW Profile is publicly available from `http://www.pnp-software.com/fwprofile`.

1.3 What are the target applications for the C1 Implementation?

Its minimal memory and CPU requirements, its high reliability, and the availability of a Qualification Data Package (FAQ 1.4) make the C1 Implementation especially well-suited for mission-critical embedded real-time applications. However, there is nothing to prevent the deployment of the C1 Implementation in any context where reliability and efficiency are valuable.

1.4 What is the Qualification Data Package?

The Qualification Data Package is a set of documents which demonstrate the correct specification and implementation of the C1 Implementation software. It consists of the following items:

- A **Test Suite** with 100% code, branch, and condition coverage for the entire implementation (excluding error branches for dynamic memory errors)
- A set of **User Requirements** which formally specify the implementation
- An **Implementation Traceability Matrix** which shows how each requirement is implemented
- A **Verification Traceability Matrix** which shows how the implementation of each requirement is verified
- A **Validation Traceability Matrix** which justifies each requirement with respect to the intended use of the C1 Implementation
- A **User Manual** which discusses implementation issues which are relevant to end-users

The Qualification Data Package can be used by end-applications to support the certification of their software.

1.5 Is the C1 Implementation certified?

We have not yet certified the C1 Implementation but we have put together the documentation required for the certification process in a Qualification Data Package (FAQ 1.4). The Qualification Data Package can be used by end-applications to support the certification of their software.

1.6 How is the C1 Implementation documented?

The C1 Implementation is extensively documented both at source code level (doxygen comments) and through a User Manual and a User Requirements Document.

1.7 Is a GUI-Based Tool Available for the C1 Implementation?

Use of the C1 Implementation is straightforward and a GUI-based tool is hardly necessary. For convenience, a web application is provided which allows user to create state machine diagrams graphically and to generate their implementation code automatically. The web application is accessible from the FW Profile web site at: `http://www.pnp-software.com/fwprofile`.

2 Distribution & Installation

2.1 How is the C1 Implementation distributed?

As one single zip file containing both the code and its documentation. Users who choose to use the C1 Implementation as free software in accordance with the terms of Mozilla Public Licence v2 can download the delivery file from the distribution site at `http://code.google.com/p/fwprofile/`. Users who purchase a commercial licence receive the delivery file from the developers (P&P Software GmbH of Switzerland).

2.2 How is the C1 Implementation installed?

No installation is required. The distribution zip file is unzipped and the source code is ready to be included in the target application.

Unix shell scripts provided with the distribution to build the demo application and the test suite.

Use of the RT Containers (which is optional) requires an implementation of a C Posix library to be available. On most linux systems, a standard library called `libpthread` is available.

2.3 Do I have to use all modules of the C1 Implementation?

No. The C1 Implementation offers three modules covering: State Machines, procedures and RT Containers. These three modules are completely independent of each other and can be used either together or separately.

3 State Machine Definition

3.1 Does the C1 Implementation comply with UML?

The definition of the State Machine concept in UML is complex, often unclear, and sometimes ambiguous. The C1 Implementation adopts the state machine model of the FW Profile (see reference [1]). This is a subset of the UML model and it is clearly and unambiguously defined.

3.2 Is embedding of State Machines supported?

Yes. There is no restriction on the depth of embedding of state machines.

3.3 What kind of pseudo-states are supported?

Obviously, we support the initial and final pseudo-states. We also support the choice pseudo-state with a semantic which allows it to also cover the junction pseudo-state.

3.4 How is the elapsing of time modelled?

Time is not modelled in the C1 Implementation. The State Machine Model underlying this implementation is that of the FW Profile (see reference [1]) which only includes the functional features of the standard UML State Machine Model. Thus, for instance, it is not possible to specify time-triggered transitions in a state machine. Transitions in a state machine only take place in response to a transition command.

As a substitute for time, the FW Profile has introduced two *Execution Counters*: the *State Machine Execution Counter* counts the number of times a state machine has been executed since it was started, and the *State Execution Counter* counts the number of times a state machine has been executed since the current state was entered. Since, in many applications, state machines will be executed periodically, the execution counters can serve as proxies for time.

3.5 Are do-actions supported?

Yes. A do-action can be attached to any state. The do-action is executed when the state machine is sent an "Execute" transition command. State machines only perform some action in response to a transition command. The "Execute" transition command has a special status in the sense that, in addition to potentially triggering a transition from one state to another, it also triggers the execution of the do-action of the current state.

3.6 Are "else" guards supported?

"Else" guards are not directly supported but their effect can be achieved as follows. The out-going transitions from a choice pseudo-state are evaluated in the order in which they were added to the state machine. If the last transitions to be added to a choice pseudo-state is given a guard which always returns true, then this transition will behave like an "Else" transition.

