The Framework Profile
- FW Profile -

Alessandro Pasetti & Vaclav Cechticky

13 October 2016

Revision 1.3.1
PP-DF-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 defines the Framework Profile (or *FW Profile* for short). The FW Profile provides the means to model the behaviour of a software application.

The FW Profile is built on three concepts: (a) State Machines as a means to describe state-dependent functional behaviour; (b) Procedures as a means to describe sequential functional behaviour; and (c) RT Containers as a means to describe non-functional behaviour.

The main characteristics of the FW Profile are: (a) Focus on the definition of behaviour independently of software-level design and implementation issues; (b) Separate definition of functional and non-functional behaviour; and (c) Support for the definition of the behaviour of reusable software assets (*Software Frameworks*).

Contents

List of Figures

List of Tables

Listings

1 Change History

This section lists the changes made in the current revision. Changes are classified according to their type. The change type is identified in the second column in the table according to the following convention:

- "**E**": Editorial or stylistic change
- "**L**": Clarification of existing text
- "**D**": A feature present in the previous revision has been deleted
- "**C**": A feature present in the previous revision has been changed
- "**N**": A new feature has been introduced

Table 1: Changes introduced in Revision 1.3.1

Section	Type	Description
4.2	L	Constraint C7, changed "transition command" to "transition trigger" for consistency with terminology used in constraint C5
4.5, 5.5	E	Spelling corrected

Table 2: Changes introduced in Revision 1.3.0

Section	Type	Description
all	E	Change of latex header to bring the document into line with formatting rules used in related documents
2.1	E	Replaced reference to UML 2 with reference to UML
2.2	E	Stylistic changes
2.4	E	Clarified definition of functional and non-functional behaviour
2.5	E	Stylistic changes
2.6	N	New section on formal verification
3.2	L	Clarified that a control flow "target" can also be called "destination"
3.2	E	Minor editorial changes
3.2	L	Clarified that a procedure control flow with no guard is equivalent to a control flow with a guard which always evaluates to true
3.3	E	Corrected "logical time" to: "logical execution time"
3.4	C	Descrition of how a procedure action is specified is no longer in terms of "mechanisms" but rather in terms of typical examples
3.5	E	Minor editorial changes
3.6	E	Section has been entirely re-written for greater clarity
3.7	E	Minor editorial and stylistic changes
3.8	E	Minor editorial and stylistic changes

Section	Type	Description
4.2	L	Clarified that a state machine transition with no guard is equivalent to a transition with a guard which always evaluates to true
4.3	E	Minor editorial and stylistic changes
4.4	E	Minor editorial and stylistic changes
4.5	E	Minor editorial and stylistic changes
4.6	E	Minor editorial changes; modification of example of state machine adaptation
5.1	E	Minor stylistic change
5.3	L	Clarified that, when a container is stopped, a notification is sent to the Activation Thread by incrementing the Notification Counter; added activity diagrams to describe the Start and Stop operations; other minor clarifications
5.3	C	Introduced operation Notify to represent an execution of the Notification Procedure; Moved the start and initial execution of the Notification and Activation Procedures from the Activation Thread to the Start Operation (this brings all initialization actions within the Start operation)
5.4	L	Clarified usage constraint C-1 for RT Container
5.4	E	Minor editorial changes
5.4	C	Modified property P4 to reflect the change in the location of the start of the Notification and Activation Procedures from the Activation Thread to the Start operation
5.4	D	Deleted properties P1 and P2 in tables 3 and 4 (the tables only cover properties which arise from the interaction of the Notification and Activation Threads)
5.5	N	Added table with adaptation points of a RT container
5.6	C	Simplified discussion of mapping of RT containers to design level
6	N	New section on formal verification of FW Profile models
App. A	E	Editorial changes in the title of the appendix section and in the caption of the Promela listing
App. B	N	New appendix section with complete Promela model of formal verification example

2 Introduction

This document defines the Framework Profile (or *FW Profile* for short). The FW Profile provides the means to model the behaviour of software applications. Part of the FW Profile consists of a modelling language defined as a restriction of the UML. The main characteristics of the FW Profile are:

1. Focus on the definition of behaviour independently of software-level design and implementation issues.
2. Separate definition of functional and non-functional behaviour.
3. Support for the definition of the behaviour of reusable software assets (Software Frameworks).

Each of the above features of the FW Profile is discussed in greater detail later in this section.

2.1 Basic Concepts

The FW Profile is built on three basic concepts:

1. **State Machines** to describe state-dependent functional behaviour
2. **Procedures** to describe sequential functional behaviour
3. **RT Containers** to describe non-functional behaviour

The State Machine and Procedure concepts are defined as a restriction of the State Machine and Activity concepts of the UML. They can therefore be represented graphically (as state charts and activity diagrams) using standard UML tools. The RT Container concept is instead specific to the FW Profile.

2.2 Heritage

An early version of the FW Profile was proposed by the authors of this document in the ASSERT Project (see reference [4]) and was later used in the CORDET Project under a contract funded by the European Space Agency (see reference [5]). Variants of the FW Profile have been used by P&P Software in the the following industrial projects:

- Specification of the unit management and failure handling logic of the Attitude and Orbit Control System of the BepiColombo Satellite.
- Definition of a software framework for the real-time part of a line of medical instruments for a major Swiss pharmaceutical company.

The viability and usefulness of the concepts proposed by the FW Profile has thus been demonstrated in practice. This document uses the experience gained in previous projects to extend and refine the FW Profile.

2.3 Behaviour Specification

The FW Profile allows the behaviour of a software application to be specified in a manner that is independent of the choices made at software-level for the

design and implementation of that application.

The objective of the FW Profile is to provide the means to build a logical model for the target application. The logical model must capture all functional and non-functional aspects of the application that are relevant to its user.

The FW Profile is therefore intended to be used in the requirements definition phase of the software development process.

In the design definition phase, the application developer decides how to map the concepts offered by the FW Profile (State Machines, Procedures, and RT Containers) to design-level constructs. This document provides examples of how this mapping could be done but it is important to stress that the definition of the application design is outside the scope of the FW Profile.

2.4 Functional and Non-Functional Behaviour

The FW Profile promotes the separation between the specification of functional and non-functional aspects of an application. The term *functional behaviour* designates behaviour which depends neither on time nor on the interaction between different flows of executions.

The distinction between functional and non-functional behaviour can be understood in terms of the concept of *logical execution time*. The logical execution time of a behaviour is the execution time of that behaviour on a processor with infinite speed and in the absence of pre-emption by higher-priority activities or blocking by lower-priority activities. Functional behaviour is behaviour with zero logical execution time. Non-functional behaviour is behaviour with non-zero logical execution time. Thus, for instance, the presence of wait conditions or of inter-thread handshaking mechanisms makes a behaviour non-functional.

Of the three modelling concepts offered by the FW Profile, the first two (State Machines and Procedures) are exclusively aimed at the definition of functional behaviour whereas the last one (RT Containers) is primarily aimed at the definition of non-functional behaviour.

2.5 Support for Reusability

The FW Profile explicitly supports the specification of *Software Frameworks*. Software Frameworks are a kind of Product Family. A Product Family offers reusable software assets for applications within a certain domain. The reusable assets offered by a software framework are adaptable software components embedded within an architecture. Thus, a software framework predefines the architecture for the applications within its domain and it offers pre-defined components to help instantiating that architecture for a specific application.

The framework instantiation process is the process through which the reusable assets offered by the framework are used to build a specific application. This process is illustrated in figure 1. The framework components are first adapted to meet the needs of the target application and are then assembled to build the target application.

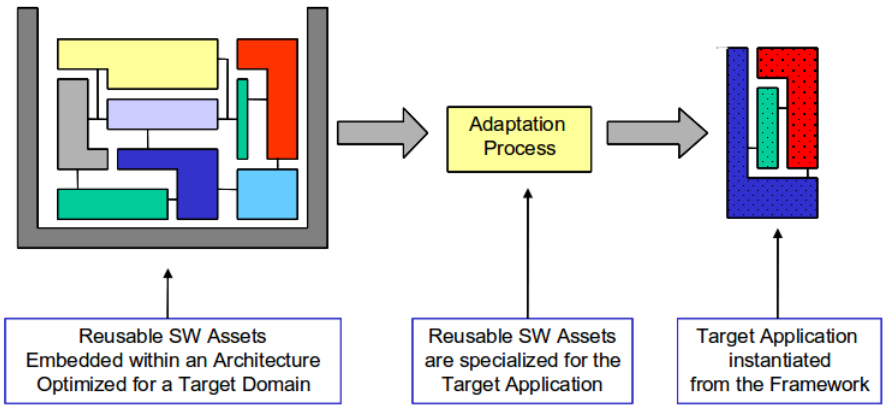

Fig. 1: The Software Framework Concept

The FW Profile supports the specification of software frameworks through the concepts of *Adaptation Point* and *Invariant Property*. An Adaptation Point is a point where the behaviour of a pre-defined component offered by the software framework can be modified to meet the needs of a target application. An Invariant Property is a property that is guaranteed to hold on all applications instantiated from the framework. Thus, an invariant property expresses an aspect of the behaviour of the framework that remains unchanged even after the framework components have been adapted for the target application.

The concept of adaptation point supports the specification of software frameworks because the distinguishing characteristic of a framework component (as opposed to a component which is intended for use in a single application) is its adaptability, namely its ability to be modified to match the needs of several related applications. Hence, specification of a framework requires the ability to specify adaptability. The Adaptation Point concept of the FW Profile fulfills this need.

In the FW Profile, adaptability is supported by allowing certain elements of the State Machine and Procedure models to be marked as “adaptation points”. The objective of adaptability in the specification of a software framework is to cover the variability within the framework domain. Adaptability must therefore be controlled to remain limited at a specific domain. For this purpose, the FW Profile imposes restrictions on the type of elements which can be marked as adaptation points. These restrictions are defined so as to allow the definition of invariant properties for a framework.

The role of the invariant properties in the framework instantiation process is illustrated in figure 2. The framework is specified to encapsulate the properties that are invariant within its domain (i.e. the properties that must be satisfied by all applications that can be instantiated from the framework). The instantiation process (namely the adaptation of the framework components) guarantees that these properties are preserved at application level and application developers can therefore concentrate on adding their own application-specific properties and can assume that framework-level properties remain satisfied.

The adaptation mechanisms offered by the FW Profile are discussed further in section 3.7 for Procedures and in section 4.6 for state machines.

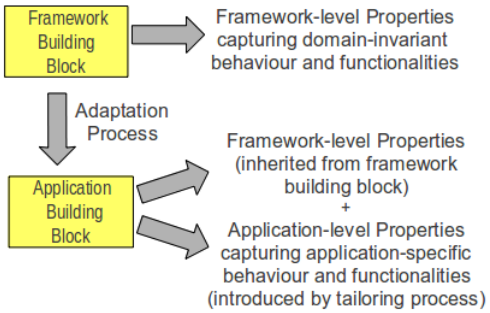

Fig. 2: Invariant Properties

2.6 Support for Formal Verification

The FW Profile allows the requirements of an application to be expressed through a complete and unambiguous model of a target application. This in principle allows formal verification techniques to be used to validate the model (i.e. to prove that the model actually satisfies certain properties). This issue is discussesd at greater length in section 6.

3 The Procedure Model

Procedures are one of the three modelling concepts offered by the FW Profile (see section 2.1). This section defines the procedure model of the FW Profile. The procedure semantics defined in this section can be mapped to a subset of the semantics of the activity concept in UML 2.

3.1 Role of Procedures

Together with the twin concept of state machines (see next section), procedures are intended to capture the functional behaviour of an application (see section 2.4). To some extent, state machines and procedures are interchangeable in the sense that the same abstract behaviour can often be modelled using either one or the other of these two concepts. Procedures are, however, especially well-suited to modelling self-contained behaviour, namely behaviour that is started by some external command but which then continues execution according to an internal logic. Procedures are also better suited at modelling behaviour that consists of a linear or quasi-linear sequence of actions.

3.2 Definition of Procedures

A procedure in the FW Profile consists of the following elements:

- One *initial node*
- One or more *actions nodes* (or actions)
- One or more *control flows*
- Zero or more *decision nodes*
- Zero or more *final nodes*
- Two *execution counters*

The *initial node* is characterized by one control flow which has the initial node as its source and has either an action node or a decision node as its target.

An *action node* (or action) is characterized by the following elements:

- One or more incoming control flows
- One outgoing control flow
- The behaviour associated to the action

The incoming control flows are control flows that have the action as its target. The outgoing control flow is a control flow that has the action as its source.

An action represents a single step within a procedure. It encapsulates behaviour that is not decomposed further within the procedure. The action's behaviour can be defined using natural language or some formalism (e.g. an "action language").

A *control flow* is characterized by the following elements:

- One source

- One target (or destination)
- Zero or one guards

The source and the target are either action nodes or decision nodes. Additionally, the initial node can be the source of a control flow and the final node can be the target of one or more control flows.

The guard is a specification which evaluates either to TRUE or FALSE and which has no side effects. If a control flow has no guard attached to it, then this is equivalent to the control flow having a guard which always evaluates to true.

A *decision node* is characterized by the following elements:

- One or more incoming control flows
- Two or more outgoing control flows

The incoming control flows are control flows which have the decision node as its target. The outgoing control flows are control flows which have the decision node as their source.

For control flows issuing from a decision node, the pre-defined "else" guard is available. This guard returns TRUE if and only if all the other guards attached to control flows issuing from the same decision node return FALSE.

The *final node* is characterized by one or more incoming control flows (namely control flows that have the final node as their target). Note that all final nodes are equivalent and therefore it would be legitimate to allow only one single final node. The option to have more than one is introduced as a matter of convenience.

The *execution counters* are unsigned integers which are exclusively characterized by their value. The first execution counter is called the *Procedure Execution Counter* and the second one is called the *Node Execution Counter*.

The execution counters of a procedure count the number of times the procedure has been executed (one counts the number of times the procedure has been executed since it was started and the other counts the number of times the procedure has been executed since its current node was entered). Since procedures will often be executed periodically, the execution counters can serve as proxies for measuring the elapsing of time.

The following syntactical constraints apply to the definition of the procedure elements:

- C1. The control flows out of a decision node must have a guard.

The following dynamical constraints must be satisfied when a procedure is executed:

- D1. Among the outgoing control flows from a decision node, at least one must have a guard which evaluates to true;
- D2. The evaluation of the guards of a control flow must be free of side-effects;

- D3. The procedure actions and guards must execute in zero logical execution time.

The last constraint implies that the behaviour encapsulated by the actions and by the guards must be purely functional. In practice, this means that actions and guards cannot include time- dependent behaviour or behaviour that depends on synchronization with other flows of execution.

The control flow guards and the actions can act as adaptation points (see section 2.5). For this purpose, the FW Profile pre-defines a stereotype called $\langle\langle AP \rangle\rangle$ that can be attached to these elements. The use of the $\langle\langle AP \rangle\rangle$ stereotype only makes sense in the context of a framework specification. This is discussed further in section 3.7.

3.3 Procedure Behaviour

Four operations may be performed on a procedure: (a) the procedure may be *started*; (b) the procedure may be *executed*; (c) the procedure may be *stopped*; or (d) the procedure may be *run*.

Procedures are purely reactive: they wait for one of these four operations to be performed upon them and they only execute a behaviour in response to one of these operations.

Operations are performed in response to *commands*: the command Start triggers the start operation; the command Execute triggers the execute operation; the command Stop triggers the stop operation; and the command Run triggers the run operation.

