Development of Real-Time Mission-Critical and Reusable Software with the FW Profile

Vaclav Cechticky*[1] and Alessandro Pasetti†[1]

[1]P&P Software GmbH, High Tech Center 1, Tägerwilen, CH-8274,
www.pnp-software.com

The Framework (FW) Profile is a UML profile and a methodology for real-time, safety-critical, reusable software applications. It was initially developed at ETH-Zürich and has recently been extended and provided with C-language support. *Real-time design* is supported by enforcing separation of functional and timing aspects. *Mission criticality* is supported by providing modelling concepts (State Machine, Activity Diagrams, Active Objects) with simple and unambiguous semantics to make them compatible with formal verification. C implementations of these concepts are provided with a Qualification Data Package which can be used for the certification of end-applications. *Reusability* is supported by allowing modelling of adaptable and extensible behaviour in reusable components.

The Framework Profile (or *FW Profile* for short) was born in 2005 as an offshoot of the ASSERT Project [4] at the Department of Automatics of ETH-Zürich. Its initial aim was to support the modelling of functional behaviour of reusable software architectures (software frameworks). The profile was later extended to encompass the complete modelling of software applications. The extensions were partly funded by the European Space Agency [5].

The FW Profile is a requirements-level modelling language but, from 2012, it is provided with C-language support which allows the behavioural models created with the profile to be translated into C-code. The C libraries which implement the profile are supplied with a Qualification Data Package to make them suitable for deployment in safety-critical applications. More recently, the translation of FW Profile models into Promela and their validation with the Spin Model Checker [6] have been investigated with the aim of creating an environment where users can specify their requirements using a simple but precise language; validate their requirements using a model checker; and implement them using a certifiable library of C modules. This environment is shown in schematic form in figure 1.

This paper begins by presenting the modelling concepts proposed by the FW Profile. It then discusses the use of formal verification for FW Profile models (section 2) and finally describes the C-language library to implement them (section 3).

*pasetti@pnp-software.com
†vaclav.cechticky@pnp-software.com

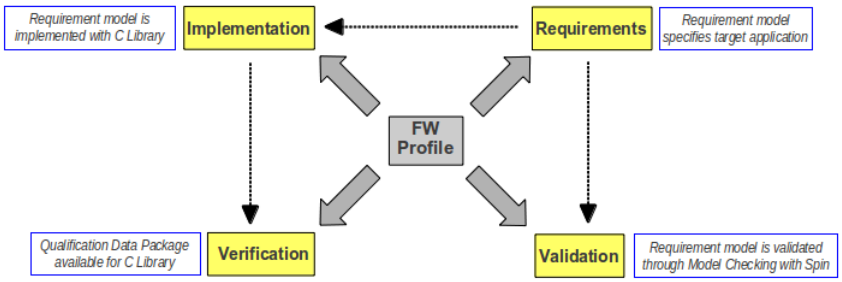

Fig. 1: The FW Profile Environment

1 Modelling Concepts

The objective of the FW Profile is to provide the means to build a logical model for a 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. 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 (timing) behaviour

The State Machine and Procedure concepts are defined as 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.

The restriction of the state machine and activity concepts of UML was motivated by the over-complexity and semantic ambiguity of these concepts in standard UML. To illustrate, consider UML state machines and consider the following questions: in which state is a state machine when a transition action is executed? Under which conditions is a do-action executed? What happens if a transition request arrives while a transition is already under way? The answers to these questions are at best unclear in standard UML. The implication of this lack of clarity is that the behaviour of an application which has been specified with standard UML state machines or activity diagrams may not match the intention of the author of the specification and that analysis of the expected behaviour of an application cannot be done at requirements level but must instead take into account software-level design and implementation. In the safety-critical domain where predictability of behaviour is of paramount importance these limitations are simply unacceptable.

The full definition of the state machine and procedure concepts of the FW Profile is presented in [8]. It cannot be reproduced here in full for reasons of space but its guiding principles were *simplification* and *disambiguation*.

Simplification was done at three levels. Firstly, all time-related constructs of UML have been dropped. Secondly, both State Machines and Procedures are defined as purely reactive and passive entities (state machines, for instance,

only do something if they receive a transition request; they never spontaneously initiate a transition). Finally, all buffering mechanisms for transition and execution requests have been dropped. All three restrictions follow naturally from the choice (discussed in the next section below) to use State Machines and Procedures exclusively to model non-time-related behaviour.

Disambiguation was obtained by carefully defining the response a State Machine or Procedure to an external transition or execution request. The behaviour of State Machines and Procedures in the FW Profile is thus fully predictable.

