9

Composable Framework Support for Software-FMEA through Model Execution

Valentina Bonfiglio[1], Francesco Brancati[1], Francesco Rossi[1], Andrea Bondavalli[2,3], Leonardo Montecchi[2,3], András Pataricza[4], Imre Kocsis[4] and Vince Molnár[4]

[1]Resiltech s.r.l., Pontedera (PI), Italy
[2]Department of Mathematics and Informatics, University of Florence, Florence, Italy
[3]CINI-Consorzio Interuniversitario Nazionale per l'Informatica-University of Florence, Florence, Italy
[4]Dept. of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

9.1 Introduction

Performing Failure Mode and Effects Analysis (FMEA) during software architecture design is becoming a basic requirement in an increasing number of domains. However, due to the lack of standardized early design-phase model execution, classic Software-FMEA (SW-FMEA) approaches carry significant risks and are human effort-intensive even in processes that use Model-Driven Engineering (MDE).

From a dependability-critical development process point of view, FMEA – more generally, the identification of hazards and planning their mitigation – should be performed in the early phases of system design; for software, this usually translates to the architecture design phase [1]. Additionally, for some domains, standards prescribe the safety analysis of the software architecture – as is the case, e.g., with ISO 26262 in the automotive domain.

However, historically, software architecture specifications in the most widely used modelling languages either do not represent behaviour, only structure, or the behavioural models do not have standardized operational semantics. This is a major problem for SW-FMEA; in contrast to hardware, relatively small changes of “internals” of a software component (essentially the program logic) can lead to wide variations in the response of executed software components to various external and internal faults. This means that in addition to computing error propagation from component to component, the sensitivity of each component for internal and external faults has to be explored on a case by case basis, and this can be done only by using specifications of behaviour.

In the absence of this capability, the system modeller has to either make strong guarantees in advance (“this component will be fail-silent under all circumstances”), or make too pessimistic assumptions (e.g., “all kinds of output failures are possible”). Significant risk is introduced by the fact that the error propagation assumptions made at this stage have to hold for the final system – otherwise the constructed hazard mitigation arguments will not hold, either. Thus, without rolling back the development process, we run the risk of having to enforce not easily enforceable guarantees, or having to use dependability mechanisms that are actually superfluous in the given system.

This chapter addresses the aforementioned problem on the basis of a new standard for the UML 2 modelling language. Throughout the next sections, we will introduce the reader to advances in standardized model execution semantics, the outline of a composable framework built on top of executable software architecture models to help SW-FMEA, as well as a realization of such a framework applied on a case study from the railway domain.

9.2 Software-FMEA Using fUML/ALF

For UML 2, the status quo of not having standardized operational semantics has changed with the standardization of “Foundational UML” (fUML) [2]: a core subset of UML 2 has been given standardized execution semantics. Although fUML mainly contains facilities for describing structured activities of communicating, typed objects, in theory, the whole UML 2 language can be mapped to it. To facilitate practical application, fUML also has an action language called Alf, the “Action Language for Foundational UML” [3].

Alf is a quasi-imperative, Java-like programming language. As a “surface language” for fUML, its structure and execution is directly and unambiguously mapped to fUML. Whole programs can be written purely in Alf, but it can be also used to define specific behaviours in an encompassing UML 2 model. However, in this case, the operational semantics of the embedding model containing the Alf code snippets also has to be specified, e.g., by translating the whole model to pure Alf. This is not necessarily a shortcoming; our approach actively exploits the partially “underdefined” composite structure semantics. That said, it is worth to note that the newly finalized standard “Precise Semantics of UML Composite Structures” (PSCS) [4] addresses exactly this issue.

9.2.1 Tooling for fUML and Alf

Due to the novelty of the languages, fUML and Alf tooling is still maturing, but the progress is steady. For both fUML and Alf, reference interpreters exist [5]; for fUML, additional execution engines are also available [6]. Papyrus, the popular Eclipse-based modelling environment includes an Alf editor for UML 2 language elements and supports direct compilation of Alf code into UML 2 [7].

The compilation of fUML/Alf to other languages and the formal analysis of fUML/Alf specifications are much less developed, with no directly (re)usable solutions known. That said, notably [8] presents a full Model-Driven Engineering approach where Alf code is translated into an intermediate model that, in turn, is translated to C++. On the formal analysis side, initial progress has been made both for theorem proving [9] and classic model checking [10].

9.2.2 Software-FMEA through Alf Execution

Earlier work performed in the CECRIS project (“Certification of Critical Systems”) [11] has proposed an approach for the SW-FMEA of component-based systems through Alf execution (using an interpreter) [12]. The main idea of the approach is summarized in the following three steps.

1. Components as well as their Alf code are translated into a single Alf program. During translation, the code for a cyclic scheduler component is also woven into the Alf source (with a simple logical clock). The static component activation schedule is determined by the modeller.
2. As a form of model-level fault injection, the translation can inject some simple errors into the scheduler as well as replace output/input port behaviours with “programmed” error behaviours.
3. Error propagation is analyzed by comparing simulation runs of the fault-free case to various fault activations.