A procedure may be in two states: STOPPED or STARTED. Initially, by default, the procedure is in state STOPPED. When the procedure is in state STARTED, it has a *current node*. The current node is either the procedure's initial node or one of its action nodes.

When a procedure is *started*, the following behaviour is executed:

1. If the procedure is in state STARTED, then no further action is performed;
2. If the procedure is in state STOPPED, then it is put in state STARTED, its current node is set equal to its initial node and its execution counters are reset.

When a procedure is *stopped*, the following behaviour is executed:

1. If the procedure is in state STOPPED, then no further action is performed;
2. If the procedure is in state STARTED, then it is put in state STOPPED and its current node is set to an invalid value.

Thus, the Stop and Start commands toggle the state of a procedure and update its current node. This is shown in the state diagram of Figure 3.

When a procedure is *executed*, the following behaviour is executed:

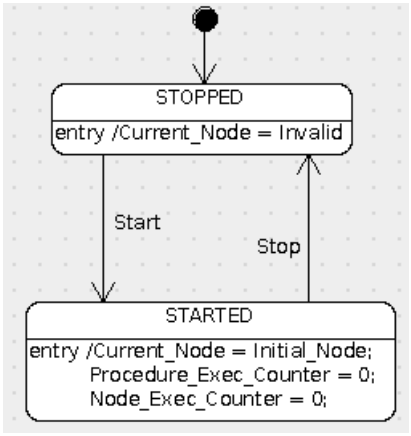

Fig. 3: Procedure Start/Stop Commands

1. If the procedure is in state STOPPED, then no further action is performed;
2. If the procedure is in state STARTED, then its execution counters are incremented by 1 and the guard attached to the outgoing control flow of the current node is evaluated;
3. If the guard evaluates to FALSE, then no further action is performed;
4. If the guard evaluates to TRUE and the target of the outgoing control flow attached to the current node is an action node T, then: (a) the current node is set equal to T, (b) the node execution counter is reset, (c) the behaviour associated to T is executed, (d) the guard on the out-going control flow of T is evaluated and steps 3 and 4 are (recusively) repeated;
5. If the guard evaluates to TRUE and the target of the outgoing control flow attached to the current node is a decision node, then: (a) the guards of the outgoing control flows attached to the decision node are evaluated; (b) if the target of the outgoing control flow whose guard evaluates to TRUE is another decision node, then steps (a) to (d) are performed upon it; (c) if the target of the outgoing control flow whose guard evaluates to TRUE is an action node T, then the current node is set equal to T, the behaviour associated to T is executed, the guard on the out-going control flow of T is evaluated and steps 3 and 4 are (recusively) repeated; (d) if the target of the outgoing control flow whose guard evaluates to TRUE is a final node, the state of the procedure is set to STOPPED and the current node is set equal to an invalid value.
6. If the guard evaluates to TRUE and the target of the outgoing control flow attached to the current node is a final node, then the state of the procedure is set to STOPPED, and the current node is set equal to an invalid value.

Thus, in summary, when a procedure is executed, it tries to traverse the control flow issuing form the current node. If this can be done (i.e. if the guard associated to the control flow evaluates to true), then it advances the execution of the procedure until it finds a guard that evaluates to false or until it finds a final node. Whenever an action node is traversed, its associated behaviour is executed.

The Execute command may carry parameters. These parameters may be passed to any of the actions that are executed as part of the processing of the Execute command.

Note that, at any given time, only one flow of control may be traversing a procedure. This flow of control is advanced every time that the procedure is executed.

The behaviour associated to the execution of a procedure is shown as an activity diagram in Figure 4.

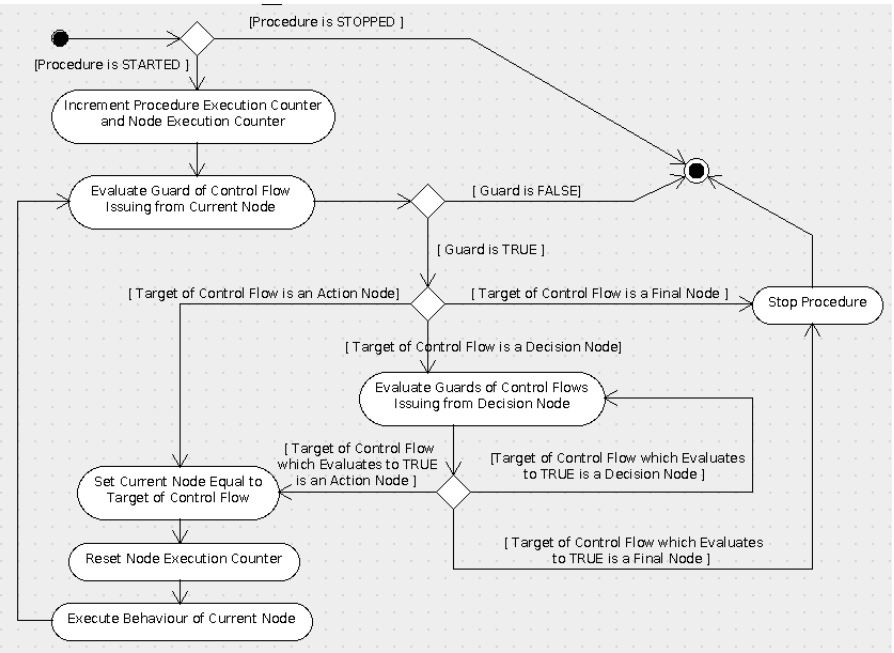

Fig. 4: Procedure Execution Logic

Finally, when a procedure is run, the following behaviour is executed:

1. The procedure is started;
2. The procedure is executed;
3. The procedure is stopped.

Thus, the Run operation is defined in terms of the previous three operations. The Run operation may take parameters which are passed to the Execute operation which is performed as part of the Run operation (step 2 above).

The Run operation is only useful for procedures which execute in one single cycle. It is typically used to perform the actions associated to a state in a state machine.

The execution of the various actions associated to the four procedure operations (Start, Execute, Stop, and Run) is performed in sequence: an action is executed only when the previous one has completed. Note that, since actions are constrained to execute in zero logical execution time, the execution of a procedure

operation will also execute in zero logical execution time.

Requests to perform an operation upon a procedure are executed in sequence. A new request can only be processed by a procedure when the previous one has been fully processed. Procedures have no queues to buffer incoming operation requests.

Note that the procedure operations do not return any values.

3.4 Specification of Procedure Actions

The FW Profile does not mandate any formalism for specifying the behaviour encapsulated in a procedure action. Within a FW Profile Model, a procedure action is typically specified in terms of the operations which can be performed upon a state machine or upon another procedure. Thus, examples of typical actions performed by a procedure include:

- Starting a procedure
- Executing a procedure
- Stopping a procedure
- Running a procedure
- Starting a state machine
- Executing a state machine (i.e. sending an Execute command to it)
- Sending a transition command to a state machine
- Stopping a state machine

3.5 Specification of Procedure Guards

The FW Profile does not mandate any formalism for specifying a guard. However, it pre-defines the following guard: "Wait n Cycles", where n is a positive integer. This guard is true only when the node execution counter is greater than or equal to n. Thus, this guard implies that the flow of control is held for n consecutive execution cycles of the procedure.

The case where the procedure has to wait for one cycle (namely where it has to wait for the next execution) is especially common and for this case the predefined guard "Next Execution" may also be used.

3.6 Graphical Representation

FW Profile procedures can be conveniently represented using standard UML Activity Diagrams. The mapping from their graphical elements to the elements defined in section 3.3 for the procedures of the FW Profile is the obvious one.

As an example, consider the procedure in figure 5. In this figure, when an action consists of performing an operation upon a state machine or upon another procedure (see section 3.4), the following syntax is used: "$\langle operationName \rangle$: $\langle SM\_or\_ProcName \rangle$". Thus, for instance, if an action consists in starting procedure `Proc_A`, the content of the action is expressed as follows: "Start: Proc_A".

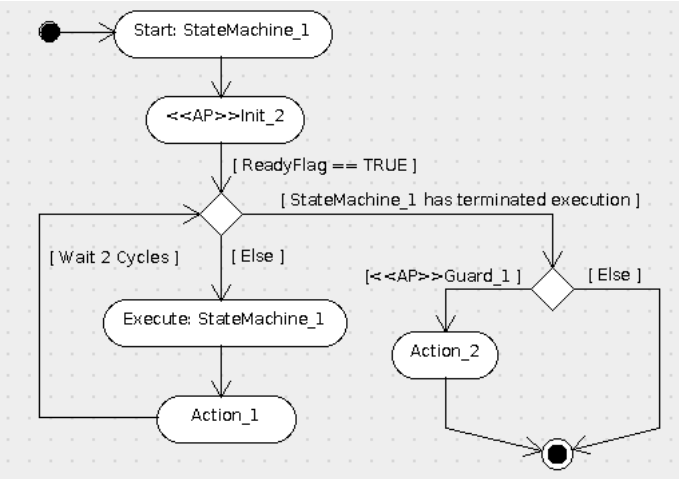

Fig. 5: Procedure Example

After the example procedure of figure 5 has been started and when it is executed for the first time, the procedure starts the `StateMachine_1` state machine and then performs the `Init_2` action. Note that these two actions are performed as part of the first execution cycle for the procedure. In fact, if `ReadyFlag` happens to be true when `Init_2` has terminated execution, then, also as part of its first execution cycle, the procedure will evaluate the guards on the top decision node and will proceed along the control flow with a true guard.

After the guard on `ReadyFlag` has been passed and assuming that `StateMachine_1` does not immediately terminate execution, the procedure remains in a loop where it executes the `StateMachine_1` and the `Action_1` every second execution cycle until the `StateMachine_1` terminates. At that point, the procedure itself terminates either immediately or after having executed `Action_2`. The choice between these two options (immediate termination or termination after execution of `Action_2`) depends on the value of the `Guard_1` guard.

The example procedure of Figure 5 has two adaptation points (see next section): users may adapt this procedure by changing the definition of `Initi_2` and/or `Guard_1`.

3.7 Procedure Adaptation

If a procedure is used to specify the behaviour of a framework, then it is necessary to identify its adaptation points (see section 2.5). The FW Profile relies on the use of the $\langle\langle AP \rangle\rangle$ stereotype to identify adaptable elements within a procedure. The FW Profile allows this stereotype to be associated to the following elements:

- Actions in Action Nodes
- Guards on Control Flows

The presence of the $\langle\langle AP \rangle\rangle$ stereotype on any of the elements listed above may mean one of two things:

1. The content of the stereotyped element is not defined at framework level and the definition must be done at application level, or
2. A default content for the stereotyped element is defined at framework level but this can be overridden at application level.

The FW Profile does not provide the means to discriminate between the two cases above.

In section 2.5 it was explained that the adaptation mechanisms supported by the FW Profile are designed to preserve certain invariant properties defined at framework level. What, then, are the properties which are invariant with respect to the adaptation mechanism defined above?

In order to answer this question, it is necessary to consider what *cannot* be modified through the allowed adaptation mechanisms: neither can new nodes or new control flows be added to a procedure, nor can new guards be added to existing control flows. Thus, the features of a procedure which cannot be changed are: (a) the *topology* of the procedure, (b) the *conditions*, expressed in terms of the outcomes of guards, which lead to a control flow being traversed, and (c) the *sequence of actions* which are executed when a control flow is traversed.

The invariant properties of a procedure are therefore those which describe behaviour which depends on the topology of a procedure and on the sequence of control flows traversed and actions performed by the procedure. By contrast, properties which depend on the *content* of the actions or of the guards are typically not invariant since the content of the actions may change during the framework instantiation process.

As an example, consider again the example procedure of the previous section. The following property holds on the procedure: "`Action_1` is executed as many times as `StateMachine_1` is executed". This property is invariant because it depends merely on the procedure topology and cannot be broken by the adaptation process: all procedures obtained by adapting the procedure of figure 5 satisfy this property.

Similarly, consider the following property: "if `StateMachine_1` is executed, then the `ReadyFlag` was true at least once in the past". This property is invariant because it depends on the topology of the procedure and on the conditions attached to its control flows. Note that `ReadyFlag` may be undefined at framework level (or it may be defined at framework level but may be overridden at application level) but this does not affect the validity of the property because the property does not depend on the value of the `ReadyFlag` (which may change during the framework instantiation process) but simply on its presence on a certain transition guard in the procedure (which cannot change during the framework instantiation process the guard is not marked as an adaptation point).

Finally, as an example of a property which is not invariant with respect to the adaptation mechanisms allowed by the FW Profile, consider the following: "if the procedure executes `Action_1`, then it will also execute `Action_2` when it terminates". This property may hold in some cases (whenever `Guard_1` happens to be true) but it is not possible to say whether it holds in all cases because the

definition of `Guard_1` (which is an adaptation point for the procedure) may be modified during the framework instantiation process.

3.8 Mapping to Design Level

The FW Profile is aimed at the modelling of behaviour. The concepts offered by the FW Profile to model behaviour can be mapped in many different ways to software-level design artifacts. In the case of the procedure concept, examples of possible mappings to the software level include:

- A procedure is mapped to a single class with an Execute method as execution trigger; actions and guards are mapped to dedicated methods in the class; adaptation is through inheritance.
- A procedure is mapped to a single class associated to classes representing its actions nodes and guards (a class for each action node or guard); adaptation is through inheritance.
- A procedure is mapped to a C-style module; actions and guards are mapped to functions in the module; adaptation is through delegation.

Obviously, the list above is non-exhaustive but the point it tries to make is that the use of the FW Profile to model the behaviour of an application does not dictate its software-level design.

3.9 UML 2 Compliance

The procedure model offered by the FW Profile complies with the UML 2 activity model in the sense that the elements of the procedure concept of the FW Profile and their semantics can be mapped in an obvious way to a subset of the elements of the activity concept of UML 2.

The execution counters are specific to the FW Profile. They have been introduced as a substitute for the concept of time (which does not exist in the FW Profile Procedures): if a procedure is executed periodically, then the value of its execution counters is proportional to the time elapsed since the procedure was started (Procedure Execution Counter) or since the current node was entered (Node Execution Counter).

It should be emphasized that the procedure model proposed by the FW Profile is far more restrictive than the activity model supported by UML 2. This is because the FW Profile uses state machines to model purely functional (non-time-related and non-concurrent) behaviour.

4 The State Machine Model

State machines are one of the three modelling concepts offered by the FW Profile (see section 2.1). This section defines the state machine model of the FW Profile. This model is defined as a restriction of the state machine model of the UML 2.

4.1 Role of State Machines

Together with the twin concept of procedures, state machines are intended to capture the functional behaviour of an application (see section 2.4). To some extent, state machines and procedures are interchangeable in the sense that the same abstract behaviour can often be modelled using either one or the other of these two concepts.

State machines are, however, especially well-suited to modelling reactive behaviour, namely behaviour that is triggered by external commands. State machines are also better suited at modelling behaviour that is state-dependent, namely behaviour that is dependent on the past history of a component or application.

4.2 Definition of State Machines

A state machine in the FW Profile consists of the following elements:

- One *initial pseudo-state*
- One or more *states*
- One or more *state transitions*
- Zero or more *choice pseudo-states*
- Zero or more *final pseudo-states*
- Two *execution counters*

The *initial pseudo-state* is characterized by one transition which has the initial pseudo-state as its source and has either a state or a choice pseudo-state as its target.