RT Containers complement the State Machine and Procedure concepts 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 (e.g. for a behaviour which is modelled through State Machnes and Procedures). 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.

Additionally, the RT Container model guarantees certain properties. Thus, for instance, a RT Container guarantees that certain "initialization actions" are executed before the container starts processing notification requests; or that certain "finalization actions" are executed before it is shut down; or that all notification requests are eventually processed. All these properties are guaranteed by design in the sense that they are inherent in the way the RT Container concept is defined and can be relied upon in any compliant implementation.

The focus of the FW Profile on the description of behaviour independently of software-level decision is found in all requirements-level modelling languages. To this common feature, the FW Profile adds two distinctive features: the *separation between functional and non-functional behaviour*; and the *support for the modelling of reusable software assets*. These features are discussed in the next two subsections.

1.1 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 hand-shaking 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. Thus, the methodology promoted by the FW Profile is one where developers use state machines and procedures to specify the functional part of their application in a manner that is independent of its timing-related requirements and then, as a separate activity, they use RT Containers to specify the timing-related behaviour of the applications. In practice, the latter means that they can use RT Containers to clearly identify the conditions under which the various flows of functional behaviour must be activitated.

The separation between the modelling of functional and non-functional aspects of an application is especially valuable in the real-time domain. The development of real-time applications is especially challenging because of: (a) the difficulty of capturing the real-time needs of an application in unambiguous requirements; and (b) because real-time requirements often impact an application as a whole so that a change in a real-time requirement may disrupt the entire architecture of an application. The latter point weighs heavily on overall costs because real-time requirements normally derive from system-level considerations and can therefore only be defined late in the application development process.

The RT Container concept addresses both issues because, on the one hand, it provides a pre-defined framework within which real-time needs can be formulated as requirements and, on the other hand, it allows the specification and implementation of these needs to be insulated from the specification and implementation of the rest of the application so the impact of late changes to real-time requirements can be contained.

1.2 Support for Reusability

Traditional modelling languages do not make explicit provisions for software reusability. This is a significant drawback where there is a need to develop infrastructure software which is intended to be used as a basis for the development of families of related applications.

The FW Profile specifically caters to this need and it does so within the *software framework paradigm*. 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 2: the framework components are first adapted to meet the needs of the target application and are then assembled to build the target application.

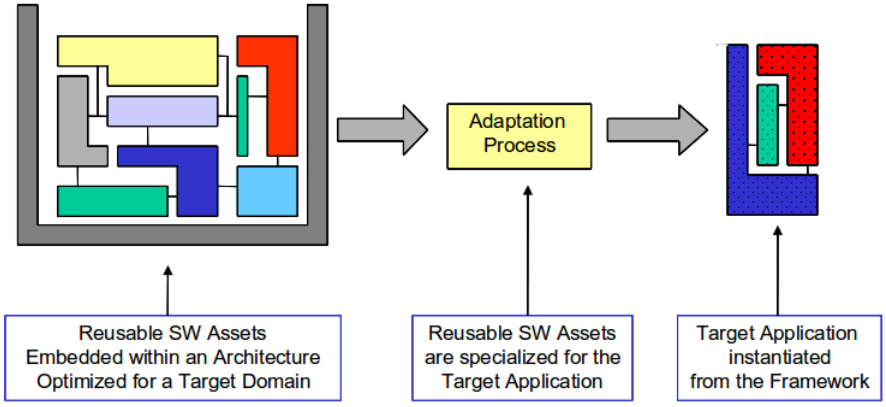

Fig. 2: The Software Framework Concept

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 a 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 3. 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.

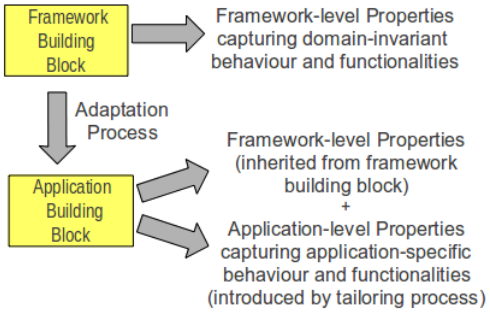

Fig. 3: Invariant Properties

The adaptation mechanisms of the FW Profile are described in full in [8]. For purposes of illustration, the adaptation mechanisms of State Machines are summarized below. Two such mechanisms are supported. The first one relies on the use of the $\langle\langle AP \rangle\rangle$ stereotype (see section 1.2) 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: either that the content of the stereotyped element is not defined at framework level and the definition must be done at application level; or that a default content for the stereotyped element is defined at framework level but can be overridden at application level.