In general, simulation certainly has its drawbacks; e.g., it is hard to ensure that all execution paths have been exercised in a nondeterministic system, though this is not a major issue for three reasons. First, the reference simulator performs sequential execution with deterministic choices (a semantic variation that the fUML standard fully allows). Second, although the embedded system models we apply our approach to do not exhibit parallelism at either the micro or the macro level, there is at least one fUML virtual machine called *moliz* that supports very fine-grained external control of model execution [13]. This means that if the need arises, the various interleaving executions can be tracked, accounted for and controlled. Thirdly, we do expect solutions for the application of formal methods (at least model checking) on fUML/Alf models to appear in the near future; these, by their nature, cover the entire state space of models.

These considerations demonstrate an important strength of the approach proposed in Bonfiglio et al. [12] and provides one of the main motivations for the framework presented in this chapter. If, during the translation of the component model to pure Alf, we are able to equip the Alf code with all the facilities that transform model execution into explicit error propagation execution, then we can reap the benefits of advances in fUML/Alf tooling without additional effort.

9.2.3 Framework Support for Executable Error Propagation

Along the previous consideration, we describe the design of a model transformation framework that transforms component models with Alf behaviour specifications into a pure Alf program that simulates error propagation by passing error tokens between the components and computing (or approximating) the input-output error transformation that a potentially faulty software component exhibits upon (erroneous or correct) activation (Figure 9.1). As the user-supplied Alf code cannot always be used to compute error propagation (e.g., the component itself might have an active internal fault), in some cases, error transformation draws on a library of behavioural patterns (e.g., “fail-silent”).

The orchestration of the execution of components is broken into a number of configurable, cooperating functions. These functions have generic variants; these are drawn from a framework library of options (Figure 9.2).

9.2.4 Error Tokens, Component Activation

The composite error tokens passed between components carry a reference value – the object that should be seen during the interaction in a fault free system – as well as error information. The error being passed is either a standard category (succinctly introduced, e.g., in TanjaMayerhofer [14]), a refinement thereof, or a specific one (e.g., a specific erroneous value that is late by a known amount of time).

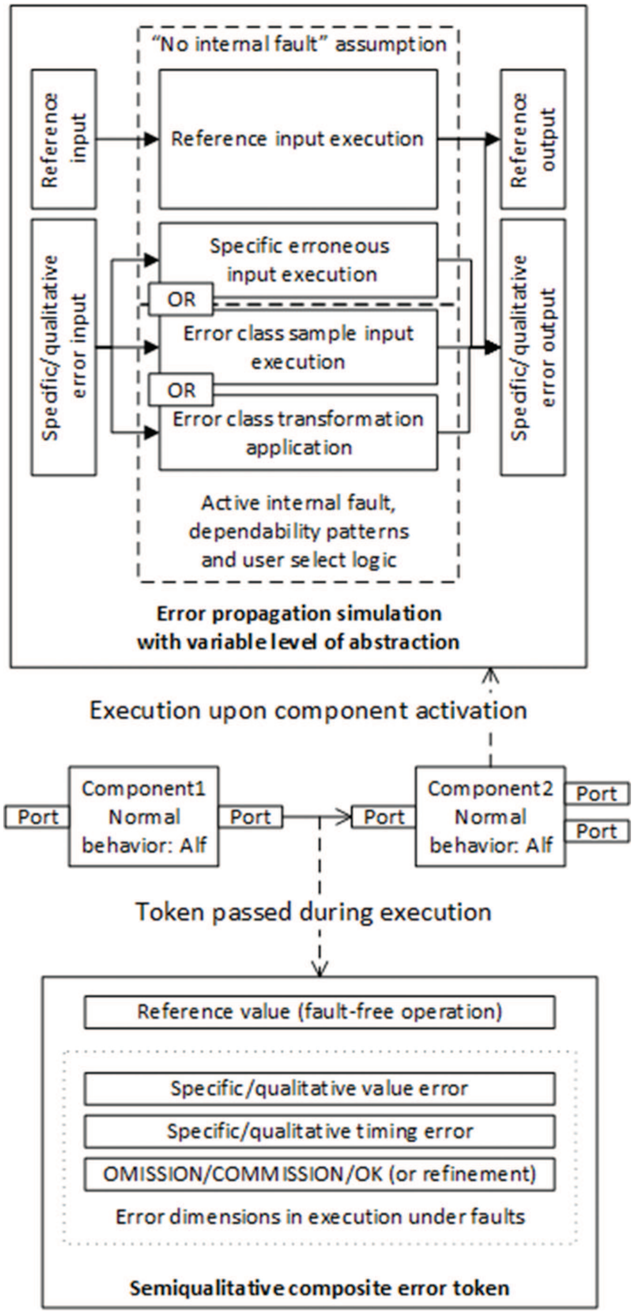
"No internal fault" assumption
Reference input
Reference input execution
Reference output
Specific/qualitative error input
Specific erroneous input execution
OR
Error class sample input execution
OR
Error class transformation application
Specific/qualitative error output
Active internal fault, dependability patterns and user select logic
Error propagation simulation with variable level of abstraction
Execution upon component activation
Port
Component1 Normal behavior: Alf
Port
Port
Component2 Normal behavior: Alf
Port
Port
Token passed during execution
Reference value (fault-free operation)
Specific/qualitative value error
Specific/qualitative timing error
OMISSION/COMMISSION/OK (or refinement)
Error dimensions in execution under faults
Semiqualitative composite error token