3.7 Can transition commands carry parameters?

The FW Profile allows transition commands to carry parameters and to return values. The parameters represent the parameters passed to the actions and guards triggered by the transition command and the return values represent the values returned by these actions. In the C1 Implementation a transition command is represented by an integer identifier and does not directly carry parameters or generate return values. However, all actions and guards triggered by a transition command are passed a State Machine Data (FAQ 4.7) which can be used to exchange data with them. Thus, although callers cannot directly attach parameters to a transition command or receive return values from them, they can use the state machine data to exchange data with the actions and guards of a state machine.

3.8 What is State Machine Extension?

The C1 Implementation supports an extension mechanism for state machines which is similar to the inheritance-based extension mechanism of object-oriented languages.

A state machine (the *base state machine*) can be extended to create a new state machine (the *derived state machine*). After being created, a derived state machine is a clone of its base. It can then be configured by: overriding one or more of its actions; overriding one or more of its guards; or embedding new state machines in its states.

The extension mechanism is useful where there is a need to define a large number of state machines which share the same topology (same set of states, of choice pseudo-states, and of transitions) but differ either in their actions, or in their guards, or in the internal behaviour of their states.

As an example, consider an application which manages a set of external hardware devices all of which are characterized by the same basic states (e.g. OFF, STANDBY, OPERATIONAL) and by the same behaviour in states OFF and OPERATIONAL but which have different – and device-specific – behaviour in state STANDBY. In this case, it is convenient to proceed as follows:

- A base state machine is defined to model the behaviour which is shared by all devices
- For each device, a state machine is derived which overrides the behaviour in state STANDBY in a manner that is specific to each device.

3.9 In what order are out-going transitions evaluated?

When a state machine receives a transition command, it checks whether any out-going transition from the current state matches the transition command and has a true guard. The FW Profile specifies that if a state or choice pseudo-state has two or more matching out-going transitions with a guard which evaluates to true, the transition which will be executed is the one whose guard is evaluated first. The order of evaluation of the guards is, however, left undefined by the FW Profile.

The C1 Implementation has made the following choice: outgoing transitions from the same state or choice pseudo-state are evaluated in the order in which they were added to the state machine during the state machine configuration process.

4 State Machine Usage

4.1 Are there any usage examples?

Usage examples are provided in the distribution web site `http://code.google.com/p/fwprofile/`; in the doxygen documentation which includes a *state machine example* page which describes how a test state machine is created, configured and used; and in the User Manual.

A *Demo Application* is included in the distribution and can be used as a source of examples. In particular: modules `FwDaHwDev.h` and `FwDaFDCheck.h` can serve as examples of how simple state machines are instantiated and configured and modules `FwDaCurCheck.h`, `FwDaDeltaCheck.h` and `FwDaTempCheck.h` can serve as examples of how derived state machines can be created by extending other state machines.

Finally, a test suite (FAQ 10.3) is available. Its module `FwSmMakeTest.h` consists of a collection of functions each of which creates and configures a state machine. This module can be used as an additional source of examples.

4.2 How is a new state machine created?

A state machine is created by instantiating its state machine descriptor (SMD) (FAQ 9.1). The simplest way to do this is through function `FwSmCreate`. This function, however, uses dynamic memory allocation. If this is undesirable, then either macro `FW_SM_INST` or macro `FW_SM_INST_NOCPS` can be used to instantiate the SMD data structures followed by a call to function `FwSmInit` to initialize them.

After being created, an SMD must be configured (i.e. the states and transitions in the state machine must be defined). This is done using the functions in the `FwSmConfig.h` header file.

4.3 How is a new state machine configured?

After being created, a state machine descriptor (SMD) (FAQ 9.1) must be configured (i.e. the states and transitions in the state machine must be defined). This is done using the functions in the `FwSmConfig.h` header file. Functions are provided to add a new state or a new transition to the SMD or to embed a state machine within a state. At the end of the configuration process, function `FwSmCheck` allows correctness and completeness of the configuration to be checked.

4.4 Can I check whether a state machine is correctly configured?

Yes, with function `FwSmCheck`.

4.5 Which operations can I perform on a state machine?

After being created and configured, state machines can be *started* (with function `FwSmStart`), *stopped* (with function `FwSmStop`) and they can be sent state transition commands (with functions `FwSmMakeTrans` and `FwSmExecute`). A state

machine is initially stopped and, before being started, it does not respond to state transition commands (but it is not an error to send a transition command to a state machine which has not been started or has been stopped). After it is started, the state machine responds to state transition commands (FAQ 4.8) until it is stopped. A state machine can be started and stopped multiple times.

4.6 How are state machine actions and guards implemented?

As function pointers. The C1 Implementation defines the prototype for the state machine actions and guards (through a `typedef`) and users must provide a function complying with this prototype for each action or guard defined in their state machine.

4.7 How is the data upon which a state machine operates defined?

The state machine descriptor (SMD) (FAQ 9.1) includes a field holding a pointer to the *State Machine Data*. The State Machine Data are data which are manipulated by the state machine actions and guards. The exact type of the State Machine Data is defined by applications for each state machine. In most cases, it will take the form a `struct` whose fields represents the inputs and outputs for the state machine actions and guards. The SMD treats the pointer to the State Machine Data as a pointer to `void`. The C1 Implementation provides functions `FwSmSetData` and `FwSmGetData` to set this pointer in, and to retrieve it from, an SMD.

