From Temporal Models to Property-based Testing

Nasser Alzahrani, Maria Spichkova and Jan Olaf Blech
RMIT University, Melbourne, Australia

Keywords: Formal Methods, Property based Testing.

Abstract: This paper presents a framework to apply property-based testing (PBT) on top of temporal formal models. The aim of this work is to help software engineers to understand temporal models that are presented formally and to make use of the advantages of formal methods: the core time-based constructs of a formal method are schematically translated to the BeSpaceD extension of the Scala programming language. This allows us to have an executable Scala code that corresponds to the formal model, as well as to perform PBT of the models functionality. To model temporal properties of the systems, in the current work we focus on two formal languages, TLA+ and FocusST.

1 INTRODUCTION

Safety-critical systems, e.g., in the automotive domain (Kühnel and Spichkova, 2007), become more and more software-intensive with every year. While specifying such systems, a precise formal model, i.e., a mathematical model at some level of abstraction, might be essential to eliminate ambiguity and to detect possible errors early in the software development life-cycle (SDL). Also, in most cases the system properties have to be analysed in relation to the time, thus, verification/testing of the temporal aspects is crucial.

To achieve the integration of formal models into SDL, the development process should be human-oriented. Thus, aspects of human factors engineering should be taken into account, cf. (Spichkova et al., 2015). Moreover, using Formal Methods (FMs) can be beneficial while developing not only safety-critical systems, but also web services, cf. (Newcombe et al., 2015). FMs were successfully applied to design and analyse systems since many years, cf. (Bowen and Hinchey, 1995; Yu et al., 1999). Despite all the advantages of FMs, software engineers are not keen to include them into the software development process. This problem was discussed 15-20 years ago, e.g., in (Hinchey, 2003). This problem is still unsolved now. Lack of readability and usability is one of the reasons for very limited use of FMs in industrial projects (Zamansky et al., 2016). However, in some cases even simply implementable improvements can make an FM more readable and understandable, cf. (Spichkova, 2012).

In many cases, FMs require huge amount of training, as they use a very specific syntax that is unreadable for novices. In general, testing approaches are perceived by practitioners as more appropriate for a real-life development process. However, they are usually comfortable with concepts from property-based testing (PBT), which require a little bit of mathematical thinking. PBT approach allows to use randomly generated test cases based on properties to test systems against their specifications.

To led programmers in formulating and testing properties of programs, Claessen and Hughes introduced a tool named *QuickCheck* that is focusing on Haskell programming language. They demonstrated that *QuickCheck* allowed them to discover hundreds of bugs, e.g., DropBox file sharing service (Claessen and Hughes, 2011; Hughes, 2010). In its first edition, *QuickCheck* was proposed as a testing framework for testing only functional programs. However, recent development in the area of PBT incorporates the statefulness of systems. That provides functionality for the testing of state-ful systems as well as for testing programs written in imperative languages, e.g., C (Gerdes et al., 2015; Hughes, 2010).

We propose to apply PBT on top of temporal formal models. This might help software engineers to understand temporal formal models (which describe the state of a system in every discrete time point), as the FM constructs will be expressed in terms of system code. This might contribute to the understandability of FMs indirectly, and allow software engineers

Alzahrani, N., Spichkova, M. and Blech, J.
From Temporal Models to Property-based Testing.
DOI: 10.5220/0006340302410246
In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2017)*, pages 241-246
ISBN: 978-989-758-250-9

to make use of the advantages of FMs. To achieve this goal, we suggest to translate the core time-based constructs of an FM to the BeSpaceD extension of the Scala programming language, specified in (Blech and Schmidt, 2014). This allows us to have an executable Scala code that corresponds to the formal model, as well as to perform PBT of the models functionality. To model temporal properties of the systems, in the current work we focus on two formal languages, Temporal logic of actions (TLA+) and FOCUSST. TLA+ combines temporal logic with a logic of actions, and is used to describe behaviours of concurrent systems, cf. (Lamport, 1994; Lamport, 1993). FOCUSST is a formal language providing concise but easily understandable specifications that is focused on timing and spatial aspects of the system behaviour (Spichkova et al., 2014; Spichkova, 2007).

To implement the proposed ideas, we selected Scala programming language, as on the PBT level this allows us to apply an extension to *ScalaCheck library*. Early ideas of this approach was presented at Software Technologies: Applications and Foundations Conference, cf. (Alzahrani et al., 2016). In this paper we go further and discuss the developed framework and how it can be applies to TLA+ and FOCUSST. This approach is based on a completed Minor Master Thesis of the first author.

2 PROPOSED FRAMEWORK

Figure 1 depicts the proposed framework that will allow for combining FMs with PBT. The general idea is to start with specifying the system using human-oriented modelling techniques based on FMs. After the specification phase, the software of the system under test is designed according to the specification. The framework will then generate random test cases to exercise and verify that the system runs according to the specification. If a test fails, it will be the judgment of the engineer to decide whether the errors were in the system software or in the specification formulas for which the system was not correctly specified. If the test passes without any errors, the system under test meets the specification.

