Verification of multiple models of a safety-critical motor controller in railway systems

José Proença[1][0000−0003−0971−8919], Sina Borrami[2][0000−0003−1737−6509], Jorge Sanchez de Nova[2], David Pereira[1][0000−0002−7561−6649], and Giann Spilere Nandi[1][0000−0002−3206−0599]

[1] CISTER, Polytechnic Institute of Porto, Portugal
{pro,drp,giann}@isep.ipp.pt
[2] Alstom SA
{sina.borrami,jorge.sanchez-de-nova}@alstomgroup.com

Abstract. Motor controllers, such as the ones used in signalling systems, include critical embedded software. Alstom is a company that produces such embedded systems, which must follow complex certification processes that require formal modelling and analysis. The formal analysis of these real-time systems have to balance between including enough details to be useful and abstracting away enough details to be verifiable. This paper describes our work in the context of the European VALU3S project to integrate the analysis of such systems with the Uppaal model checker during the development cycle, involving both developers from Alstom and academic partners. We use special Excel tables to configure the underlying Uppaal models and requirements, bridging these two stakeholders. We follow Software Product Line Engineering principles, e.g., allowing features to be turned on and off and periodicities to be changed, and verify different properties for each of such configuration. We automate the instantiation and verification in Uppaal of a set of selected configurations via an open-source prototype tool named *Uppex*.

Keywords: Verification, Variability, Railway, Real-Time Automata

1 Introduction

In railway systems, motor controllers play a crucial and safety-critical role in point switch machines. Guaranteeing its correct design and development is a challenging but essential task to avoid catastrophic accidents that could cause severe damage to the environment and property, or even result in the loss of human lives. Most state of the art approaches address this safety concerns using formal modelling and verification, including abstract interpretation [15] and Event B [1,7], to enforce compliance with certification processes and railway-specific safety standards, such as EN-50126 [10], EN-50128 [11], and EN-50129 [12]. In these systems, safety means that faults are detected with very high probability, leading to a fallback state.

The design of motor controllers is usually performed by multidisciplinary teams composed of experts in hardware, embedded software, and verification.

Guaranteeing that all stakeholders with different backgrounds have the same understanding of the critical aspects of the system development can be challenging. We model the behaviour of a railway motor controller using the Uppaal model checker [8], in the context of the European project VALU3S (https://valu3s.eu). This paper reports on how we integrate and automate the formal verification of this controller during its development by the rail manufacturer Alstom, while improving the trade-off between fine-grained details in the formal models and its verifiability, and efficiently involving all team members in this process.

Our use-case uses a controller with software components that interact with a dashboard and a circuit board (Fig. 1). Intermediate components are used to poll the circuit, to add and verify CRC error codes, etc. We compiled a set of safety requirements for the controller's software to be verified using model checking. However, when trying to build a network of automata to model the controller with enough details to cover all requirements, we concluded that it generated a state-space too large to be feasible when model-checking. For example, the requirement *"the controller component should take less than 100ms to send a given command to the circuit"* should not need to consider all combinations of states involving the sending of messages to the dashboard. Similarly, the requirement *"if the controller component receives an error message it should go to a fallback state and the dashboard should be informed within 100ms"* should not need to consider the mechanisms to interact with the circuit.

This lead to a family of formal models with different parameters and levels of detail, each targetting different requirements. This lead us to 3 challenges: **C1: maintain the model**, to kept it up-to-date with the system under development; **C2: manage variability**, as too many models with commonalities are needed; and **C3: improve the collaboration** between developers and modellers of the formal specifications.

Our approach uses a high-level representation of the configurations of the family of formal models for real-time systems. This representation consists of Microsoft Excel spreadsheets with parameters and requirements to be used in the formal models, read by our prototype tool *Uppex* that automatically generates and verifies the full family of models and requirements. These spreadsheets include, for example, the time-bounds of certain components, the size of buffers, and the initial values of certain variables. Furthermore, these values vary according to the set of active *features*; for example, by activating a feature named *SelfTesting*, a variable named `TSelfTest` is set to 200, otherwise it is set to 0. A special table compiles a set of *configurations*, each listing its active features. For example, a given configuration could activate *SelfTesting*, deactivate unrelated monitoring features, and activate its associated requirements.

