Fault tolerant sensor network using formal method Event-B

Fault Tolerant Sensor Network Using Formal Method Event-B

Ammar Alhaj Ali [0000-0003-3406-5120], Bronislav Chramcov [0000-0002-3252-1578], Roman Jasek [0000-0002-9831-9372], Rasin Katta [0000-0002-1075-1508], Said Krayem [0000-0003-3827-670X]

Faculty of Applied Informatics, Tomas Bata University in Zlin, Zlin, Czech Republic

Ammar282n@hotmail.com,{chramcov,jasek}@utb.cz,
Katta@utb.cz,drsaid@seznam.cz

Abstract. Cyber physical systems (CPS) are being increasingly deployed in different critical infrastructures such as transportation, healthcare, power, water, and other networks, these deployments witnessed a growing complexity of components by increased use of advanced technologies like sensors, actuators, communication networks and multicore processors: this posed reliability as a major challenge. We face many issues to ensure the reliability of the CPS, especially in the presence of system failures, any sensor fault that occurs during the readings from the physical systems manipulates the knowledge extracted from the physical systems and is most likely to cause the loss of the CPS reliability unless appropriate measures are taken.

In this paper we propose formal development of a software-based mechanism for fault tolerance in cyber physical system by the Event-B Method, in addition we present an approach to translate the Event-B model into C# code; which can be executed as a computer program or as simulation.

Keywords: Fault Tolerance; CPS; Cyber Physical System; Modeling; Simulation; Event-B; Rodin; C#.

1 Introduction

Cyber-physical systems (CPSs) represent a new field in automatic control that has been recently used increasingly in life critical systems, where the probability of tragic failure has to be kept below very low levels.

Cyber-physical systems represent an interconnection between processing systems and physical systems through communication, sensor and actuator technologies [1], see Figure 1.

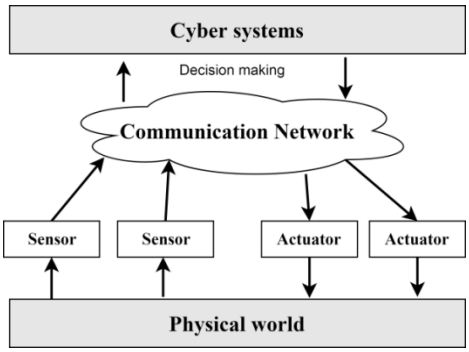

Fig. 1. Cyber physical system view

Where a sensor fault refers to a sensor's reading that is inconsistent with the expected behavior of the physical phenomenon, fault tolerance is used to face failures and achieve best performance of systems, thus; it becomes critical to develop models that are tolerant towards a system failure . This will make it possible for the CPS to continue its operation even in the presence of faults.

Modeling and simulation enables designers to test whether design specifications are met by using virtual rather than physical experiments. The use of virtual prototypes significantly shortens the design cycle and reduces the cost of design. It further provides the designer with immediate feedback regarding design decisions which, in turn, promises a more comprehensive exploration of design alternatives and a better performing of the final design [2].

In this paper; we present a mechanism for fault tolerance in cyber physical systems for the case where faults occur in the sensor interface of CPSs between the cyber and the physical parts, we propose Event-B as formal method for system-level modeling and analysis, and we recommend using the Rodin platform as modeling tool for Event-B that integrates modeling and proving, finally we present a method to the translation of our Event-B model into C# code that can be executed as a computer program.

The paper is organized as follows: in Section 2; the related works are presented, Section 3 gives definitions for basic terms and concepts, In Section 4 we briefly introduce modeling in Event-B and structure of model in Event-B, In Section 5 we introduce a fault tolerant model in cyber-physical systems. Section 6 presents modeling the FMS in Event-B, Section 7 gives a brief of transforming of Event B models into verified C# implementations, and finally, in Section 8, we present our conclusions.

2 Related works.

There is different literature covering the concept of Fault Tolerance in Cyber Physical Systems and formal development.

Sudeep Ghimire [3]; they presented the model for defining sensing model for representing sensors and actuators and creating virtual objects that will enable data re-

generation when the physical devices fails. The data regeneration algorithm is based on the virtual instance attributed with contextual details of the deployed physical devices.