A *state* is characterized by the following elements:

- Zero or more *entry actions*
- Zero or more *do actions*
- Zero or more *exit actions*
- Zero or one *embedded state machine*
- One or more *incoming transitions*
- Zero or more *outgoing transitions*

The state actions represent behaviour which is not decomposed further within the state machine. Actions' behaviour can be defined using natural language or some formalism (e.g. an "action language"). An embedded state machine is a state machine that is embedded within the state. Embedded state machines

are defined in the same way and have the same semantics as other FW Profile state machines. An incoming transition is a state transition that has the state as its target. An outgoing transition is a state transition that has the state as its source.

A *state transition* is characterized by the following elements:

- One *transition source*
- One *transition target* (or *transition destination*)
- Zero or one *transition trigger* (or *transition command*)
- Zero or one *transition guard*
- Zero or more *transition actions*

The transition source and the transition target are either a state or a pseudo-state. The transition trigger is the command that triggers the execution of the transition. A transition guard is a specification that evaluates either to TRUE or to FALSE and has no side effects. A transition with no guard is equivalent to a transition with a guard which always evaluates to TRUE. A transition action represents behaviour which is not decomposed further within the state machine. A transition action behaviour can be defined using natural language or some formalism (e.g. an "action language").

Transition commands may carry parameters and may return values. The parameters and return values are not defined further by the FW Profile. They represent parameters that are passed to the actions and guards and values that are returned by the actions.

A *choice pseudo-state* is characterized by the following elements:

- One or more *incoming transitions*
- One or more *outgoing transitions*

An incoming transition is a state transition that has the choice pseudo-state as its target. An outgoing transition is a state transition that has the choice pseudo-state as its source.

For transitions issuing from a choice pseudo-state, the pre-defined "else" guard is available. This guard returns TRUE if and only if all the other guards attached to transitions issuing from the same choice pseudo-state return FALSE.

The *final pseudo-state* is characterized by one or more incoming transitions (namely state transitions that have the final pseudo-state as their target). Note that all final pseudo-states are equivalent and therefore it would be legitimate to allow only one single final pseudo-state. The option to have more than one is introduced as a matter of convenience.

The *execution counters* are unsigned integers which are characterized by their value. The first execution counter is called the *State Machine Execution Counter* and the second one is called the *State Execution Counter*.

The following syntactical constraints apply to the definition of the state machine

elements:

- C1. The same pseudo-state cannot be both source and target for a transition;
- C2. The source and target of a transition cannot both be choice pseudo-states;
- C3. The transition that has the initial pseudo-state as source can have neither a guard nor a trigger;
- C4. Deleted;
- C5. Transitions that have a choice pseudo-state as source cannot have a transition trigger;
- C6. Deleted;
- C7. Transitions that have a state as a source must have a transition trigger;
- C8. Transitions can only link states and/or pseudo-states that belong to the same state machine.

The last constraint implies that transitions from an outer state machine to an embedded state machines or vice-versa are not allowed. Note, however, that the same transition command may trigger a transition both in an outer state machine and in one of its embedded state machine. This is discussed in the next section.

The following dynamical constraints must be satisfied when a state transition is executed:

- D1. Among the outgoing transitions from a choice pseudo-state, at least one must have a guard which evaluates to true;
- D2. The evaluation of the guards of a transition must be free of side-effects.
- D3. The state actions (entry, do, and exit actions) and the transition actions and guards must execute in zero logical execution time.

The last constraint implies that the behaviour encapsulated by actions and guards is constrained to be *purely functional*. In practice, this means that actions and guards cannot include time-dependent behaviour or behaviour that depends on synchronization with other flows of executions.

One type of transition command – the *Execute* command – has a special status in that it triggers the execution of the current state's do-action. The Execute command models the situation (common in embedded control systems) of a cyclical scheduler periodically triggering an application and advancing its execution.

As a matter of terminology, when a state machine is sent the Execute command, the state machine is said to be *executed*.

The execution counters of a state machine count the number of times the state machine has been executed (one counts the number of times the state machine has been executed since it was started and the other counts the number of times

the state machine has been executed since its current state was entered). Since state machines will often be executed periodically, the execution counters can serve as proxies for measuring the elapsing of time.

The transition guards, the transition actions and the state actions can act as adaptation points (see section 2.5). For this purpose, the FW Profile pre-defines a stereotype called $\langle\langle AP \rangle\rangle$ that can be attached to these elements. The use of the $\langle\langle AP \rangle\rangle$ stereotype only makes sense in the context of a framework specification. This is discussed further in section 4.6.

4.3 State Machine Behaviour

Three operations may be performed on a state machine: (a) the state machine may be *started*; (b) the state machine may be sent a *transition command*; or (c) the state machine may be *stopped*.

State machines are purely reactive: they wait for one of these three operations to be performed upon them and they only execute some behaviour in response to one of these operations.

A state machine can be either in a *defined state* or in an *undefined state*. A state machine is in a defined state from the time it has completed the transition out of its initial pseudo-state to the time it has either completed the transition into one of its final pseudo-states or has been stopped.

When a state machine is in a defined state, it has a *current state*. The current state is one of the states of the state machine.

When a state machine is *started*, the following behaviour is executed:

1. If the state machine is in a defined state, then no further action is taken.
2. If the state machine is in an undefined state, then its execution counters are reset and the action associated to the transition out of its initial pseudo-state is executed. If several transition actions are present, they are executed in the order in which they are listed.
3. If the destination of the transition out of the initial pseudo-state is a choice pseudo- state, then the guards of the outgoing transitions from the choice pseudo-state are evaluated and the actions associated to the transition with a guard evaluating to true is executed. If several transition actions are present, they are executed in the order in which they are listed.
4. If the destination of the transition out of the initial pseudo-state is a state, then the current state of the state machine is set equal to that state.
5. If the destination of the transition out of the initial pseudo-state is a choice pseudo- state and if the selected transition out of the choice pseudo-state has a state as a target, then the current state of the state machine is set equal to that target state.
6. The entry action of the current state is executed. If several entry actions are present, they are executed in the order in which they are listed.
7. If the current state has an embedded state machine, then the embedded state machine is started.

8. If the destination of the transition out of the initial pseudo-state is a choice pseudo- state and if the selected transition out of the choice pseudo-state has the final pseudo- state as a target, then the state machine remains in an undefined state.

With reference to point 3, it is noted that at least one of the guards on the outgoing transitions from a choice pseudo-state is guaranteed to be true because of constraint D1 in the previous section.

When a state machine is *stopped*, the following behaviour is executed:

1. If the state machine is in an undefined state, no further action is taken.
2. If the state machine is in a defined state and its current state has an embedded state machine, the embedded state machine is stopped.
3. The exit action of the current state is executed. If several exit actions are present, they are executed in the order in which they are listed.
4. The state machine is set to an undefined state.

The logic of the start and stop commands for state machines is shown in Figure 6 as two activity diagrams.

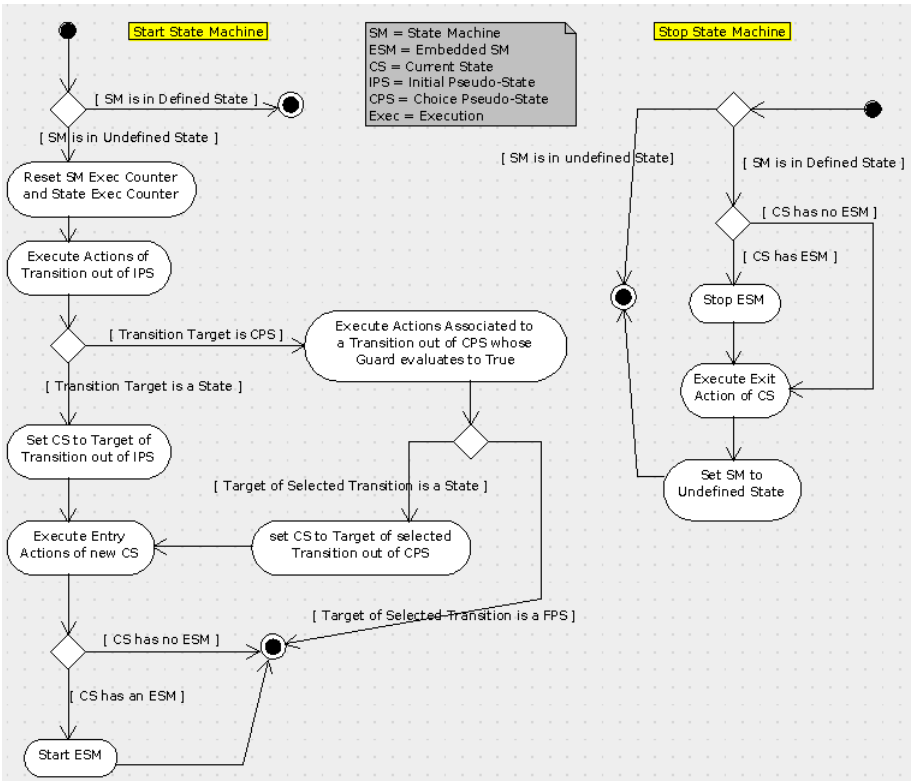

Fig. 6: Logic for the Start and Stop Commands to a State Machine

When a *transition command T is sent* to a state machine S, then the following behaviour is executed:

1. If S is in an undefined state, then no further action is taken.

2. If T is the Execute command, then the execution counters of the state machine are incremented and the do-action associated to the current state of S is executed. If several do-actions are present, they are executed in the order in which they are listed.
3. If S is in a defined state and the current state of S has an embedded state machine SE, then the transition command T is propagated to SE.
4. If there are no transitions from the current state of S that have T as their trigger, then no further action is taken.
5. If there are one or more transitions from the current state of S that have T as their trigger, then their guards are evaluated in sequence. The order of the evaluation is undefined. The absence of a guard is equivalent to a guard that returns TRUE.
6. When the first transition is found whose guard evaluates to TRUE, then that transition is executed.

The logic that governs the processing of a transition command by a state machine is shown in Figure 7 as an activity diagram. Note that this logic merely describes the circumstances under which a transition within a state machine is executed but it does not define the logic according to which the transition is executed. This is done below (see also Figure 8).

When a *transition is executed*, then the following behaviour is executed:

1. If the source state of the transition is a state and that state has an embedded state machine, then the embedded state machine is stopped.
2. If the source state of the transition is a state, then the exit action associated to the source state is executed. If several exit actions are present, they are executed in the order in which they are listed.
3. The transition action associated to the transition is executed. If several transition actions are present, they are executed in the order in which they are listed.
4. If the target of the transition is a choice pseudo-state, then the guards of the out-going transitions from the choice pseudo-state are evaluated in sequence until one is found that evaluates to true and that transition is executed.
5. If the target of the transition is a final pseudo-state, then the state machine is set to an undefined state and no further action is taken.
6. If the target state of the transition is a state, then the current state of the state machine is updated to be equal to the target state of the transition and the state execution counter is reset.
7. If the target state of the transition is a state, then the entry action of the target state is executed. If several entry actions are present, they are executed in the order in which they are listed.
8. If the target state of the transition is a state and that state has an embedded state machine, then the embedded state machine is started.

With reference to point 4, it is noted that at least one of the guards on the outgoing transitions from a choice pseudo-state is guaranteed to be true because

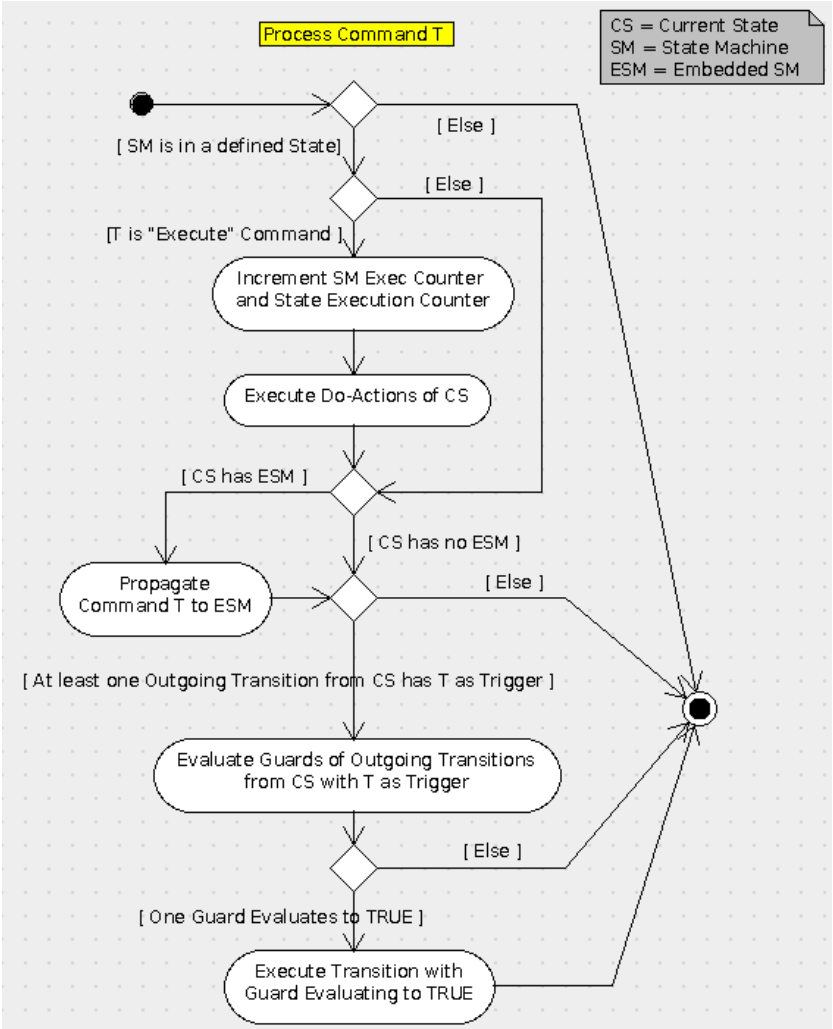

Fig. 7: Logic for Processing Transition Commands by a State Machine

of constraint C6 in the previous section.

The logic according to which a transition is executed is shown as an activity diagram in Figure 8. Note that this logic is called up by the logic shown in the activity diagram of Figure 7.

Transition commands may carry parameters. These parameters may be passed to any of the state or transition actions that are executed as part of the processing of the transition command.

The execution of the various actions associated to the three state machine operations is performed in sequence: an action is executed only when the previous one has completed. Note that, since state and transition actions are constrained to execute in zero logical execution time, the execution of a state machine operation will also execute in zero logical execution time.

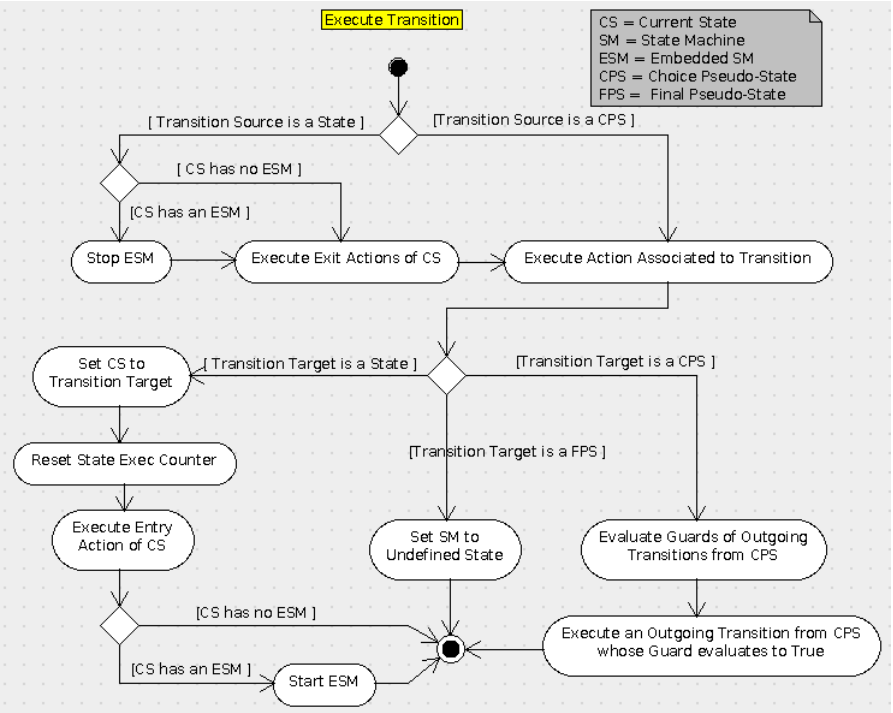

Fig. 8: Logic for Executing Transitions in a State Machine

Transition commands arrive and are processed in sequence. A new command can only arrive and be processed by a state machine when the previous one has been fully processed. State machines have no queues to buffer incoming transition commands.