4.8 How do I trigger a state transition?

State transitions are exclusively triggered by transition commands. Each state machine reacts to a finite number of transition commands. Each transition command has an identifier (a non-negative integer). Function `FwSmMakeTrans` is used to send a transition command to state machine.

4.9 Can I change the size of a state machine after creation?

No. When a state machine is created, the user must specify its size (number of states, of choice pseudo-states, of transitions, of actions and of guards) and this is used to allocate the memory for the state machine descriptor (FAQ 9.1). Changing the size dynamically would require re-allocating or releasing memory. This is normally undesirable in mission-critical embedded applications and is therefore not allowed.

4.10 How do I create and configure a derived state machine?

The simplest way to derive a state machine from an existing state machine is through function `FwSmCreateDer`. This function, however, uses dynamic memory allocation. If this is undesirable, then macro `FW_SM_INST_DER` can be used to instantiate the SMD data structures followed by a call to function `FwSmInitDer` to initialize them. After being thus created, the derived state machine is a clone of its base. Selected actions or guards can then be overriden using functions `FwSmOverrideAction` and `FwSmOverrideGuard`. New state machines can be embedded in the states of the derived state machine with function `FwSmEmbed`.

4.11 Can I prevent an action/guard from being overridden?

The C1 Implementation does not offer any means to mark an action or guard in a base state machine as "final". However, a similar effect can be achieved as follows. An action or a guard in a derived state machine is overridden by means of functions `FwSmOverrideAction` and `FwSmOverrideGuard`. These functions require the name of the action or guard to be overridden to be known. Hence, a state machine can prevent one of its actions or guards from being overridden by keeping the name of the function implementing it "hidden" (e.g. by declaring it as `static`).

4.12 Can I directly configure a state machine descriptor?

Yes, but you need to understand the internal structure of the state machine descriptor (this is defined in header file `FwSmPrivate.h`). The potential advantage of direct instantiation and configuration is that a user no longer needs to link to the functions which perform instantiation and configuration (these functions are declared in header files `FwSmDCreate.h`, `FwSmSCreate.h`, and `FwSmConfig.h`). This reduces the memory footprint of the state machine module of the C1 Implementation to, typically, less than 2 kBytes. This option is therefore interesting for applications which are severely memory-constrained.

An example of direct configuration of a state machine descriptor can be found in function `FwSmMakeTestSM5Dir` in the Test Suite (FAQ 10.3).

4.13 Can I print the configuration state of a state machine?

Yes, with function `FwSmPrintConfig`.

5 Procedure Definition

5.1 What is a Procedure?

The FW Profile (FAQ 1.2) uses the name "Procedure" for the entity which in UML is represented as an Activity Diagram. Thus, procedures can be used to model a sequential flow of conditionally executed actions. The definition of Activity Diagrams in UML is complex and often unclear. The FW Profile (FAQ 1.2) has a adopted a simpler model which is clearly and unambiguously defined.

5.2 What are the constitutive elements of a Procedure?

They are a subset of the constitutive elements of a UML Activity Diagram: *action nodes* (which model the execution of some behaviour), *decision nodes* (which model decision points), and *control flows* (which connect nodes to each other). A procedure models a single flow of execution which traverses a set of nodes in sequence.

5.3 What does it mean to "execute a procedure"?

A procedure consists of a set of *action nodes* and *decision nodes* connected by *control flows*. Control flows have *guards*. A control flow can only be traversed if its guard is true. When a procedure is executed, an attempt is made to move from the *current node* to the next node and, from there, to the following node and so on until either the procedure terminate or a guard is found which evaluates to false. Every time an action node is traversed, its action is executed.

5.4 How is the elapsing of time modelled?

Time is not modelled in the C1 Implementation. The Procedure Model underlying this implementation is that of the FW Profile (FAQ 1.2) which only includes the functional features of the standard UML Activity Diagram Model. Thus, for instance, it is not possible to specify time-triggered transitions in a procedure. Movement from one node to the next in a procedure only takes place in response to an execution request to the procedure.

As a substitute for time, the FW Profile has introduced two *Execution Counters*: the *Procedure Execution Counter* counts the number of times a procedure has been executed since it was started, and the *Node Execution Counter* counts the number of times a procedure has been executed since the current node was entered. Since, in many applications, procedures will be executed periodically, the execution counters can serve as proxies for time.

5.5 Can data be exchanged with a procedure?

The FW Profile allows execution commands to carry parameters and to return values. The parameters represent the parameters passed to the actions and guards triggered by the execution request and the return values represent the values returned by these actions. In the C1 Implementation, the execution command to a procedure does not directly carry any parameters. However, all actions and guards triggered by an execution request are passed the Procedure

Data (FAQ 6.7) which can be used to exchange data with them. Thus, although callers cannot directly attach parameters to an execution request to a procedure or receive return values from it, they can use the Procedure Data to exchange data with the actions and guards of a procedure.

