Runtime observable and adaptable UML State Machines: Models@run.time approach

Miren Illarramendi
Mondragon Goi Eskola Politeknikoa S.Coop
Mondragon, Spain
millarramendi@mondragon.edu

Leire Etxeberria
Mondragon Goi Eskola Politeknikoa
Mondragon, Spain
letxeberria@mondragon.edu

Xabier Elkorobarrutia
Mondragon Goi Eskola Politeknikoa
Mondragon, Spain
xelkorobarrutia@mondragon.edu

Goiuria Sagardui
Mondragon Goi Eskola Politeknikoa
Mondragon, Spain
gsagardui@mondragon.edu

ABSTRACT

An embedded system is a self-contained system that incorporates elements of control logic and real-world interaction. UML State Machines constitute a powerful formalism to model the behaviour of these types of systems. In current industrial environments, the software of these embedded systems have to cope with the increasing complexity and robustness requirements at runtime. One way to manage these requirements is having the software component's behaviour model available at runtime (models@run.time). Thus, it is possible to enhance the safety of the software component by enabling verification and adaptation at runtime. In this paper, we present a model-driven approach to generate software components (namely, RESCO framework), which are able both to provide their internal information in model terms at runtime and adapt their behaviour automatically when an error or an unexpected situation is detected. The aforementioned runtime introspection and adaptation abilities are added automatically to the software component and it does not require the developer make any extra effort. The solution has been tested in the design and implementation of an industrial Burner controller. Results indicate that the software components generated by the presented solution provides introspection at runtime. Thanks to this introspection ability at runtime, the software components are able to adapt automatically from their normal-mode behaviour to a safe-mode behaviour which was defined to be used in erroneous or unexpected situations at runtime. Therefore, it is possible to enhance the safety of the systems consisting of these software components.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**; *Embedded software*;

KEYWORDS

Models@runtime, Runtime Adaptation, Embedded Systems, UML State Machines.

ACM Reference Format:
Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, and Goiuria Sagardui. 2019. Runtime observable and adaptable UML State Machines: Models@run.time approach. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC'19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297459

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3297459

1 INTRODUCTION

Cyber-Physical Systems (CPSs) integrate digital cyber computations with physical processes. These CPSs are composed of embedded systems and networks that monitor and control physical processes with sensors and actuators [11].

The scope, complexity, and pervasiveness of CPSs continue to increase dramatically. The consequences of these systems failing can range from the mildly annoying to catastrophic. As stated in [1], CPSs include embedded systems, which are inherently self-adaptive, because they are embedded in an environment and are meant to observe and/or influence it. Event-driven architecture is a commonly used pattern to develop these systems, and UML State Machines (UML-SM) constitute a widely used formalism to design the behaviour of such systems. In addition, following Model Driven Engineering (MDE) approach, the final code can be obtained automatically if assisted by a code generation tool.

However, generating the code automatically is not enough. The software of these embedded systems increasingly assumes the responsibility of providing functionality in systems so there is a need to provide low-cost mechanisms to ensure correct and safe behaviour at runtime. Work on models@run.time seeks to extend the applicability of models produced in MDE approaches to the runtime environment. Having the model at runtime is the first step towards the runtime verification; and having the mechanism to adapt the model at runtime implying a software component change is the next step once an unexpected situation or an error is detected. Thus, the safety of the system is enhanced.

In this paper, we present a models@run.time approach to automatically generate UML-SM code with runtime introspection, verification and adaptation ability. In addition, an externalized runtime checker and adaptation system is presented. The main contribution provides the following benefits:

(1) runtime monitoring, verification and adaptation ability which are based on information provided by the software components in terms of their model elements at runtime.
(2) the software developer is not involved in instrumenting the code and thus, can focus exclusively on modelling the behaviour of the software components by UML-SMs. Additional infrastructure for having introspection and adaptation ability at runtime is automatically generated.

Section 2 presents background concepts of the work, and Section 3 describes a Model-driven approach for automatically generating software components that are observable and adaptable at runtime. Section 4 presents the empirical evaluation of the solution. In Section 5 related work is detailed and, finally, Section 6 closes the paper with conclusions and future lines.

2 BACKGROUND

In this section, we will define some concepts that will help to understand the work presented in this document.

One way to perform runtime verification is to observe the runtime information (traces) sent by the software controller to the externalized runtime checker/adapter. Since correct traces will be finite and predefined in the checker/adapter, when the received trace is not defined as a correct one, the checker/adapter comes to a state that a Trace-Violation has been detected.

As the runtime checker needs to receive traces/information from the monitored software component, we need to instrument the latter. There are different approaches for instrumentation:

- Source code instrumentation: this technique is used to modify source code to insert appropriate code (usually conditionally compiled so you can turn it off). Programmers implement instrumentation in the form of code instructions that monitor different aspects of a system.
- Model instrumentation: this technique specifies which elements of the model will be monitored at runtime and it automatizes the source code instrumentation.

In our solution we are using the second approach.

The software components of the presented solution are generated automatically from the UML-SM model. Even if we apply model checking methods to the models, due to the complexity of the systems there may be residual faults. In this scenario, solutions that support runtime verification are needed.

The faults that can be detected by the solution are:

- random hardware faults such as bit inversions or changing errors,
- random software faults such as heisenbugs,
- remaining software faults: errors that remain after the validation and verification in design and development phase
- unanticipated environmental faults: not considered in the design and development phase.

Runtime Verification can be performed in different ways. One of them being performed using the information of model elements (current state, event, next state,...) of the UML-SM model of the software component under study. This enables using a common language to design and verify software components at runtime.

In order to maintain the model at runtime, the software component has to be observable by the externalized checking and adapting system. In order to support this characteristic, the software components that are monitored need to have the introspection and reflection ability. Introspection supports runtime monitoring of the program execution with the goal of identifying, locating and analyzing errors [23].

Reflection [25] can be defined as the property by which a component enables observation and control of its own structure and behavior from outside. This means that a reflective component provides a meta-model of itself, including structural and behavioral aspects, which can be handled by an external component. A reflective system is basically structured around a representation of itself or meta-model that is causally connected to the real system.