Organisation of the paper. Section 2 describes the motor controller use-case and its requirements, formalised in Section 3 using the Uppaal model checker. Section 4 describes how we configure and verify many variations of a Uppaal model. Section 5 summarises what we have learned during this process and the plans for future developments, and Section 6 concludes this paper.

Fig. 1. Architecture of the motor controller system under verification

2 Use-case: Motor controller

Our running use-case consists of a motor controller, or *controller* for short, running in a resource constraint device with a Real Time OS. This controller is connected both to a physical *circuit* and to a *dashboard*, as depicted in Fig. 1. The circuit includes a DC motor that is being controlled, receives simple commands from the controller to turn left, turn right, or to stop, and sends back a status report, including the information of whether the limit of a rotation has been reached or if a problem has been found. The dashboard sends instructions to the controller, including commands to be sent to the circuit, which in turn informs the dashboard of internal state updates.

We focus on the behaviour of the software part of the controller, and on its formal verification via model-checking of timed-behaviour. This is complementary to other analysis and tests performed by other stakeholders involved in the same use-case, e.g., to inject faults in hardware and to generate batches of tests with enough coverage. We expect our underlying formalizations and tools to also benefit, directly or after repurposed, the other stakeholders in this use-case.

This paper includes behavioural details only of the core controller component, and the full Uppaal models are not publicly available since they are intellectual property of Alstom.[3] We believe that these descriptions, supported by our open-source prototype tool, are rich enough to convey our approach and its benefits.

Safety-critical behaviour Hazard analysis for the controller has been performed to justify the desired criticality levels. This analysis guided the architecture of the software components deployed on the controller board. Most components are replicated and executed in two diversified processing units available in the selected board, to detect when their behaviour diverges. Also, CRC codes are applied to incoming and outgoing packages to ensure message consistency.

The replicated components are: a core *controller*, a *monitor* to check if the state of the controllers are consistent, a *decoder* to compare incoming messages against their CRC error code and against the messages from the neighbour decoder, a *buffer* to store messages to be sent to the dashboard, an *encoder* to add CRC codes to messages to be sent to the dashboard, and a *reader* of messages received from the circuit. Non-replicated components are: a *scheduler* to start runtime self-tests, a simulator of the dashboard, a simulator of the circuit, and a fault-injector to cause some components to fail. The simulators exist only on the

[3] These can be made available to the reviewers if needed.

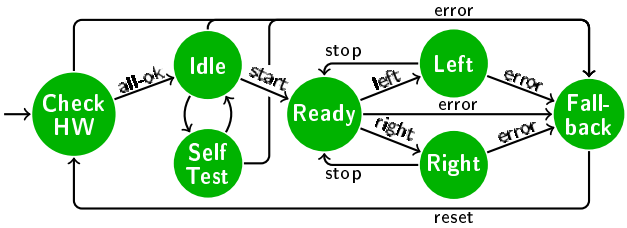

Fig. 2. General behaviour of the controller component

formal models, to mimic the environment, while capturing the minimum information required to perform formal analysis, represented as predefined sequences of messages to be sent.

The behaviour of the core controller task is depicted in Fig. 2. The controller performs some initialisation in Check-HW, tests the interaction with the circuit in Self-Test, and can trigger the rotation of the motor to the left or to the right. At any moment, it can receive an error and go to a Fall-back state.

Parameterised requirements Following the hazard analysis, we compiled a set of requirements to be verified using model checking based on Uppaal. The most relevant ones are listed in Table 1. Requirements follow some syntactic structure to tighten the gap between formal and informal requirements, following the EARS approach [17]. For example, the 3rd requirement reads *"**In** $Conf_3$, **when** $controller_1$ fails **the** $controller_2$ **shall** go to a fallback state within 100ms."* Configurations specify the parameters of the model when validating the requirement. This covers both general parameters of the system, such as the time to decode messages and the frequency of operation of monitors, and the scenario consisting of the messages sent by the dashboard, by the circuit, and by the fault-injector. In our example $Conf_3$ defines a scenario where the dashboard sends a `start` and a `left` command after 20ms and 100ms, respectively, and the fault-injector causes $controller_1$ to fail after 120ms.