5.6 Can a procedure remain stuck in an endless loop?

Yes, this possibility is inherent to the Procedure Concept. As an example, consider a procedure with two actions nodes N1 and N2 and with a control flow from N1 to N2 and a control flow from N2 to N1. Suppose also that the guards on the two control flows are both always true. In such a situation, the execution request that results in node N1 being entered with also cause the procedure to enter an endless loop where nodes N1 and N2 are traversed in sequence and forever.

5.7 Are "else" guards supported?

"Else" guards are not directly supported but their effect can be achieved as follows. The out-going transitions from a decision node are evaluated in the order in which they were added to the procedure when the procedure was configured. If the last transition to be added to a decision node is given a guard which always returns true, then this transition will behave like an "Else" transition.

5.8 What is Procedure Extension?

The C1 Implementation supports an extension mechanism for procedures which is similar to the inheritance-based extension mechanism of object-oriented languages.

A procedure (the *base procedure*) can be extended to create a new procedure (the *derived procedure*). After being created, a derived procedure is a clone of its base. It can then be configured by: overriding one or more of its actions; or overriding one or more of its guards.

The extension mechanism is useful where there is a need to define a large number of procedures which share the same topology (same set of nodes and control flows connecting them) but differ either in their actions or in their guards.

As an example, consider an application which manages a set of external hardware devices all of which are characterized by the same basic switch-on procedure consisting of the following steps: power-on, switch-on, execution of a self-test, collection and evaluation of self-test results, start of normal operation. If the power-on and switch-on actions and the commands to start normal operation are the same for all hardware devices but the self-test is device-specific, then, it is convenient to proceed as follows:

- A base procedure is defined to model the behaviour which is shared by all devices (sequence of operation and power-on, switch-on, and normal operation start commands)
- For each device, a procedure is derived which overrides the actions associated to the nodes performing the self-test.

5.9 In what order are control flows out of a decision node evaluated?

When a procedure must traverse a decision node, it evaluates the guards of all out-going control flows from the decision node and selects the first one which has a true guard. The order of evaluation of the guards is left undefined by the FW Profile.

The C1 Implementation has made the following choice: outgoing control flows from the same decision node are evaluated in the order in which they were added to the procedure during the procedure configuration process.

5.10 Can (or must) procedures and state machines be used together?

Procedure and state machines are distinct and independent concepts both in the way they are defined by the FW Profile and in the way they are implemented by the C1 Implementation. Users can make use of both concepts or of only one.

Often, procedures are used to model the do-actions associated to a state machine.

6 Procedure Usage

6.1 Are there any usage examples?

Usage examples are provided in the distribution web site `http://code.google.com/p/fwprofile/`; in the doxygen documentation which includes a *procedure example* page which describes how a test state machine is created, configured and used; and in the User Manual.

A *Demo Application* is also provided and can be used as a source of examples. In particular modules `FwDaCurRecAction` and `FwDaTempRecAction` can serve as examples of how simple procedures are instantiated and configured.

Finally, a test suite (FAQ 10.3) is available. Its module `FwPrMakeTest.h` consists of a collection of functions each of which creates and configures a procedure. This module can be used as an additional source of examples.

6.2 How is a new procedure created?

A procedure is created by instantiating its procedure descriptor (PRD) (FAQ 9.2). The simplest way to do this is through function `FwPrCreate`. This function, however, uses dynamic memory allocation. If this is undesirable, then macro `FW_PR_INST` or macro `FW_PR_INST_NODEC` can be used to instantiate the PRD data structures followed by a call to function `FwPrInit` to initialize them.

After being created, a PRD must be configured (i.e. the nodes and control flows in the procedure must be defined). This is done using the functions in `FwPrConfig.h`.

6.3 How is a new procedure configured?

After being created, a procedure descriptor (PRD) (FAQ 9.2) must be configured (i.e. the nodes and control flows in the procedure must be defined). This is done using the functions in the `FwPrConfig.h` header file. Functions are provided to add a new node or a new control flow to the PRD. At the end of the configuration process, function `FwPrCheck` allows correctness and completeness of the configuration to be checked.

6.4 Which operations can I perform on a procedure

After being created and configured, procedures can be *started* (with function `FwPrStart`), *stopped* (with function `FwPrStop`) and *executed* (with function `FwPrExecute`). A procedure is initially stopped and, before being started, it does not respond to execution commands (but it is not an error to execute a procedure which has not been started or has been stopped). After it is started, the procedure responds to execution requests (FAQ 5.3) until it is stopped. A procedure can be started and stopped multiple times.

6.5 Can I check whether a procedure is correctly configured?

Yes, with function `FwPrCheck`.

6.6 How are procedure actions and guards implemented?

As function pointers. The C1 Implementation defines the prototype for the procedure actions and guards (through a `typedef`) and users must provide a function complying with this prototype for each action or guard defined in their procedure.