The above rule in particular implies that transition commands cannot be "nested", namely the processing of a transition command by a state machine cannot result in a new command being sent to the same state machine (*nesting rule*).

As an example where the nesting rule would be violated, consider the following situation. A first transition command is sent to state machine A that triggers a transition from state A1 to state A2. The entry action of state A2 sends a second transition command to state machine A.

As a second example of violation of the nesting rule, consider a transition command that is sent to state machine A that triggers a transition from state A1 to state A2. The entry action of state A2 sends a new transition command to state machine B. State machine B, as part of its processing of this command, sends a new transition command to state machine A.

Forwarding of transition commands from one state machine A to another state machine B is instead allowed provided that neither of the two state machines is embedded in the other one.

Forwarding of transition commands from an embedded state machine to its embedding state machine or vice-versa is forbidden. This restriction helps to

avoid the ambiguities that would arise when, for instance, the entry action of a state in an embedded state machine triggers a transition in the embedding state machine.

4.4 Specification of State and Transition Actions

Although the FW Profile does not mandate any formalism for specifying the content of a state or transition action, it offers two mechanisms through which this can be done.

Firstly, a state or transition action can be modelled by a procedure. Note, however, that, since state and transition actions must execute in zero logical execution time, procedures that specify state machine actions must execute in one single execution cycle (i.e. they must consist of a sequence of steps that are executed in one single execution of the procedure).

If a procedure is used to define a state or transition action, then the execution of that action must result in the procedure being started and then executed and its execution must result in the procedure terminating.

Secondly, a state or transition action can be defined in terms of operations performed upon another state machine or upon a procedure. More specifically, a state or transition action can be defined to do one of the following:

- Start a procedure
- Execute a procedure
- Stop a procedure
- Start a state machine
- Execute a state machine (i.e. send an execute command to it)
- Send a transition command to a state machine
- Stop a state machine

4.5 Graphical Representation

FW State Machines can be conveniently represented using standard UML State Machine diagrams. The mapping from the graphical elements to the elements defined above for the state machines of the FW Profile is the obvious one.

As an example, consider the procedure in figure 9. In this figure, when an action consists of performing an operation upon another state machine or upon a procedure, the following syntax is used: "$\langle operationName \rangle$: $\langle SM\_or\_ProcName \rangle$". Thus, for instance, if an action consists in starting procedure `Procedure_A`, the content of the action is expressed as follows: "Start: Procedure_A".

When the example state machine of figure 9 is started, it enters either `STATE_1` or `STATE_2`, depending on the outcome of the evaluation of `Guard_1` and `Guard_2` (and note that, by virtue of constraint D1 in section 4.2, at least one of the two guards must be true). If `STATE_1` is entered, then the following sequence of actions is executed: `TransAction_0`; `n=0`; `TransAction_1`; and `Entry_1`. If instead `STATE_2` is entered, the following sequence of actions is executed `TransAction_0`;

`n=0`; `Entry_21`; and `Entry_22`.

State `STATE_3` can only be entered from `STATE_2` in response to command transition `Trigger_3`. In `STATE_3`, the state machine increases the value of the counter n every time it is executed and as long as `Guard_1` remains false. If, for instance, `Guard_1` becomes true for the first time when Execute is called for the fourth time, then, at the time the state machine enters state `STATE_1`, the value of n is equal to 4.

The state machine terminates execution when it is executed at a time when it is in `STATE_1`. If the state machine is stopped before it has terminated execution, it is set to an undefined state without any action being performed.

The use of the AP stereotypes is discussed in the next section.

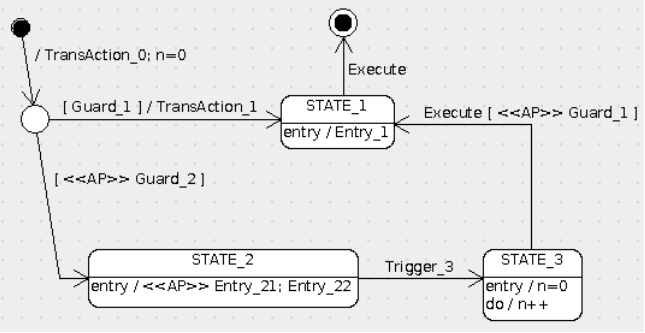

Fig. 9: Example of FW-Compliant State Machine

4.6 State Machine Adaptation

If a state machine is used to specify the behaviour of a framework, then it is useful to identify its adaptation points. The FW Profile offers two adaptation mechanisms for state machines.

The first one relies on the use of the $\langle\langle AP \rangle\rangle$ stereotype (see section 2.5) to identify adaptable elements within a state machine. The FW Profile allows this stereotype to be associated to the following elements:

- Entry Actions
- Do Actions
- Exit Actions
- Transition Actions
- Transition Guards

The presence of the $\langle\langle AP \rangle\rangle$ stereotype on any of the elements listed above may mean one of two things:

1. The content of the stereotyped element is not defined at framework level and the definition must be done at application level, or
2. A default content for the stereotyped element is defined at framework level but this can be overridden at application level.

The FW Profile does not provide the means to discriminate between the two cases above.

The second adaptation mechanism allowed by the FW Profile is as follows: if a state does not have an embedded state machine, then a state machine can be embedded in that state during the framework instantiation process.

In section 2.5 it was explained that the adaptation mechanisms supported by the FW Profile are designed to preserve certain invariant properties defined at framework level. What, then, are the properties which are invariant with respect to the two adaptation mechanisms defined above?

In order to answer this question, it is necessary to consider what *cannot* be modified through the allowed adaptation mechanisms: neither can the transition commands be changed, nor can new actions or new guards be added to existing states or transitions, nor can new states or new pseudo-states be added to the state machine. Thus, the features of a state machine that cannot be changed are: (a) the topology of the state machine, (b) the conditions, expressed in terms of the outcomes of guards, that lead to a state transition taking place, and (c) the sequence of actions which are executed when a transition is performed.

The invariant properties of a state machine are therefore those which describe behaviour which depends on the topology of a state machine and on the sequence of transitions and actions performed by the state machine.

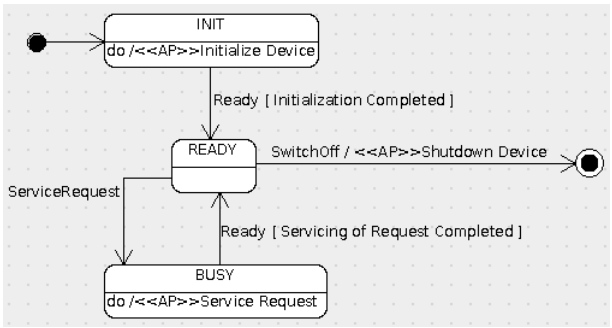

Fig. 10: Hardware Device with a Busy Wait

As an example, consider the state machine of figure 10. This represents a family of hardware devices which can be initialized and shutdown and which, while operational, are capable of servicing requests from their users. The initialization, shutdown and servicing procedures vary across the family of devices and are therefore modelled as adaptation points. The operation logic of the device is instead common to all devices in the family and it is captured in the topology of the state machine.

Examples of invariant properties for this state machine are: (a) a device can only service a user request, if, at the time the request is made, it is in state READY; (b) a device only executes its shutdown procedure if it is switched off when it is state READY. These properties are invariant because they will hold irrespective of how the adaptation points in the state machine are filled at application level. These properties will also continue to hold if the application designer adds behaviour to the INIT, READY or BUSY states by embedding

new state machines into them.

Consider instead the following property: when a device receives the `Ready` command, it enters the READY state. This property may hold on some devices (depending on how the device is commanded and on how its initialization and request servicing procedures are implemented) but it is not guaranteed by the state machine diagram of figure 10 and should therefore not be relied upon when designing a generic framework.

Figure 11 illustrates the general concept of property invariance for state machines. The figure shows a base state machine that is extended with the addition of a new embedded state machine and, possibly, with re-definitions of some of its actions or guards (not shown explicitly in the figure). The embedded state machine can be used to endow the derived state machine with additional (application-level) properties but it cannot violate the properties inherited from the framework level.

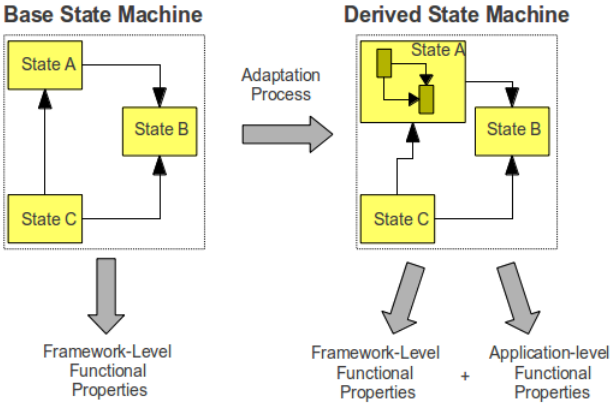

Fig. 11: State Machine Adaptation Process

4.7 Mapping to Design Level

The FW Profile is aimed at the modelling of behaviour. The concepts offered by the FW Profile to model behaviour can be mapped in many different way to software design-level artifacts. In the case of the state machine concept, the following mappings would be possible:

- A state machine is mapped to a single class; the transition triggers and all the actions and guards are mapped to methods in the class; adaptation is through inheritance.
- A state machine is mapped to a single class associated to classes representing its states (a class for each state); the transition triggers and all the actions and guards are mapped to methods in the class; adaptation is through inheritance.
- A state machine is mapped to a C-style module; the transitions are controlled by a single function in the module (a transition is identified by an argument to the function); actions and guards are mapped to functions in the module; adaptation is through delegation.

Obviously, the list above is non-exhaustive but the point it tries to make is that the use of the FW Profile to model the behaviour of an application does not dictate its software-level design

4.8 UML 2 Compliance

The state machine model offered by the FW Profile complies with the UML 2 state machine model in the sense that the elements of the state machine concept of the FW Profile and their semantics can be mapped in an obvious way to a subset of the elements of the state machine concept of UML 2 with the following provisos:

- The semantics of choice pseudo-states in the FW Profiles subsumes that of junction pseudo-states in UML2. Thus, in the FW Profile, choice pseudo-states can also be used to join together incoming transition flows.
- The execution counters are specific to the FW Profile. They have been introduced as a substitute for the concept of time (which does not exist in the FW Profile State Machines): if a state machine is executed periodically, then the value of its execution counters is proportional to the time elapsed since the state machine was started (State Machine Execution Counter) or since the current state was entered (State Execution Counter).

It should be emphasized that the state machine model proposed by the FW Profile is far more restrictive than that supported by UML 2. This is because the FW Profile uses state machines to model purely functional (non-time-related, non-concurrent) behaviour.

5 RT Containers

RT Containers are one of the three modelling concepts offered by the FW Profile (see section 2.1). This section defines the RT Container concept of the FW Profile.

Note that, unlike state machines and procedures (the other two modelling concepts offered by the FW Profile), RT Containers are not derived from a UML 2 concept.

5.1 Role of RT Containers

State machines and procedures allow all functional aspects of a software application to be modelled. RT Containers complement them by offering a means to capture one aspect of the time-related behaviour of an application.

It is important to stress that full modelling of an application's timing behaviour is beyond the scope of the FW Profile. This is because the FW Profile is aimed at modelling individual applications. Applications normally run on a software/hardware platform which they share with other applications. Timing behaviour is a system-level aspect (it depends, for instance, on the relative priorities of the threads allocated to the various applications in a system) and cannot therefore be fully captured at application level.

RT Containers provide a way to encapsulate the activation logic for a functional behaviour. More specifically, a RT Container can be seen as a representation of a thread that controls the execution of some functional behaviour. The RT Container model defined by the FW Profile allows the conditions under which the thread is released to be specified.

Conceptually, a RT Container can be seen as a software structure that encapsulates some functional code and endows it with certain timing properties. Thus, RT Containers are a means of separating the specification of the timing aspects of an application from its functional aspects.

There is a difference between procedures and state machines on the one hand, and RT Containers on the other hand. All three concepts are offered as means to express the behaviour of a software application but they exist at different levels of abstraction: state machines and procedures constitute a generic modelling language for the functional part of an application; RT Containers allow the timing behaviour of a software application to be modelled but they presuppose the use of a certain design pattern for handling the activation of functional code. The RT Container concept is thus less generic than the state machine and procedure concepts.

The design pattern behind the concept of RT Containers is a notification-based model of thread activation where the notification can be either time-triggered or sporadic (event-driven notification).

5.2 Definition of RT Container

A RT Container is defined by the following elements:

- One *Activation Procedure*
- One *Activation Thread*
- One *Notification Procedure*

The *Activation Procedure* is a FW Profile Procedure which executes the functional behaviour encapsulated by the RT Container.

The *Activation Thread* is the thread responsible for executing the Activation Procedure (and hence for executing the functional behaviour encapsulated by the RT Container).

The *Notification Procedure* is a FW Profile Procedure which encapsulates the logic for notifying the Activation Thread.

5.3 RT Container Behaviour

Three operations may be performed on a RT Container: (a) the RT Container may be *started*; (b) the RT Container may be *stopped*; and (c) the RT Container may be *notified*.

A RT Container may be in two states: STOPPED or STARTED. Initially, by default, the container is in state STOPPED. When a RT Container is started, the behaviour shown in the activity diagram in the left-hand side of figure 12 is executed. The Start operation only has an effect if the container is in state STOPPED when the operation causes the Activation and Notification Procedures to be started and executed once and the Activation Thread to be created and released. The Notification and Activation Procedures are started "atomically" in the sense that neither procedure can be executed or stopped before both have been started. Reference to figure 13 shows that the first execution of the Activation and Notification Procedures results in their initialization actions being executed and, in the case of the Activation Procedure, in the first Set-Up Notification action being executed.

When a RT Container is stopped, the behaviour shown in the activity diagram in the right-hand side of figure 12 is executed. The Stop operation only has an effect if the container is in state STARTED when the operation causes the container to be placed in state STOPPED and the Notification Counter to be incremented. The latter results in one last notification being sent to the Activation Thread. This notification is necessary to ensure an orderly termination of the thread and of the Activation and Notification Procedures.

When a RT Container is notified, the following behaviour is executed:

1. If the RT Container is in state STOPPED, then no further action is performed;
2. If the RT Container is in state STARTED, then its Notification Procedure is executed.

