# Increasing Dependability in Safety Critical CPSs Using Reflective Statecharts

Miren Illarramendi(✉), Leire Etxeberria, Xabier Elkorobarrutia, and Goiuria Sagardui

Embedded Systems Research Group,
Mondragon Goi Eskola Politeknikoa (MGEP),
Mondragón, Spain
{millarramendi,letxeberria,xelkorobarrutia,gsagardui}@mondragon.edu
http://www.mondragon.edu/eps

**Abstract.** Dependability is crucial in Safety Critical Cyber Physical Systems (CPS). In spite of the research carried out in recent years, implementation and certification of such systems remain costly and time consuming. In this paper, a framework for Statecharts based SW component development is presented. This framework called CRESC (**C**++ **RE**flective **S**tate**C**harts), in addition to assisting in transforming a Statechart model to code, uses reflection to make the model available at Run Time. Thus, the SW components can be monitored at Run Time in terms of model elements. Our framework helps the developer separate monitoring from functionality. Any monitoring strategy needed to increase dependability can be added independently from the functional part. The framework was implemented in C++ because this programming language, together with the Statechart formalism constitute widely used choices for the Safety Critical CPS domain.

**Keywords:** Fault Tolerance · Monitoring · Statecharts · Safety-critical embedded systems · Cyber Physical Systems · Reflection · Introspection

## 1 Introduction

Cyber-Physical Systems (CPSs) integrate digital cyber computations with physical processes. These CPSs are composed of embedded computers and networks that monitor and control physical processes with sensors and actuators [1]. A Safety Critical Cyber-Physical System (SCCPS) is a system whose failure or malfunction may result in very severe consequences.

CPSs are applied in several domains including aerospace, energy, automotive, railway or health-care, which are considered safety critical domains. In comparison with CPSs, SCCPSs are more complex in terms of functionality, integration and networking interoperability, reliance on software, and the number of non-functional constraints (e.g., dependability, robustness, scalability, safety).

Functional Safety is one of the key properties of SCCPS. Safety is aimed at protecting the systems from accidental failures in order to avoid hazards. Many


S. Tonetta et al. (Eds.): SAFECOMP 2017 Workshops, LNCS 10489, pp. 114–126, 2017.
DOI: 10.1007/978-3-319-66284-8_11

safety critical CPS, are required to pass a certification process and must provide evidence that they have been developed according to the domain functional safety standards [2–6].

The scope, complexity, and pervasiveness of SCCPSs continue to increase dramatically. As the software of today assumes more of the responsibility of providing functionality and control in systems, it has became more complex and more significant to the overall system performance and dependability. Given the current state of the art, fewer development faults are committed because of the use of best practices and better tools, but not all are prevented.

Verification and validation techniques applied during development could help to reduce the errors introduced in the systems but, if we want to increase the dependability of the systems, there is still a need for Run Time Checking. There are Fault Detection and Fault Tolerance techniques that assist in this task but they are not easy to implement and require much effort as Laprie et al. stated in [7].

In a process of developing SW components, designers add the requirements to their software models in the design phase. Normally, in the next phases, most of this information is lost. Fault Detection strategies need these requirements and specification information. In the approach presented in this paper the specifications and requirements added in the design phase, when modelling the system, are kept at Run Time. The SW components generated by the CRESC framework can use the Fault Detection mechanisms to check the current internal state of the controller by means of reflection.

The contribution of this research is to introduce the CRESC framework that generates SW components based on Reflective Statecharts in C++ programming language. This CRESC framework is easy to use and the generated SW control components have the ability of introspection and adaptation.

The solution separates the functionality and safety aspects of the system. We use a combination of classic mechanisms (such as Reflection and Statecharts). However, from that combination we have created a new efficient tool to develop SW control components for SCCPSs.

In Sect. 2 of the paper the Technical Background is presented. In Sect. 3 we present the CRESC Framework. After that, in Sect. 4 a Toy Example and the Use Case for Productive 4.0 project are shown. The Conclusion of the developed Framework is presented in Sect. 5 and finally, the Future Lines section closes the paper.

## 2 Technical Background

In the domain of CPSs, there are different techniques to design and develop robust systems. The main aim of this research is to increase the dependability of SCCPSs.