6.7 How is the data upon which a procedure operates defined?

The procedure descriptor (PRD) (FAQ 9.2) includes a field holding a pointer to the *Procedure Data*. The Procedure Data are data which are manipulated by the procedure actions and guards. The exact type of the Procedure Data is defined by applications for each procedure. In most cases, it will take the form of a `struct` whose fields represents the inputs and outputs for the procedure actions and guards. The PRD treats the pointer to the Procedure Data as a pointer to `void`. The C1 Implementation provides functions `FwPrSetData` and `FwPrGetData` to set this pointer in, and to retrieve it from, a PRD.

6.8 Can I change the size of a procedure after creation?

No. When a procedure is created, the user must specify its size (number of action nodes, of decision nodes, of control flows, of actions and of guards) and this is used to allocate the memory for the procedure descriptor (FAQ 9.2). Changing the size dynamically would require re-allocating or releasing memory. This is normally undesirable in mission-critical embedded applications and is therefore not allowed.

6.9 How do I create and configure a derived procedure?

The simplest way to derive a procedure from an existing procedure is through function `FwPrCreateDer`. This function, however, uses dynamic memory allocation. If this is undesirable, then macro `FW_PR_INST_DER` can be used to instantiate the PRD data structures followed by a call to function `FwPrInitDer` to initialize them. After being thus created, the derived procedure is a clone of its base. Selected actions or guards can then be overriden using functions `FwPrOverrideAction` and `FwPrOverrideGuard`.

6.10 Can I prevent an action/guard from being overridden?

The C1 Implementation does not offer any means to mark an action or guard in a base procedure as "final". However, a similar effect can be achieved as follows. An action or a guard in a derived procedure is overridden by means of functions `FwPrOverrideAction` and `FwPrOverrideGuard`. These functions require the name of the action or guard to be overridden to be known. Hence, a procedure can prevent one of its actions or guards from being overridden by keeping the name of the function implementing it "hidden" (e.g. by declaring it as `static`).

6.11 Can I directly configure a procedure descriptor?

Yes, but you need to understand the internal structure of the procedure descriptor (this is defined in header file `FwPrPrivate.h`). The potential advantage of

direct instantiation and configuration is that a user no longer needs to link to the functions which perform instantiation and configuration (these functions are declared in header files `FwPrDCreate.h`, `FwPrSCreate.h`, and `FwPrConfig.h`). This reduces the memory footprint of the procedure module of the C1 Implementation to, typically, less than 2 kBytes. This option is therefore interesting for applications which are severely memory-constrained.

An example of direct configuration of a procedure descriptor can be found in function `FwPrMakeTestPR2Dir` in the Test Suite (FAQ 10.3).

6.12 Can I print the configuration state of a procedure?

No. There is as yet no equivalent of function `FwSmPrintConfig` for procedures.

7 RT Container Definition

7.1 What are RT Containers?

RT Containers are one of the three modelling concepts offered by the FW Profile. State Machines and Procedures allow the functional aspects of a software application to be modelled. RT Containers complement them by offering a means to capture the time-related behaviour of an application. The acronym "RT" stands for "Real-Time".

RT Containers provide a way to encapsulate the activation logic for a functional behaviour. A RT Container can be seen as a representation of a thread which controls the execution of some functional behaviour. The RT Container allows the conditions under which the thread is released to be specified.

7.2 What exactly is "functional behaviour"?

We use this term to designate behaviour which depends neither on time nor on the interaction of different flows of execution. We also say that functional behaviour has "zero logical execution time". The logical execution time of a function is the execution time of that function on a processor with infinite speed and in the absence of pre-emption by higher-priority activities or blocking by lower-priority activities.

Examples of non-functional behaviours include waiting for the elapsing of a certain time interval or waiting for a signal from another thread.

In the FW Profile, functional behaviour is modelled using State Machines or Procedures and non-functional behaviour is modelled using RT Containers.

7.3 What is the advantage of RT Containers over plain threads?

Firstly, a RT Container wraps a thread and offers a ready-made and easy-to-use mechanism through which the thread can be created and notified. Secondly, the RT Container adds logic around a thread which guarantees certain aspects of the behaviour of the thread. For instance, it guarantees that all notifications will eventually be processed (unless the container is stopped); that certain actions (the "finalization actions") will eventually be executed if a container is stopped; that notifications will be correctly buffered and will never be lost; etc.

7.4 What is the threading model of RT Containers?

RT Containers are implemented on top of POSIX threads.

7.5 What if I do not have a POSIX library on my platform?

In this case, you cannot use the RT Container components of the C1 Implementation. Note that this neither limits nor constrains usage of the State Machine and Procedure components of the C1 Implementation.

8 RT Container Usage

8.1 Are there any usage examples?

Usage examples are provided in the distribution web site `http://code.google.com/p/fwprofile/` and in the User Manual. Also, a test suite (FAQ 10.3) is available. Its module `FwRtMakeTest.h` consists of a collection of functions each of which creates and configures a RT Container. This module can be used as an additional source of examples.