Volkan Gunes [4], in his dissertation, he focuses on sensor fault mitigation and achieving high reliability in CPS operations. He examines the falling ball example (FBE) using binary event detectors, a controller, and a camera for timely motion detection and estimation of a falling ball. Another challenge he tackle is satisfying thermal comfort and energy efficiency under faulty sensor conditions in a multi-room building incorporating temperature sensors, controllers, and heating, ventilation, and air conditioning (HVAC) systems. For both cases, he adopts a model-based design (MBD) methodology to analyze the effect of sensor faults on the system outcome. In this regard, he develops well-defined fault and system evaluation models and incorporates them into the traditional CPS model that comprises the cyber, interface (e.g., sensors and actuators) and physical models.

Dubravka Ilić [5], in their paper they propose formal development of a software-based mechanism for tolerating transient faults in the B Method. The mechanism relies on a specific architecture of the error detection actions called the evaluating tests. These tests are executed (with different frequencies) on the predefined subsets of the analyzed data.

3 Definition of Basic Terms and Concepts.

System: is a construct or collection of different elements that together produces results that are not obtainable by the elements separately or alone. The elements can include people, hardware, software, facilities, policies, documents -all things required to produce a system - level qualities, properties, functions, behavior, and performance. Vitally, the value of the system as a whole is the relationship among the parts [6].

Modeling: is the process of representing a model which includes its construction and working. This model is a physical, mathematical, or otherwise logical representation of a system and it is similar to the real system, it helps the analyst to predict the effect of changes to the system. In other words, modeling is creating a model which represents a system including its properties.

Simulation of a system: is the operation of a model in terms of time or space, which helps to analyze the performance of an existing or a proposed system. In other words, simulation is the process of using a model to study the performance of a system [7].

4 Modeling in Event-B

Event-B is a formal method for system-level modeling and analysis. Key features of Event-B are the use of set theory as a modeling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proof to

verify consistency between refinement levels. Event-B is an extension of the B method, which was developed by Jean Raymond Abrial several years ago [8].

4.1 Why Event-B

- Event-B is a mature formal method which has been widely used in several industry projects in many different domains, such as automotive, transportation, space, business information, medical device and so on.
- An FP7 project supported by the European Commission, called the Deploy Project, is using Event-B to improve system dependability, to handle system complexity and to reduce the capital spent on the testing and debugging stages of software development [9].
- A recent technical report [10] written by Mery & Singh described a formal development of a Cardiac Pacemaker using Event-B, which shows the feasibility and validity of using Event-B during the development cycle of medical devices.
- Another reason for using Event-B is that it is a tool-supported formal specification language. The Rodin platform, an Eclipse-based IDE for Event-B, provides a user-friendly interface to create, refine and mathematically prove properties of models. Proof obligations are automatically generated after models have been created. The most powerful aspect of the Rodin platform is that a semiautomatic theorem proves is embedded in the tool, which saves lots of time compared to creating the proofs manually. The Rodin platform is open source, so it supports a large number of plug-in tools [9].

4.2 Event-B structure:

A model in Event-B consists of contexts and machines. Contexts contain the static part (types and constants) of a model while machines contain the dynamic part (variables and events), there are various relationships between contexts and machines [11, [12,13,14]. See next Figure2.

5 Fault tolerant model in cyber-physical systems

We face enormously increasing challenges to ensure the reliability of CPS, especially in the presence of system failures, as any sensor fault that occurs during the readings from the physical systems damages the knowledge extracted from the physical systems and presents the risk of reducing or even losing the reliability of CPS unless appropriate measures are taken.

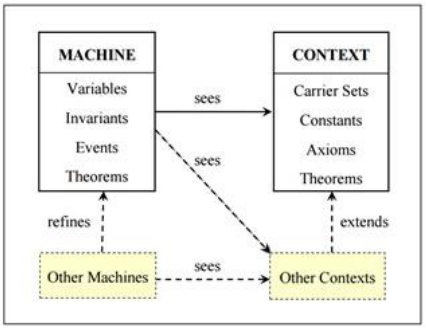

Fig. 2. Relationships between contexts and machines

Transient faults are the temporal defects within the system, they frequently occur in the hardware functioning. In our case study, we focus on designing a controller that is able to tolerate transient faults, in the complex fault-tolerant control systems; a controller largely consists of the mechanisms for implementing fault tolerance.

This is often perceived as a separate subsystem dedicated to fault tolerance, this subsystem is traditionally called Failure Management System (FMS) [5].