The term dependability has been studied by different researchers and one definition by Laprie and Kanoun [8] is “trustworthiness of a computer system such

that reliance can justifiably be placed on the service it delivers". Laprie classifies dependability in terms of threats, attributes and means. The means described by Laprie are based on fault prevention, tolerance, removal or forecasting.

### 2.1 Fault Classification and Fault Detection

Faults can be classified in many ways and different type of faults have their particular characteristics. Avizienis et al. after an extensive analysis presented a classification in [9].

In our research, we considered the following faults:

- Controller faults. These faults may be due to:
  - SW design and development faults: committed either during the system initial design, from requirements specification to implementation, or during subsequent modifications.
  - HW malfunction faults: adverse physical phenomena either internal (such as short circuits, open circuits and threshold changes) or external (such as temperature and electromagnetic perturbations).
- Environmental faults: these faults occur during the operation phase, therefore they are also called operational faults. They are caused by elements of the environment that interact with the system (such as sensors, actuators and communication systems).

Fault Detection is one of the initial and necessary steps to prevent the failure of the system. Even if other elements of a system stop a failure by using other techniques, it is important to detect and remove faults to prevent the exhaustion of the fault tolerance resources of a system.

The faults we have classified could be detected by HW Redundancy (HW and some SW faults) [10], SW Diversity (SW faults) [10] and/or Run-Time Monitoring (all the faults but specially Environmental faults) of the SW control components [11].

### 2.2 Run Time Verification

Although a model based checking approach in the design and development phases can give enough confidence that the implementation is correct, for SCCPSs we need to continue checking their behaviour respect to its specification also after the deployment.

Run Time verification is the study of how to design artifacts for monitoring and analyzing program executions. The information extracted from the running systems is used to asses satisfaction or violation of specified properties. Those properties are expressed formally using different notations such as finite state machines, regular expressions and linear temporal logic formulas [12].

To monitor a program, we need to log the events and the controller status while it is running. Program instrumentation consist of the addition of code for such information gathering. Different types of program instrumentation could be used to implement the error detection mechanism. These are some examples of program instrumentation: Hooks, Design by Contract approach and Assertions.

There are tools that automatically add program instrumentation by transforming source code as CIL (C) [13] or byte/object code as Valgrind (C) [14] or BCEL (Java) [15].

Some middleware or OS can also offer basic services to implement an error detection mechanism. These services use the infrastructure that is behind the application level (middleware and OS). However, it is not a platform independent solution.

### 2.3 Reflective Principle

Reflection makes the model used in the design phase available at Run Time. Computational reflection [16] is an approach that:

- Helps in separating the application and dependability mechanisms to reduce complexity.
- Adds the introspection capacity to increase dependability.

As an example of research in Reflection carried out in recent years, in [17,18] Lu et al. developed techniques and mechanisms to detect errors and adapt the system to change to a non error mode at Run Time. Their approach is based on multi-layered architectures using the AUTOSAR [19] standard middleware and specific OS services.

### 2.4 Statecharts Formalism and Development Tools

UML statechart formalism allows constructing a state-based model of the controller, describing both its internal behavior and its reaction to external events.

Ferreira and Rubira [20] created the Reflective State Pattern for finite state-machine aiming at reflecting the state structure of a component and changing its behaviour at Run Time for tolerating environmental faults. However, to the basis of our knowledge, it is Barbier [21] who first created a framework for Statechart based components that implicitly supported introspection of a component at Run Time. Based on this work, Elkorobarrutia et al. [22] defined a framework for Java that supports Run Time modification of the behaviour of a Statechart-based software component. Elkorobarrutia's solution does not consider real time constraints nor the resource limitation of the execution environments in embedded and real time systems.

There are a lot of patterns and proposals for transforming statecharts to code but as far as we know none of them is well suited for our purposes. As an example there is a framework, the Boost Statechart library [23], able to transform UML Statecharts to executable C++ code and viceversa, but they are not aimed at creating reflective code. Another drawback is that it makes an extensive use of C++ templates and it becomes impractical for large sized Statecharts.

Finally, there are many commercial tools that transform Statecharts specification to code, however this is their only aim. In addition, the transformation rules are quite tool and version specific. Therefore, they are not suitable for adding Run Time introspection.

## 3 Design and Development of the CRESC Framework

### 3.1 Selected Technology

CRESC was developed as a Framework that generates components based on Reflective Statecharts. One of the reasons for this decision was that Statecharts are accepted by the functional safety standards due to its simplicity and to the fact that they constitute a formalism widely used in SCCPSs [24–26].