8.2 How is a new RT Container created?

A RT Container is created by instantiating its descriptor (RTD) (FAQ 9.3). An RTD is an object of type `struct FwRtDesc`. Hence, a RT Container is created by simply instantiating a variable of this type. After being created, an RTD must be configured. This is done using the functions in `FwRtConfig.h`.

8.3 How is a new RT Container configured?

After being created, a RT Container Descriptor (RTD) (FAQ 9.3) must be configured. This is done using the functions in the `FwRtConfig.h` header file. Functions are provided to change the attributes of the POSIX objects in the container and to define the functional behaviour attached to the container. In all cases, defaults are pre-defined and hence configuration operations are optional. The only mandatory configuration operation is `FwRtInit` which initializes the internal data structures of the POSIX objects used by the RT Container.

8.4 Which operations can I perform on a RT Container

After being created and configured, RT Containers can be *started* (with function `FwRtStart`), *stopped* (with function `FwRtStop`) and *notified* (with function `FwRtNotify`). A container is initially stopped and, before being started, it does not respond to notification requests (but it is not an error to notify a container which has not been started or has been stopped). After it is started, the container responds to notification requests until it is stopped. A container can be started and stopped multiple times.

8.5 Can I attach data to a RT Container?

The RT Container descriptor (RTD) (FAQ 9.3) includes a field holding a pointer to the *Container Data*. The Container Data are data which are manipulated by the functional behaviour attached to the container. The exact type of the Container Data is defined by applications for each container. In most cases, it will take the form of a `struct` whose fields represents the inputs and outputs for the container actions. The RTD treats the pointer to the Container Data as a pointer to `void`. The C1 Implementation provides functions `FwRtSetData` and `FwRtGetData` to set this pointer in, and to retrieve it from, an RTD.

8.6 How do I attach a functional behaviour to a RT Container?

Functional behaviour is encapsulated in functions which are passed to the RT Container as function pointers.

8.7 How do I notify the thread in a RT Container?

The RT Container component offers a function called `FwRtNotify`. With the default configuration of a RT Container, a call to this function causes the thread in the container to be notified. Additionally, users have the option to define a function which filters notifications.

8.8 What kind of notification sources are possible?

There is no restriction. Typically, the setting up of a notification may consist of: (a) a request that the container's thread be notified at some time in the future; (b) a call-back registration to be notified when a certain software condition arises (e.g. a variable changes value, a message arrives, etc); (c) a request to be notified when a hardware interrupt is asserted. The notification source may change dynamically during the container's life.

8.9 How do I change thread characteristics?

RT Containers are built on POSIX threads. The characteristics of POSIX threads are encapsulated in "attribute" objects which define the thread's priority, stack size, etc. By default, a RT Container uses the POSIX-defined default values of these characteristics but it gives access to the thread's attribute object and users can therefore change its characteristics using standard POSIX functions.

9 Implementation Issues

9.1 What is a State Machine Descriptor?

The C1 Implementation represents a state machine through a State Machine Descriptor (SMD). An SMD is a data structure which holds all the information required to describe a state machine. Users normally only manipulate the pointer to an SMD. This is defined as an instance of type `FwSmDesc_t`. Its internal structure is described in header file `FwSmPrivate.h`. Most applications need not be concerned with the internal structure of an SMD and can ignore this header file.

Applications manipulate a state machine by passing its SMD to the functions defined by the C1 Implementation. Thus, for instance, an application executes a state machine through the following function call: `FwSmExecute(smDesc)`. Here, `smDesc` is the pointer to the SMD of the state machine to be executed.

9.2 What is a Procedure Descriptor?

The C1 Implementation represents a procedure through a Procedure Descriptor (PRD). A PRD is a data structure which holds all the information required to describe a procedure. Users normally only manipulate the pointer to a PRD. This is an instance of type `FwPrDesc_t`. The internal structure of a PRD is described in header file `FwPrPrivate.h`. Most applications need not be concerned with the internal structure of a PRD and can ignore this header file.

Applications manipulate a procedure by passing its PRD to the functions defined by the C1 Implementation. Thus, for instance, an application executes a procedure through the following function call: `FwPrExecute(prDesc)`. Here, `prDesc` is the pointer to the PRD of the procedure to be executed.

9.3 What is a RT Container Descriptor?

The C1 Implementation represents a RT Container through a RT Container Descriptor (RTD). An RTD is a data structure which holds all the information required to describe a RT Container. It is defined as an instance of type `struct FwRtDesc`. Users normally only manipulate the pointer to an RTD. This is an instance of type `FwRtDesc_t`.

The internal structure of an RTD is described in header file `FwRtConstants.h`. Most applications need not be concerned with the internal structure of an RTD and can ignore this information.

Applications manipulate a RT Container by passing its RTD to the functions defined by the C1 Implementation. Thus, for instance, an application notifies a container through the following function call: `FwRtNotify(rtDesc)`. Here, `rtDesc` is the pointer to the RTD of the container to be notified.