Figure 9.1 Composite error token passing during execution and component activation.

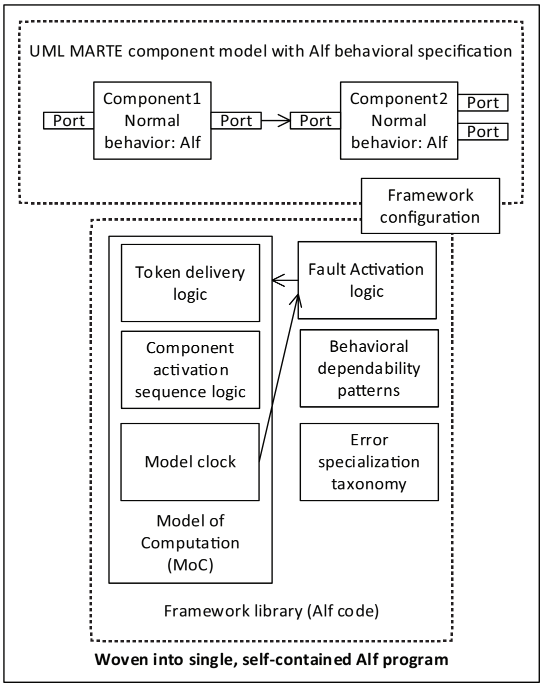

Figure 9.2 Framework components for program composition.

Component activation computes the output error tokens of the component based on the input ones. The logic for producing the error outputs depends on numerous, but mostly straightforward factors (Figure 9.2); to note is that for computing the error output when the specific error is not known, only the category, the modeller may decide to either run the user-supplied Alf code on a sample from the class or use a predefined error category transformation logic from a library of dependability behavioural patterns.

9.2.5 Execution Orchestration

Component models in UML 2 do not have standard execution semantics; the cyclic execution logic with a static schedule in Bonfiglio et al. [12] (summarized in Section 9.2.2) came from the domain of the modelled system. As a matter of fact, the overall approach is able to support numerous models of computation (MoC) – rules defining the semantics of control, concurrency, communications and time. Synchronous data flow networks, discrete events, static scheduling, and workflow-like execution all fit the approach through configurable, reusable implementations in Alf (with varying complexity, of course). In order to be able to account for orchestration errors, the framework is also intended to support runtime fault injection on the MoC implementations.

9.2.6 Fault Injection

Fault injection is performed by configurable fault activation logic implementations. These determine active faults of the various components (including orchestration) at various points in time (if the MoC defines a notion of time).

9.3 Case Study: Application of Software-FMEA through Model Execution

The case study used for the Software FMEA process was based on the railway domain, more specifically the European Rail Traffic Management System (ERTMS) and its Control Command part European Train Control System (ETCS) [15]. ERTMS/ERTCS is an automatic train protection system, and as such, a safety-critical system. ERTMS is composed of trackside units (e.g. beacons for positioning and information reporting) and on-board units. The full system is rather complex, thus in the case study only a small, simplified part of the specification was modelled. The focus of the modelling was on the safety function of receiving and consistency checking of messages from trackside beacons called *balises*.

9.3.1 Definition of the Modelled System

The case study system was based on the balise-related basic functionality of ERTMS/ETCS. The system was modelled using UML and Alf (Action Language for Foundational UML) [16]. The static structure part of the system was also described with the textual syntax of Alf; however, some of these will be represented on graphical diagrams for convenience.

As the envisioned Software-FMEA approach should be applied early in the design phase, no actual implementation or detailed design model is available in this stage. Therefore, in order to exercise the behaviour defined in the executable model, the “environment” of the modelled system had to be simulated. This scaffolding was also developed in Alf, thus the model consists of two main parts.

- **Target system**: the On-board Unit (OBU) of ETCS, the core software running in the train.
- **Environment**: simulation of the track, trackside equipment and movement of the train.