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.

As explained above, 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.

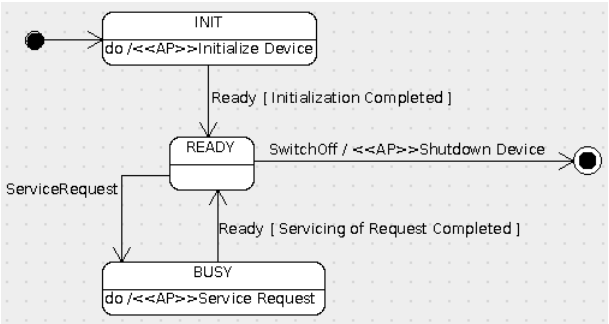

Fig. 4: Hardware Device with a Busy Wait

As an example, consider the state machine of figure 4. 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 operational 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 4 and should therefore not be relied upon when designing a generic framework.

The example of figure 4 is straightforward but in more realistic cases with several interacting state machines possibly controlled by different flows of execution, the invariant properties are often non-trivial and proving that they actually hold on a given model can be a challenging task. Formal verification techniques as discussed in the next section are especially valauble for the latter purpose.

Figure 5 recasts figure 3 for the special case of state machines. It 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.

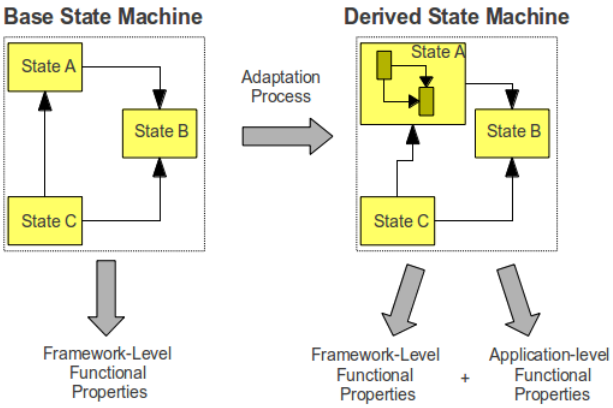

Fig. 5: State Machine Adaptation Process

2 Support for Formal Verification

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 ex-

pressed in a formalism with an unambiguous semantics. They can therefore be applied to FW Profile models. There are several formal verification methods. The one which has been applied to the FW Profile is *Model Checking*. 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. For the FW Profile, 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].

Figure 6 shows the basic approach to model-checking with the FW Profile. The inputs to the 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, Linear Temporal Logic (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 6 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.

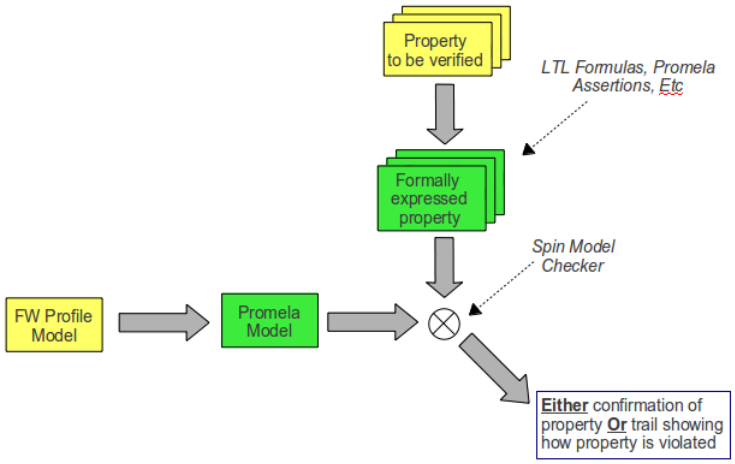

Fig. 6: Basic Approach for Model Cheking of FW Profile Models

Thus, a more realistic approach to formal verification is as in figure 7 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, the model checking approach selected for the FW Profile is based on the definition of general "Promela patterns" of how FW Profile models can be transformed into Promela models rather than on the development of a tool which would, at most, only automatize a small part of the transformation. Space does not permit the Promela Patterns to be presented in this paper but their description can be found in [8]. The important point to make here is that their availability helps a designer to rapidly create Promela models which are adapted to a certain property to be verified. They therefore constitute a kind of verification front-end for the FW Profile.