Table 1. Some functional and non-functional requirements for the motor controller

Config.	State	Trigger	Comp.	Expected
In $Conf_1$	while $controller_1$ is ready	when decoder receives a `left` command	the $controller_1$	shall send a `left` command within 100ms
In $Conf_2$		when $monitor_1$ or $reader_1$ fail	the $controller_2$	shall go to a fallback state within 100ms
In $Conf_3$		when $controller_1$ fails	the $controller_2$	shall go to a fallback state within 100ms
In $Conf_4$		when $controller_1$ receives an error message	the $controller_1$	shall send immediately a `stop` command to the circuit
In $Conf_4$		when $controller_1$ receives an error message	the $encoder_1$	shall notify the dashboard within 100ms
In $Conf_5$	while dashboard can send messages		the full system	shall never get stuck

When formalising requirements (c.f. Table 1) using the temporal logic supported by Uppaal, the notions of *state*, *component*, and *expected observation* followed in a relatively straightforward manner. Specifying the *triggers* often required manually enriching the model with new variables, since the logic does not express events. Specifying *configurations* were the most complex operations, and the core challenge addressed by this paper and our prototype tool. Traditionally for each configuration a new model would have to be specified, fine-tuning values of variables spread throughout the model, often deactivating some components to simplify the model-checking of more complex properties. Maintaining a collection of such models, in a context where neither the system specification nor the full set of requirements are fixed, quickly becomes infeasible. We provide support to specify all configurations and properties in a single Excel file, and to automatically use these with a single annotated Uppaal model.

3 Formal specification in Uppaal

Uppaal [8] is a well-known model-checker for real-time systems, successfully used in many industrial applications and in the context of embedded systems [5]. Systems are specified as a set of timed-automata that interact both by using synchronisation on actions and by using shared variables. In a nutshell, each timed-automaton is a state machine whose edges are labelled by a guard and an update over shared variables, and by an optional action name used to synchronise with neighbour automata. Special variables named clocks capture the time that has passed since they were last reset, and are incremented automatically by the rules that guide the automata evolution.

The topology of the timed automata network used in the specification of our use-case is depicted in Fig. 3, one for each task mentioned in Section 2. This topology is built iteratively by both developers and formal modellers, during the development of the system. Each node depicts the timed-automaton of a component, and arrows depict interactions between nodes: $\longrightarrow$ denote synchronous interactions that block until both automata can trigger the associated action; $\dashrightarrow$ and $\cdots\!\!\rightarrow$ denote synchronous interactions that do not block the sender – the former requires the receiver to be always ready and the latter discards data if the receiver is not ready; and $\Longrightarrow$ denotes asynchronous communication by atomic writes and reads to a shared variable.

The dashboard, circuit, and fault-injector components are parameterised by a scenario, i.e., a sequence of actions with timestamps. The dashboard sends commands to the encoders, the circuit sends reports to the readers describing if there are errors and if the motor reached a limit, and the fault-injector sends messages that cause some components to go to a faulty state with no behaviour. Furthermore, the circuit reports errors for a predefined time-window during the self-test phase, and the controllers validate that an error is indeed reported.

The behaviour of the components involved is expressed using Uppaal's notion of timed automata. We depict the automata of the controller's behaviour in

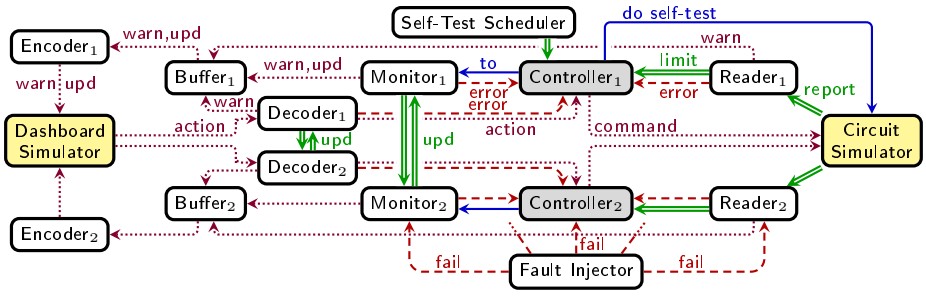

Fig. 3. Topology of the network of communicating timed-automata of the use-case

Fig. 4. All the 5 states of Fig. 2 appear in this automata, extended with extra details. The arrows pointing to and from the Controllers in Fig. 3 appear in this diagram either as channels in the labels, represented by names prefixed with '?' or '!', or as shared variables such as limit, which is read to detect if the motor reached its target position. The non-blocking behaviour of the error and fail channels is captured by including an extra transition labelled by this channel in every node where time can pass.