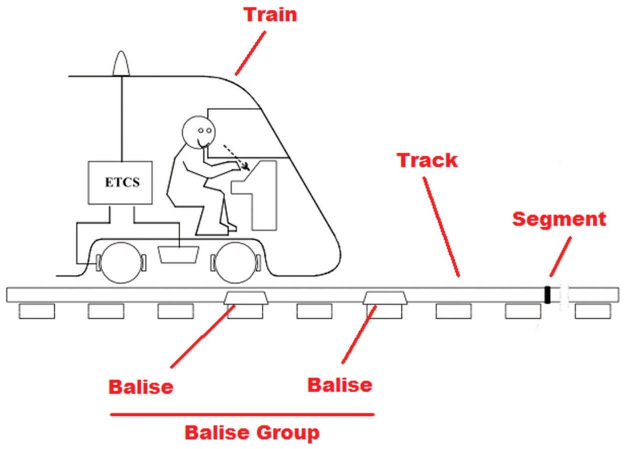

Figure 9.3 Parts of the simulated environment in the case study (figure based on European Railway Agency [15]).

Note that to reduce the complexity, the simulation of the environment focuses only on the necessary details to support the modelled functionality. Hence, the simulation is based on discrete events, and speed, distance and braking are all abstracted.

The main elements in the environment of the system are depicted on Figure 9.3 (based on Figure 2.6 in ETCS Subsection 026 Chapter 2 [15]) and are briefly explained below.

- *Track*. The train is moving on a track (the actual physical dimensions of the track are abstracted in the case study).
- *Segment*. The track is composed of neighboring segments. The train can move from one segment to another neighboring one.
- *Train*. The train can move in forward or backward one segment in each simulation step (the actual speed of the train is abstracted).
- *Balise*. A passive beacon deployed onto the track. When the train passes over a balise, it powers it up remotely via radio waves, causing the balise to send a so called telegram to the train.
- *Balise Group*. Balises can be organized in balise groups. A balise group can contain up to 8 balises. By giving position numbers to individual balises inside a group, the train can identify direction and detect missed balises.

The modelled target system consists of the main parts depicted on Figure 9.4 and explained below.

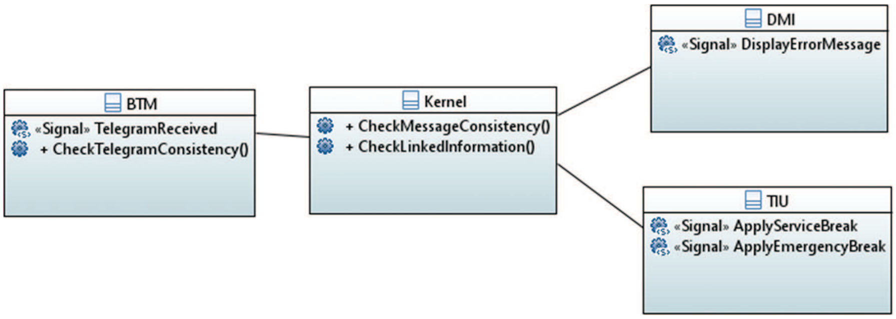
BTM
«Signal» TelegramReceived
+ CheckTelegramConsistency()
Kernel
+ CheckMessageConsistency()
+ CheckLinkedInformation()
DMI
«Signal» DisplayErrorMessage
TIU
«Signal» ApplyServiceBreak
«Signal» ApplyEmergencyBreak

Figure 9.4 Main components of the modelled system.

- *Balise Transmission Module* (BTM). Responsible for receiving raw, individual telegrams from the balises, checking and then forwarding them to the Kernel.
- *Kernel*. Responsible for the core functionality in ETCS. In the current case study, it collects telegrams from different balises to form and analyze balise group messages. If an error is detected, it can notify the driver through DMI or control the train through TIU.
- *Driver Machine Interface* (DMI). Can display information on the driver interface.
- *Train Interface Unit* (TIU). Can control the train. In the current case study, it can apply breaking.

The simulated environment and the target OBU system is connected by sending and receiving balise telegrams. The structure of a telegram is defined in Chapter 8 of the ETCS Subset 26 (SRS) [15], and is summarized in Figure 9.5.

The telegram itself was modelled using data types in Alf. The simulated balise and the BTM module directly work on this data structure, while the Kernel receives an object structure built by the BTM based on the telegrams.

The modelled behaviour is attached to the active classes in the system. Basically, they are all waiting for signals to receive, and then perform the signal handler behaviour specified in Alf. For example, upon receiving a raw telegram, the BTM checks the consistency of the header fields. This was implemented in the Alf activity presented in Figure 9.6.