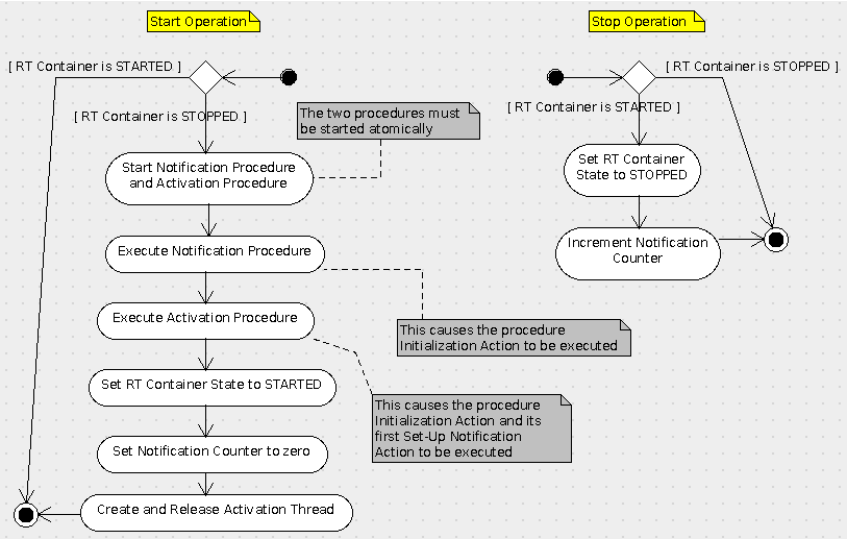

Fig. 12: Start and Stop Operations for RT Containers

The behaviour of the *Activation Thread* is expressed by the following pseudo-code:

```
while true do {
  wait until Notification Counter is greater than 0;
  decrement Notification Counter;
  execute Activation Procedure;

  if (Activation Procedure has terminated) then {
    put RT Container in STOPPED state;
    execute Notification Procedure;
    break;
  }

  if (RT Container is in state STOPPED) then {
    execute Activation Procedure;
    execute Notification Procedure;
    break;
  }
}
```

Listing 1: Pseudo-code of Activation Thread

The thread executes a loop which starts with a check on whether there are any pending notifications (the Notification Counter holds the number of pending notifications). If there is a pending notification (i.e. if the Notification Counter is greater than zero), the thread decrements the Notification Counter and then executes the Activation Procedure (which causes the container's functional behaviour to be executed). The thread terminates when the Activation Procedure has terminated or when the RT container has been stopped. In the former case (Activation Procedure has autonomously terminated), the RT Container is put in the STOPPED state and the Notification Procedure is executed one last time before the thread exits; in the latter case (RT Container has been stopped), both procedures are executed one last time. This last execution is intended to give the procedures a chance to perform their finalization behaviour.

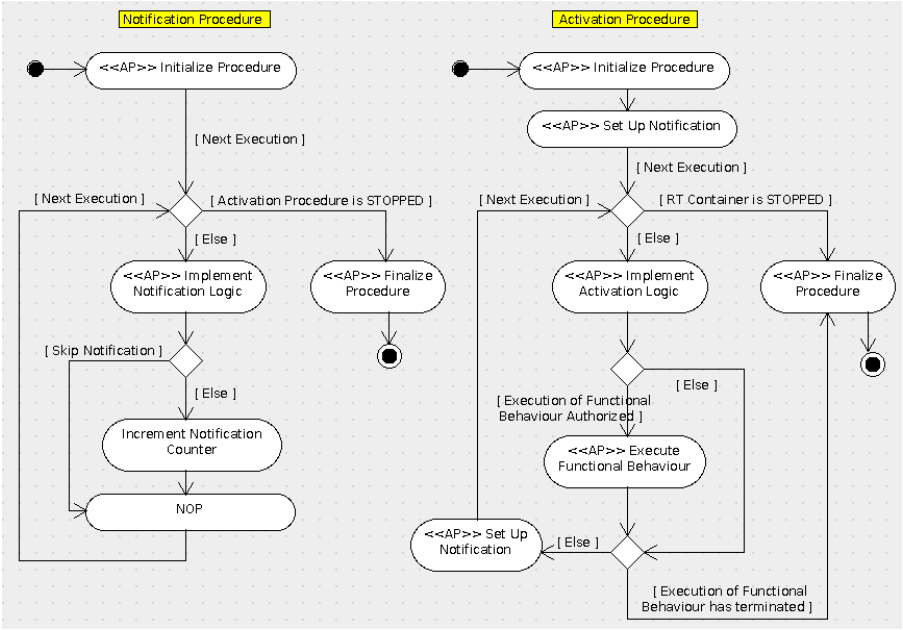

Fig. 13: RT Container Procedures

The behaviour of the *Activation Procedure* and of the *Notification Procedure* is shown in the activity diagrams in Figure 13. The definition of the two procedures makes use of the "adaptation point" stereotype to identify the parts of the container behaviour which are application-specific. Applications are therefore expected to extend the two procedures by inserting their own application-specific behaviour (by contrast, the behaviour of the Activation Thread is invariant and is fully defined at FW Profile level).

When the Activation Procedure is executed for the first time (i.e. after the Activation Thread has been started), it initializes itself and sets up the first notification of the Activation Thread. The form of the notification is application-specific. Typically, the setting up of a notification may consist of one of the following:

1. A request that the Activation Thread be notified at some time in the future;
2. A call-back registration to request to be notified when a certain software condition arises (e.g. a variable changes value, a message arrives, etc);
3. A request to be notified when a hardware interrupt is asserted.

Note that the notification may only need to be set up once when the Activation Procedure is initialized or it may need to be set up at every execution cycle. Note also that the same RT Container may set up different notification requests in the same execution cycle or it may set up notification requests of different kinds at different execution cycles. For this reason, the "Set-Up Notification" in the Activation Procedure is placed both at the beginning of the procedure (to be executed once at initialization time) and inside the loop (to be executed after each execution of the functional behaviour).

When a notification arrives (i.e. when the user of the container executes the Notification Procedure and this increments the Notification Counter), the Activation Thread is woken up and it executes the Activation Procedure. The procedure checks whether the RT Container has been stopped. If this is the case, the procedure performs its finalization action and then terminates. Otherwise, the procedure checks whether the functional behaviour should be executed (this is done by the "Implement Activation Logic" action) and, if so, it executes it. Afterwards, the procedure sets up the next notification (if one is needed) and then checks whether the execution of the functional behaviour has been completed. If this is so, the procedure terminates. Otherwise it waits for the next notification.

The procedure initialization and finalization actions are adaptation points which are defined at application level. Similarly, the action to set up the notification for the Activation Thread and to implement the activation logic must also be defined at application level. The latter could, for instance, be used to implement a filter which decides which notifications to process and which ones to ignore.

The Notification Procedure acts as an intermediary between the source of the notification event and the notification trigger to the Activation Thread. Such an intermediary may be useful to: (a) filter notification events, or (b) buffer notification requests so as to allow the Activation Procedure to handle bursts of notifications. With reference to the activity diagram in Figure 13, the filtering and buffering of notification requests is done in the (application-specific) action "Implement Notification Logic".

As already noted, the Notification Procedure runs on a thread that is external to the RT Container: the Notification Procedure is executed by an external thread when the notification event has occurred. Thus, the logic leading to the notification of the Activation Thread is as follows:

1. The Activation Procedure makes a request to be notified when a certain event occurs (this could, for instance, be done by registering with an external component to be notified when a certain condition occurs);
2. When the event occurs, the Notification Procedure is executed by the source of the event;
3. The Notification Procedure evaluates the event and may decide to notify the Activation Thread;
4. The Notification Procedure notifies the Activation Thread by incrementing the Notification Counter;
5. In response to the notification, the Activation Thread executes the Activation Procedure which may execute the functional behaviour encapsulated by the RT Container;
6. The Activation Procedure sets up the next notification request.

This cycle is broken when either the Activation Procedure decides that the execution of the functional behaviour has been completed or when the RT Container is stopped. Either of these events results in the RT Container and its two procedures terminating.

The Notification Procedure may be executed both by the Activation Thread and by an external thread. For this reason, in many cases, it will be necessary to ensure that it is executed in mutual exclusion.

Note finally that, in this section, the term "event" encompasses both asynchronous occurrences (such as the arrival of hardware interrupts from an external source) or synchronous occurrences (such as periodic signals generated by an operating system).

5.4 RT Container Properties and Usage Constraints

The RT Container logic defined in the previous section guarantees that certain properties (the *RT Container Properties*) are satisfied when the usage of the RT Container complies with certain constraints (the *RT Container Usage Constraints*). The properties are listed in table 3 in rows P-3 to P-7. The usage constraints are listed in the same table in rows C-1 to C-3.

Table 3: RT Container Properties and Usage Constraints

N	RT Container Properties and Usage Constraint
P-3	The Activation Thread shall never deadlock.
P-4	If the RT Container is stopped after the Activation Thread has been released, then, at some later time, the Activation Procedure shall terminate.
P-5	If the Activation Procedure stops or terminates (it enters the STOPPED state), then, at some later time, the RT Container shall be stopped.
P-6	If the Activation Procedure stops or terminates (it enters the STOPPED state), then, at some later time, the Notification Procedure shall terminate.
P-7	Whenever the Activation Procedure is running (it is in state STARTED), then the Notification Procedure shall be running, too (it shall be in state STARTED).
P-8	If notifications cease but the RT Container and the Activation Procedure continue to run, then, at some later time, the Activation Thread shall consume all pending notifications (the Notification Counter will become equal to zero).
C-1	If the RT Container is started and then, at some later time, it is stopped, then it can be re-started only after its Activation and Notification Procedures have terminated execution and after its Activation Thread has terminated (i.e. the user of a RT Container cannot re-start it before it has completed its orderly shutdown)
C-2	The Activation Procedure is started, stopped and executed exclusively by the RT Container (i.e. the user of the container has no access to the Activation Procedure)
C-3	The Notification Procedure is started and stopped exclusively by the RT Container itself (i.e. the user of the RT Container can execute the Notification Procedure through the Notify operation but it cannot start or stop it)

The usage constraints define the conditions for the legal use of a RT Container. If these constraints are satisfied, then the user can assume that the RT Container will comply with its properties. Note that the container's properties hold under all circumstances, irrespective of the scheduling and notification/triggering policies adopted for the Activation Thread and for the thread controlling the Notification Procedure and irrespective of the way in which the adaptation points in the container's procedure are filled.

Properties P-4 and P-5 guarantee that, if the RT Container is stopped or the Activation Procedure terminates, then the entire container will terminate in the sense that the container itself and its two procedures will all enter the STOPPED state. Property P-8 ensures that, if thread scheduling is fair and the rate at which notifications are generated is compatible with the rate at which they are processed, then no backlog of unprocessed notifications will build up.

Some notifications may instead remain unprocessed if either the Activation Thread autonomously terminates or the RT Container is stopped by the user. Thus, in informal language, the semantics of the Stop operation on the RT Container is not: "Process all pending notifications and then terminate"; but rather: "Discard any pending notifications and then terminate".

Note that the container's procedures can only terminate execution "naturally" (as opposed to being forcefully stopped). This is because the RT Container logic never stops them and usage constraints C-2 and C-3 ensure that they are not stopped by any external agent. This is important because it means that the procedure will always execute their finalization behaviour before terminating.

Constraint C-1 states that a RT Container can only be re-started after it has completed its shutdown. This is a legitimate constraint because properties P-4 and P-6 guarantee that, if the container is stopped, then its two procedures will eventually terminate. This means that the user of a RT Container can always rely on the container completing its shutdown in a finite amount of time.

The container properties can be formally verified. This is done in table 4. The verification is partially based on the Promela model of the RT Container presented in appendix A.

Table 4: Verification of RT Container Properties

N	Verification of Property
P-3	Absence of deadlock is verified because the Promela model of appendix A has no invalid end states.
P-4	If the following settings are made: p = "RT Container is in state STOPPED" q = "Activation Procedure is in state STOPPED" and z = "Activation Thread has been released"; then property P-4 can be stated in LTL as follows: []((p && z) →⟨⟩q). This property is satisfied by the Promela model of appendix A (under "weak fairness" conditions).

N	Verification of Property
P-5	If the same settings are used as for the previous property, then this property can be stated in LTL as follows: []($!q \rightarrow [](q \rightarrow \langle\rangle p)$). This property is satisfied by the Promela model of appendix A (under "weak fairness" conditions).
P-6	If the following settings are made: p = "Notification Procedure is in state STOPPED" and q = "Activation Procedure is in state STOPPED"; then property P-6 can be stated in LTL as follows: []($q \rightarrow \langle\rangle p$). This property is satisfied by the Promela model of appendix A.
P-7	If the same settings are used as for the previous property, then this property can be stated in LTL as follows: []($!q \rightarrow !p$). This property is satisfied by the Promela model of appendix A.
P-8	If the following settings are made: p = "RT Container is in state STARTED", q = "Activation Procedure is in state STARTED" u = "No notifications are generated" and v = "Notification Counter is equal to zero", then this property can be stated in LTL as follows: $\langle\rangle [] (u \,\&\&\, p \,\&\&\, q) \rightarrow \langle\rangle [] v$. This property is satisfied by the Promela model of appendix A.

5.5 RT Container Adaptation

There are no adaptation mechanisms that are specific to RT Containers. RT Containers can be adapted by adapting their procedures in accordance with the rules of the FW Profile (see section 3.7). The adaptation points of a RT Container are the adaptation points of the container's procedures. These are listed in table 5.

Table 5: Adaptation Points of RT Containers

Adaptation Point	Description
Initialize Procedure (Notification Procedure)	Implement the initialization action for the Notification Procedure.
Initialize Procedure (Activation Procedure)	Implement the initialization action for the Activation Procedure.
Implement Notification Logic	Determine whether or not a notification request is forwarded to the Activation Thread.
Set Up Notification	Set up (or update) the mechanism for the next notification.
Implement Activation Logic	Determine whether or not reception of a notification results in execution of the container's functional behaviour.
Execute Functional Behaviour	Execute the container's functional behaviour.
Finalize Procedure (Notification Procedure)	Implement the finalization action for the Notification Procedure.
Finalize Procedure (Activation Procedure)	Implement the initialization action for the Activation Procedure.

5.6 Mapping to Design Level

As in the case of state machines and procedures, there are many ways in which the behaviour specified by a RT Containers can be mapped to the design level. Also as in the case of the state machines and procedures, the type of mapping is not mandated by the FW Profile.

6 Formal Verification of FW Profile Models

The FW Profile offers the means to build a behavioural model of an application. Such a model is developed in response to the needs of a higher-level system within which the application is to be deployed. The question then arises of how one can verify that the model matches the higher-level needs of its embedding system. Traditional answers to this question rely on analysis of the model through reviews by the system's stakeholders. More recently, *formal verification* techniques have been proposed for the same purpose.

Formal verification consists in proving that a certain formally expressed model satisfies certain formally expressed properties. When applied to a requirements model, formal verification techniques are a way of checking the correctness of the requirements: the properties to be verified express the higher-level needs which the requirements are expected to satisfy and formal verification helps establish that the requirements actually do satisfy them (i.e. that the requirements are correct).

As an example, consider an application which manages a stream of commands to a hardware actuator and which is responsible for controlling its operation. One of the desirable properties of such an application might be to ensure that commands for the actuator never remain pending forever within the application itself (due to, for instance, the actuator having been shut down before the command buffer had been flushed). Establishing at requirements level that this property holds involves analysing the application's requirements to verify that they guarantee that, under all operating conditions, commands are always eventually sent to the actuator. With a formal verification approach this task takes the form of a proof carried out on a formal model of the application's requirements.

Formal verification techniques can be applied whenever requirements are expressed in a formalism with an unambiguous semantics. They can therefore be applied to FW Profile models. The objective of this section is to show how this can be done in practice using a *model checking approach* which is arguably the most common formal verification technique.

