



Mondragon Biblioteka
Unibertsitatea Biblioteka

biblioteka@mondragon.edu

Illarramendi, M., Etxeberria, L., Elkorobarrutia, X., Sagardui, G. (2017). Increasing Dependability in Safety Critical CPSs Using Reflective Statecharts. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds) Computer Safety, Reliability, and Security . SAFECOMP 2017. Lecture Notes in Computer Science(), vol 10489. Springer, Cham. https://doi.org/10.1007/978-3-319-66284-8_11

This version of the proceeding has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at:

https://doi.org/10.1007/978-3-319-66284-8_11

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) Arrasate-Mondragon, Spain
{millarramendi, letxeberrria, 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**++ **R**eflective **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 [10]. 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 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 [22] [24][9] [2] [15].

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 become more complex and more significant to the overall system performance and dependability. Given the current state of the art, fewer development errors 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 [26].

In a SW components developing process, 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 this 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 section II of the paper the Technical Background is presented. In section III we present the CRESC Framework. After that, in section IV a Toy Example and the Use Case for Productive 4.0 project are shown. The Conclusion of the developed Framework is presented in section V 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 [25] 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. Most of the means described by Laprie are fault oriented mechanisms.

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 [5].

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 are detected by HW Redundancy (HW and some SW faults) [19], SW Diversity (SW faults) [19] and/or Run-Time Monitoring (all the faults but specially Environmental faults) of the SW control components [3].

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 [18].

To monitor a program, we need to extract the events from the program 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) [31] or byte/object code as Valgrind (C) [33] or BCEL (Java) [8].

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 [29] and [28] 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 [4] 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 [17] 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 [7] who first created a framework for Statechart based components that implicitly supported introspection of a component at Run Time. Based on this work, Elkorobarrutia [13] 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 [21], 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 Reflective Statecharts based SW components. 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 [6] [32] [30].

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++ [1]).

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 et. al [14]. 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 figure 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.

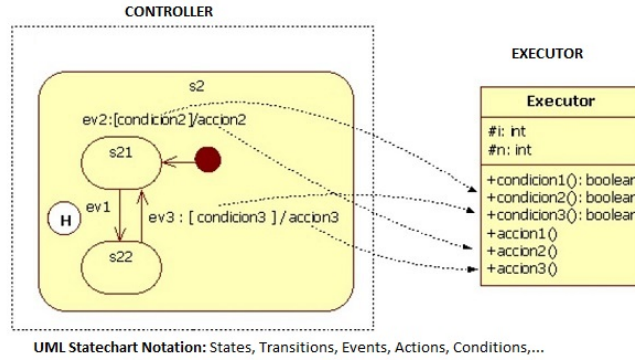


Fig. 1. Controller part and its references to the Executor

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 et al. [27], 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.

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.

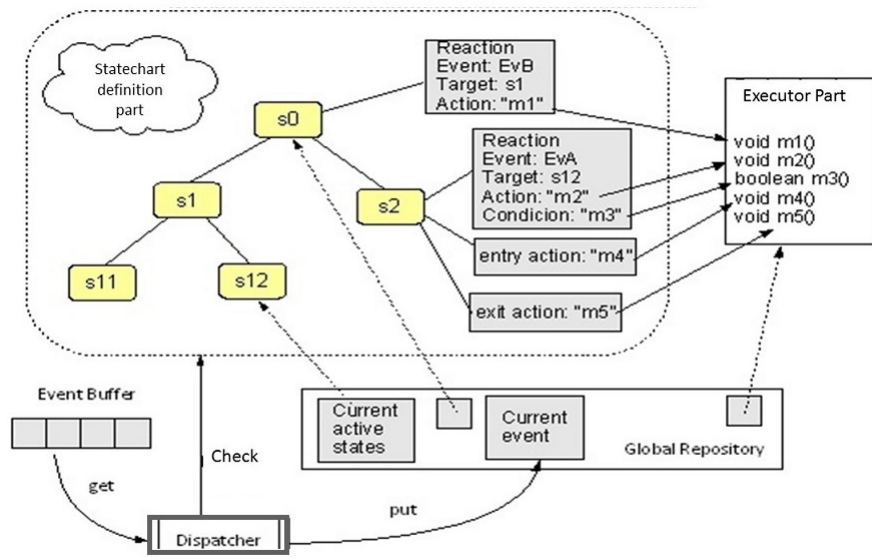


Fig. 2. Framework Infrastructure

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 figure 3 the elevator moves the load up and down by two synchronized engines. The details of this toy example were defined in [23]. 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.