Specifically we decided to use Reflective Statecharts because this way we are able to introspect SW components in term of model elements and if necessary adapt them at Run Time. Introspection and adaptability, provide the means of adding a Fault Tolerance infrastructure to SW Control Components. Additionally, the use of reflection reduces complexity as Fabre affirms in [16].

As in SCCPS domain usually they are real time and resource limited systems, we decided to implement the framework in C++. This programming language is more suitable for real time and resource limited systems than languages or execution platforms as Java. Additionally, the majority of the Functional Safety standards accept it (a subversion of C++, MISRA C++ [27]).

### 3.2 The Reflective Framework and Error Detection Mechanism

In the next paragraphs we are going to show the main elements of the CRESC framework. We can not explain all the details due to the space limitations.

In order to create a reflective structure for software components, first we had to define which elements of those components we want to reflect. We considered previous work developed by Elkorobarrutia [28]. This work was developed in Java and it was not thought to be used in CPSs and real time execution platforms.

First, we divided the design of the framework into two important parts: the *controller* part and the *executor* part.

In the former, we define the behaviour of the controller implemented by Statecharts. This part is the one that specifies and reflects the statechart model.

The second part is the one that executes the actions and conditions specified by the controller. In Fig. 1 we can see the relationship between the two parts.

To implement the *Controller*, the statechart model of the application is transformed to an object structure. These objects are representing concepts as states, transitions and actions of the statecharts. This object structure is the element that reflects the statechart model. Any change in this structure implies changes in the model and vice versa. Thus, our code reflects the application model.

This reflection enables us to query the status of the component at Run Time. Thus, the framework allows adding fault detection, fault tolerance and adaptability mechanisms to the SW Control components.

The framework needs some extra elements to manage the Run Time information. For that issue, the State Machine Global Repository object was developed. This object keeps Run Time information such as active states and the event currently being processed.

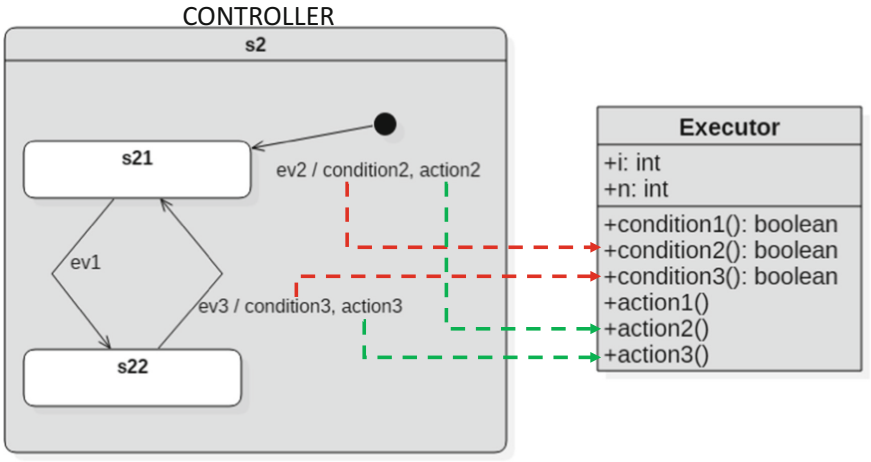


**Fig. 1.** Controller part and its references to the Executor

The Dispatcher is the object that directs the execution of our SW components. When an event is launched, it is stored in the Buffer object. The Dispatcher gets the event from the Buffer and it checks the current status of the application. Then, the Controller orders what to do (i.e. transition to another state performing specific actions) and the Executor part executes the methods (controller's actions) that the controller orders. Figure 2 sets out the whole picture.

Based on introspection ability of the CRESC Framework, internal error detection can be added to the controllers. To this end and based on the work carried out by Lu [29], software *hooks* were used in the CRESC framework. The hooks were added in the entry and exit actions of each of the states. These hooks will log information of the current status provided by the Global Repository and this logged information will be structured using the UML statecharts notation. Thus, an Error Detection mechanism will use this information to check the correctness of the monitored system.

So we can add introspection ability in any of the objects structure and it is also possible to adapt the controller. Once an early error is detected, and before the failure is generated, the Error Recovery mechanism will be started. Depending on the safety properties of the use case and the degradation modes defined for the current application, the Error Recovery Mechanism will initiate the adaptation process to the defined degradation mode at Run Time.