With a model checking approach, compliance of a model to a certain property is verified by systematically exploring all possible states of the model and verifying that, in all cases, the target property is met. Practical use of model checking techniques requires tool support. Here, the Spin Model Checker of reference [3] is considered. This is one of the most widely used model checking tools; it has a strong heritage in industrial projects; and it is publicly and freely available from [6].

This section thus discusses the applicability of model checking techniques using the Spin Model Checker to FW Profile models. The discussion assumes the reader to be familiar with model checking in general and with the Spin Model Checker in particular.

6.1 Model-Checking for FW Profile Models

Figure 14 shows the basic approach to model-checking with the FW Profile. The inputs tothe verification process (yellow boxes in the figure) are the FW Profile model itself and the properties which one wishes to verify. The model is translated into an equivalent Promela model and the properties are formalized by being translated into, for instance, LTL formulas or assertions within the Promela code. The Spin Model Checker can then be used to verify whether the properties are satisfied. The outcome of the check is either a confirmation that the properties hold or else an execution trail showing how a certain property is violated.

The appeal of the approach in figure 14 is that it would allow the Promela model to be automatically generated from the FW Profile model. Building a tool to translate a model from the FW Profile to the Promela world would be straighforward and would automate the verification process. In reality, this approach would suffer from two major flaws. Firstly, in practical cases, verification is not possible on a *full* model of the target application because the verification would either take too long or exhaust the memory of the computer platform on which it is carried out. Verification must be done on a *reduced* version of the model which only retains the features which are relevant to the properties which are being verified. Secondly, verification of the properties of a certain application normally requires that the environment within which the application is embedded be specified.

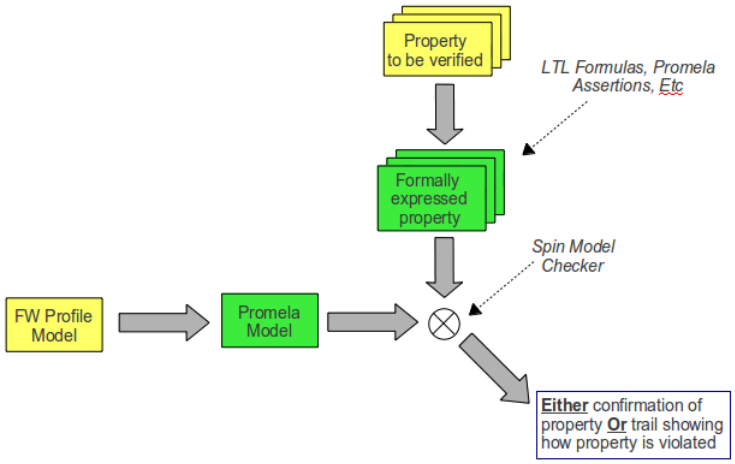

Fig. 14: Basic Approach for Model Cheking of FW Profile Models

Thus, a more realistic approach to formal verification is as in figure 15 where the generation of the Promela model requires a "simplification" step which removes all non-relevant features of the FW Profile model and requires, as a second input, the specification of the environment within which the application resides. Automatic generation of the Promela code is still possible but its benefits are much smaller because: (a) the most time-consuming part of the verification process is likely to be the identification of the features to be discarded from the full model; (b) the reduced model must be comparatively simple (or else verification takes too much time or memory) and hence the added value from

automating its generation is limited; and (c) the Promela model must cover the application's environment and this cannot be derived automatically from the model of the application requirements.

For all the above reasons, it seems more useful to define general "Promela patterns" of how FW Profile models can be transformed into Promela models rather than to provide a tool which would, at most, only automatize a small part of the transformation. Understanding of these patterns may help a designer to rapidly create Promela models which are adapted to a certain property to be verified. These Promela Patterns are introduced in the next section while section 6.3 presents a concrete example of their use.

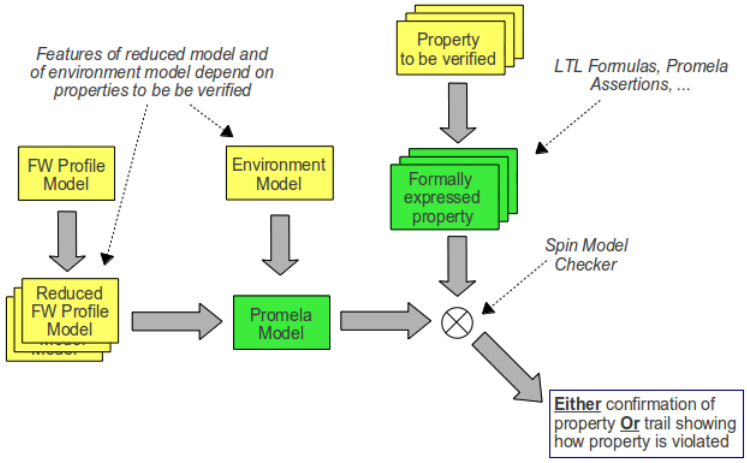

Fig. 15: Realistic Approach to Model Cheking of FW Profile Models

6.2 Promela Patterns

The FW Profile uses state machines and procedures to capture the functional (i.e. purely sequential, non-time related) behaviour of an application and uses RT containers to capture their non-functional behaviour. The basic rules for mapping a FW Profile model to Promela code therefore are:

- State machines and procedure are mapped to non-blocking code; and
- RT containers are mapped to Promela processes.

The next three sub-sections discuss in greater detail the mapping to Promela of each of the three constructs of the FW Profile.

6.2.1 State Machines

This section explains how a FW Profile state machine can be represented in a Promela program. The example Promela listings in this section have been built for the state machine of figure 16. This state machine represents a buffer controller and is part of the example discussed in section 6.3.

The basic pattern for modelling state machines in Promela is to map them to macros with one single argument representing the start/stop commands and the transition commands for the state machine. For the example state machine, this

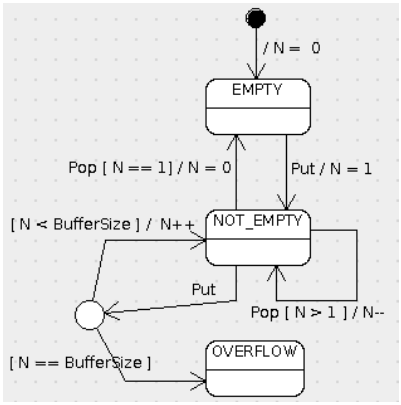

Fig. 16: State Machine for a Buffer Control Logic

is illustrated in listing 2. The listing begins with the declaration of a set of `mtype` types which define the set of commands for the state machine and the set of states for the state machine. The set of commands includes both the `Start/Stop` commands and the transition commands (`Put` and `Pop` in the example). The set of states includes an INACTIVE state to represent the situation where the state machine is stopped.

```
1  mtype = {INACTIVE};     /* State common to all SMs and Procedures */
2  mtype = {Start, Stop};              /* Commands common to all SMs */
3  mtype = {EMPTY, NOT_EMPTY, OVERFLOW};      /* States of Buffer SM */
4  mtype = {Put, Pop};                       /* Commands of Buffer SM */
5  mtype bufferState = INACTIVE;               /* State of Buffer SM */
6  byte nItems = 0;                      /* Number of items in buffer */
7
8  /* ------------------------------------------------------------------ */
9  /* A call to this macro models the sending of a command to the    */
10 /* Buffer State Machine. The buffer size is assumed equal to 3.  */
11 inline BufferSM(cmd) {
12     if
13     :: (bufferState==INACTIVE) && (cmd==Start) ->
14         bufferState = EMPTY;
15         nItems = 0;
16     :: (bufferState==EMPTY) && (cmd==Put) ->
17         nItems = 1;
18         bufferState = NOT_EMPTY;
19     :: (bufferState==NOT_EMPTY) && (cmd==Put) && (nItems<3) ->
20         nItems++;
21     :: (bufferState==NOT_EMPTY) && (cmd==Put) && (nItems==3) ->
22         bufferState = OVERFLOW;
23     :: (bufferState==NOT_EMPTY) && (cmd==Pop) && (nItems>1) ->
24         nItems--;
25     :: (bufferState==NOT_EMPTY) && (cmd==Pop) && (nItems==1) ->
26         nItems = 0;
27         bufferState = EMPTY;
28     :: (bufferState!=INACTIVE) && (cmd==Stop) ->
29         bufferState = INACTIVE;
30     :: else -> skip;
31     fi;
32 }
33 . . .
34 BufferSM(Put);                    /* Send command Put to Buffer SM */
```

Listing 2: Representation of a State Machine Promela

At line 5, variable `bufferState` is defined to represent the current state of the state machine. Such a variable is normally initialized to INACTIVE. It may be initialized to a different value if the verification scenario starts with the state machine in an active state.

Obviously, the set of commands in the `mtype` declarations does not need to encompass *all* commands which the target state machine may accept. Some of these commands may not be relevant to the verification objectives and may be dropped. For the same reason, the declaration of the states may only cover a subset of the state machine's states.

The variable `nItems` defined at line 6 of the listing matches variable `N` in the state machine. Variable `BufferSize` in the example state machine is not directly modelled in the Promela program which simply assumes that the buffer has a size of 3. This choice does not imply any loss of generality since, in a verification context, there is no difference between a situation where the buffer size is equal to 3 and a situation where it has a value greater than 3. This transformation from a variable of arbitrary value to a constant with a well-defined value is an example of the "complexity reduction" step shown in figure 15 .

The macro is defined at lines 11 to 32. A call to this macro represents the sending of a command to the state machine. The macro consists of an `if` clause which covers all combinations [current state, transition command, guard] which may trigger a transition in the state machine. The combinations of state/command pairs which cannot trigger any transition (or which are simply not relevant to the verification objective) are caught by the `else` part of the `if` clause. The behaviour associated to each state/command pair should in principle be the one defined in section 4.3 as the behaviour of a state machine in response to an external command. In practice, aspects of this behaviour which are not relevant to the verification objective should be dropped. Thus, for instance, the example in listing 2 does not model the execution counters which are associated to each state machine. This is legitimate in the case of the example state machine because none of its guards depends on the passage of time or on the execution cycle of the state machine.

Line 34 shows how the macro is called. In this line, transition command `Put` is sent to the state machine.

One variant to the code in listing 2 occurs when the implementation adds a mutual exclusion mechanism in order to let the Buffer state machine be accessed by multiple threads. If the mutual exclusion mechanism is important for verification purposes, it must be modelled in the Promela code. One way of doing this is to associate a boolean variable to the state machine and then to use it as a mutex. The variable is indivisibly tested-and-set at the beginning of the state machine macro (to model the seizing of the mutex) and is reset at the end of the macro (to model the release of the mutex). Listing 3 shows how this would be done in the case of the Buffer state machine.

The examples of the mutex and of the execution counters (both of which may be included or omitted in a Promela model depending on the verification objective) and the fact that the set of states or of transition commands may be tailored

to the verification objectives illustrate how the need may arise to maintain different Promela representations of the same state machines which are aimed at different verification objectives. One way to handle this variability is to put the definition of the state machine macro in a separate file which is then included in the Promela program (using the `#include` directive). This lets a user create different versions of the state machine macro which are stored in separate files and have the same calling interface. It then become possible to rapidly switch between alternative representations of the same state machine with minimal impact on the overall Promela program.

```
1  mtype = {INACTIVE};     /* State common to all SMs and Procedures */
2  mtype = {Start, Stop};              /* Commands common to all SMs */
3  mtype = {EMPTY, NOT_EMPTY, OVERFLOW};      /* States of Buffer SM */
4  mtype = {Put, Pop};                      /* Commands of Buffer SM */
5  mtype bufferState = INACTIVE;              /* State of Buffer SM */
6  bool bufferMutex = false;                 /* Mutex for Buffer SM */
7  byte nItems = 0;                    /* Number of items in buffer */
8
9  /* ------------------------------------------------------------- */
10 /* A call to this macro models the sending of a command to the   */
11 /* Buffer State Machine. The buffer size is assumed to be equal  */
12 /* to three.                                                     */
13 inline BufferSM(cmd) {
14    atomic{(bufferMutex==false) ->
15           bufferMutex = true;}                 /* Seize the mutex */
16    . . .
17    bufferMutex = false;                      /* Release the mutex */
18 }
```

Listing 3: Representation of a Mutex-Protected State Machine in Promela

6.2.2 Procedures

This section explains how a FW Profile procedure can be represented in a Promela program. The example Promela listing in this section has been built for the procedure of figure 17. This procedure controls a hardware actuator and is part of the example discussed in section 6.3.

The basic pattern for representing procedures in Promela is to split their mapping to Promela into two parts. The first part consists of dedicated code which models the start/stop logic for the procedure. The second part consists of a parameterless macro which models the execution of the procedure. This is illustrated in listing 4 for the example procedure.

The listing begins with the declaration of an `mtype` type which defines the possible states of the procedure. All procedures have a STOPPED and a STARTED state which correspond to the procedure being stopped or having just been started. Additionally, procedures have one state for each node at the end of a control flow with a guard. This state represents the condition of a procedure which is being executed and which has found the guard to be false. The example procedure has two control flows with a guard attached to them but both these control flows end in the same node (the decision node after the "Start Actuator SM" action node) and hence only one additional state needs to be defined for this procedure. In listing 4 this state is called WAITING.

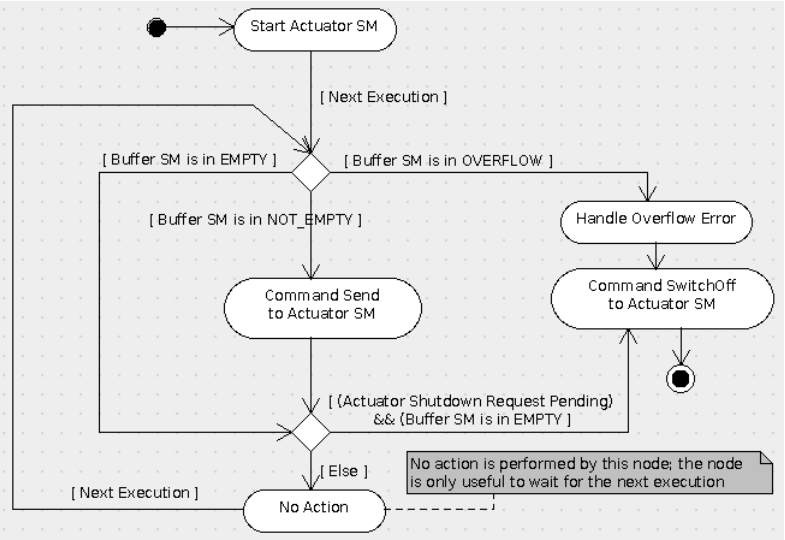

Fig. 17: Procedure to Control a Hardware Actuator

The procedure macro consists of an `if` clause (see lines 7 to 31 in the listing) which covers all states of the procedure and associates to each the behaviour of the procedure when it receives an Execute command in that state. This is in principle the behaviour defined in section 3.3 but, as in the case of the state machine macros, behaviour which is not relevant to the verification can (and should) be dropped. Thus, for instance, the example in the listing does not model the procedure execution counters.

The starting and stopping of the procedure is modelled by setting the procedure state to, respectively, STARTED and STOPPED (see lines 38 and 44 in the listing). The execution of the procedure is modelled by a call to the procedure macro (see lines 40 and 42 in the listing).

As in the case of state machines, there is one obvious variant to the code in listing 4 which occurs when the implementation adds a mutual exclusion mechanism in order to let the procedure be accessed by multiple threads. If the mutual exclusion mechanism is important for verification purposes, it must be modelled in the Promela code. One way of doing this is to associate a boolean variable to the procedure and then to use it as a mutex. The variable is indivisibly tested-and-set at the beginning of the procedure macro (to model the seizing of the mutex) and is reset at the end of the macro (to model the release of the mutex). The mechanism is the same as for the state machine of listing 3.