The model and the modelled scenarios were executed in the Alf Reference Implementation [17]. The model includes logging to create execution traces. For example, the output in Figure 9.7 shows a simple, valid execution trace where the OBU receives a consistent telegram from a single balise. The same

GENERAL FORMAT OF BALISE TELEGRAM

Field No.	VARIABLE	Length (bits)	Remarks
1	Q_UPDOWN	1	Defines the direction of the information: Down-link telegram (train to track) (0) Up-link telegram (track to train) (1)
2	M_VERSION	7	Version of the ERTMS/ETCS system.
3	Q_MEDIA	1	Defines the type of media: Balise (0)
4	N_PIG	3	Position in the group. Defines the position of the balise in the balise group.
5	N_TOTAL	3	Total number of balises in the balise group.
6	M_DUP	2	Used to indicate whether the information of the balise is a duplicate of the balise before or after this one.
7	M_MCOUNT	8	Message counter (M_MCOUNT) – 8 bits. To enable detection of a change of balise group message duringpassage of the balise group.
8	NID_C	10	Country or region.
9	NID_BG	14	Identity of the balise group.
10	Q_LINK	1	Marks the balise group as linked (Q_LINK = 1) or unlinked (Q_LINK = 0).
	Packet 0 (optional)	14	Virtual Balise Cover marker.
	Information	Variable	This information is composed according to the rules applicable to packets.
	Packet 255	8	Finishing flag of the telegram.

Figure 9.5 Structure of a balise telegram [15].

```
privateactivityCheckTelegramConsistency(in t : Telegram) : Boolean {
let consistent: Boolean = true;
if (t.Q_UPDOWN != UpDown.Up || t.Q_MEDIA != Media.Balise ||
t.N_PIG<0 || t.N_PIG>7 ||
t.N_TOTAL<0 || t.N_TOTAL>7) {
        consistent = false;
    }
//further checks ...
return consistent;
}
```

Figure 9.6 Alf implementation of a BTM behaviour.

```
   [test  ] SingleBalise_Valid_ReceiveTelegram
[train ] Received MoveForward
[train ] Moved to segment s2
[train ] train  -> s2     : TelePower
[s2    ] Received TelePower from train
[s2    ] s2     -> b1     : TelePower
[b1    ] Received TelePower from train
[b1    ] b1     -> BTM    : TelegramReceived
[BTM   ] Received Telegram from Balise with position 0 in BG 2
[BTM   ] BTM    -> KERNEL : TelegramReceived
   [KERNEL]  Received  Telegram  from  Balise  1  in  BG  2,  consistent:
true
```

Figure 9.7 Log trace of a fault-free execution of the case study model.

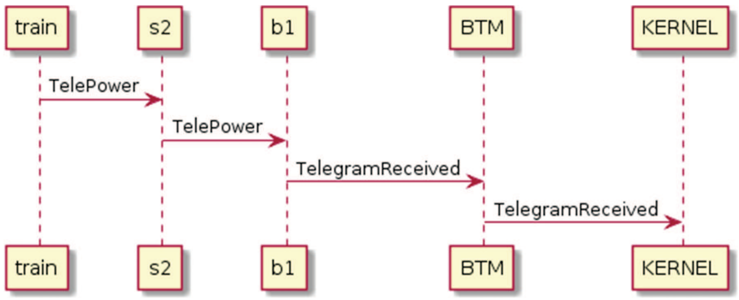

Figure 9.8 Visualization of a fault-free execution tree of the case study model.

trace can be visualised by PlantUML as a sequence diagram (Figure 9.8). In the modelled environment, there is a train, a track with two segments (s1, s2) and a balise (b1). The train initially stands on the first segment (s1) and it moves to the second (s2) as the first step of the test case. Note that for the sake of simplicity, the component representing the train in the case study is not associated with the balise component, so powering up a balise is mediated by the segment component.

9.3.2 Process Evaluation

As the discussion in section "Software-FMEA Using fUML/ALF" pointed out, the fundamental tenet of our method is to perform SW-FMEA on component-based systems through Alf execution (as of now, using an interpreter).

Based on the presented use case, process-wise, it is apparent that the SW-FMEA approach can be used in a "drop-in" fashion in existing safety processes, replacing classic approaches during software architecture analysis. The major added value is delivering much tighter bounds on error propagation characteristics (certainly not probabilities!) at the point in design where the major dependability mechanisms are most probably decided upon. While much more sophisticated than classic FMEA (and even such composable methods as HiP-HOPS [18]), the approach largely remains an FME(C)A – and thus there is no real reason it cannot be a candidate method in virtually all safety processes where SW-FMEA is necessary.

Importantly, the ability to "mix and match" specific errors and error categories in evaluating and propagating errors may enable new process patterns. Refinement of our knowledge of the error propagation characteristics in the system is a definite (and largely new) option in this setting; thus, in theory, safety arguments could very well evolve *cooperatively* with the refinement of system and software design. Future research will explore this possibility.

Certainly, there are some apparent drawbacks, too.