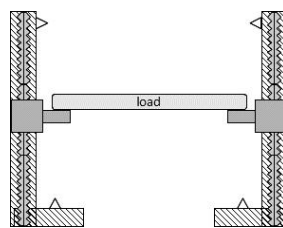


Fig. 3. Elevator Toy Example

As shown in the figure 4, the system starts in the Idle state and once the SwitchOn event is detected, it goes to the StartingUp state. Here, the controller

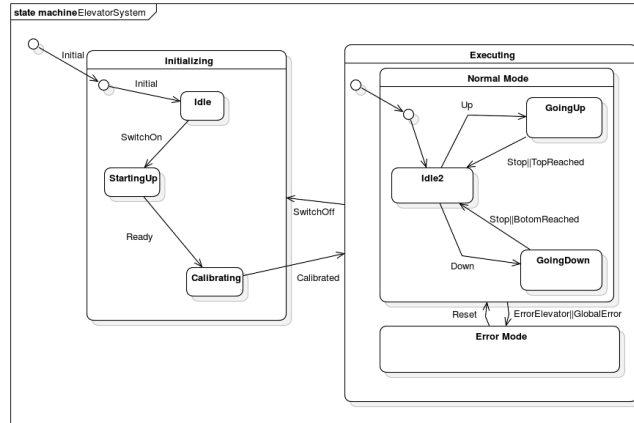


Fig. 4. Toy Example Statechart

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 eCclipse CDT [11].

4.1 Development Process of the Toy Example using the CRESC Framework

In this section we show how the described toy example was carried out from the developer role. In the figure 5 we can see the steps to follow in the main function of the CRESC framework.

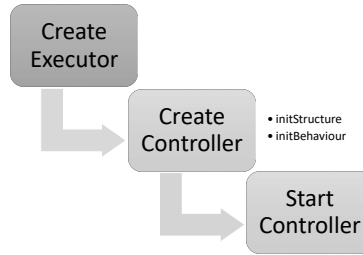


Fig. 5. CRESC: Main function steps

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).

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.

To create the code of those two functions automatically, a visual tool called Papyrus [20] for modelling both the structure and the behaviour of the system by UML Statecharts is used.

The developer has to model the system using this tool (definition of the root-State, subStates, events, actions, conditions and transition) and automatically the model is transformed to text (code). Thus, these two functions, `initStructure` and `initBehaviour` are constructed with the use case specific information extracted from the model.

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.

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 led by DANOBAT S.Coop

(industrial partner) shown in the figure 6. In this Use Case, the results and future works of the presented research are going to be implemented and verified. For that, MGEP will develop fault tolerant and safety critical SW components based on introspection. These component will be integrated to communicate with the manufacturing HW and devices of the Use Case. The researchers are going to develop an evolution of the presented work and add the Run Time Monitoring and Checking infrastructure and validate it in the use case.

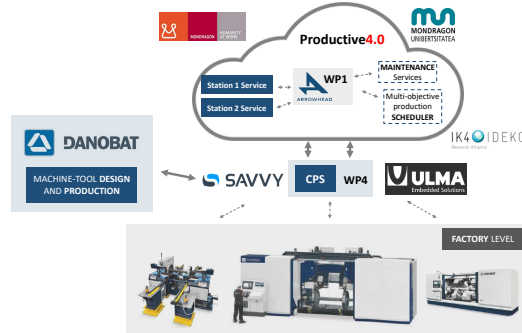


Fig. 6. Machinery for railway wheels.