9.4 How much memory does a State Machine Descriptor take?

This depends on the size of the state machine and on the compiler. The C1 Implementation User Manual gives a formula which allows the size of the state

machine descriptor to be computed from the number of: states, choice pseudo-states, transitions, actions, and guards.

In general, the C1 Implementation is optimized to minimize memory footprint. A medium-sized state machine with 5 states, 3 choice pseudo-states, 10 transitions and with 10 actions and 5 guards takes 195 bytes (assuming "tight" packing).

If the state machine has been derived (through the extension mechanism (FAQ 3.8) from another state machine, then its memory requirement is reduced to 109 bytes.

9.5 How much memory does a Procedure Descriptor take?

This depends on the size of the procedure and on the compiler. The C1 Implementation User Manual gives a formula which allows the size of the procedure descriptor to be computed from the number of: action nodes, decision nodes, control flows, actions, and guards.

In general, the C1 Implementation is optimized to minimize memory footprint. A medium-sized procedure with 5 action nodes, 3 decision nodes, 10 control flows and with 3 actions and 8 guards takes 104 bytes.

If the procedure has been derived (through the extension mechanism (FAQ 5.8) from another procedure, then its memory requirement is reduced to 65 bytes.

9.6 How much memory does a RT Container Descriptor take?

This depends on the compiler and on your POSIX implementation. The C1 Implementation User Manual gives a formula which allows the memory used by the C1-specific data structures of a RT Container to be computed. In addition to this, a RT Container uses one POSIX thread, one POSIX mutex and one POSIX conditional variable.

9.7 What kind of error checking is performed?

The functions which operate on a state machine or on a procedure perform a limited amount of error checking. If they find an error, they report it through the *error code* which stores the identifier of the last error encountered by the implementation. The value of the error code can be read with the `FwSmGetErrCode` function (state machines) or with the `FwPrGetErrCode` function (procedures). Nominally, the error code should be equal to `smSuccess` for function `FwSmGetErrCode` and `prSuccess` for function `FwPrGetErrCode`. If this is not the case, the behaviour of the state machine or procedure is undefined.

Functions which operate on a RT Container only report an error if a POSIX function call has failed. The error code is stored in the RTD and can be accessed with function `FwRtGetErrCode`. It should nominally be equal to zero. If this is not the case, the behaviour of the container is undefined.

9.8 Is the C1 Implementation thread-safe?

The functions offered by the state machine and procedure parts of the C1 Implementation do not use any global data and only operate on data passed as function argument. They are therefore inherently thread-safe in the sense that they can be used by different threads to operate on different state machine or procedure instances (but, obviously, if different threads try to manipulate the same state machine or procedure instance, conflicts may arise).

The RT Container components of the C1 Implementation define data which may be accessed either by the container's internal thread (the Activation Thread) or by an external (user) thread. The functions to start, stop and notify a container (`FwRtStart`, `FwRtStop` and `FwRtNotify`) use a mutex to guarantee access in mutual exclusion and are therefore thread-safe. All other container functions (in particular its configuration functions) are not thread-safe and it is the responsibility of the user to ensure that they are called in mutual exclusion.

9.9 Does the C1 Implementation use dynamic memory allocation?

Users are given the option to create a new state machine or procedure instance using either dynamic memory allocation or static instantiation. In the latter case, a macro is provided to facilitate the instantiation of all data structures required to represent a state machine or a procedure.

Instantiation of a RT Container does not require any dynamic memory allocation at the level of the C1 Implementation. However, a RT Container internally uses a POSIX thread, a POSIX mutex and a POSIX condition variable. These POSIX objects must be initialized as part of the container's configuration (this is done through function `FwRtInit`). Depending on the POSIX library implementation, this may entail dynamic memory allocation.

9.10 Does the C1 Implementation use recursion?

Yes. Use of recursion is inevitable because recursion is intrinsic to the execution model of state machines: a state machine may (recursively) contain other state machine embedded in one of its states and transition requests are propagated (recursively) to the embedded state machines. The depth of recursion is, however, bounded and it is equal to the depth of nesting of state machines.

If an application wishes to avoid the use of recursion, it should not embed any state machine in the state of other state machines.

The procedure and RT Container part of the C1 Implementation do not use recursion.

9.11 Does the C1 Implementation require any special libraries?

The state machine and procedure parts of the C1 Implementation only need the `stdlib` of the C language. The RT Container part needs an implementation of the POSIX library.

9.12 Does the C1 Implementation comply with a coding standard?

The C1 Implementation aims to comply with the Misra C coding standard with a small number of deviations which are individually justified. The level of compliance has, however, not yet been fully verified.