As we are working in the Safety Critical CPS domain, these degradation modes have to be designed previously and the adaptation in this case must be a controlled one.

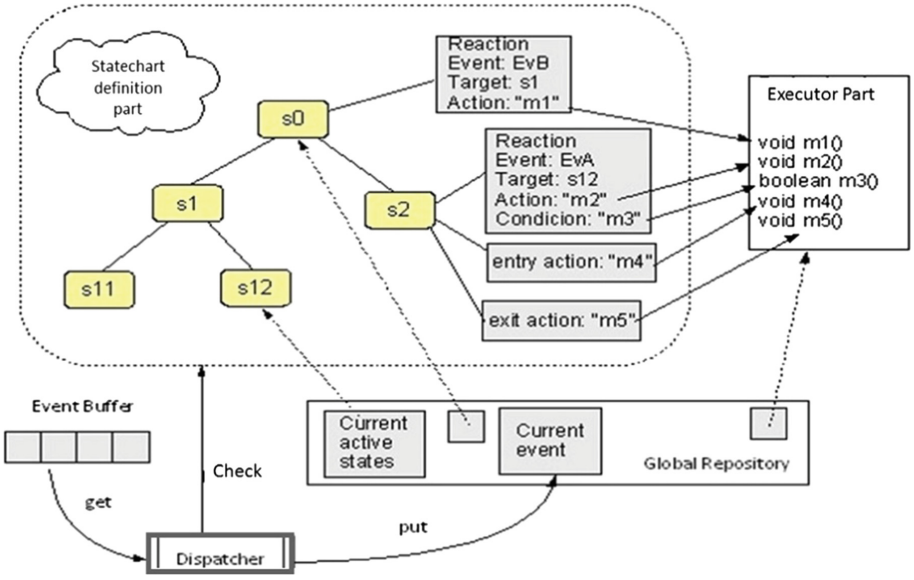


**Fig. 2.** Framework infrastructure

One of the benefits of the CRESC Framework is its ability to detect controller internal errors in early phases before they are transformed to erroneous output control signals.

## 4 Toy Example and Productive 4.0 Use Case

In this section the development of a SW Control Component that controls a distributed elevator is presented. As shown in Fig. 3 the elevator moves the load up and down by two synchronized engines. The details of this toy example were defined in [30]. Each of the synchronized subsystems is composed of an engine and different sensors: top and bottom detectors and a shaft rotation sensor that is used to infer position and speed. The elevator has two movements: up and down.