The FM specification gets translated to host programming language (Scala in this case). These specification gets formal verification depending on the flavour of FM being used. For example, in case of TLA+, the TLA+ model checker (TLC) is used to check the specification. On the other hand, in case of FOCUSST, the theorem prover Isabelle/HOL via the framework *Focus on Isabelle* is used to verify systems specification, cf. (Nipkow et al., 2002) and (Spichkova, 2007).

The workflow within the proposed framework includes the following steps:

- To create an (informal) requirements specification of the system;
- To transform the informal specification to a formal specification (model) of the system, using TLA+ or FOCUSST;
- To verify formal model, using TLA+ model checker or Isabelle/HOL theorem prover, respectively;
- To translate the formal model to Scala using the provided translation schema;
- To add the specified in Scala model to the extended ScalaCheck library;
- To check the extended ScalaCheck library against the behaviour generated by FM specification.

In this section, we show the applicability of the proposed framework to TLA+ and FOCUSST. The goal is to demonstrate how the proposed framework can be applied to many types of FMs with similar syntax. Each subsection presents systematic informal program transformation schemas. Using these schemas makes transforming FM formulas to any hosting language, Scala in this case, an easy mechanical task. We start by analysing TLA+ syntax and semantics. After that, we show the design and model the API for the TLA+ flavour. After that, we show the designed API and the testing it using small example (*One Bit Block*). Similar process applied to FOCUSST, showing the analysis of FOCUSST syntax and semantics. Restricting FOCUSST to it's major parts that is related to temporal properties.

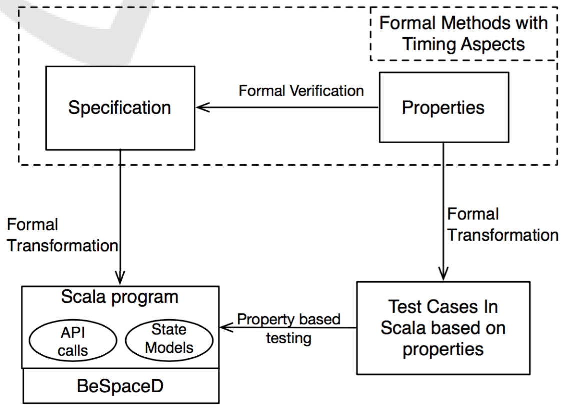


Figure 1: Proposed Framework.

2.1 Application to TLA+

TLA provides a toolbox which includes an integrated development environment (IDE) for the TLA+.

The IDE allows create and edit specifications, it also shows parsing errors and can be used to turn TLA+ model checker. To decrease the cognitive load of the developer and tester, it also includes an error trace viewer and explorer: these components provide a structured view of the states, illustrate how the states/values are changed at each step, and allow to run the TLA+ proof system.

A TLA formula such as $Init \wedge \Box[Next]v$ specifies the initial states and the allowed transitions of a system. It allows for transitions that do not change the value of v. This kind of transitions is called *stuttering transitions*. Most TLA system specifications are of the form $Init \wedge \Box[Next]_v \wedge L$. The semantics of such formulas are shown in Table 1. Table 2 shows logic operators in TLA+ and their mappings in Scala, many of the logical operators in Scala are provided by BeSpaceD.

Table 1: Semantics of TLA formula.

Init	State formula describing the initial state(s)
Next	Action formula formalizing the transition relation – usually a disjunction $A1 \vee .. \vee An$ of possible actions (events) Ai
L	Temporal formula asserting liveness conditions

Table 2: Operator mapping from TLA+ to Scala.

TLA+	Scala
/\	AND
\/	OR
$\Rightarrow$	IMPLIES
TRUE	TRUE
FALSE	FALSE
BOOLEAN	Boolean
$\{TRUE, FALSE\}$	List(TRUE, FALSE)
$\leq$	lessThanEq
$\geq$	greaterThanEq
$>$	greaterThan
$<$	lessThan
$\nleq$	lessThanEqNot
$\nless$	lessThanNot
$\ngeq$	greaterThanEqNot
$\ngtr$	greaterThanNot
$\in$	IN
$x == e$	defined(x, e)
$x = e$	assign(x, e)
$\forall x \in S : p$	for $\{x \leftarrow S$; if p$\}$ yield x
$\exists x \in S : p$	exists(x, S, p)
CHOOSE $x \in S$	choose(x, List(S))