Obviously, correctness of the FMS itself is essential for ensuring dependability of the overall system. Formal methods are traditionally used for reasoning about software correctness; we will demonstrate how to develop the FMS by stepwise refinement in the Event-B, then translate final proven model to C# code to get the simulation.

The Failure Management System may be part of CPS as shown in Figure 3.

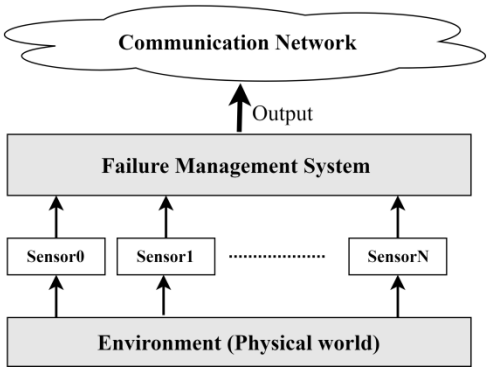

Fig. 3. Failure Management system in CPS

The sensor readings are considered as the inputs to the FMS and the outputs from the FMS are forwarded to the communication networks and the task of the FMS is to detect incorrect inputs and prevent their propagation into the communication networks, Hence the main purpose of the FMS is to supply the communication networks of the system with the fault-free inputs from the environment.

We assume that initially the system is error-free and the FMS tests the inputs by applying a certain detection procedure; As a result, the inputs are categorized as fault-

free or faulty, then the FMS analyses the categorized faulty inputs to distinguish between recoverable and non-recoverable faulty inputs.

This is achieved by assigning a status to each analyzed input; the status can be [5]:

- The fault-free inputs are marked as **ok**.
- The recoverable inputs are marked as **suspected**.
- The non-recoverable inputs are marked as **confirmed failed**.

After completing analysis, the FMS takes the corresponding actions; these actions can be classified as healthy, temporary or confirmation see Figure 4.

1. Healthy action:
   If the FMS is in the Normal state, a received input is fault-free (ok), then the input is forwarded unchanged to the communication networks and the FMS continues its operation by accepting another input from the environment.
2. Temporary action:
   If the FMS is in the Normal state and detects the first faulty input, it changes the operating state from Normal to Recover, see Figure 4,While in the Recover state, the FMS counts the number of faulty inputs in successive operating cycles and at the same time the status of the faulty input is marked as suspected, One of the requirements imposed on the FMS is to give a fault-free output even when the input is faulty, Hence, while operating in the Recover state, the FMS calculates the output using the last good values of this inputs obtained before entering the state recover. Once a temporary action is triggered, it will keep the system in the state Recover until the counting mechanism determines whether the input (i.e., the corresponding sensor) has recovered. In this case, the system changes its state from recover to normal.
3. Confirmation action:
   If the system has been operating in the state Recover and the input fails to recover, the counting mechanism triggers the confirmation action, Then the input is marked as confirmed failed and the system changes the operating state to Failed[5].

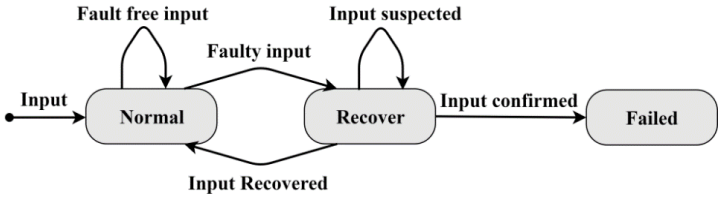

Fig. 4. Specification of the FMS behavior

6 Modeling the FMS in Event-B

The abstract specification defines the behavior of the FMS during one operating cycle is modeled using the variable `FMS_State`. The type `STATES` of `FMS_State` is defined in the context Global, as follows:

```
STATES = {env, det, anl, anlloop, act, output, freeze};
```

Where the values of FMS_State define the phases of the FMS execution in the following way:

- env: obtaining inputs from the environment.
- det: performing tests on the inputs and detecting erroneous inputs,
- anlloop and anl: deciding upon the input status,
- act: setting the appropriate actions,
- output: sending output to the communication network either by simple forwarding one of the obtained inputs or by calculating the output based on the last good values of the inputs,
- freeze: freezing the system

The variable FMS_State models the progress of the system behavior in the operating cycle. At the end of the operating cycle the system either reaches the terminating (freezing) state or produces a fault-free output. In the latter case, the operating cycle starts again.