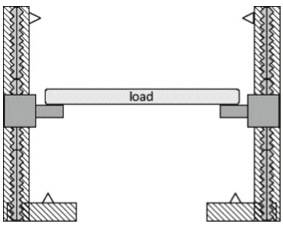


**Fig. 3.** Elevator toy example

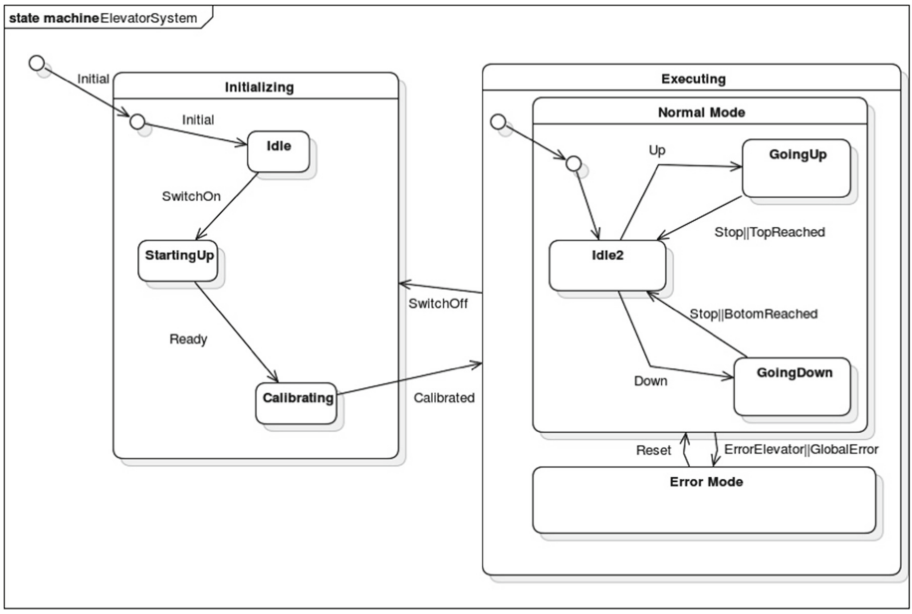


**Fig. 4.** Toy example statechart

As shown in the Fig. 4, the system starts in the Idle state and once the SwitchOn event is detected, it goes to the StartingUp state. Here, the controller checks all the elevators and if they are ready a new transition is performed. Next, the system goes to the Calibrating state. In this state, the controller performs the calibration action of sensors and the system is ready to work. At this moment, the initializing phase is finished and the system enters the Executing state.

The Executing state has two substates: Normal and Error Mode. The default substate is the Normal Mode. Here, there are three substates: Idle (default state), Going Up and Going Down.

The system is in the Idle state until the user sends a command (Up or Down). Once an Up or Down event is detected, a transition is performed to Going Up or Going Down state.

- If the activated state is Going Up, the system does not change until Stop, TopPositionReached or an Error event is detected.
- If the activated state is Going Down, the system does not change until Stop, BottomPositionReached or an Error event is detected.

In Normal Mode, in any of the substates, if an Error event is detected, the system performs a transition to the Error Mode. In this state, once the system is restored/reset, a Reset event is launched and the system goes to the default state, the Normal Mode (Idle).

We developed and tested the case study in a Linux machine with Ubuntu 14.04 LTS and our development environment was eClipse CDT [31].

### 4.1 Development Process of the Toy Example Using the CRESC Framework

In this section we show how the design and development of the described toy example was carried out from the developer role.

First, at the design phase, the SW designer has to model the behaviour of the controller by UML statecharts (definition of the rootState, subStates, events, actions, conditions and transition) using an eClipse tool called Papyrus Modeling Environment [32]. This statechart model is translated to text generating automatically the use case specific code of the CRESC framework.

Next, the CRESC framework creates the SW component automatically adding reflectivity to the modelled controller.

In the Fig. 5 we can see the steps to follow in the main function of the CRESC framework.

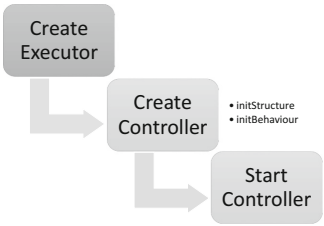


**Fig. 5.** CRESC: main function steps

In the first step, the developer only had to define the name of the Executor in order to identify it. The framework allows having more than one Executor (for example to use as Error Recovery Mechanisms to adapt the behaviour of the system when an error is detected). This information could be extracted from the model automatically.

In the second step, when creating the controller, the developer had to define the structure of the Controller (ToyExampleSM) using the initStructure function and the behaviour of the system by the initBehaviour function. As mentioned before, this step is carried out automatically extracting the model information.

The following extract of code was created automatically with the model to text transformation. In this case, the initStructure and initBehaviour functions were filled with the toy example specific information:

**Listing 1.1.** initStructure

```
states="idle";
idle=new
    control::XorState(states,0);
root->addState(idle);
states="StartingUp";
startUp=new
    control::XorState(states,0);
root->addState(startingUp);
```

**Listing 1.2.** initBehaviour

```
methName="SwitchOff";
fName[methName]=&Executor::SwitchOff;
action::Action
    *actSwitchOff=new
    action::MethodInvocation(methName);
rName="rExecuting2Idle";
control::SimpleReaction(...);
executing->addReaction(EvSOff,rExecuting2Idle);
```

At this point, the controller was created and the system was ready to start.

**Evaluation of Results.** A Toy Example was implemented in order to show how to use the CRESC framework. In the next list, some of the positive aspects found in the experiment are presented:

- runtime introspection ability is added automatically to the controller,
- easy to use framework,
- SW development process: the implementation of the solution is generated automatically from the design phase.

It is true that the listed benefits are not measurable and at this stage we are not able to specify how much the dependability and/or efficiency have been increased. In the next steps of the research we will add mechanisms that will benefit from runtime introspection ability and these benefits will be measured.