Also as in the case of state machines, the need to tailor a Promela model to a certain verification objective and to keep it as simple as is compatible with that objective will inevitably result in several models being developed for a given procedure with each model encapsulating different aspects of the procedure's behaviour. One way of handling this variability is to proceed as recommended for the state machines and to give the various versions of the procedure the same calling interface and place them into `#include` files which may then be included in the main Promela program.


```
mtype = {STOPPED, STARTED, WAITING};  /* Act. Cont. Proc. States */
mtype actContState = STOPPED;    /* State of Actuator Cont. Proc. */

/* ------------------------------------------------------------------ */
/* A call to this macro models the execution of the Actuator          */
/* Controller Procedure.                                              */
inline ActuatorControllerPR() {
   if
   :: (actContState==STOPPED) ->
      skip;
   :: (actContState==STARTED) ->
      ActuatorSM(Start);                    /* Start Actuator SM */
      actContState = WAITING;
   :: (actContState==WAITING) ->
      if
      :: (bufferState==EMPTY) -> skip;
      :: (bufferState==NOT_EMPTY) ->
         ActuatorSM(Send);          /* Send Send to Actuator SM */
      :: (bufferState==OVERFLOW) ->
         overflowHandled = true;
         ActuatorSM(SwitchOff);   /* Send SwitchOff to Act. SM */
         actContState = STOPPED;         /* Terminate procedure */
      fi;
      if
      :: (shutdownReqPending==true) && (bufferState==EMPTY) ->
         ActuatorSM(SwitchOff);    /* Send SwitchOff to Act. SM */
         actContState = STOPPED;         /* Terminate procedure */
      :: else -> skip;
      fi;
   fi;
}
. . .
/* ------------------------------------------------------------------ */
/* Promela Process executing Actuator Controller Procedure.           */
active proctype SomeProcess() {
  . . .
  /* Start Actuator Controller Procedure */
  actContState = STARTED;
  . . .
  ActuatorControllerPR();     /* Execute Actuator Cont. Procedure */
  . . .
  ActuatorControllerPR();     /* Execute Actuator Cont. Procedure */
  . . .
  actContState = STOPPED;        /* Stop Actuator Cont. Procedure */
  . . .
}
```

Listing 4: Representation of a Procedure in Promela

6.2.3 RT Containers

This section explains how a RT container can be represented in a Promela program. Mapping of RT containers to Promela can be done at two levels. A detailed mapping requires modelling of the two procedures in the container (the Activation Procedure and the Notification Procedure, see section 5.3). In this case, the Promela model in appendix A can be used.

Such a detailed modelling is, however, unlikely to be necessary. The RT container procedures are designed to guarantee that containers are well-behaved (to guarantee, for instance, that they initialize correctly or that they do not miss any

notification requests). Applications will normally take this well-behavedness for granted (because it has been proven at the level of the container) and will instead focus on the interaction between the containers and the functional behaviour which they control. A more coarse-grained modelling of the RT containers then becomes more appropriate where a RT container is seen as a source of transition commands to state machines and of execution requests to procedures. In this case, the RT container is simply represented by a Promela process whose body sends the commands and execution requests.

As a simple example, listing 5 shows a RT container which represents a thread which starts the Buffer State Machine of listing 2 and then repeatedly sends it `Put` commands until it eventually stops it.

```
1 active proctype ActuatorHardware() {
2   BufferSM(Start);
3   do
4   :: true -> BufferSM(Put);
5   :: true -> break;
6   od;
7   BufferSM(Stop);
8 }
```

Listing 5: Representation of a RT Container in Promela

6.3 Example Application of Promela Patterns

This section illustrates the concepts discussed above to present a complete example of how a FW Profile model can be mapped to a Promela program and how its properties can be verified using the Spin Model Checker. The example problem considered in this section is that of a software controller for a hardware actuator with the following characteristics:

1. By default, the hardware actuator is unpowered in the OFF state.
2. The hardware actuator is switched on by sending it command `SwitchOn`. In response to this command, the hardware actuator performs some internal initialization actions and then enters its normal operational state.
3. The hardware actuator signals completion of its initialization and entry into its normal operational state by sending a `Done` signal to its software controller.
4. When it is in its normal operational state, the actuator accepts actuator commands from its controller.
5. The processing of an actuator command takes some time; when the hardware actuator has terminated processing an actuator command, it generates a `Done` signal to its software controller.
6. The hardware actuator can only process actuator commands one at a time (i.e. there is no internal buffering of commands in the hardware actuator).
7. An orderly switch off of the hardware actuator is performed by sending it command `SwitchOff` at a time when it is not busy processing an actuator command.

Figures 18 and 19 capture the FW Model of a controller for the hardware ac-

tuator. Such a model might be used as a specification for a software module in an embedded control application. The model consists of two state machines (the *Buffer State Machine* and the *Actuator State Machine* in figure 18) and one procedure (the *Actuator Controller Procedure* in figure 19).

Since the hardware actuator has no buffering capability for the commands it receives, the Buffer State Machine is introduced to buffer software-level requests for actuator commands. The application sends `Put` commands to this state machine when it wishes to enqueue an actuator command. The buffer has two nominal states (EMPTY and NOT_EMPTY) and one error state (OVERFLOW) which is entered when the rate at which the application makes actuator command requests is greater than the rate at which the hardware actuator can process them.

The Actuator State Machine controls the hardware actuator. When the state machine is started, it sends a `SwitchOn` command to the hardware actuator and it starts the Buffer State Machine. It then waits in state INIT until the hardware actuator confirms that it has completed its initialization by issuing a `Done` command (in practice, this command would originate from an interrupt processing routine which is triggered by the `Done` signal generated by the hardware actuator). After initialization, the Actuator State Machine enters state READY where it waits for a `Send` command which triggers the collection of a command from the Buffer and its forwarding to the hardware actuator. While the hardware actuator is busy processing a command request, the Actuator State Machine remains in state BUSY and returns to state READY when the hardware actuator sends it a `Done` command to signal its readiness to process the next command. When the state machine is in state READY, it may be switched off. This causes a `SwitchOff` command to be sent to the hardware actuator and the Buffer State Machine to be stopped.

The Actuator Controller Procedure of figure 19 controls the operation of the two state machines and, through them, of the hardware actuator. Initially, both state machines are inactive. When the application wishes to start using the hardware actuator, it starts the Actuator Controller Procedure. This causes the Actuator State Machine to be started and (indirectly) the Buffer State Machine to be started. Subsequently, the application periodically executes the procedure. At each execution, the procedure checks whether there are any pending commands in the buffer and, if so, it triggers the Actuator State Machine to process them. The procedure can terminate either nominally (if there has been a shutdown request for the actuator) or abnormally (if the command buffer has overflown).

Table 6 lists four desirable properties which a well-designed actuator controller should satisfy. The list is of course not exhaustive but it gives an idea of the kind of properties for which a formal verification approach might be useful.

Table 6: Properties of Actuator Controller

N	Property Definitio
P-1	Pending commands in the buffer are always eventually sent to the hardware actuator.
P-2	If the Actuator State Machine is inactive, then the Buffer State Machine is inactive too.
P-3	If a shutdown request is made, then, eventually, the entire actuator function is shut down.
P-4	Buffer overflows are always eventually handled.

The Promela program which was used to verify the properties in the table is listed in full in appendix B. The bulk of the program consists of three macros which represent the two state machines and the procedure of the actuator function. Their structure is in line with the patterns presented in sections 6.2.1 and 6.2.2.

The application within which the actuator function is embedded is represented by process `ApplicationSoftware`. This process defines how the application uses the actuator function. The model in the Promela program assumes that the application initially starts the Actuator Controller Procedure and then sends a sequence of command requests to the Buffer State Machine interleaved with a sequence of execution requests to the Actuator Controller Procedure until, eventually, a shutdown request is made to stop operation of the hardware actuator.

The hardware actuator is modelled as a second process `HardwareActuator` which implements the behaviour defined in the bulleted list at the beginning of this section. In terms of the terminology of figure 15, the `ApplicationSoftware` and the `HardwareActuator` processes represent the "environment" around the behaviour which must be verified.

The properties listed informally in table 6 are formalized at the end of the Promela program in appendix B as LTL formulas which are then used by the Spin Model Checker as positive forms of `never` claims. As stated in the program, only the last property P-4 is satisfied. Properties P-1 to P-3 are *not* satisfied. This fact is somewhat counter-intuitive and deserves some discussion to highlight the importance of a formal verification approach.

Property P-1 ("Pending commands in the buffer are always eventually sent to the hardware actuator") is formalized as follows:

```
#define p (actReqDone==false) /* No Put request to Buffer is done*/
#define q (nItems==0)         /* No pending items in Bufer       */

/* If no new commands are placed in the buffer, then, eventually,*/
/* all pending commands are sent to the actuator.                 */
ltl P1 {(<>[]p) -> (<>[]q)}
```

One would expect this property to be violated exclusively when, at the time the actuator function is shut down, there are some pending commands left in

the buffer. The property would thereore seem to be guaranteed by the fact that the Actuator Controller Procedure only services a shutdown request if the buffer is empty (i.e. if all pending actuator commands have been processed). In reality, the Spin Model Checker shows that a violation may arise in the following scenario:

- The hardware actuator is switched on and is initializing.
- The Actuator State Machine is in state INIT and the Buffer State Machine is in state EMPTY.
- A shutdown request is made by the application.
- Since the buffer is empty, the Shutdown request is serviced at the next execution of the Actuator Controller Procedure which sends a `SwitchOff` command to the Actuator State Machine and then terminates.
- Since the Actuator State Machine is still in state INIT, the `SwitchOff` command has no effect and both it and the Buffer State Machine remain active.
- Some time later, a command is deposited in the buffer and remains permanently pending - this violates property P-1.

Violation of the property could probably be avoided either by constraining the application not to shut down the actuator function while initialization is under way; or by modifying the Actuator State Machine to respond to `SwitcOff` commands when it is in the INIT state.

Property P-2 ("If the Actuator State Machine is inactive, then the Buffer State Machine is inactive too") is formalized as follows:

```
#define r (actState==INACTIVE)      /* Actuator SM is inactive    */
#define s (bufferState==INACTIVE)   /* Buffer SM is inactive      */

/* If Actuator SM is inactive, then Buffer SM is inactive too    */
ltl P2 {[](r->s)}
```

One would expect this property to be satisfied because the Buffer State Machine is started as part of the starting of the Actuator State Machine and it is stopped as part of its stopping. In fact, the Spin Model Check shows a violation under the following conditions:

- When the Actuator State Machine is started, the transition action out of its Initial Pseudo State is executed (at this time, the state machine is still in an inactive state).
- The transition action starts the Buffer State Machine which therefore leaves its inactive state *before* the Actuator State Machine reaches its INIT state. This creates a violation of property P-2.

Property P-3 ("If a shutdown request is made, then, eventually, the entire actuator function is shut down") is formalized as follows:

```
#define r (actState==INACTIVE)      /* Actuator SM is inactive    */
#define s (bufferState==INACTIVE)   /* Buffer SM is inactive      */
#define t (shutdownReqPending==true) /* Shutdown request made     */
```



```
#define u (actHwState==OFF)          /* HW Actuator is switched off*/

/* If a shutdown request is made, then, eventually, Actuator SM  */
/* and Buffer SM terminate and Actuator HW is switched off       */
ltl P3 {[](t -> <>[](r && s && u))}
```

One would expect this property to be verified because a shutdown request causes the Actuator Controller Procedure to terminate by switching off the Actuator State Machine which in turns triggers commands to switch off the hardware and to stop the Buffer State Machine. In fact, this property is violated for the same reason that property P-1 is violated: if the application makes a shutdown request before the initialization of the hardware actuator has been completed, the hardware actuator will never be switched off.

Finally, property P-4 ("Buffer overflows are always eventually handled") is formalized as follows:

```
#define v (overflowHandled==true)     /* Overflow error handled  */
#define w (actContExec==true)      /* Act. Cont. PR is executed   */
#define x (actState==READY)           /* Actuator SM is in READY  */
#define y (actState==BUSY)            /* Actuator SM is in BUSY   */
#define z (bufferState==OVERFLOW)     /* Buffer SM is in OVERFLOW */
#define a (actContState==WAITING)      /* Act. Cont. PR is active */

/* If, during normal operation (i.e. after the Actuator has      */
/* completed its initialization and while the Controller         */
/* Procedure is active), the Buffer overflows and then the       */
/* Actuator Controller Procedure executes, then the overflow     */
/* error is handled.                                             */
ltl P4 {[]((z && a && (x || y) && <>w) -> <>v)}
```

This property is satisfied but it should be stressed that it is only satisfied because, in the light of the violations of properties P-1 and P-3, it has been restricted to apply only after the actuator initialization has been completed. If this restriction had been omitted, then the property would have been violated for exactly the same reasons which led to violation of properies P-1 and P-3.

In summary, this example shows that FW Profile models can be easily and systematically mapped to Promela programs where formal verification of their properties can be performed using the Spin Model Checker.