Uppaal supports imperative code using a C-like language inside a global *Declarations* block, accessible by all automata in the network. These variables and functions can be used by the expressions in the timed-automata. For example, the concrete actions (e.g., goLeft), time-bounds ((e.g., TLeft[id][max])), shared variables (e.g., limit), and channel names (e.g., action) are declared in this block.

4 Parameterisation and verification with Uppex

In order to cope with the multiple configurations of Uppaal's models, we developed **Uppex** to provide a mechanism based on annotations to customise many aspects, including channels, shared variables, data types, time-bounds, and requirements. Uppex is an open-source tool that uses the workflow depicted in Fig. 5: it reads both an Excel file with the configurations and an Uppaal file with annotations, and it creates a new Uppaal model for each configuration found. Either one of the new models is used to replace the original Uppaal file, or they are verified by Uppaal and a report is produced. Uppex is developed in Scala, uses the Apache POI libraries for Microsoft documents [13], and is available at https://cister-labs.github.io/uppex.

4.1 Annotating Uppaal models

Declarations in the input Uppaal model are annotated with special blocks starting with "// @Name", which act as hooks that Uppex uses to inject and update the values that configure the model. XML blocks from "<Name>" until "</Name>" also act as hooks for annotations, which we use to inject and update the properties being verified in the <queries> block. We call these *@-annotations* and *xml-annotations*, respectively.

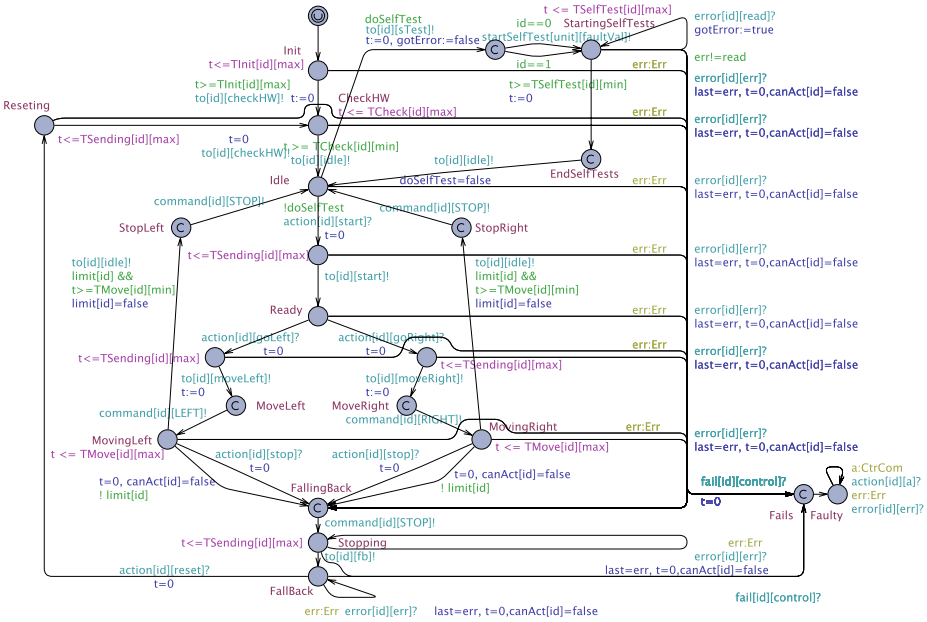

Fig. 4. Specification in Uppaal of the a controllers' timed-automata with identifier `id`

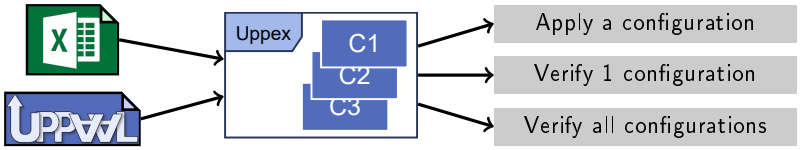

Fig. 5. Uppex workflow: updating and verifying models based on configuration tables