As explained in [17], there are two main dynamic software adaptation approaches using runtime models of the software: planned and unplanned adaptation. Our solution is based on the unplanned adaptation approach. This type of adaptation is triggered by unexpected events or when a fault is detected at runtime and reactive decisions are needed to dynamically adapt the system to avoid system failures.

In our approach, the adaptation of the component follows the idea presented in [17]. In this work, they modelled a basic adaptation by a state machine with three states: Active (normal-mode of operation), Passive (stopped the initiated transition and will not initiate new transitions) and Quiescent (no longer operational).

Our solution is inspired by the work presented in [15] where the adaptation is externalized. When using an externalized runtime adaptation, the behaviour of the software component is monitored by components outside the running system. These external components are responsible for determining when a software component's behaviour is within the envelope of acceptable system parameters. When the software component's behaviour fall outside of the expected limits, the external components start the adaptation process (failsafe system). A failsafe device/system is expected to eventually fail but when it does it will be in a safe way. In figure 1, the architecture of the solution is shown.

We define as normal-mode of operation the situations in which all elements of the system are functioning as intended and the software component's behaviour is within the envelope of acceptable system parameters. When the software component's behaviour is not working in the expected limits, the adaptation process starts and the software component is sent to a safe-mode of operation (graceful degradation). The safe-mode operation is an aspect of a fault tolerant software system, where in case of some faults, system functionality is reduced to a smaller set of services/functionalities that can be performed by the system [12].

To accomplish the adaptation process, the externalized components (1) maintain the model of the monitored running software component and (2) support reasoning about system problems.

3 GENERATING RUNTIME OBSERVABLE AND ADAPTABLE STATE MACHINE SOFTWARE COMPONENTS

This section presents the model-driven approach for generating runtime observable and adaptable UML-SMs components following a models@runtime approach.

In the following subsections we will present the details of the process of generating reflexive software components that provide

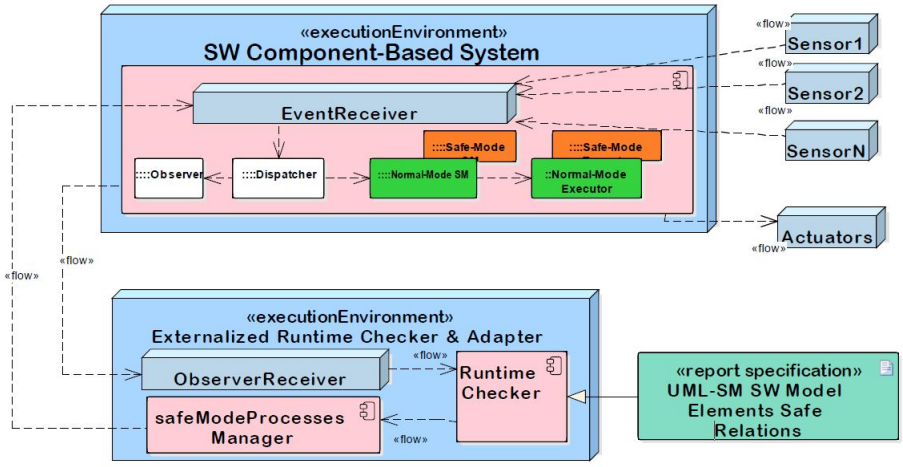

Figure 1: Runtime Architecture: Controller and Externalized Checker/Adapter

information in terms of UML-SM model elements at runtime and allow adapting their behaviour dynamically when an unexpected situation or error is detected.

3.1 RESCO infrastructure

The solution is based on the RESCO (REflective State machines-based observable software COmponents) metamodel. This metamodel is composed of two packages: (1) a design package corresponding to the component specific UML-SM with observability information and (2) a runtime package corresponding to the runtime infrastructure.

As background, and before continuing with the process of generating the RESCO software components, the infrastructure of the RESCO is explained. RESCO metamodel has two packages (1) a design package that is used for modeling the application specific part and (2) a runtime package that enables a UML-SM based software component to reflect the model it comes from (models@runtime approach).

The application specific part of the model is modeled using the design package and it includes the *StateMachine* and *Executor* concepts. The *StateMachine* describes an hierarchical state machine along with a description of the *Reactions* defined in each of the *States*. The *Executor* has the implementation of the *Actions* that need to be triggered and the *Conditions* that need to be evaluated. The *Reactions* have the references to them. The *StateMachine* and the *Executor* are generated automatically from the UML-SM model defined by the designer.

The runtime part, that is generic for all the components and applications, is modelled using the runtime package. It includes generic elements used for providing models@run.time observation capabilities: the *Dispatcher*, the *Observer* and the *EventReceiver*. The runtime elements are used for implementing the execution semantics of state machines.

Using this two packages, an object structure that reflects the model is generated. Thus, it is possible to adapt the model without recompiling the solution.

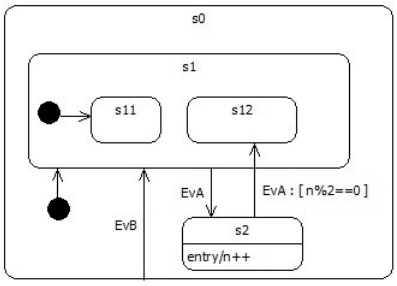

Figure 2: State Machine, guiding example

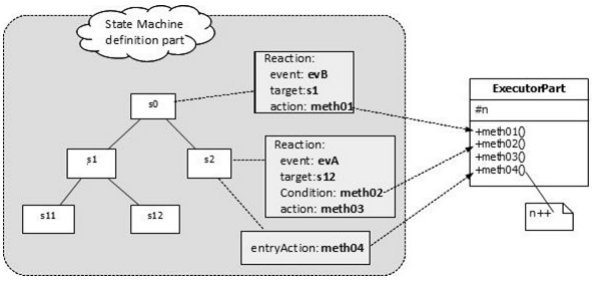

Figure 3: Guiding Example: Transformation of the Design package of RESCO metamodel

Let us consider the state machine of figure 2 as a guiding example. The representation of this state machine developed by means of RESCO is illustrated in figure 3. The state definition part is a tree-like composite object-structure that reflects the state structure of the model and each state has the specification of the behavior attached to its different elements so that changing this object structure means model modification and vice-versa. Later in the document, figure 6 shows how this structure is transformed to C++ code.