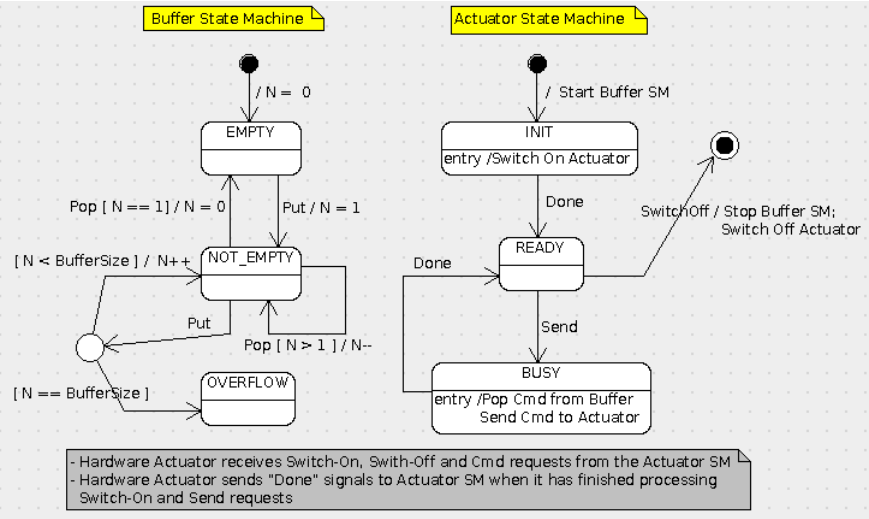

Fig. 18: State Machines to Control the Hardware Actuator

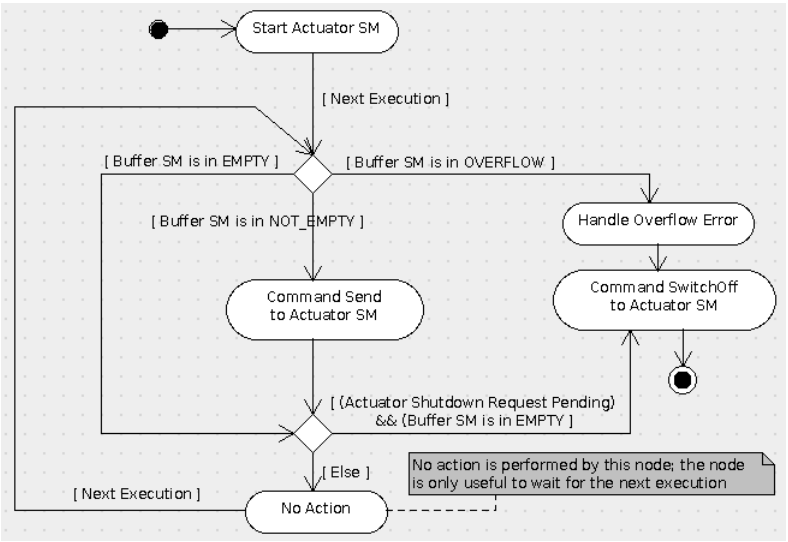

Fig. 19: Actuator Controller Procedure

A Verification Model for RT Container

This appendix presents the Promela model used to verify the properties of the RT Container defined by the FW Profile (see section 5.4).


```
/*
 * A. Pasetti - P&P Software GmbH - Copyright 2010 - All Rights Reserved
 *
 * RT Container Model (single start/stop cycle)
 *
 * The following aspects of a RT Container are modelled:
 * 1. The Activation Thread
 * 2. The Activation Procedure
 * 3. The Notification Procedure
 *
 * Of the actions in the Notification Procedure, only the "Implement Notification
 * Logic" action is modelled.
 *
 * Of the actions in the Activation Procedure, only the "Execute Functional Behaviour
 * Action" is modelled.
 *
 * In addition to modelling the RT Container itself, the following aspects
 * of its environment are modelled:
 * 1. A process representing a source of a Start command followed by a Stop command
 * 2. A process representing a source of Notification Requests
 */

mtype = {STARTED, STOPPED};
mtype RTContState = STOPPED;
mtype activProcState = STOPPED;
mtype notifProcState = STOPPED;
byte notifCounter = 0;
bool notifyActivThread = false;
bool activationThreadReleased = false;

/* A call to this macro corresponds to one execution of the loop in the
 * Notification Procedure.
 * The Notification Procedure is started and stopped by setting notifProcState
 * to, respectively, STARTED and STOPPED.
 */
```

```
36 inline NotifProc() {
37    if
38    :: (notifProcState==STOPPED) -> skip;
39    :: else ->
40       if
41       :: (activProcState==STOPPED) ->
42          notifProcState = STOPPED;     /* Terminate procedure */
43       :: else -> /* Activation procedure is still running */
44          if
45          :: true ->    /* Activation Thread should be notified */
46             notifyActivThread = true;
47          :: true ->    /* Notification of Activ. Thread should be skipped */
48             notifyActivThread = false;
49          fi;
50          if
51          :: notifyActivThread -> notifCounter++;
52          :: else -> skip;
53          fi;
54       fi;
55    fi;
56 }
57
58 /* A call to this macro corresponds to one execution of the loop in the
59  * Activation Procedure.
60  * The Activation Procedure is started and stopped by setting activProcState to,
61  * respectively, STARTED and STOPPED.
62  */
63 inline ActivProc() {
64    if
65    :: (activProcState==STOPPED) -> skip;
66    :: else ->
67       if
68       :: (RTContState==STOPPED) -> /* RT Container has been stopped */
69          activProcState = STOPPED; /* Terminate procedure */
70       :: (RTContState!=STOPPED) -> /* RT Container is still running */
71          if
```

```
72            :: true ->                    /* Functional behaviour has terminated */
73               activProcState = STOPPED; /* Terminate procedure */
74            :: true -> skip;              /* Functional behaviour has not yet terminated */
75            fi;
76        fi;
77    fi;
78  }
79
80  /* Process representing source of start/stop requests for RT Container  */
81  active proctype Environment() {
82    /* Start the RT Container (this also releases the Activation Thread
83     * and starts the Notification and Activation Procedures) */
84    atomic{notifProcState = STARTED;
85           activProcState = STARTED;}
86    RTContState = STARTED;
87    /* Stop the RT Container (this also notifies the Activation Thread */
88    RTContState = STOPPED;
89    notifCounter++;
90  }
91
92  /* Process representing the thread executing the notification procedure.
93   * This model assumes that only up to 2 pending notifications can be
94   * buffered.
95   */
96  active proctype NotificationThread() {
97   end1:
98    do
99    :: (notifCounter<3) -> NotifProc();
100   :: true -> skip;
101   od;
102 }
103
104 /* Process representing the Activation Thread */
105 active proctype ActivationThread() {
106
107   /* Wait until RT Container is started and released */
```

```
108  end2:
109   (RTContState==STARTED);
110   activationThreadReleased = true;
111
112   do
113   :: atomic { (notifCounter>0) -> notifCounter--; }
114         ActivProc();    /* Execute Activation Procedure */
115         if       /* Check if Activation Procedure has terminated */
116         :: (activProcState==STOPPED) -> RTContState = STOPPED;
117                                         NotifProc();
118                                         break;
119         :: (activProcState==STARTED) -> skip;
120         fi;
121         if      /* Check if RT Container is stopped */
122         :: (RTContState==STOPPED) -> ActivProc();
123                                      NotifProc();
124                                      break;
125         :: (RTContState==STARTED) -> skip;
126         fi;
127   od;
128 }
129
130 /* Define variables used to formulate never claims */
131 #define p (RTContState==STOPPED)     /* RT Container is stopped */
132 #define q (activProcState==STOPPED)    /* Activation Procedure is stopped */
133 #define r (notifProcState==STOPPED)    /* Notification Procedure is stopped */
134 #define s (activProcState==STARTED)    /* Activation Procedure is started */
135 #define t (notifProcState==STARTED)    /* Notification Procedure is started */
136 #define u (notifyActivThread==false)  /* No notification requested by Notif. Procedure */
137 #define v (notifCounter==0)        /* All notification requests consumed by Activ. Thread */
138 #define z (activationThreadReleased==true)  /* Activation thread has been released */
139
140 /* The following formulas are used as (positive forms of) never claims */
141
142 /* If the RT Container is stopped after the activation thread has been released, then
143  * eventually the Activation Procedure terminates. */
```

```
144 ltl P4 {[] ( (p && z) -> <>  q )}
145
146 /* If the Activation Procedure is started and then it is terminated,
147  * then eventually the RT Container is stopped. */
148 ltl P5 {[] (!q  ->  [] (q -> <>  p ))}
149
150 /* If the Activation Procedure terminates, then eventually the Notification Procedure terminates. */
151 ltl P6 {[] (q -> <>  r )}
152
153 /* If the Activation Procedure is running, then the Notification Procedure is running too. */
154 ltl P7 {[] (s -> t )}
155
156 /* If notifications cease and RT Container and Activation Procedure remain active,
157  * then eventually all notifications are consumed by Activation Thread. */
158 ltl P8 {<>[] (u && !p && q) -> <>[] v }
```

Listing 6: Verification Model for RT Container

B Promela Program for Actuator Control Example

This appendix presents the complete Promela program for the formal verification of the actuator control example discussed in section 6.3. The Promela code includes (at the very end) the definition of the LTL formulas which represent the properties to be satisfied by the actuator controller.


```
/* P&P Software GmbH - Copyright 2013 - All Rights Reserved          */
/*                                                                   */
/* Example of Formal Verification of a FW Profile Model              */
/*                                                                   */
/* Model of the actuator example described in section 6.5 of the     */
/* FW Profile Definition Document.                                   */

mtype = {INACTIVE};                          /* State common to all SMs */
mtype = {Start, Stop};                     /* Commands common to all SMs */
mtype = {EMPTY, NOT_EMPTY, OVERFLOW};            /* States of Buffer SM */
mtype = {INIT, READY, BUSY};                   /* States of Actuator SM */
mtype = {Send, Done, SwitchOff};             /* Commands of Actuator SM */
mtype = {Put, Pop};                            /* Commands of Buffer SM */
mtype = {STOPPED, STARTED, WAITING}; /* Actuator Cont. Proc. States */
mtype = {OFF, HW_INIT, HW_BUSY, HW_AVAIL}; /* States of Actuator HW */

mtype bufferState = INACTIVE;     /* State of Buffer SM              */
mtype actState = INACTIVE;        /* State of Actuator SM            */
mtype actContState = STOPPED;     /* State of Actuator Controller PR */
mtype actHwState = OFF;           /* State of Hardware Actuator      */
byte nItems = 0;                  /* Number of items in buffer       */
bool shutdownReqPending = false;
bool overflowHandled = false;
bool actReqDone = false;
bool actContExec = false;

/* ------------------------------------------------------------------ */
/* A call to this macro models the sending of a command to the Buffer */
/* State Machine. The buffer size is assumed to be equal to 3.        */
inline BufferSM(cmd) {
   if
   :: (bufferState==INACTIVE) && (cmd==Start) ->
       bufferState = EMPTY;
       nItems = 0;
   :: (bufferState==EMPTY) && (cmd==Put) ->
```

```
36          nItems = 1;
37          bufferState = NOT_EMPTY;
38      :: (bufferState==NOT_EMPTY) && (cmd==Put) && (nItems<3) ->
39          nItems++;
40      :: (bufferState==NOT_EMPTY) && (cmd==Put) && (nItems==3) ->
41          bufferState = OVERFLOW;
42      :: (bufferState==NOT_EMPTY) && (cmd==Pop) && (nItems>1) ->
43          nItems--;
44      :: (bufferState==NOT_EMPTY) && (cmd==Pop) && (nItems==1) ->
45          nItems = 0;
46          bufferState = EMPTY;
47      :: (bufferState!=INACTIVE) && (cmd==Stop) ->
48          bufferState = INACTIVE;
49      :: else -> skip;
50      fi;
51  }
52
53  /* ------------------------------------------------------------------ */
54  /* A call to this macro models the sending of a command to the        */
55  /* Actuator State Machine.                                            */
56  inline ActuatorSM(cmd) {
57      if
58      :: (actState==INACTIVE) && (cmd==Start) ->
59          BufferSM(Start);            /* Send command Start to Buffer SM */
60          actState = INIT;
61          actHwState = HW_INIT;       /* Switch-On command to Actuator HW */
62      :: (actState==INIT) && (cmd==Done) ->
63          actState = READY;
64      :: (actState==READY) && (cmd==Send) ->
65          actState = BUSY;
66          BufferSM(Pop);                /* Send command Pop to Buffer SM */
67          actHwState = HW_BUSY;           /* Send command to Actuator HW */
68      :: (actState==BUSY) && (cmd==Done) ->
69          actState = READY;
70      :: (actState==READY) && (cmd==SwitchOff) ->
71          BufferSM(Stop);             /* Send command Stop to Buffer SM */
```

```
72          actHwState = OFF;          /* Switch-Off command to Actuator HW */
73          actState = INACTIVE;
74      :: else -> skip;
75      fi;
76  }
77
78  /* ------------------------------------------------------------------- */
79  /* A call to this macro models the execution of the Actuator          */
80  /* Controller Procedure.                                              */
81  inline ActuatorControllerPR() {
82      if
83      :: (actContState==STOPPED) ->
84          skip;
85      :: (actContState==STARTED) ->
86          ActuatorSM(Start);                      /* Start Actuator SM */
87          actContState = WAITING;
88      :: (actContState==WAITING) ->
89          if
90          :: (bufferState==EMPTY) -> skip;
91          :: (bufferState==NOT_EMPTY) ->
92              ActuatorSM(Send);              /* Send Send to Actuator SM */
93          :: (bufferState==OVERFLOW) ->
94              overflowHandled = true;
95              ActuatorSM(SwitchOff);    /* Send SwitchOff to Actuator SM */
96              actContState = STOPPED;            /* Terminate procedure */
97          fi;
98          if
99          :: (shutdownReqPending==true) && (bufferState==EMPTY) ->
100             ActuatorSM(SwitchOff);  /* Send SwitchOff to Actuator SM */
101             actContState = STOPPED;            /* Terminate procedure */
102         :: else -> skip;
103         fi;
104     fi;
105 }
106
107 /* ------------------------------------------------------------------- */
```

```
108 /* Process representing the Hardware Actuator.                    */
109 /* The actuator hardware responds to Switch-On and Send requests by  */
110 /* sending a Done command to the Actuator SM.                      */
111 active proctype HardwareActuator() {
112  end1:
113   do
114   :: (actHwState == HW_INIT) ->
115        actHwState = HW_AVAIL;
116        ActuatorSM(Done);
117   :: (actHwState == HW_BUSY) ->
118        actHwState = HW_AVAIL;
119        ActuatorSM(Done);
120   od;
121 }
122
123 /* ------------------------------------------------------------------*/
124 /* Process representing the application software which starts the    */
125 /* Actuator Controller Procedure, then periodically executes it and  */
126 /* puts commands in the Actuator Buffer until it eventually stops the*/
127 /* Actuator Controller Procedure.                                    */
128 active proctype ApplicationSoftware() {
129   actContState = STARTED;       /* Start Actuator Controller Procedure*/
130  end2:
131   do
132   :: true ->
133      actReqDone = true;
134      BufferSM(Put);
135      actReqDone = false;
136   :: true ->
137      actContExec = true;
138      ActuatorControllerPR();
139      actContExec = false;
140   :: (!shutdownReqPending) ->
141      shutdownReqPending = true;
142   od;
143 }
```

```
144
145 /* ------------------------------------------------------------------ */
146 /* Define variables used to formulate never claims:                   */
147 #define p (actReqDone==false)    /* No Put request to Buffer is done     */
148 #define q (nItems==0)            /* There are no pending items in Buffer */
149 #define r (actState==INACTIVE)     /* Actuator SM is inactive            */
150 #define s (bufferState==INACTIVE)  /* Buffer SM is inactive              */
151 #define t (shutdownReqPending==true) /* A shutdown request has been made*/
152 #define u (actHwState==OFF)     /* The Hardware Actuator is switched off */
153 #define v (overflowHandled==true)    /* Overflow error has been handled */
154 #define w (actContExec==true)        /* The Act. Cont. PR is executed   */
155 #define x (actState==READY)          /* Actuator SM is READY            */
156 #define y (actState==BUSY)           /* Actuator SM is BUSY             */
157 #define z (bufferState==OVERFLOW)    /* Buffer SM is in OVERFLOW        */
158 #define a (actContState==WAITING)    /* Act. Cont. PR is active         */
159
160 /* The following formulas are used as (positive forms of) never claims  */
161
162 /* If no new commands are placed in the buffer, then, eventually,       */
163 /* all pending commands are sent to the actuator.                       */
164 ltl P1 {(<>[]p) -> (<>[]q)}
165
166 /* If the Actuator SM is inactive, then the Buffer SM is inactive too  */
167 ltl P2 {[](r->s)}
168
169 /* If a shutdown request is made, then, eventually, the Actuator SM    */
170 /* and the Buffer SM terminate and the Actuator HW is switched off     */
171 ltl P3 {[](t -> <>[](r && s && u))}
172
173 /* If, during normal operation (i.e. after the Actuator has completed */
174 /* its initialization and while the Controller Procedure is active),  */
175 /* the Buffer overflows and then the Actuator Controller Procedure    */
176 /* executes, then the overflow error is handled.                       */
177 ltl P4 {[]((z && a && (x || y) && <>w) -> <>v)}
```

Listing 7: Verification Model for Actuator Control Example

References

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

[2] 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

[3] Gerald J. Holzmann: *The Spin Model Checker - Primer and Reference Manual*. PP-UM-COR-0001, Revision 1.2.0, Addison-Wesley, U.S.A., 2004

[4] Assert Project Web Site, `www.assert-project.net`

[5] CORDET Project Web Site, `www.pnp-software.com/cordet`

[6] Spin Model Checker Web Site, `http://spinroot.com/spin/whatispin.html`