Each annotation can be defined in the Excel file in a sheet named with the same name (c.f. Fig. 6). The first line of these sheets describe the pattern used to produce code that will be injected for each line of the table, followed by a table with a *header* of names in row 2 and their values below. E.g., in the `@TimeBounds` table (Fig. 6), row 4 injects the line "`const int TCheck[Ids][Intrv] = {{4,4},{6,6}};...`" to the Uppaal code in the corresponding block. The first column acts as unique identifier: if multiple lines are found, the last one prevails. The column named `Features` associates feature names that must be *active*, otherwise the line is discarded. In our example, when the feature `SelfTesting` is active the variable for `SelfTest` is set to 200, otherwise it is set to 0. The `<queries>` table on the top-left of Fig. 6 depicts some of the requirements from Table 1.

4.2 Verifying multiple configurations

Using Uppex it is possible to specify a list of *configurations*, each regarded as a set of features that can be active or not. These feature selection guides which rows from

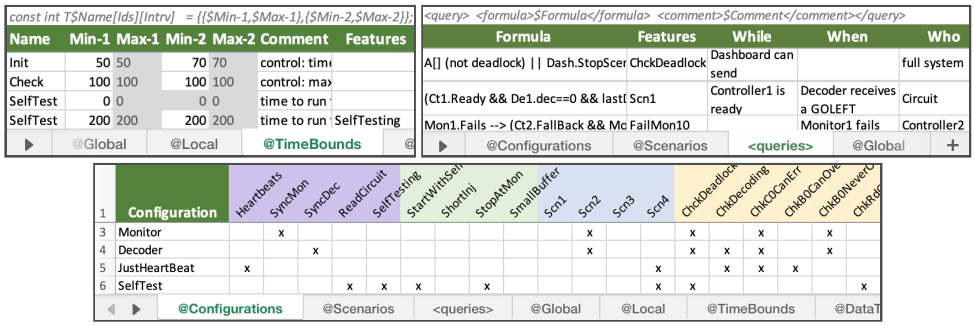

Fig. 6. Special Excel tables: @-annotation, xml-annotation, and configurations

annotation should be included. The list of configurations is specified in an Excel sheet named `@Configurations`, such as the one in the bottom of Fig. 6. In this example the configuration `SelfTest` includes the features `ReadCircuit`, `SelfTesting`, and `StartWithSelfTest`, among others, and not the feature `SyncMon` nor `Heartbeats`. Hence, when selecting the `SelfTest` configuration, the `SelfTesting` will be active, triggering the last row visible in the `@TimeBounds` table to be used to define the `SelfTest` variable. When selecting instead the configuration `JustHeartBeat`, the `SelfTesting` feature will not be active, thus the previous row will be used instead. Similarly, the selected features will also influence which queries will be used during verification.

Uppex can be used as a command line tool to modify the annotated blocks of an Uppaal model according to a given configuration, or to verify one or all configurations. For example, the command "`java -jar uppex.jar -runAll motorController.xlsx`" will verify all configurations in the given Excel file using the Uppaal model with the same name, producing a report such as the one in Table 2. This report states that 3 properties of configuration `SelfTest` passed and the verification timed-out while verifying the 4th property. This property would pass using a slighly larger timeout when calling Uppex. We write ellipsis '...' to omit parts of the report. Configurations `Monitor` and `JustHeartBeat` also passed and failed some properties.

Table 2. Report produced when verifying all properties and all configurations

```
> Reading Uppaal file 'motorController.xml'

--Verifying 'SelfTest'--
[OK] @SelfTest: the Controller1 shall be able to
     start the self-tests.
[OK] @ChkSelfTest1: when Self-test ends, the Reader
     shall have received some error.
[OK] @ChckDeadlock: while Dashboard can send
     messages, the full system shall not deadlock.
| Time-out. Missing 1 properties.
| Failed on property:
| "@SelfTest, StartWithSelfTest: the Controller1
     shall be able to end the self-tests."

--Verifying 'Monitor'--
[OK] @SyncMon: the Monitor1 shall be able
     to send a warning.
[OK] @ChkB0NeverOverflows: the Buffer1
     shall never overflow.
...
--Verifying 'JustHeartBeat'--
[OK] @ChkDecoding: the Decoder1 shall be
     able to send a warning.
[FAIL] @ChkB0NeverOverflows: the Buffer1
     shall never overflow.
...
```