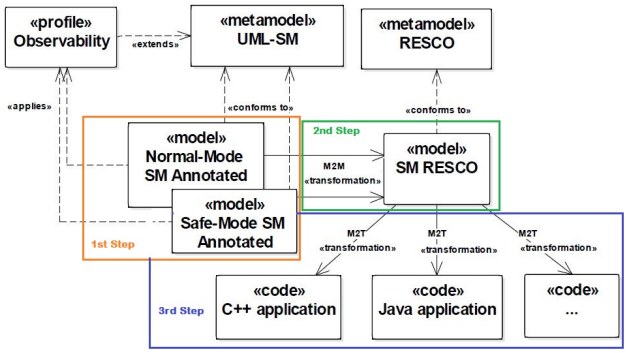

Figure 4: Model-driven workflow

3.2 Generating software components using RESCO

In this subsection we are going to present the different steps of the Model-driven workflow shown in figure 4.

First, the behaviour of the components is modelled using UML-SMs by Papyrus [18] and the states to be observable at runtime are annotated using the defined observability profile. This first step is performed by the designer and in order to have the adaptation ability at runtime, the designer has to design two or more annotated UML-SM models: normal-mode UML-SM and one or more safe-mode (degraded-mode) UML-SM. Thus, when an error is detected, the software component will adapt from normal-mode UML-SM to the safe-mode (degraded-mode) UML-SM.

In a second step, a RESCO metamodel conformant model (instrumented model) is generated automatically by ATL [4] model-to-model (M2M) transformations. Note that the model generation corresponds mainly to the (1) design package as the (2) runtime package remains the same for any component. In the last step, the model conforming to the RESCO metamodel is transformed to code by means of a model to text (M2T) transformation using Acceleo [20].

3.2.1 1st step: Behaviour design of the software component. The first step is to model the behaviour of software components by UML-SM models using Papyrus [18] modeling tool. In the presented solution, runtime adaptation is one of the main contributions and for that, in this first step the software designer has to design also the alternative safe-mode behaviour(s) of the software component. All these models will be transformed by the RESCO M2M transformation rules. In addition, the designer has to define for each of the original state the initial state in the safe-mode model to be adapted.

Different types of systems and operation-modes require different monitoring needs. Although it is possible to obtain information about all the states, events and transitions, due to reduced processing resources, it may be desirable to monitor only a subset of available monitoring information that will be verified. We have defined an observability profile that provides an « Observable » stereotype to select at design level those parts of the controllers which need to be observed. The designer defines which of the states of the UML-SM models will be « Observable ».

3.2.2 2nd step: Automatic generation of the RESCO Model. Once the designer finishes the first step, RESCO framework continues with the work. First, it takes the annotated UML-SMs and performs a M2M transformation. As a result, it generates an instrumented model for each of the designed UML-SM. For doing this, we defined some platform-independent transformation rules.

Our approach is based on a platform-independent model instrumentation process. As in [5], our approach uses M2M transformation techniques for creating an instrumented version of the user-defined model supporting introspection, checking and adapting activities at runtime. This support is added by adding reflection and adaptation capability to the model in the (2) runtime package (runtime infrastructure). Thus, without having to instrument the code, we are able to generate applications providing advanced capabilities such as component introspection by themselves.

To formalize our approach, we considered only the computations that occur in actions and conditions attached to transitions.

Figure 5 shows how a transition chain between two states is instrumented in order to provide debugging and observation ability at runtime. The left side of the figure shows the original transition when event *EvA* happens being the software component in *S1*. The right side shows the equivalent version of the transition after model instrumentation. The new model introduces a choice point and a composite state that will get the observed information and share/log it. In our case, this composite state will be shared by all the transitions. We do not have to implement different Observer States for each of the transitions. The (2) runtime package is an application independent solution that provides the infrastructure that enables observing the current information of the states at runtime.

Summarizing, this is the overall behaviour of RESCO-SMs: when an event is sent to the state machine based software component, the dispatcher analyzes the current status and calculates if a transition has to be performed. If the transition is going to be performed, and the *current*, *next* or *parent* state is annotated as *Observable*, the current state information is observed and sent to the externalized checker. Having this observed information at runtime, we can localize bugs analyzing execution traces in model terms.

3.2.3 3rd step: C++ reflective UML-SM based software components generation (CRESCO). In this section we will present the concrete implementation of RESCO approach for C++: CRESCO framework. As we have mentioned, the RESCO metamodel is platform independent.

In order to generate an application with *Observable* software components in terms of model elements at runtime, CRESCO framework includes: (1) M2T transformations of the elements of the design package part of the RESCO metamodel into C++ code by the Acceleo [20] tool, and (2) an implementation in C++ of the runtime infrastructure. In figure 6 an excerpt of the result of the State Machine M2T transformation is shown.

This specific solution addresses embedded and resource limited systems. In this vein, for CRESCO, we performed a M2T transformation to C++ code and we did not use dynamic memory allocation.

3.3 RESCO Software components' Self-Adaptation

In the presented solution, an externalized adaptation system is observing the system behaviour in model terms at runtime. When an error or an unexpected situation is detected by this external

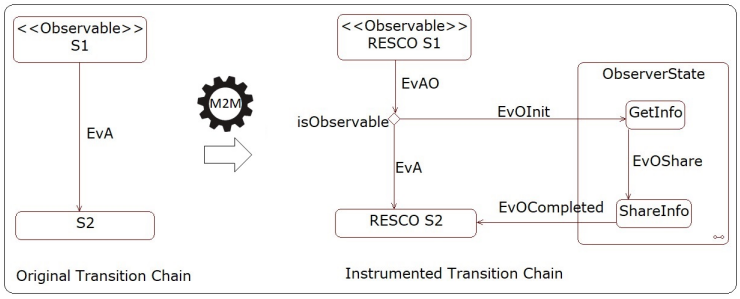

Figure 5: Model Instrumentation: Transformation Rule of the runtime package of RESCO metamodel. Thanks to the (2) runtime package, the same ObserverState State object is used in all the transitions.