6.1 Variables

Table 1 shows all variables in abstract, First refinement and second refinement model.

Table 1. Variables in FMS model

No	Variables	description	invariants
abstract			
1	Indx	Index of The input values by the sensors	$0 \cdots max\_indx$ where $max\_indx \in \mathbb{N}$
2	InputN	The input values produced by the sensors (N multiple homogeneous sensors.)	Indx→ $\mathbb{N}$
3	Input_StatusN	it is an array that for each of N inputs contains a value of the type: $I\_STATUS = \{ok, suspected, confirmed\_failed\}$; (previous cycle)	Indx→ **I_STATUS**
4	Input_In_ErrorN.	It is **TRUE**, if an error is detected on the sensor reading of a particular input, and **FALSE** otherwise.	Indx→**BOOL**
5	Last_Good_InputN	Last good input or last input – fault-free	Last_Good_InputN ∈ Indx →$\mathbb{N}$
6	Output	Out of FMS	Output ∈ $\mathbb{N}$
7	FMS_State	The variable *FMS_State* models the evolution of the system behavior in the operating cycle.	FMS_State ∈ **STATES** Where $STATES = \{env, det, anl, anlloop, act, out, freeze\}$;
First refinement			
8	Input_StatusN1	it is an array that for each of N inputs contains a value of the type: $I\_STATUS = \{ok, suspected, confirmed\_failed\}$;(current cycle)	Indx→ **I_STATUS**
9	Processed	It is **True**,After the operation AnalysisLoop is completed else **False**.	Processed ∈ Indx →BOOL
Second refinement			
10	cc	For N inputs, we introduce counters cc_i (i∈0.. *max_indx*), which contain accumulated values determining how trustworthy a particular input *i* is.	cc ∈ Indx ↛ $\mathbb{N}$
11	num	For N inputs, the counter which counts the number of the consequent recovering cycles for each suspected input	num ∈ Indx ↛ $\mathbb{N}$

6.2 Events

In abstract model we use the following events:

1. Environment: The input values produced by the sensors are modeled by the variable InputN. The variable represents the readings of N multiple homogeneous sensors, see Figure 5.

```
event Environment
  where
    @grd1 FMS_State = env
  then
    @act1 InputN :∈ Indx →ℕ
    @act2 FMS_State ≔ det
end
```

Fig. 5. Specification of environment Event

2. Detection: After obtaining the sensor readings from the environment, the FMS starts the operation Detection, we assign the value TRUE to the variable Input_In_ErrorN , if an error is detected on the sensor reading of a particular input, and FALSE otherwise, see Figure 6.

```
event Detection
  where
    @grd1 FMS_State = det
  then
    @act1 Input_In_ErrorN :∈ Indx →BOOL
    @act2 FMS_State ≔ anlloop
End
```

Fig. 6. Specification of detection Event

3. AnalysisLoop: After the detection phase, the FMS performs the operation AnalysisLoop, the operation Analysis sets the results of the analysis for all the inputs at once, see Figure 7.

```
event AnalysisLoop
  where
    @grd1 FMS_State = anlloop
  then
    @act1 FMS_State :∈ { anlloop , anl }
End
```

Fig. 7. Specification of analysisloop Event

4. Analysis: The FMS decides upon the status of an input (Based on the results obtained at the previous state); fault-free (i.e., ok), suspected or confirmed failed. The variable Input_StatusN is an array that for each of N inputs contains a value of the type: I_STATUS = {ok, suspected, confirmed_failed}, see Figure 8.

```
event Analysis
    where
      @grd1 FMS_State = anl
    then
      @act2 FMS_State ≔ act
      @act3 Input_StatusN :∈ { ff | ff ∈ Indx →I_STATUS ∧
            (∀ee·ee∈Indx ∧ Input_In_ErrorN(ee)=FALSE ⇒
ff(ee)∈{ok, suspected,confirmed_failed} ) ∧
            (∀ee·ee∈Indx ∧ Input_In_ErrorN(ee)=TRUE ⇒
ff(ee)∈{suspected, confirmed_failed})}
  End
```

Fig. 8. Specification of analysis Event

5. Action: here we have four action events, the operation Action will decide if system will go to freeze status or out status, see Figure 9.

6. Return: in this Event FMS will send output to the communication network either by simple forwarding one of the obtained inputs or by calculating the output based on the last good values of the inputs, then move to Environment event to read new values, see Figure 10.