In TLA+, a representation of an abstraction of a system is modelled using the standard model. The Standard Model states that an abstract system is described as a collection of behaviours, each representing a possible execution of the system, where a behaviour is a sequence of states and a state is an assignment of values to variables. In this model, an event (step) is the transition from one state to the next in a behaviour. For example, In one-bit clock, formulas are defined as follows:

```
VARIABLE b
Init ==  (b = 0) \/ (b = 1)
Next == \/ /\ b = 0
           /\ b' = 1
        \/ /\ b = 1
           /\ b' = 0
```

These two TLA+ statements define *Init* and *Next* to be two formulas. Therefore, referencing *init* or *Next* is completely equivalent to typing ((b = 0) \/ (b = 1)). The equality symbol = (typed ==) is read *is defined to equal*. To transform these formulas into a host programming language, it is necessary to capture the essential aspects of the formula to be transformed, i.e., to create a translation schema. Each transformation step will consist of two elements: one to capture the TLA+ formula and one to capture the corresponding programming language function. The two schemata together can then be used to do the transformation. The TLA+ elements for the above formulas:

```
f1 == p \/ q
f2 ==  \/ /\ p
          /\ q
       \/ /\ q
          /\ p
```

That is, f1 represent Init, f2 represent Next, p represent (b=0), q represent (b=1) respectively. According to the translation schema, the translation of one bit clock from TLA+ to Scala is as follows:

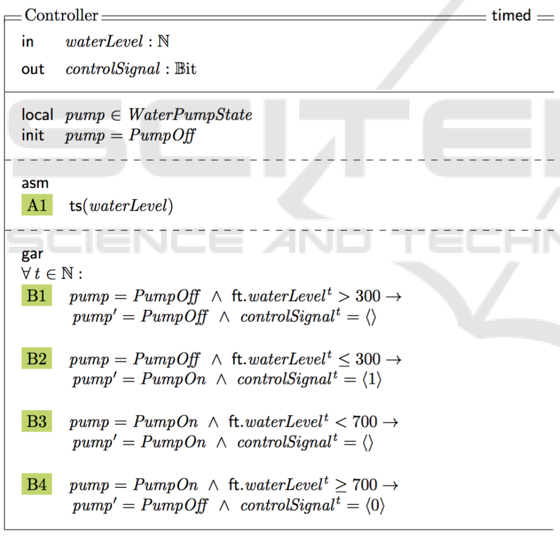
```
val b: TLAVariable = TLAVariable(IN(List(0, 1)))
val init: TLAInit =  OR(defined(b,0), defined(b,1))
val next: TLANext = {
    while(true) {
        if defined(b, 0)
           return assign(b, 1)
        else
          return assign(b, 0)
      }
}
```

2.2 Application to FOCUSST

The FOCUSST language was inspired by Focus, a framework for formal specification and development of interactive systems. In both languages, specifications are based on the notion of streams, cf. (Broy and Stølen, 2001). The syntax of FOCUSST is particularly devoted to specify spatial (S) and timing (T) aspects in a comprehensible fashion, which is the reason to extend the name of the language by ST: FOCUSST stream is a mapping from natural numbers to lists of messages within the corresponding time intervals. Table 3 shows a partial mappings between FOCUSST basic operators and their Scala representations.

Table 3: Operator mapping from FOCUSST to Scala.

FOCUSST	Scala
$\bigwedge$	AND
$\bigvee$	OR
$\rightarrow$	IMPLIES
TRUE	TRUE
FALSE	FALSE
BOOLEAN	Boolean
$\leq$	lessThanEq
$\geq$	greaterThanEq
$>$	greaterThan
$<$	lessThan
$\not\leq$	lessThanEqNot
$\not<$	lessThanNot
$\not\geq$	greaterThanEqNot
$\not>$	greaterThanNot
$\in$	IN
$x == e$	defined(x, e)
$x = e$	assign(x, e)
$\forall x \in S : p$	for $\{x \leftarrow S$; if p$\}$ yield x
$\exists x \in S : p$	exists(x, S, p)
$\langle\rangle$	List()
$\langle a_1, \ldots, a_m \rangle$	a_1 to a_m


Controller timed

in $\mathit{waterLevel} : \mathbb{N}$

out $\mathit{controlSignal} : \mathbb{B}\mathrm{it}$

local $\mathit{pump} \in \mathit{WaterPumpState}$

init $\mathit{pump} = \mathit{PumpOff}$

asm

A1 $\mathrm{ts}(\mathit{waterLevel})$

gar

$\forall\, t \in \mathbb{N}:$

B1 $\mathit{pump} = \mathit{PumpOff} \;\wedge\; \mathrm{ft}.\mathit{waterLevel}^t > 300 \rightarrow$
$\mathit{pump}' = \mathit{PumpOff} \;\wedge\; \mathit{controlSignal}^t = \langle\rangle$