5 Lessons learned and future work

During the development of the motor controller system at Alstom in collaboration with ISEP and other academic partners, we iterated over core design architectural decisions and agreed upon different synchronisation mechanisms. Using the model-checking capabilities of Uppaal, we verified different properties, including the possibility of sending warnings, of buffer overflows, and of reaching deadlocks (or timelocks). These models are useful both to predict possible problems and bottlenecks, and to be used in certification processes. Our configuration-driven approach using Excel spreadsheets emerged as a solution to the growth in complexity of the underlying formal models, which typically must remain simple in order to be useful. We were able to find time-bounds that satisfy our requirements, e.g., the periodicity at which monitors and decoders check consistency, or the periodicity at which reports should be polled from the circuit, under different scenarios simulated by the `dashboard`.

Uppex adds a negligible overhead over the model-checking process, involving the parsing of the configuration tables and the Uppaal file, and the writing of an updated set of Uppaal files. In our use-case we use the 16 automata from Fig. 1 in a file with ∼1.7K lines excluding queries. Our tables currently include around 25 requirements, 15 configurations, and 135 different entries (including scenarios, time parameters, data channels, and data constructors). Invoking Java to produce a concrete instance takes less than 5s in our 1.4 GHz Quad-Core Intel Core i5 machine.

Related work The verification of complex embedded systems has been investigated, e.g., by Basten et al. [3] who generate Uppaal models (and Petri net models) using a model-driven approach with the Octopus toolset, focusing on design-space exploration and schedule optimisation. Gario et al. [14] and Dureja and Rozier [9] provide an exhaustive analysis of a large air traffic control, in a joint effort with NASA team of engineers, using 3 concrete models specified in the OCRA architectural language with SMV component models. They validate the 3 models using a combination of different techniques based on the property at hand, and analyse dependencies among properties to avoid the verification of unnecessary queries. In contrast to these approaches, Uppex allows the manual definition and fine-tuning of models in the host model-checker instead of using generated models, and provides mechanisms to control the variability of the models in a way that can be perceived by both tool- and formal-developers.

The variability in Uppex is given as a set of tables that inject code in the annotated specifications, but it is not reasoned upon. Other approaches, such as the formal framework by Kim et al. [16] in the context of embedded systems, can be used to analyse valid configurations based on feature models [4].

Future work We are pursuing the following two directions of work.

1. Valid configurations. Currently one can specify any combination of features, sometimes leading to incorrect configurations because of missing dependencies or incompatibilities. These restrictions can be captured by a set of constraints, usually taking the form of a Feature Model [4] in the context of Software Product Lines. One could, for example, make the feature `StartWithSelfTest` dependent on `SelfTesting`, marking any configuration with only the first one as invalid. Following existing work in this community, we could further exploit these validity constraints over features, e.g., by considering all configurations that satisfy these constraints, or to aim at finding the

best configuration using some cost function. In the context of this work, the properties validated by Uppaal could also play a role in the validity of a configuration.

2. Other backends. Our work targets Uppaal models using a frontend for developers based on Excel spreadsheets. However, these tables can also be used with different backends besides Uppaal. For example, to generate configuration files used in the implementation, or to use a different model-checker for verification, such as Imitator [2] for real-time systems, which supports the optimisation of some parameters, or mCRL2 [6] that supports a temporal logic over events and can handle very large state-spaces. We are also working on an intermediate domain specific language that can generate Uppaal models, among other analysis, with a better support to reason over the architectural topology, such as the one in Fig. 3, which emerges only implicitly in Uppaal.

Uppaal is free to use only for non-commercial purposes. It is currently being used by academic partners, and our use-case is not being commercialised and is representative of other ongoing projects. This work may lead to the adoption of Uppaal in commercial projects of Alstom, or to a different backend supported by Uppex.

6 Conclusions

This paper presents our approach to formalise the timed-behaviour in Uppaal of a motor controller system, under development by the Alstom railway company, in the context of the VALU3S European project. We use parameterised configuration tables that adapt a core Uppaal model, facilitating the customisation of the model so it can better suit different requirements. This paper also describes how we integrated the usage of model-checking within the development cycle of a safety-critical system, involving stakeholders with different background, relying on intelligible tables and architectural topologies. We produced a prototype open-source tool Uppex to automatise the extraction of parameters and adaptation of the formal models, and to verify many configurations on a single run. In the future we plan to further exploit the validity of configurations and to experiment with different backends.