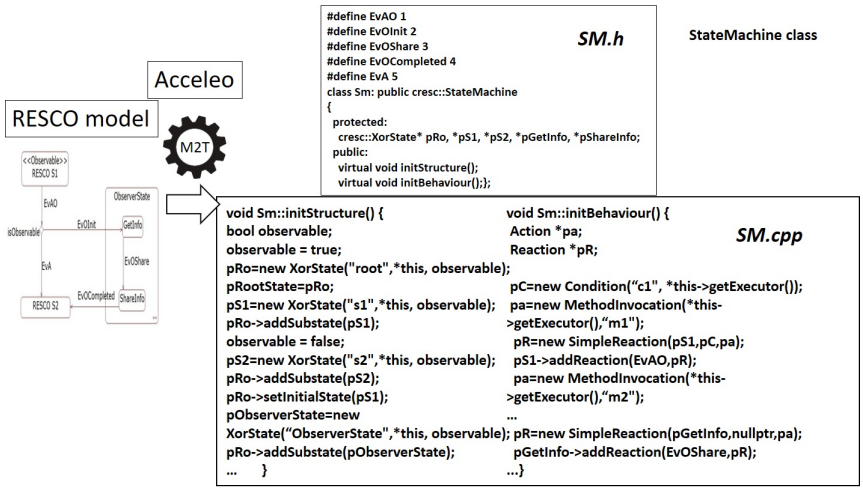

Figure 6: CRESCO: RESCO State Machine M2T Transformation to C++ code

adaptation system it activates the adaptation mechanism sending a safeAdapt event to the affected software component. Once the software component receives this safeAdapt event, the current active UML-SM is sent to Passive mode. Once the system is controlled and all the pending activities of the last transition are processed/stopped, it is sent to the Quiescent mode. At this moment, the safe-mode UML-SM is initialized and sent to the corresponding initial state of the new SM (defined in the first step of the process) and finally this alternative safe UML-SM is Activated.

The next code fragment (listing 1) shows the runtime adaptation infrastructure added to the automatically generated code.

```
void Dispatcher::processEventinError(Event ev)
{ ...
  cresc::State *activeState;
  ...
  this->context->getOwnerSm()->setActive(0);
  this->context->getOwnerSm()->reinit();
  this->context->getOwnerSm()->id=1;
  activeState=this->context->getWorkingState();
  ...
  this->context->getOwnerSm()->setActive(1);
...}
```

Listing 1: Fragment of code managing the CRESCO software component's adaptation ability

4 EMPIRICAL EVALUATION

4.1 Case Study Description

The selected case used for evaluation is an industrial software component that controls a micro-generation device: the Burner controller of the Whispergen commercial device [29]. It covers electrical generation, heat generation as well as micro CHP (Combined Heat and Power). Additionally, we generated 6 synthetic controllers to evaluate the effects on performance and time when we have UML-SMs with different size and complexity levels. We defined the 6 synthetic cases based on the original Burner's controller UML-SM.

For evaluation purposes, the Burner controller shown in figure 7 was implemented in the CRESCO framework, in the SinelaboreRT version 3.7.2.2 [27] tool (specific tool for real time systems), and in the Sparx Systems Enterprise Architecture (EA) version 11 tool [19] (generic tool). The selected Burner controller's normal-mode UML-SM had 13 simple states, 2 composite states, 13 transitions and 13 events. The safe-mode UML-SM had 7 simple states, 2 composite states, 9 transitions and 9 events.

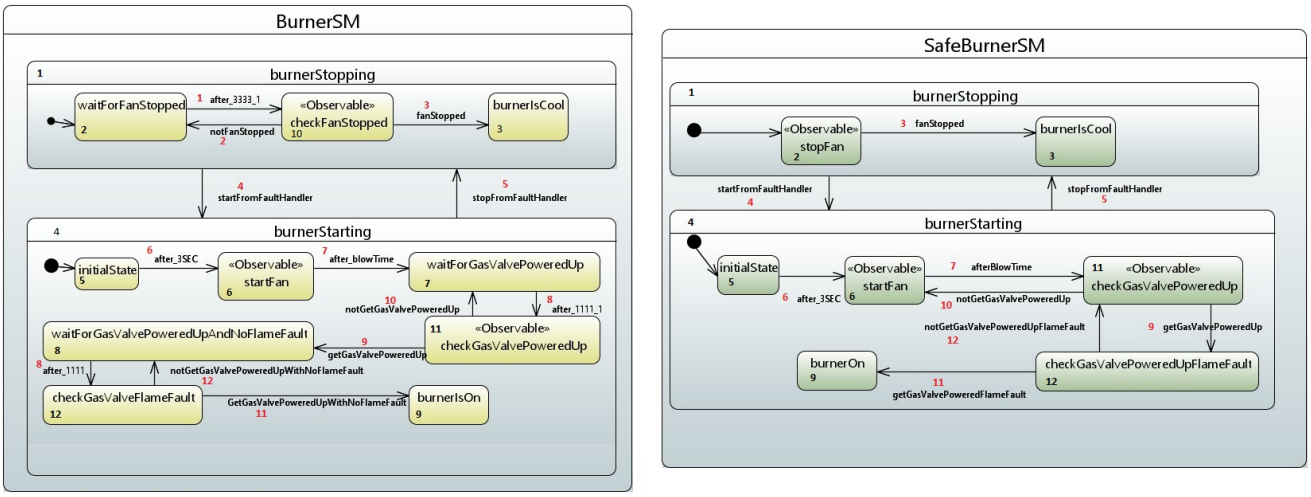

Figure 7: Burner's Normal and Safe state machine models

4.2 Research Questions

One of the objectives of the experiments was to show how it is possible to observe the information of the running system in model terms at runtime (RQ1). Another objective was to demonstrate the runtime verification and adaptation ability of the software components generated by the RESCO framework (RQ2). Finally, the overhead of the solution was measured (RQ3). We have defined the following research questions:

RQ1. Is it possible to obtain the model of the software component by analyzing observed information at runtime?