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 [13] 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 [12]) 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 Validation 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 European H2020 research and innovation programme, ECSEL Joint Undertaking, and National Funding Authorities from 19 involved countries under grant agreement no. GAP-737459 - 999978918.

References

1. Misra c++: Guidelines for the use of the c++ language in critical systems,.
2. *IEC, 61511: Functional safety- Safety instrumented systems for the process industry sector.*, 2016.
3. H. Al-Asaad, B.Murray, and J.Hayes. Online bist for emebdeded systems. *IEEE , Design Test of Computers*, 15:17–24, 1998.
4. AUTOSAR. Automotive open system architecture.
5. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Transactions on dependable and secure computing*, volume 1, 2004.
6. Michele Banci and Alessandro Fantechi. Geographical versus functional modelling by statecharts of interlocking systems. *Electronic Notes in Theoretical Computer Science*, 2005.
7. F. Barbier. Mde-based design and implementation of autonomic software components. In *International Conference on Cognitive Informatics (ICCI)*, 2006.
8. BCEL. Byte code engineering library.
9. CENELEC. *EN 50128 Railway applications- Communications, signalling and processing systems- Software for railway control and protection systems*, 2012.
10. Patricia Derler, Edward A. Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber-physical systems. volume Special issue on CPS, pages 13–28. IEEE, January 2012.
11. eclipse. Eclipse ide for c/c++ developers (mars). Technical report, 2016.

12. Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 2013.
13. X. Elkorobarrutia, Mikel Muxika, G. Sagardui, F. Barbier, and X. Aretxandieta. A framework for statechart based component reconfiguration. In *Engineering of Autonomic and Autonomous Systems (EASE)*, 2008.
14. Xabier Elkorobarrutia. Iscart: Framework para la reconfiguracin dinmica de componentes software basados en statecharts. Master's thesis, Mondragon University, 2010.
15. RTCA & EUROCAE. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.
16. Jean-Charles Fabre, Marc Olivier Killijian, and Franois Taiani. Robustness of automotive applications using reflective computing: Lessons learnt. 2011.
17. L. L. Ferreira and C. M. Rubira. Reflective design patterns to implement fault tolerance. 1998.
18. Klaus Havelund. Reliable software: Testing and monitoring. <http://www.runtime-verification.org/course09/>, 2009.
19. Walter L. Heimerdinger and Charles B. Weinstock. A conceptual framework for system fault tolerance. Technical report, Carnegie Mellon University, 1992.
20. <https://eclipse.org/papyrus/>. Papyrus. Technical report.
21. "http://www.boost.org". The boost statechart library. Technical report, 2015.
22. IEC. *61508:Functional safety of electrical/electronic/programmable electronic safety related systems*, 2010.
23. Miren Illarramendi, Leire Etxeberria, and Xabier Elkorobarrutia. Reuse in safety critical systems: Educational use case final results. 2015.
24. ISO. *ISO26262:Road vehicles- Functional Safety*, 2012.
25. J. Laprie and K Kanoun. *Software reliability and system reliability*, volume Handbook of software reliability engineering. 1996.
26. Jean-Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 1990.
27. Caroline Lu. Robustesse du logiciel embarqu multicouche par une approche reflexive: application l'automobile. Master's thesis, LUNIVERSIT DE TOULOUSE, 2009.
28. Caroline Lu, Jean-Charles Fabre, and Marc-Olivier Killijian. An approach for improving fault-tolerance in automotive modular embedded software. pages 132–147, Paris, France, October 2009. inria.
29. Caroline Lu, Jean-Charles Fabre, and Marc-Olivier Killijian. Robustness of modular multi-layered software in the automotive domain:a wrapping-based approach. In *Regular paper submitted to ETFA*, 2009.
30. Marco Pradelly Luca Pazzi, editor. *Using Part-Whole Statecharts for the safe modeling of clinical guidelines*, 2010.
31. George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer, editors. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*, 2002.
32. Zsigmond Pap, Istvn Majzik, and Andrs Pataricza. Checking general safety criteria on uml statecharts. In *International Conference on Computer Safety, Reliability, and Security*, 2001.
33. Valgrind. Valgrind.