6.3 Refinement

First refinement: we specify in detail the operation AnalysisLoop. The operation gradually performs the input analysis, considering inputs one by one until all the inputs are processed. The information about the input status of the processed inputs is correspondingly accumulated in the variable Input_StatusN1. After the operation AnalysisLoop is completed, the value of Input_StatusN1 is assigned to Input_StatusN in the operation Analysis.

Second refinement: For each of N inputs, we introduce counters cci (i∈ 0.. max_indx), which contain accumulated values determining how trustworthy a particular input i is. If cci=0 then the input i is ok. If 0<cci<zz, where zz is some predefined value, the input i is suspected. Otherwise, the input i is considered failed.

At every cycle the counters cci are reevaluated depending on the detection results (suspected status). Each faulty input i increments the counter cci by a certain predefined value xx. Similarly, each fault-free input i decrements the corresponding counter cci by another predefined value yy. If at some point the value of cci reaches 0, the input i is declared ok. Similarly, if the value of cci exceeds zz, the input i is declared confirmed_failed and should be removed from the set of inputs used by the FMS.

The predefined values zz, xx and yy are set after observing the real performance of the FMS. The input may behave in such a way that the counter cc is practically oscillating between some values but never reaches the limit zz or zero. To overcome this problem, we introduce the second counter Num which counts the number of the consequent recovering cycles for each suspected input (i.e., when 0<cci<zz). When a certain limit for Num is exceeded, the recovery terminates and, if cc is different from zero, the input is confirmed_failed [5].

```
event ActionAllOut
    where
      @grd1 FMS_State = act
      @grd2 confirmed_failed ∉ ran (Input_StatusN)
      @grd3 Indx ≠ ∅
    Then @act1 FMS_State ≔ output
  end
  event ActionSomeOut
    where
      @grd1 FMS_State = act
      @grd2 confirmed_failed ∈ ran ( Input_StatusN )
      @grd3 Input_StatusN~[ { ok , suspected } ] ≠ ∅
    then
      @act1 Indx ≔ Input_StatusN~[ { ok , suspected } ]
      @act2 InputN ≔ Input_StatusN ~ [ { ok , suspected } ]◁
InputN
      @act3 Input_StatusN ≔ Input_StatusN ▷{ ok , suspected }
      @act4 Input_In_ErrorN ≔ Input_StatusN ~ [ { ok , sus-
pected } ] ◁ Input_In_ErrorN
      @act5 Last_Good_InputN ≔ Input_StatusN ~ [ { ok , sus-
pected } ] ◁ Last_Good_InputN
      @act6 FMS_State ≔ output
  End
  event ActionAllFailed
    where
      @grd1 FMS_State = act
      @grd2 confirmed_failed ∈ ran ( Input_StatusN )
      @grd3 Input_StatusN~[ { ok , suspected } ] = ∅
    then
      @act1 FMS_State ≔ freeze
  end
  event ActionNoOut
    where
      @grd1 FMS_State = act
      @grd2 confirmed_failed ∉ ran (Input_StatusN)
      @grd3 Indx = ∅
    then
      @act1 FMS_State ≔ freeze end
```

Fig. 9. Specification of action Events

```
event Return
    any In
    where
      @grd1 FMS_State = output
      @grd2 In = ( Last_Good_InputN ⩤ ( Input_StatusN ~ [ { ok
} ] ◁ InputN ) )
      @grd3 ran(In)≠∅
    then
      @act1 Last_Good_InputN ≔ In
      @act2 Output :∈ ran ( In )
      @act3 Input_In_ErrorN ≔ Indx × { FALSE }
      @act4 FMS_State ≔ env
  End
```

Fig. 10. Specification of return Event

6.4 Proof Statistics.

In table 2, we can see proof statistics for our model using the Rodin3.2 platform, the statistics give us the proof obligations generated and discharged by the Rodin. The finial development of our model results in 227 POs (Proof obligations), around (98%) of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment [13,14,15].