RQ2. Can this runtime information be employed for runtime verification and adaptation?

RQ3. Is the performance of SW components generated by CRESCO framework as good as the SW components generated by other existing tools (EA v11 and SinelaboreRT v3.7.2.2)?

4.3 Experimental Setup

All the experiments were executed as a standalone application over a Linux virtual machine configured with a 1 Core processor, 2196MB of RAM, 20GB SSD, and running 64-Bit Ubuntu 16.4 LTS. We have used Eclipse IDE for C/C++ Developers version Oxygen.1a Release (4.7.1a) for generating the executable state machines using the code generated by both CRESCO framework, SinelaboreRT (v3.7.2.2) and EA tool (v11).

To analyse RQ1, RQ2 and RQ3, we defined 10 experiments. In order to have more reliable results, each experiment was repeated 1000 times. Table 1 shows the characteristics of each of the experiments. The state machine SM1 is the original Burner Controller. SM2 to SM7 are the synthetic state machines created for testing purposes to perform experiments with different size and complexity level state machines. To that purpose, we added different number of states to the original one: some of them in a flat way and others hierarchically. Thus, we experimented with state machines that have different size and complexity levels.

Considering the works in [9] and [16], to measure the size and complexity of state machines, we used the next metrics: Number of Simple States (NSS- Size metric), Number of Composite States (NCS- Size metric) and Cyclomatic Number of McCabe (Structural Complexity metric) adapted to state machines.

4.4 Results

4.4.1 RQ1 Results. To answer RQ1, first we initialized the controller of SM1, sent to it 10.000 random events and the externalized monitoring and adapter system received and stored the internal status information in model terms of the software component at runtime. Analyzing the runtime observed information, the externalized system was able to represent the structure and behaviour of the software component under study (SCUS). In this experiment, all the states were annotated as *observable*.

Listing 2 shows a fragment of the logged information at runtime. In the figure 7 (original state machine) we can see the interpretation of the received information (numbers) that represent the name of states and events.

```
EvId 4; CurrentState 2;NextState 4; FatherState 1;
EvId 6; CurrentState 5;NextState 6; FatherState 4;
EvId 7; CurrentState 6;NextState 7; FatherState 4;
EvId 8; CurrentState 7;NextState 11;FatherState 4;
```

Listing 2: Fragment of the logged information at runtime

4.4.2 RQ2 Results. Regarding RQ2, in order to answer this question, we used the externalized runtime verification and adaptation system. This system was developed inspired by the solution defined in [3] (the CoMA runtime monitor) and it compares the current running SCUS's logged information at runtime with the correct information of the SCUS it (externalized system) has previously configured. This correct information is based on the designed UML-SM model elements. We used trace-violation checking techniques.

We evaluated the failsafe detection by fault injection campaigns modifying the source code with the libfiu tool [24]. In this first evaluation we measured the detected number of faults but we did not activate the adaptation process.

We injected 6 faults using the libfiu tool emulating the effect of random hardware, software and unanticipated environmental

Table 1: Experiments Setup

State Machine	Applied to RQ	Observ. %	NSS	NCS	McCabe	Fault Injection
SM1	RQ1, RQ3	100	13	2	13	No
SM1	RQ2	100	13	2	13	Yes
SM1	RQ3	50	13	2	13	No
SM1	RQ3	0	13	2	13	No
SM2	RQ3	0	25	4	26	No
SM3	RQ3	0	49	4	52	No
SM4	RQ3	0	113	9	117	No
SM5	RQ3	0	25	5	26	No
SM6	RQ3	0	49	11	52	No
SM7	RQ3	0	113	26	117	No

faults. In order to emulate the random hardware and software faults we injected 2 faults using the *fiu_enable_random()* (Random) option of the libfiu library. This option enables the point of failure in a non-deterministic way, which will fail with the given probability.

Concerning the remaining software faults, we emulated them inserting another 4 faults using the *fiu_enable()* (Unconditional) option of the libfiu library. This option enables the point of failure in an unconditional way, so it always fails.

As a result, the externalized runtime verification system detected all the faults injected in the SCUS before the transition was performed and, thus, the runtime verification applicability of the solution was demonstrated.

Listing 3 shows two of the faults that the externalized runtime verification system detected at runtime.

```
EvId 8; CurrentState 7;NextState 8; FatherState 4;
EvId 8; CurrentState 8;NextState 7; FatherState 4;
```

Listing 3: Fragment of the logged information at runtime

Both of the faults, were injected in the fault injection campaigns and, if we check the UML-SM of the SCUS, we can conclude that these transitions are faulty because they are not represented in the UML-SM model.

In the second part of the experiment, we activated the adaptation process and reinitialized the system. After that, we started injecting the same faults as in the first experiments but in this case, when the first error was detected, the original state machine was deactivated and the alternative safe-mode state machine was activated. Therefore, the behaviour of the controller was changed automatically at runtime. The externalized runtime checker continued logging and analyzing the runtime information. Once the experiment was finished we analyzed the logged information and concluded that after the runtime error detection the safe-mode state machine started working.

Listing 4 shows the logs of the safe-mode state machine. Comparing with the state machine of the figure 2 we can conclude that its behaviour was correct and system failure avoided.

```
EvId 3; CurrentState 2;NextState 3; FatherState 1;
EvId 7; CurrentState 6;NextState 11; FatherState 4;
```

Listing 4: Fragment of the logged information at runtime

4.4.3 RQ3 Results. In RQ3, performance was evaluated in terms of execution time (milliseconds) and percentage of CPU usage. To

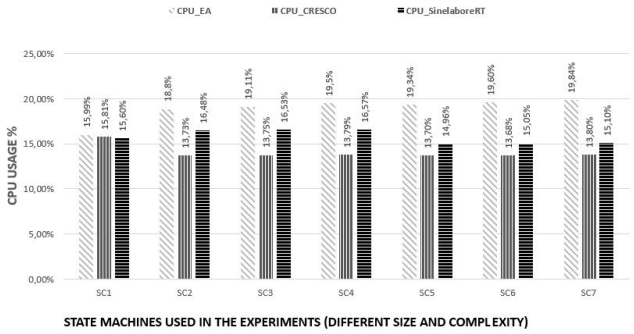

Figure 8: RQ3 results: CPU usage % results for SinelaboreRT, EA and CRESCO tools (when observability level 0%).