### 4.2 Productive 4.0 Use Case

In the project Productive 4.0 (European ECSEL project), our research group will work in the *Machinery for railway wheels* Use Case leaded by DANOBAT S.Coop (industrial partner) shown in the Fig. 6. In this Use Case, the results and future works of the presented research are going to be implemented and evaluated. For that, MGEP will develop fault tolerant and safety critical SW control components based on introspection. These components will be integrated with the manufacturing HW and devices of the Use Case.

These new SW controllers will monitor the internal signals and sensors of the machines. Different machining processes will be considered and each of them will

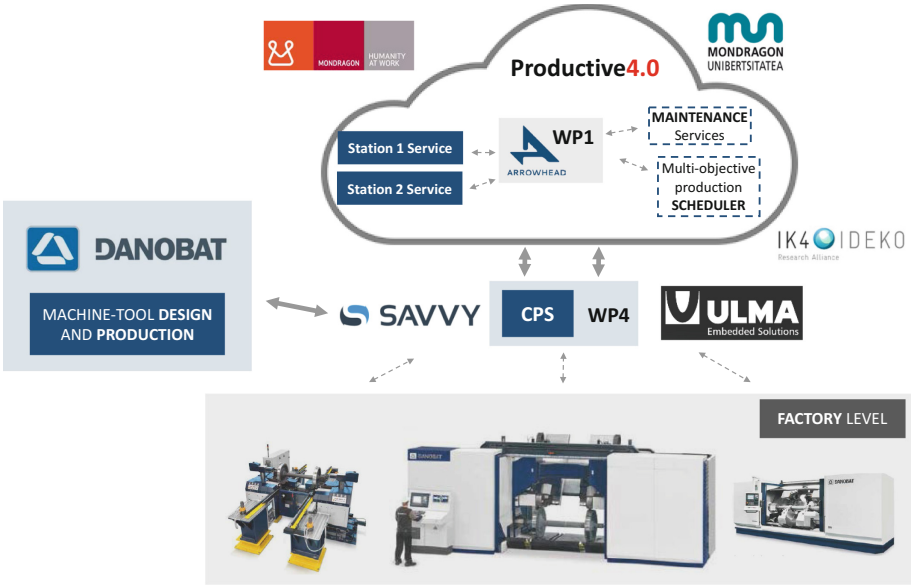


**Fig. 6.** Machinery for railway wheels.

have different optimization objectives and constraints. Based on those objectives and constraints, the controller will send optimized control signals such as spindle speed and/or feed rate to the machine.

The controller will be modelled by UML statecharts and automatically a CRESC SW controller component will be generated. The researchers are going to develop an evolution of the CRESC framework that will include a Run Time Monitoring and Checking infrastructure. Thus, the increased dependability of the controller will be measured and the benefits of the framework quantified.

## 5 Conclusion

Safety, dependability, adaptability, reliability and maintainability of CPSs are crucial issues due to the increasing complexity, development cost and the supporting Run Time environment.

In this research a framework that generates SW Control Components with introspection capacity was developed. This ability supports the addition of Fault Tolerance mechanisms.

It is true that currently there are tools that generate code automatically taking as starting point the system model. Some of them are reliable tools but the developer does not know how this transformation is carried out. They are not in control of their code and a lot of the tools are not designed for use in CPS and they are not reflective.

The presented solution is based on Reflective Statecharts and while there are other tools [22] that provide similar characteristics, they are not written in C or C++. All related implementations we have found are written in Java or in languages that are not widely accepted in the CPSs domain, or the adopted solution is very complicated which increases complexity in the solution and decreases the dependability level.

The use of Reflective Statechart separates the functional and dependability mechanisms properties which adds simplicity to the solution. When solutions are simple, the integrity and the dependability level of the system increases.

In this solution, using Reflective Statecharts as modelling technique, and C++ as the programming language, the SW Control developer has a very powerful tool for the SCCPS domain.

## 6 Future Lines

As future research lines we can consider the following topics:

- Define a catalogue of mechanisms to add specific Fault Tolerance techniques (such as SW and/or HW Redundancy and Recovery [33]) using this framework and validate it with experimentation.
- As the Reflective Statecharts can also adapt their operation mode at Run Time, implement the classes and modules that will permit the adaptation of operation mode at Run Time.

- Develop methodologies and tool support to help adding introspection ability and dependability mechanisms to legacy systems.
- Application and Evaluation of the results in the Productive 4.0 use case.