Table 2. Proof Obligations of our model

Reliability (Our model)	Total	Auto	Manual
	227(100%)	223 (98%)	4 (2%)
Global	0	0	0
Global1	0	0	0
FMS	24	24	0
FMS1	133	133	0
FMS2	70	66	4

7 Transforming EVENT B Models into Verified C# Implementations.

The RODIN platform allows translation of Event-B to the C# programming language by use plugin tools; EB2ALL is a set of translator tools that automatically generates efficient target programming language code (C, C++, Java and C#) from Event-B formal specification related to the analysis of the complex problems. The EB2ALL contains four plugin namely EB2C, EB2C++, EB2J and EBC#. The goal of EB2ALL is to be able to generate a verified source code that satisfies behavioral properties of the develop formal system (abstractly). The EB2ALL tool is developed as a set of plugins for RODIN development tool under the Eclipse framework [16].

7.1 C# Translation Philosophy

After final refinement step; we can generate C# code by plugin tools (i.e., EB2ALL) of Rodin or we can convert manually Event B syntax to c# code [17,18].
A top-level C# main function must be provided to call the generated functions "INITIALISATION" and "Iterate". The calling of INITIALISATION function of FMS must be called before Iterate, see figure11.

```
if (INITIALISATION() == true)
{
Iterate();
}
//*****************************************//
public bool Iterate()
{
if (Environment() == true) return true;
if (Detection() == true) return true;
if (AnalysisLoop() == true) return true;
if (Analysis() == true) return true;
if (Action() == true) return true;
if (Return() == true) return true;
return false;
}
```

Fig. 11. Structure of calling Events in C#

7.2 FMS Simulation in C#

Simulation is a technique in which a real-life system or process is emulated by a designed model. The model encapsulates all of the system's features and behaviors, the simulation is the execution of this system over time. Due to the fact that simulation programming can be complicated, there have been many attempts to create languages that embody all the requirements of the simulation paradigm in order to ease the development. One such language is SIMULA.

Nowadays, the focus is more on creating packages, frameworks or libraries that incorporate what programmers need when creating a simulation. These libraries are meant to be called from ordinary languages like C#, C++, Java or Python [19].
We used MS Visual studio to create our simulation with C# language, see Figure 12

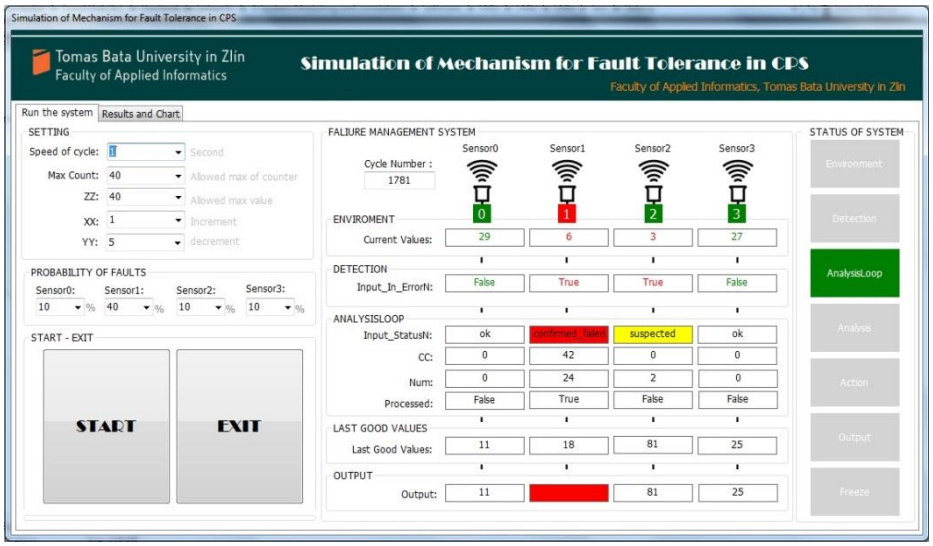

Fig. 12. FMS simulation (Run the system)

8 Conclusions:

In this paper we offered a formal pattern for specifying and refining a mechanism for fault tolerance in cyber physical systems. Our formal development of the failure management system adopts the CPS approach itself.

Formal methods are traditionally used for reasoning about software correctness, we demonstrated how to develop the FMS by stepwise refinement in the Event-B, then we translated the final proven model to C# code to get the simulation. We introduced the main principles, rules and implementation solutions for the translation tools in order to generate the target programming language code (C#) from the Event-B specifications.

The translator generates for all events of concrete modules separate code methods in the target programming language this approach may be applicable to massive parallel processing. After the final refinement step, we can generate C# code by plugin tools (i.e., EB2ALL) of Rodin or we can convert manually Event B syntax to C# code.