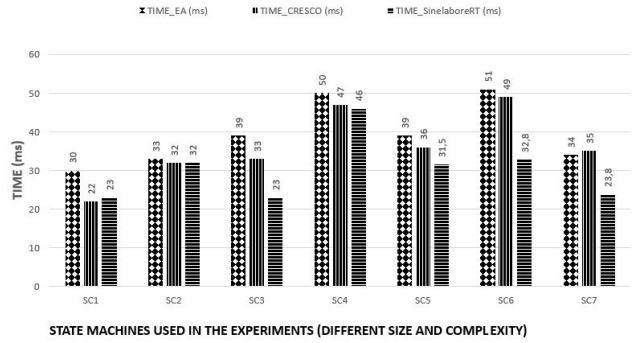

Figure 9: RQ3 results: Timing results for SinelaboreRT, EA and CRESCO tools (when observability level 0%).

measure the execution time, we used the *gettimeofday* instruction. This instruction was launched at the beginning and the end of the execution.

Figures 8 and 9 illustrate the results for RQ3. In these experiments we used 1.000 input events and the observability level of the states in RESCO was 0%. We also launched the original state machine (SM1) generated by CRESCO when 50% and 100% of the states were observed. When the observability level was 50%, the time response in milliseconds was 272, and when all the states were observed the result was 411. In all the experiments we used 1.000 input events.

In terms of time response, SinelaboreRT is the tool (specific for real time systems) that achieves the best results. However,

CRESCO's results are similar or even better when the state machines' complexity is low. Concerning the CPU usage percentage, all the results are similar and this is because the experiments were performed in very similar situations and with the same size and complexity levels of the software controllers. So, the time was the parameter that was affected by the different experiments scenarios. If we consider the synthetic UML-SM (SM2 to SM7) for CRESCO and EA tools, when the size and complexity of the state machine increase, the performance of the tools is affected negatively and it decreases: the execution time increases although the percentage of the CPU resource used is only slightly affected. As for SinelaboreRT, when the complexity and size of the state machines is increased, the execution time also increases but to a lesser extent. We have to add that the SinelaboreRT tool is specific for designing and developing real-time systems and Enterprise Architecture tool is a generic tool for different type of systems.

4.5 Discussion

The results of the RQ1 show that the solution that we are presenting is able to provide the internal status information in model terms. This runtime information can be employed for various purposes, e.g., fault detection and localization, runtime adaptation/reconfiguration, runtime enforcement and observation of software runtime behavior.

In this work, we performed experiments for the observation of the software component runtime behaviour and the fault detection, localization and adaptation. The results of RQ2 were positive and the solution, in addition to detecting the fault, indicated in which state and transition of the SCUS the fault was happened and if the adaptation ability was activated the system was adapted automatically to a predefined safe-mode state machine at runtime. When the observability level of the states of UML-SMs of the SCUS was 100%, CRESCO detected the faults and executed the adaptation before the next state transition was performed. Additionally, the solution allows the runtime externalized checker and adapter to ask the SCUS about its internal status at any time. Thus, the robustness of the system is increased.

One of the strong points of the presented approach is that, in this case, the developer does not need to be bothered about these issues. The behaviour, adaptation and the introspection ability are developed orthogonally. The developer only has to focus the efforts on the behaviour of the SCUS, and there is no need to instrument code manually, thereby avoiding new points of introduction of faults.

The general objective of RQ3 was to evaluate the CRESCO framework's performance compared with commercial tools. We compared the performance of the CRESCO framework (0% observability level) with the SinelaboreRT v3.7.2.2 and EA v11 tool results. We considered seven different state machines of different size and complexity. The results show that SinelaboreRT was the tool with best time response results but CRESCO was the next best. We have to consider that we are adding some more infrastructure and logic to enable the solution to have introspection and adaptation ability at runtime. This will be one of the main reasons for achieving worse results than SinelaboreRT. The results with different observability level show that the performance of the solution decreases when increasing the percentage of states observed at runtime.

4.6 Threats to validity

This section identifies threats that could invalidate the empirical evaluation performed. An external validity threat could arise due to considering only one industrial controller. We have configured different state machines for the different experiments but all of them were based on the initial Burner controller. Different state machine controllers might lead to different results. However, this first experiment was valid to check the correctness of the design and development of the solution since the main objectives were: (1) to show how the software components generated by CRESCO framework were able to provide internal status information in model terms at runtime; (2) to demonstrate the applicability of this information for runtime verification and adaptation; and (3) to measure the CRESCO software components' performance. The experiment employs 7 different state machines, 4 different observability levels and fault injection campaigns. A conclusion validity threat could possibly arise due to the way the execution time and percentage of the CPU usage was measured. To mitigate this threat, each execution is repeated 1000 times and results are statistically tested.

5 RELATED WORK

5.1 Runtime Software Adaptation

Gomaa et al. propose an approach for dynamic software adaptation using runtime models of the software architecture. In [17] they describe software adaptation patterns that consist of interaction models and state machine models. In this solution, they describe the pattern to perform the adaptation: a Genelarized Adaptation State Machine. This pattern has been used as a basis in the presented work to define how to execute the adaptation in UML-SM based software components when an error or an unexpected situation is detected.

In [15] Garlan et al. describe a model-based adaptation for self-healing system. In this work they identified the challenging research problems and defined an architecture of an adaptation framework. They also implemented an specific solution in Acme language in the control side.

The architecture and behaviour of our solution is based in the aforementioned works but our approach is more holistic starting from the software components design to runtime adaptation ability.