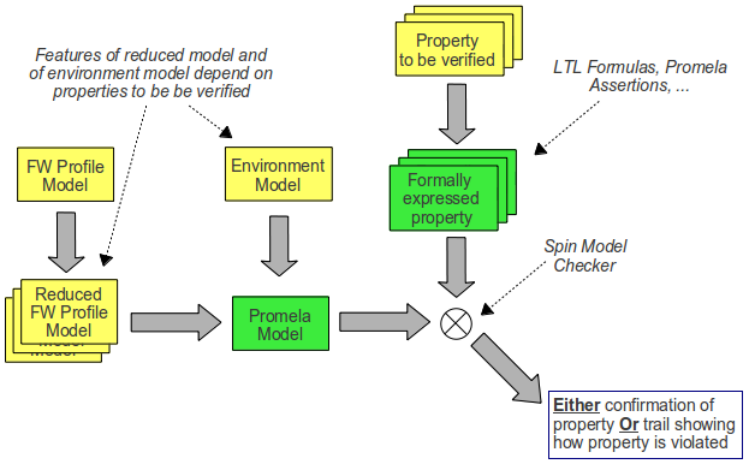

Fig. 7: Realistic Approach to Model Cheking of FW Profile Models

3 C-Language Support

The FW Profile is a requirements-level modelling language. As such it is implementation-independent: the same FW Profile model can be mapped to code in many different (but equivalent) ways. There is, however, an obvious advantage to providing a standard way to transpose a FW Profile model into code. This goal fulfilled by the *C1 Implementation*. The C1 Implementation is a C-language library which offers pre-defined and configurable components to implement in C the three modelling concepts of the FW Profile: State Machines, Procedures and RT Containers.

The C1 Implementation is provided both as free software (GNU GPLv3) and on a commercial license. The source code and its extensive documentation can be downloaded from [7].

The main characteristics of the C1 Implementation are:

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

The coverage level of the Test Suite and the availability of formal requirements with verification and traceability evidence make the C1 Implementation especially well-suited for use in safety-critical domains where applications must undergo a certification process. The documentation of the C1 Implementation can then be directly included in the certification process for the end application.

The C1 Implementation is extremely easy to use and is provided with numerous coding examples and a complete demo application. Hence, many users will be able to deploy it directly in their source code.

For users who prefer to go through a GUI-based interface, P&P Software GmbH also provide a web application where state machines can be created and configured using graphical and wizard-based operations. A code-generating back-end can then be used to generate the configuration code for the state machine. The web application is freely accesible and usable from [7]. At present it only covers the state machine part of the FW Profile but will be extended to cover Procedures and RT Containers in the near future. A screenshot is presented in figure 8.

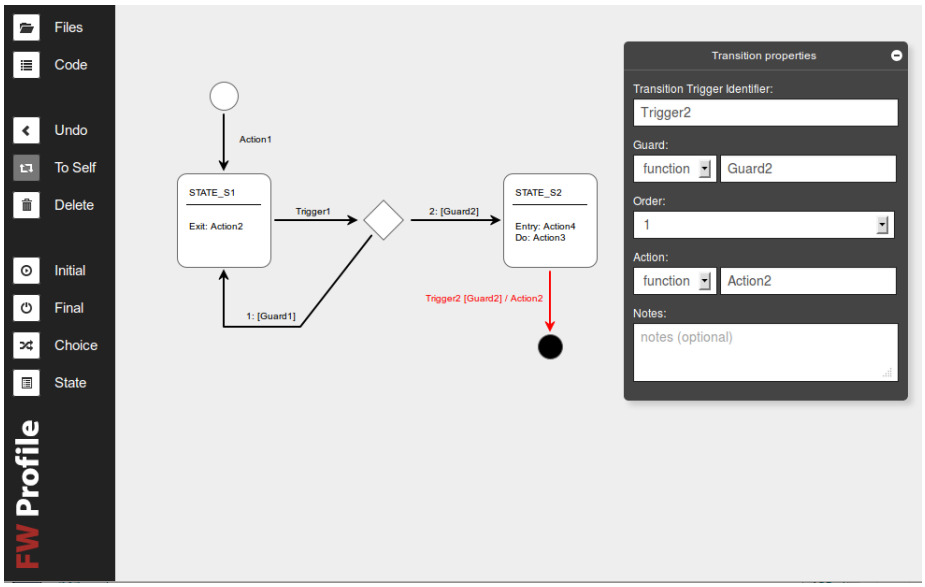

Fig. 8: Screenshot of the FW Profile Web Application

References

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

[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`

[7] FW Profile and C1 Implementation Web Site, `http://www.pnp-software.com/fwprofile`

[8] Alessandro Pasetti, Vaclav Cechticky: *The Framework Profile*. PP-DF-COR-00001, Revision 1.3.0, P&P Software GmbH, Switzerland, 2013; available from: `http://www.pnp-software.com/fwprofile`