The Rodin tool uses an internal database to handle model information which allows model generation to be based on the underlying meaning of the model and reduces the syntax dependency.

Rodin tool is a platform where verification of the program is done and it offers reactive environment for constructing and analyzing models as do most modern integrated development environments, It provides integration between modeling and proving, whereas this is an important feature for the developers that allows them to focus on the modeling task without the need to switch between different tools to check proving correctness of model at same time. Our final models are translated into

the Event-B notation to verify required properties; we can say that Event-B allows us to define a kind of modeling methodology by writing the correct mathematical notions.

We verified our complete development with Rodin platform; around 98% of proof obligations have been proved automatically by the tool, the rest have been proved using the Rodin interactive proving environment.

9 References

1. Roxana, R. B., & Eva-Henrietta, D:. Fault-tolerant Control of a Cyber-physical System. MS&E, 261(1), 012003. (2017)
2. Sinha, R., Paredis, C. J., Liang, V. C., & Khosla, P. K. Modeling and simulation methods for design of engineering systems. J. Comput. Inf. Sci. Eng., 1(1), 84-91. (2001).
3. Ghimire, S., Sarraipa, J., Agostinho, C., & Jardim-Goncalves, R.: Fault tolerant sensing model for cyber-physical systems. In Proceedings of the Symposium on Model-driven Approaches for Simulation Engineering (pp. 1-9). (2017)
4. Gunes, V.: Ensuring Reliability and Fault-Tolerance for the Cyber-Physical System Design (Doctoral dissertation, UC Irvine). (2015)
5. Dubravka Ilić, Elena Troubitsyna, Linas Laibinis, and Colin Snook.: Formal Development of Mechanisms for Tolerating Transient Faults. (2006). SBN-10 3-540-48265-2, ISBN-13 978-3-540-48265-9, Springer.
6. John A. Sokolowski,Catherine M. Banks (2010). Modeling and simulation fundamentals, ISBN 978-0-470-48674-0, published by John Wiley & Sons, Inc., Hoboken, New Jersey.
7. Modeling & Simulation Tutorial (2018). https://www.tutorialspoint.com/modelling_and_simulation/index.htm. Accessed 2020-11-20.
8. Event-B and the Rodin Platform (2018). http://www.event-b.org. Accessed 2020-11-20.
9. Xu, H.: Model based system consistency checking using Event-B (Doctoral dissertation). (2012)
10. Méry, D., & Singh, N. K.: Technical report on formal development of two-electrode cardiac pacing system. (2010)
11. Michael Jastram, Michael Butler.:Rodin User's Handbook: Covers Rodin v.2.8, CreateSpace Independent Publishing Platform. (2014). ISBN 10: 1495438147 ISBN 13: 9781495438141, USA
12. Boulanger, J. L. (Ed.).: Industrial use of formal methods: formal verification. John Wiley & Sons. (2013)
13. Ali, A. A., Krayem, S., Chramcov, B., & Kadi, M. F.:Self-Stabilizing Fault Tolerance Distributed Cyber Physical Systems. Annals of DAAAM & Proceedings, 29. (2018).

14. Ali, A. A., Jasek, R., Krayem, S., & Zacek, P.:Proving the Effectiveness of Negotiation Protocols KQML in Multi-agent Systems Using Event-B. In Computer Science On-line Conference (pp. 397-406). Springer, Cham. (2017).
15. Ali, A. A., Jasek, R., Krayem, S., Chramcov, B., & Zacek, P.:Improved Adaptive Fault Tolerance Model for Increasing Reliability in Cloud Computing Using Event-B. In Computer Science On-line Conference (pp. 246-258). Springer, Cham. (2018).
16. EB2ALL - The Event-B To C, C++, Java And C# Code Generator (2011). http://eb2all.loria.fr/. Accessed 2020-11-20.
17. Elsayed, E. K., & El-Sharawy, E.: High Quality Implementation For Automatic Generation C# Code By Event-B Pattern. International Journal of Software Engineering & Applications, 5(1), 43. (2014)
18. Degerlund, F., Grönblom, R., & Sere, K.: Code generation and scheduling of Event-B models. Turku Centre for Computer Science, Tech. Rep, 1027. (2011)
19. Discrete Event Simulation: A Population Growth Example (2016). https://msdn.microsoft.com/en-us/magazine/mt683796.aspx. Accessed 2020-11-20.