5.2 Tracing UML State Machines

Mazak et al. propose an execution-based model profiling as a continuous process to improve prescriptive models at design-time through runtime information in [26]. In order to have runtime information and logs, in model terms, they defined a observation language which determines the runtime changes to be logged. The code generator provides the appropriate logging line to that. Comparing with the presented solution, Mazak's solution instruments the code and not the model. In this sense, our solution is more general, less dependent of the specifics of code generators or deployment configurations.

In the same vein, in [10] Das et al. present their solution based on instrumenting the code (not the model) to monitor real-time

Table 2: Tools and methods for code generation

Tools	UML-SM Code Generators	Model at Runtime	Addressed Systems
MOCAS [6]	Java	Yes (automatically instrumented)	Not resource limited systems
EA [19]	Java, C++,...	No	Not resource limited systems
SinelaboreRT [27]	C++/Python/...	No	Embedded Systems
Quantum (QP/C++) [30]	C++	No	Embedded Systems
Yakindu [22]	C++/C/Java	No	Embedded Systems
Papyrus-RT [13]	C++	No	Embedded Systems
IBM Raphsody [21]	C++, Java,...	No	Embedded Systems
Pham [28]	C++	No	Embedded Systems
CRESCO	C++	Yes (automatically instrumented)	Embedded Systems

embedded systems at runtime. They combine the use of MDE, runtime monitoring, and animation for the development and analysis of components in real-time embedded systems.

The solution presented in [2] defines a textual language for trace specification of state machines. As the aforementioned works, this approach is also based on code instrumentation. They define different commands to trace different specifications at runtime but the solution does not provide information at model level and the logged information is not related to model elements: it is not a model centered solution.

In [31] Saadatmand et al. propose a solution for runtime verification of state machines based on model-based testing. Their solution is not generating automatically the final code and, as the previous solutions, they instrument the code manually to have the software components information in model terms at runtime.

In [5] they present a platform-independent model-level debugger. The solution is focused on real-time embedded systems. This approach, like ours, relies on model transformation to instrument the model to be debugged. Both approaches are in the same direction but the implementation and the model-to-model transformation rules are different. The main difference with our solution is that this one [5] is applicable to models expressed in UML for Real-Time (UML-RT). Our approach is applicable to models expressed in UML-SM. Additionally, in their solution, they add new objects to debug or trace the execution in all the transition chains. In our case, the same object that is in the (2) runtime package, the ObjectObserver, is reused in all the transitions. This way, the number of objects in the model is independent of the size and number of transitions of the state machine.

5.3 Reflective UML State Machines

In [6], [8] and [14] they propose a component model that carries models at runtime, focusing on UML state machines. So, like our, the frameworks of [6] and [14] also provide a runtime state-based component model. The main difference regarding [6] and [14] is that our solution can be oriented to different type of systems. In the last step of the model-driven workflow, we can decide which M2T transformation rules to use. In the presented work, we implemented the transformation to C++ language. Thus, the specific solution of the presented work is oriented to embedded software.

Moreover, CRESCO is an extension of the framework proposed in [14] and, in addition to code-generation capabilities, the presented model-driven approach provides the RESCO metamodel. The latter is able to represent RESCO models that are platform independent.

Additionally, the RESCO models are able to add different observability levels by the observability profile. This last characteristic enables software components generated by this model-driven approach to have runtime verification and adaptation capability as demonstrated in the presented work. For doing that, the solution also provides an externalized runtime checker and adapter.

5.4 State Machines embedded in code

In [7] authors show how state machine logic can be embedded in object-oriented code. A runtime environment extracts the state machine information at runtime and executes it. In this way the runtime environment provides control of the application, enables the logging of its workflow and debugging of events. In the implementation, Java code is connected to the state charts by means of special classes, interfaces and annotations. Rather than being created and manipulated at design time, the statemachine model is extracted from code at runtime.

This approach inverts the traditional direction of model-to-code generators. There is no model that is manipulated at design time and transformed into source code from time to time. Instead there is a permanent model representation in the source code, which is extracted for analysis within modelling tools from time to time. On the one hand, this eliminates any effort to maintain and merge different abstraction layers. On the other hand, the chosen approach is not independent from programming languages and execution environments, in this case JAVA, as it is when using other model-driven development technologies.

The solution presented in this work also maintains the model at source code level and in addition it is platform independent.

5.5 Automatic UML-SM to Code Generation Tools

Table 2 shows the different solutions to generate code and their characteristics. As we can see in table 2, there is a lack of solutions able to generate UML-SM based software components that provide runtime information in model terms and that address embedded systems.

MOCAS is the only one that provides the model of the software component at runtime but it is not a solution for embedded systems and it does not provide the externalized system verification and adaptation module. The rest of the tools, such as EA or SinelaboreRT do not provide the model at runtime. Most of them are solutions for embedded systems but we need to instrument the code manually

if we want to have internal information of the UML-SMs in model terms at runtime.

6 CONCLUSIONS

The paper first presents a model-driven approach to automatically generate software components based on UML-SMs with the ability to provide their internal status in terms of model elements and adapt their behaviour at runtime. We defined a platform independent metamodel called RESCO to represent the model of these software components based on UML-SM models annotated by the *observability* profile. Finally, the CRESCO framework, a concrete implementation of the RESCO approach for embedded and resource limited systems in C++ code, was presented.

In order to demonstrate the characteristics of the presented work, we performed an empirical evaluation of how a software component generated by CRESCO was able to represent the software components' UML-SM model elements at runtime. Additionally, a specific application to detect faults and adapt the behaviour of the software component when needed at runtime (runtime verification and adaptation) was presented. The results of the experiments showed that the solution is able to detect the errors and adapt the behaviour if the errors are in observed states at runtime. Furthermore, the solution also enables interrogating the internal status of the software component in model terms at runtime if required.

We empirically evaluated the performance of the framework (in terms of execution time and percentage of CPU usage) using state machines of different size and complexity. Some experiments were also implemented using different commercial tools (EAv11 and SinelaboreRT) in order to compare their results with those of the CRESCO framework.