- **Modelling overhead**. The least significant drawback that nevertheless has to be mentioned is that the whole approach assumes that the system under design is created in an appropriate Model-Driven System Design (MDSD) workflow. Although MDSD is becoming the default in many industries where SW-FMEA has to be performed, it is not necessarily used in all settings.
- **Early definition of behaviour**. Executable models such as Alf give us the possibility to model behaviour early on in the design process – but this does not automatically mean that it is convenient or feasible at all. Further studies are necessary to evaluate this aspect.
- **Proof of behavioural equivalence**. When executable behavior is specified early on in the development process and it is the basis of safety arguments, behavioural equivalence of the final system (and components) with this early specification has to be maintained during development.
- **Simulation.** As of now, we use simulation for model evaluation. Simulation has its drawbacks; e.g., it is hard to assure that all execution paths have been exercised in a nondeterministic system. We argued in chapter "Software-FMEA through Alf execution" that in our case this is not a major issue. In fact, the proposed approach does not rely on any specific simulation technique; all the facilities that transform model execution into explicit error propagation execution are included in the model. This way, we will be able to reap the benefits of advances in fUML/Alf tooling without additional effort.

9.4 Implementation in a Blockly-based Modelling Tool

To demonstrate the general applicability of the approach presented so far, the main points of the framework were also implemented for the modelling tool introduced in "Chapter 4 – SYSML-UML like modeling environment based on Google Blockly customization".

The tool supports the modelling of static and dynamic aspects of component-based systems by using blocks, interfaces and connections to model structure, as well as sequence diagrams to model the collective behaviour of the whole system. The former aspect defines the participants and their relations, while the latter describes their interactions and the exchanged data. The basic block of behaviour is a *Service*, which may have a specific implementation in Python. Interactions then consist of *Service Requests* and control logic (e.g. decisions). Once modelled, the tool can visualize the connections in the system, as well as the sequence diagram defined for the global behaviour. One of the strongest aspects of the tool is the ability to generate a Python program for the simulation of the system. With small modifications, the generated code is an appropriate candidate for the methods defined in the previous sections.

9.4.1 Preparation of the Model

The case study model was again based on the balise-related basic functionality of ERTMS/ETCS (Figures 9.3 and 9.4 for the structure and Figure 9.7 for the behaviour). A bird's eye view of the model itself is presented in Figure 9.9.

The generated code has been augmented with logging: values of parameters and variables after assignment result of decisions and assertions as well as service requests were all output to build an execution trace.

As before, faults activation was done by injected, configurable logic that would determine which faults are activated during execution. In practice, this involves the replacement of certain constructs (e.g., expressions) with a function call that either performs the original behaviour (e.g., returns the value of the original expression) or alters the behaviour in some way (e.g., negates a condition). In the current case study, faults affected the assignment of Boolean and integer values (altering the value of the right-hand-side expression), the conditions in decisions and the sending of service requests (causing an *omission* fault).

The fault activation logic can be fine-tuned by setting the maximum number of active faults as well as if the faults are transient of permanent. In case of permanent faults, fault distribution is balanced by an initialization logic that

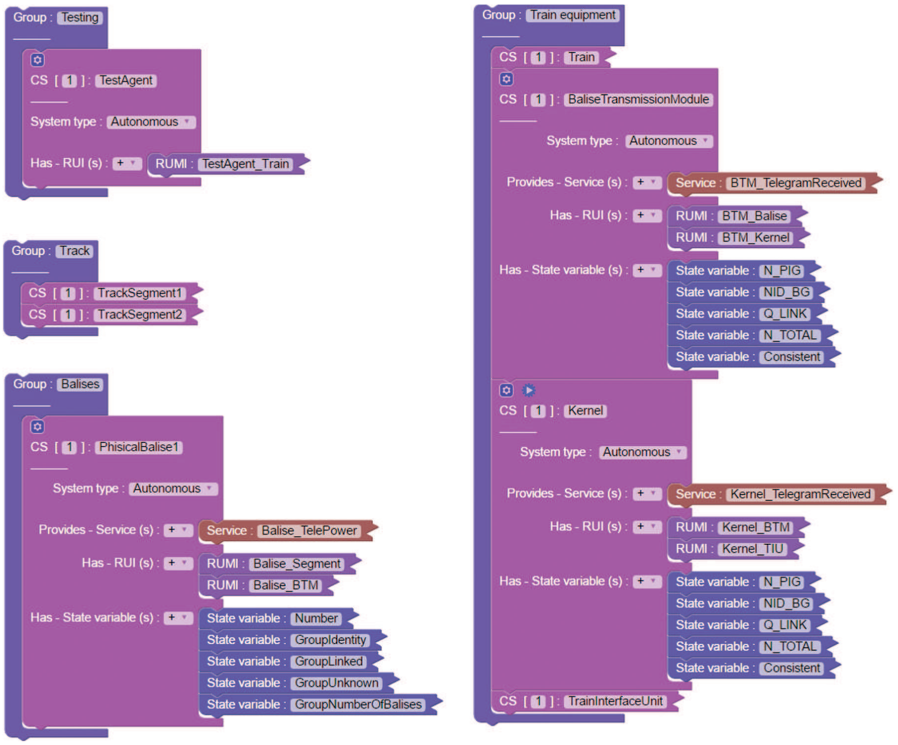

Figure 9.9 Blockly-based model of the case study system and its environment.

randomly selects a configured number of faults, which may then activate if the affected statements are executed (i.e. the injected fault activation function is called). In this case study, the logic was configured to activate *at most one* permanent fault.