9.13 How is the extension mechanism implemented?

A state machine is represented by its state machine descriptor (SMD) (FAQ 9.1). An SMD is split into two parts: the *Base Descriptor* and the *Extension Descriptor* (see the `FwSmPrivate.h` header file for details). The Base Descriptor holds the information about the state machine topology (its states, choice pseudo-states and their connections) whereas the Extension Descriptor holds the information about the state machine actions and guards and its embedded state machines. During the extension process, only the Extension Descriptor is duplicated whereas the Base Descriptor is shared between a state machine and its children. This significantly reduces memory occupation in a situation where a large number of state machines are derived from the same base state machine.

A similar approach is adopted for procedures. A PRD is split into two parts: the *Base Descriptor* and the *Extension Descriptor* (see the `FwPrPrivate.h` header file for details). The Base Descriptor holds the information about the procedure topology (its nodes and the control flows connecting them) whereas the Extension Descriptor holds the information about the procedure actions and guards. During the extension process, only the Extension Descriptor is duplicated whereas the Base Descriptor is shared between a procedure and its children.

10 Verification & Validation

10.1 How are C1 Implementation requirements verified?

The objective of verification is to demonstrate that the requirements of the C1 Implementation are correctly implemented. For each requirement one of the following verification methods is defined:

- **Verification by review**: the requirement is verified by reviewing the source code or its documentation
- **Verification by analysis**: the requirement is verified by analysing the source code (possibly with a tool)
- **Verification by testing**: the requirement is verified by means of a test in the test suite (FAQ 10.3)

The User Requirement Document in the qualification data package (FAQ 1.4) shows the result of the verification for each requirement in accordance with its verification method.

10.2 How is the C1 Implementation validated?

The objective of validation is to demonstrate that the C1 Implementation meets its intended purpose. The intended purpose of the C1 Implementation is to provide an implementation of the state machine and procedure concepts of the FW Profile (FAQ 1.2) adapted for use in embedded mission-critical applications. Validation is therefore done by analysing each requirement and showing that it supports the implementation of FW Profile in an environment where memory and processing resources are constrained and where high reliability is of paramount importance. This analysis is performed in the User Requirement Document in the qualification data package (FAQ 1.4).

10.3 What is the Test Suite?

The Test Suite is a complete application which demonstrates all aspects of the behaviour of the C1 Implementation. The main program is in file `FwTestSuite.c`. This program consists of a set of *test cases*. The test cases are declared in files `FwSmTestCases.h` (state machine test cases), `FwPrTestCases.h` (procedure test cases) and `FwRtTestCases.h` (RT container test cases). The test suite offers 100% code, branch, and condition coverage (with the exception of branches entered when a system call fails).

10.4 What is the Demo Application?

The Demo Application is a complete application which demonstrates the use of the state machine and procedure modules of the C1 Implementation by implementing a simplified but realistic monitoring system for a *Hardware Device*.

The Hardware Device can be in several states and, when it is powered, it periodically generates a measurement of its temperature and of the current it absorbs. The Demo Application monitors the Hardware Device through three *Failure Detection (FD) Checks*. When a FD Check detects a failure, it executes

a Recovery Action which either switches off the Hardware Device or commands it into Stand-By.

The Demo Application models both the Hardware Device and the FD Checks through state machines. For the FD Checks, the state machine extension mechanism (FAQ 3.8) is used. This is useful to represent the fact that part of their behaviour is common to all FD Check whereas some of their behaviour is specific to each FD Check. The Recovery Actions are modelled as procedures.

10.5 How is test coverage measured?

Test coverage is measured with `gcov` which provides statistics on code and branch coverage. Since the C1 Implementation only uses simple boolean expressions in `if` clauses, branch coverage is equivalent to condition coverage.

11 Licensing & Support

11.1 How is the C1 Implementation licensed?

The C1 Implementation and its documentation are available both as free software under the terms of the GNU General Public License and on a commercial licence.

11.2 Why should I buy a commercial license?

The commercial licence allows the C1 Implementation to be bundled and distributed with proprietary software.

11.3 How do I buy a commercial license?

By contacting the developers at `pnp-software@pnp-software.com`.

11.4 How do I get support?

Users can ask for direct support from `pnp-software@pnp-software.com`. Depending on the type of support required, this may be provided either on a goodwill basis or on a consulting basis.

11.5 How do I report a bug?

Users can either send an e-mail with a description of the problem to `pnp-software@pnp-software.com` or else they can create an entry in the FW Profile Issue Tracker. The latter option requires a user to have an account on the bug tracker. This is provided on request: users should send an e-mail to `pnp-software@pnp-software.com` stating their name and affiliation.

References

[1] Alessandro Pasetti, Vaclav Cechticky: *The FW Profile*. PP-DF-COR-00001, Revision 1.3.0, P&P Software GmbH, Switzerland, 2013

[2] Alessandro Pasetti, Vaclav Cechticky: *The Framework Profile - C1 Implementation User Requirements*. PP-SP-COR-00001, Revision 1.2.0, P&P Software GmbH, Switzerland, 2013

[3] Alessandro Pasetti, Vaclav Cechticky: *The Framework Profile - C1 Implementation User Manual*. PP-UM-COR-0001, Revision 1.2.0, P&P Software GmbH, Switzerland, 2013