In the future, we would like to expand the empirical evaluation using other real industrial cases and projects. More commercial tools will also be compared.

As another future line, as the solution enables having adaptation and intercession ability at runtime, we will integrate approaches for runtime model adaptation in uncertain environments generating new models and behaviour to be adapted also at runtime. Thus, the system will be able to redesign the behaviour and not to use a predefined safe-mode. In this vein, inspired by the work in [26], one of the future research lines is to combine the Process Mining (PM) techniques with runtime models.

ACKNOWLEDGMENT

The project has been developed by the Embedded System Group of MGEP and supported by the Department of Education, Universities and Research of the Basque Government under the projects Ikerketa Taldeak (Grupo de Sistemas Embebidos) and TEKINTZE (Elkartek 2018) and the European H2020 research and innovation programme, ECSEL Joint Undertaking, and National Funding Authorities from 19 involved countries under the project Productive 4.0 with grant agreement no. GAP-737459 - 999978918.

REFERENCES

[1] Uwe Abmann, Sebastian GÃűtz, Jean-Marc JÃľzÃľquel, Brice Morin, and Mario Trapp. 2014. A reference architecture and roadmap for Models@run.time Systems. In *Models@run.time*, Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran (Eds.). Springer International Publishing, Cham, 1–18.

[2] Hamoud Aljamaan, Timothy C Lethbridge, and Miguel A Garzón. 2015. MOTL: a textual language for trace specification of state machines and associations. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 101–110.

[3] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2014. Offline Model-based Testing and Runtime Monitoring of the Sensor Voting Module. *Communications in Computer and Information Science* (2014).

[4] ATL. 2018. ATL Transformation Language. http://www.eclipse.org/atl/

[5] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. 2017. Model-level, platform-independent debugging in the context of the model-driven development of real-time systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 419–430.

[6] C. Ballagny, N. Hameurlain, and F. Barbier. 2009. MOCAS: A State-Based Component Model for Self-Adaptation. In *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. 206–215. https://doi.org/10.1109/SASO.2009.11

[7] Moritz Balz, Michael Striewe, and Michael Goedicke. 2008. Embedding State Machine Models in Object-Oriented Source Code. In *3rd Int. Workshop on Models@run.time*.

[8] Franck Barbier. 2008. Supporting the UML State Machine Diagrams at Runtime. In *Model Driven Architecture – Foundations and Applications*, Ina Schieferdecker and Alan Hartman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 338–348.

[9] Mokhtar Beldjehem. 2013. A Granular Hierarchical Multiview Metrics Suite for Statecharts Quality. *Advances in Software Engineering* Volume 2013 (2013), 13.

[10] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. 2016. Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 36–43.

[11] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni Vincentelli. 2012. Modeling Cyber-Physical Systems, IEEE (Ed.), Vol. Special issue on CPS. IEEE, 13–28.

[12] Rohit Dhall. 2017. Designing Graceful Degradation in Software Systems. In *Proceedings of the Second International Conference on Research in Intelligent and Computing in Engineering*, Vol. 10. 171–179. https://doi.org/10.15439/2017R15

[13] eclipse. 2018. Papyrus-RT. https://www.eclipse.org/papyrus-rt/

[14] X. Elkorobarrutia, M. Muxika, G. Sagardui, F. Barbier, and X. Aretxandieta. 2008. A Framework for Statechart Based Component Reconfiguration. In *Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems (ease 2008)*. 37–45. https://doi.org/10.1109/EASe.2008.11

[15] David Garlan and Bradley Schmerl. 2002. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*. ACM, 27–32.

[16] Marcela Genero, David Miranda, and Mario Piattini. 2003. Defining metrics for UML statechart diagrams in a methodological way. In *International Conference on Conceptual Modeling*. Springer, 118–128.

[17] Hassan Gomaa, Emad Albassam, and Daniel A Menascé. 2017. Run-time Software Architectural Models for Adaptation, Recovery and Evolution.. In *MODELS (Satellite Events)*. 193–200.

[18] https://eclipse.org/papyrus/. [n. d.]. *Papyrus*. Technical Report.

[19] http://sparxsystems.com/products/ea/. 2015. *Enterprise Architecture v11*. Technical Report. Sparx Systems.

[20] https://www.eclipse.org/acceleo/. 2016. *Acceleo*. Technical Report.

[21] IBM. [n. d.]. IBM-Rhapsody. https://www.ibm.com/us-en/marketplace/rational-rhapsody

[22] itemis. 2018. Yakindu Statechart Tools. https://www.itemis.com/en/yakindu/state-machine/

[23] Mark James, Paul Springer, and Hans Zima. 2010. Adaptive fault tolerance for many-core based space-borne computing. In *European Conference on Parallel Processing*. Springer, 260–274.

[24] Libfiu. [n. d.]. Libfiu: faultinjection in userspace. https://blitiri.com.ar/p/libfiu

[25] Pattie Maes. 1987. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, Vol. 22. ACM, 147–155.

[26] Alexandra Mazak, Manuel Wimmer, and Polina Patsuk-Bösch. 2016. Execution-Based Model Profiling. In *International Symposium on Data-Driven Process Discovery and Analysis*. Springer, 37–52.

[27] Peter Mueller. 2018. *sinelaboreRT*. Technical Report.

[28] Van Cam Pham, Ansgar Radermacher, SÃľbastien GÃľrard, and Shuai Li. 2017. Complete Code Generation from UML State Machine. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*. INSTICC, SciTePress, 208–219. https://doi.org/10.5220/0006274502080219

[29] Prashant Kaliram Pradip. 2016. *Commissioning and Performance Analysis of WhisperGen Stirling Engine*. Technical Report. University of Windsor.

[30] Quantum. 2018. Quantum Platform in C++. http://www.state-machine.com/qpcpp/

[31] Mehrdad Saadatmand, Detlef Scholle, Cheuk Wing Leung, Sebastian Ullström, and Joanna Fredriksson Larsson. 2014. Runtime verification of state machines and defect localization applying model-based testing. In *Proceedings of the WICSA 2014 Companion Volume*. ACM, 6.