**Acknowledgments.** 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 LANA II ELKARTEK and by 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. Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling cyber-physical systems. In: Special issue on CPS, pp. 13–28. IEEE (2012)
2. IEC 61508: Functional safety of electrical/electronic/programmable electronic safety related systems (2010)
3. ISO 26262: Road vehicles- Functional Safety (2012)
4. CENELEC: EN50128 Railway applications- Communications, signalling and processing systems-Software for railway control and protection systems (2012)
5. IEC 61511: Functional safety- Safety instrumented systems for the process industry sector (2016)
6. RTCA & EUROCAE. DO-178B: Software Considerations in Airborne Systems and Equipment Certification (1992)
7. Laprie, J.-C., Arlat, J., Beounes, C., Kanoun, K.: Definition and analysis of hardware-and software-fault-tolerant architectures. Computer **23**(7), 39–51 (1990). doi:10.1109/2.56851
8. Laprie, J., Kanoun, K.: Software reliability and system reliability. In: Handbook of Software Reliability Engineering (1996)
9. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)
10. Heimerdinger, W.L., Weinstock, C.B.: A conceptual framework for system fault tolerance. Technical report, Carnegie Mellon University (1992)
11. Al-Asaad, H., Murray, B., Hayes, J.: Online BIST for emebedded systems. IEEE Des. Test Comput. **15**, 17–24 (1998)
12. Havelund, K.: Reliable software: testing and monitoring. http://www.runtime-verification.org/course09
13. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002). doi:10.1007/3-540-45937-5_16
14. Valgrind. http://valgrind.org. Accessed 14 June 2017
15. Byte code engineering library. http://commons.apache.org/proper/commons-bcel. Accessed 14 June 2017
16. Fabre, J.-C., Killijian, M.O., Taiani, F.: Lessons learnt, robustness of automotive applications using reflective computing (2011)

17. Lu, C., Fabre, J.-C., Killijian, M.-O.: Robustness of modular multi-layered software in the automotive domain: a wrapping-based approach. In: Regular paper submitted to ETFA (2009)
18. Lu, C., Fabre, J.-C., Killijian, M.-O.: An approach for improving fault-tolerance in automotive modular embedded software. INRIA, Paris, France (2009)
19. Automotive open system architecture. https://www.autosar.org. Accessed 14 June 2017
20. Ferreira, L.L., Rubira, C.M.: Reflective design patterns to implement fault tolerance (1998)
21. Barbier, F.: MDE-based design and implementation of autonomic software components. In: International Conference on Cognitive Informatics (ICCI) (2006)
22. Elkorobarrutia, X., Muxika, M., Sagardui, G., Barbier, F., Aretxandieta, X.: A framework for statechart based component reconfiguration. In: Engineering of Autonomic and Autonomous Systems (EASE) (2008)
23. The boost statechart library (2015). http://www.boost.org
24. Banci, M., Fantechi, A.: Geographical versus functional modelling by statecharts of interlocking systems. Electron. Notes Theor. Comput. Sci. **133**, 3–19 (2005)
25. Pap, Z., Majzik, I., Pataricza, A.: Checking general safety criteria on UML statecharts. In: Voges, U. (ed.) SAFECOMP 2001. LNCS, vol. 2187, pp. 46–55. Springer, Heidelberg (2001). doi:10.1007/3-540-45416-0_5
26. Pradelly, M., Pazzi. L.: Using part-whole statecharts for the safe modeling of clinical guidelines (2010)
27. The Motor Industry Software Reliability Association. Misra C++: Guidelines for the use of the C++ language in critical systems (2008)
28. Elkorobarrutia, X.: ISCART: framework para la reconfiguracin dinamica de componentes software basados en statecharts. Master's thesis, Mondragon University (2010)
29. Lu, C.: Robustesse du logiciel embarqu multicouche par une approche reflexive: application l'automobile. Master's thesis, LUNIVERSIT DE TOULOUSE (2009)
30. Illarramendi, M., Etxeberria, L., Elkorobarrutia, X.: Educational use case final results. Reuse in safety critical systems (2015)
31. Eclipse IDE for C/C++ developers (Mars). https://eclipse.org/mars. Accessed 14 June 2017
32. Papyrus. https://eclipse.org/papyrus. Accessed 14 June 2017
33. Egwutuoha, I.P., Levy, D., Selic, B., Chen, S.: A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. J. Supercomput. **65**, 1302–1326 (2013)