Acknowledgments. This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UIDP/UIDB/04234/2020); also by the Norte Portugal Regional Operational Programme (NORTE 2020) under the Portugal 2020 Partnership Agreement, through the European Regional Development Fund (ERDF) and also by national funds through the FCT, within project NORTE-01-0145-FEDER-028550 (REASSURE); also by COMPETE 2020 under the PT2020 Partnership Agreement, through ERDF, and by national funds through the FCT, within project POCI-01-0145-FEDER-029946 (DaVinci); also by FCT within project ECSEL/0016/2019 and from the ECSEL Joint Undertaking (JU) under grant agreement No 876852 (VALU3S). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

References

1. Robert Abo and Laurent Voisin. Formal implementation of data validation for railway safety-related systems with ovado. In Steve Counsell and Manuel Núñez, editors, *SEFM*, pages 221–236. Springer, 2014.

2. Étienne André. IMITATOR 3: Synthesis of timing parameters beyond decidability. In Alexandra Silva and K. Rustan M. Leino, editors, *CAV 2021*, volume 12759 of *LNCS*, pages 552–565. Springer, 2021.
3. Twan Basten, Emiel van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian de Smet, Lou J. Somers, and Egbert Teeselink. Model-driven design-space exploration for embedded systems: The octopus toolset. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA 2010*, volume 6415 of *LNCS*, pages 90–105. Springer, 2010.
4. David Benavides, Sergio Segura, and Antonio Ruiz Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.
5. Timothy Bourke and Arcot Sowmya. Automatically transforming and relating Uppaal models of embedded systems. In Luca de Alfaro and Jens Palsberg, editors, *EMSOFT 2008*, pages 59–68. ACM, 2008.
6. Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In Tomás Vojnar and Lijun Zhang, editors, *TACAS 2019*, volume 11428 of *LNCS*, pages 21–39. Springer, 2019.
7. Mathieu Comptier, Michael Leuschel, Luis-Fernando Mejia, Julien Molinero Perez, and Mareike Mutz. Property-based modelling and validation of a CBTC zone controller in event-b. In Simon Collart Dutilleul, Thierry Lecomte, and Alexander B. Romanovsky, editors, *RSSRail 2019*, volume 11495 of *LNCS*, pages 202–212. Springer, 2019.
8. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *STTT*, 17(4):397–415, Aug 2015.
9. Rohit Dureja and Kristin Yvonne Rozier. More scalable LTL model checking via discovering design-space dependencies (d^3). In Dirk Beyer and Marieke Huisman, editors, *TACAS 2018*, volume 10805 of *LNCS*, pages 309–327. Springer, 2018.
10. Railway Applications. The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS). Generic RAMS Process. Standard (N), CENELEC, Dec. 2017.
11. Railway applications. Communication, signalling and processing systems - Software for railway control and protection systems. Standard (N), CENELEC, Jul. 2020.
12. Railway applications. Communication, signalling and processing systems. Safety related electronic systems for signalling. Standard (N), CENELEC, Nov. 2018.
13. Apache Software Foundation. Apache POI - the Java API for Microsoft documents. https://poi.apache.org, 2021. Accessed: 2021-11-30.
14. Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, and Kristin Yvonne Rozier. Model checking at scale: Automated air traffic control design space exploration. In Swarat Chaudhuri and Azadeh Farzan, editors, *CAV 2016*, volume 9780 of *LNCS*, pages 3–22. Springer, 2016.
15. Daniel Kästner and Christian Ferdinand. Applying abstract interpretation to verify EN-50128 software safety requirements. In Thierry Lecomte, Ralf Pinger, and Alexander Romanovsky, editors, *RSSRail 2016*, pages 191–202. Springer, 2016.
16. Jin Hyun Kim, Axel Legay, Louis-Marie Traonouez, Mathieu Acher, and Sungwon Kang. A formal modeling and analysis framework for software product line of preemptive real-time systems. In Sascha Ossowski, editor, *SAC 2016*, pages 1562–1565. ACM, 2016.
17. Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (EARS). In *RE 2009*, pages 317–322. IEEE Computer Society, 2009.