Faults were injected in the relevant parts of the control logic of the train (i.e. the *Kernel* and the *Balise Transmission Module*), but message omissions were also included in the code simulating the powering of the balise to emulate failure of equipment. Every time a fault activated, its type and the affected line were logged, but not the specific value used to modify the original expression.

Two test cases were used for the simulations: in the first one, the balise sends a consistent telegram, while in the second; the balise has corrupted data (it has an invalid position value). Thus, according to the specified behaviour, correct reactions of the system would be to acknowledge the reception of the telegram in the first case, and applying emergency brake in the second.

Assertions in the model checked if the produced behaviour was in accordance with the balise data, as well as if a telegram was successfully processed in a given time frame (in more complex settings, this could be detected and handled when reaching the next balise).

9.4.2 Aggregation and Analysis of Traces

A single simulation of the fault-free model and 1000 simulations with random faults in each test case provided a satisfying number of traces to conduct a probabilistic analysis. The traces were processed through the following steps:

1. *Error traces:* Every faulty trace was compared to the reference (fault-free) trace to obtain the differences, i.e. to identify the chain of errors that led from a single fault activation to a failed assertion. Corresponding to the injected faults, the errors could be *Parameter errors*, *Data errors*, *Control errors* and *Missing calls.*
2. *Superposition of traces:* The error traces were merged to obtain a graph. An arc in this graph from A to B means that in some trace, error A was immediately followed by error B.
3. *Annotation with occurrences:* Arcs of the graph were then annotated by the ratio of the number of traces in which B has *eventually* occurred after A to the total number of traces in which A has occurred. This value corresponds to the *conditional probability* of eventually seeing B if A has occurred. This way, a probability of 1 means there is a strong correlation between errors A and B, which in this case may also suggest a causal relationship. Hence, these cases are visually distinguished by using a solid line for the potential causalities and a dashed line otherwise.
4. *Reducing noise:* Various techniques were employed to remove arcs that were the consequences of other relationships. It is worth to note that, this part of the process is the most theoretic and has a lot of room for improvements. The more efficient the techniques used here are, the more meaningful the results of the process can be.

Nodes of the graph (fault activations, errors and assertion failures) were grouped by the component which logged them. The output for the test case with the valid telegram can be seen on Figure 9.10. Things to node about the figure are the following.

- In an FMEA terminology, each box corresponding to a component contains the internal faults and the failure modes (errors) of the component. Arcs entering the box denote external failure modes that affect this

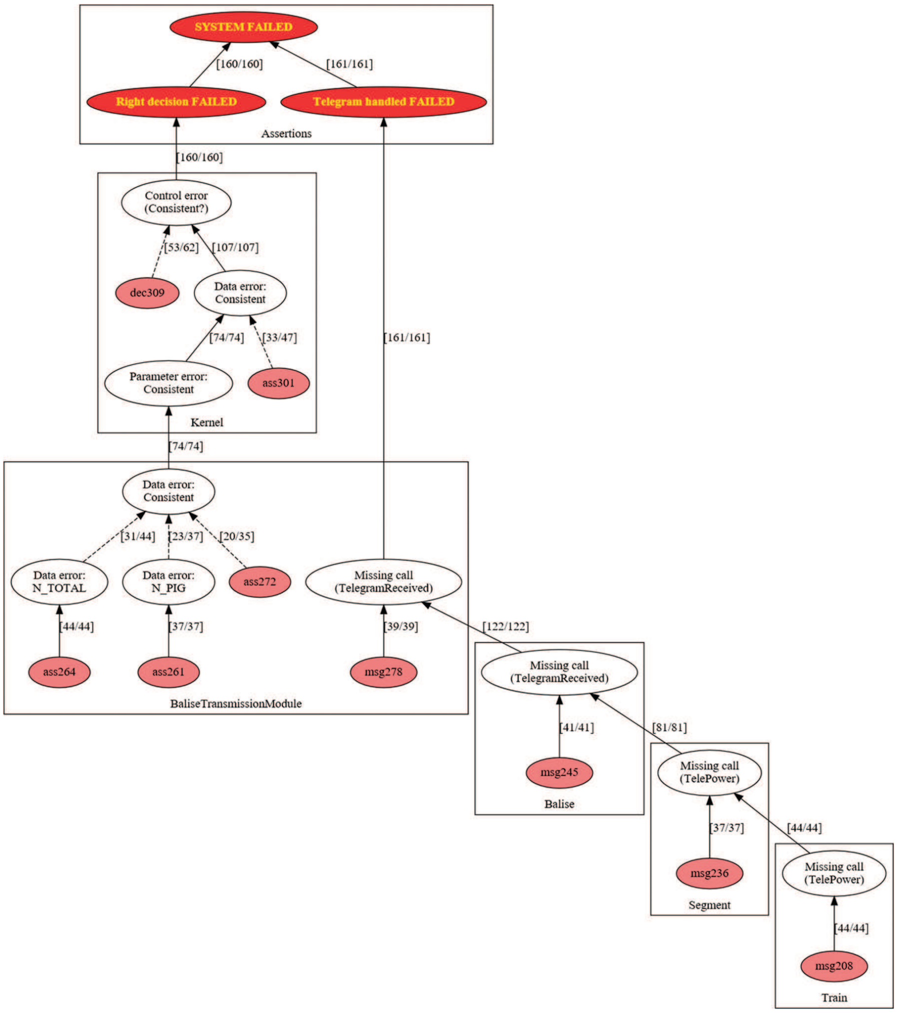

Figure 9.10 Error propagation in the case study model when input is consistent.

component, while outgoing arcs denote failure modes of the component that affect others.

- In the case study model, omission of service requests always results in a failure to handle the balise.
- On the other hand, corruption of data causes a system-level failure only if the value of the Boolean flag “Consistent” gets corrupted. Once this happens, there is no way to avoid failure, but only some corruption of the other values leads to the corruption of the flag.

- A fault affecting the value of “Consistent” does not always cause a data error, because always returning *True* is considered a correct answer in this scenario.

Analysis of the test case with the invalid telegram showed similar results.

9.5 Concluding Remarks

In the chapter, the reader was introduced to the main ideas of a novel approach to SW-FMEA for component-based systems that can be composed with existing safety processes. The method can replace or augment classic approaches during software architecture analysis and automating much of the traditional FMEA techniques.

The work transfers techniques well-known in academia into the SW-FMEA of safety-critical embedded systems, with strong potential applicability in other dependability-critical domains. These techniques include explicitly embedding fault activation logic and operational semantics into the interpreted model and constructing error automata from the specification of normal and abnormal behaviours (see e.g., [19]). At the same time, the presented approach promises to have a low effort overhead over producing the base models (that are produced in a model-driven process even in the absence of SW-FMEA); something that is sorely missing from manually performing SW-FMEA.

References

[1] Pataricza, A. (2007). “Systematic Generation of Dependability Cases from Functional Models,” in *Proceedings of the Symposium FORMS/FORMAT – Formal Methods for Automation and Safety in Railway and Automotive Systems*, Budapest, Hungary.

[2] Object Management Group. (2016). *Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.2.1.*

[3] Object Management Group. (2013). *Action Language for Foundational UML (Alf), version 1.0.1.*

[4] Object Management Group. (2015). *Precise Semantics of UML Composite Structures (PSCS), version 1.0.*

[5] GitHub. (2016). *Foundational UML Reference Implementation.* [Online].

[6] GitHub. (2016). *moliz – Model Execution Based on fUML* [Online].

[7] Seidewitz, E, and Tatibouet, J. (2015). “Combining Alf and UML in Modeling Tools – An Example with Papyrus,” in *OCL 2015 – 15th*

International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings, 105–119.

[8] Ciccozzi, F. (2014). *From Models to Code and Back: A Round-trip Approach for Model-driven Engineering of Embedded Systems*. Doctoral thesis, Mälardalen University, Sweden.

[9] Romero, G., Schneider, K., and Ferreira, M. G. V. (2014). "Using the base semantics given by fUML for verification," in *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)* (New York, NY: IEEE), 5–16.

[10] Schneider, A. S. and Treharne, H. (2011). "Towards a Practical Approach to Check UML/fUML Models Consistency Using CSP," in *Formal Methods and Software Engineering*, eds S. Qin and Z. Qiu (Berlin: Springer), 33–48.

[11] CECRIS. (2016). *CECRIS – Certification of Critical Systems, Grant Agreement no.: 324334, IAPP Marie Curie Action, 7th Framework Program.*

[12] Bonfiglio, V., Montecchi, L., Rossi, F., Lollini, P., Pataricza, A., and Bondavalli, A. (2015). "Executable Models to Support Automated Software FMEA," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering (HASE)* (New York, NY: IEEE), pp. 189–196.

[13] Object Management Group. (2011). *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1.*

[14] Mayerhofer, T. (2014). *Defining Executable Modeling Languages with fUML*. Doctoral thesis, Vienna University of Technology.

[15] European Railway Agency. (2014). "ERTMS/ETCS System Requirements Specification", SUBSET-26.

[16] Object Management Group. (2013). Semantics of a Foundational Subset for Executable UML Models (fUML).

[17] ModelDriven. (2016). ModelDriven.org: Action language for UML (Alf) open source implementation.

[18] Papadopoulos, Y., McDermid, J., Sasse, R., and Heiner G. (2001). Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliabil. Eng. Syst. Safety* 71, 229–247.

[19] Gallina, B., and Punnekkat, S. (2011). "FI4FA: A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures' Analysis," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)* (New York, NY: IEEE), 493–500.