B2 $\mathit{pump} = \mathit{PumpOff} \;\wedge\; \mathrm{ft}.\mathit{waterLevel}^t \leq 300 \rightarrow$
$\mathit{pump}' = \mathit{PumpOn} \;\wedge\; \mathit{controlSignal}^t = \langle 1\rangle$

B3 $\mathit{pump} = \mathit{PumpOn} \;\wedge\; \mathrm{ft}.\mathit{waterLevel}^t < 700 \rightarrow$
$\mathit{pump}' = \mathit{PumpOn} \;\wedge\; \mathit{controlSignal}^t = \langle\rangle$

B4 $\mathit{pump} = \mathit{PumpOn} \;\wedge\; \mathrm{ft}.\mathit{waterLevel}^t \geq 700 \rightarrow$
$\mathit{pump}' = \mathit{PumpOff} \;\wedge\; \mathit{controlSignal}^t = \langle 0\rangle$

Figure 2: FOCUSST Specification of Steam Boiler Controller (Spichkova, 2016).

The FOCUSST specification layout is based on human factor analysis within formal methods (Spichkova, 2012; Spichkova, 2013). Figure 2 provides an example on how a FOCUSST specification looks like. The *in* and *out* sections of FOCUSST specifications are used to specify input and output streams of the corresponding types. *local* and *init* sections include local variables and initial values, respectively. FOCUSST requires using assumption-guarantee templates, to avoid the omission of unnecessary assumptions about the system‘s environment. The keyword *asm* lists the assumption that the specified component expect from its environment, e.g., the assumption $ts(s)$ would mean that the input stream s should contain exactly one message per time interval. The component behaviour that should be guaranteed in the case all assumptions are fulfilled, is then described in the specification section *gar*.

3 DISCUSSION AND EVALUATION

Let us use the steam soiler (Broy and Stølen, 2001) example to discuss the applicability of the developed framework. We selected this example as it (1) is simple-enough to introduce it shortly, (2) is well-known example for analysing FMs, (3) includes most of the functionalities of the proposed framework. For this example, we start by given the TLA+ and FOCUSST specification which gets translated to Scala programming language before feeding the translated specification to the proposed framework. The translation correctness is verified manually by checking the behaviour that is generated by the tools developed and used to generate systems behaviours with the behaviour generated by the actual TLA model checker (TLC). We follow the informal definition of the example provided in (Spichkova, 2016): *The steam boiler has a water tank, which contains a number of gallons of water, and a pump, which adds* 10 *gallons of water per time unit to its water tank, if the pump is on. At most* 10 *gallons of water are consumed per time unit by the steam production, if the pump is off. The steam boiler has a sensor that measures the water level. Initially, the water level is* 500 *gallons, and the pump is off. In each time interval the system outputs it current water level in gallons and this level should always be between* 200 *and* 800 *gallons.*

The system consists of three logical components: SteamBoiler, Converter, and Controller. The specification *Controller* as shown in Figure 2 describes the controller component of the system. The controller role is to switch the steam boiler pump on and off. In addition, it knows the current state of the pump. The behaviour of this component is asynchronous to keep the number of control signals as small as possible.

Figure 3 shows TLA+ specification of the Steam Boiler controller. Unlike FOCUSST, TLA+ is weakly typed. Therefore, it uses a convention to indicated types of variables using *TypeOK* keyword as shown in the specification.

To check the framework, we provided two implementation for the steam boiler system, correct (wrt. the given FOCUSST and TLA+ specification) and in-

```
─────────────────── MODULE SteamBoiler ───────────
LOCAL INSTANCE Naturals
LOCAL INSTANCE Sequences
LOCAL INSTANCE Reals
──────────────────────────────────────────────────
OneOf(s) ≜ {⟨s[i]⟩ : i ∈ DOMAIN s}
tok(s)   ≜ {⟨s⟩}
Tok(S)   ≜ {⟨s⟩ : s ∈ S}

CONSTANT PumpOn, PumpOff
VARIABLE pump, waterLevel, controlSignal

TypeOK ≜ pump ∈ {PumpOn, PumpOff}
           ∧ waterLevel ∈ Nat

Init  ≜ pump = PumpOff
Next  ≜ ∨ ∧ pump  = PumpOff ∧ waterLevel > 300
          ∧ pump′ = PumpOff ∧ controlSignal = ⟨⟩

        ∨ ∧ pump  = PumpOff ∧ waterLevel ≤ 300
          ∧ pump′ = PumpOn ∧ controlSignal = ⟨1⟩

        ∨ ∧ pump  = PumpOn ∧ waterLevel < 700
          ∧ pump′ = PumpOn ∧ controlSignal = ⟨⟩

        ∨ ∧ pump  = PumpOn ∧ waterLevel ≥ 700
          ∧ pump′ = PumpOff ∧ controlSignal = ⟨0⟩
──────────────────────────────────────────────────
```

Figure 3: TLA+ specification of the Steam Boiler controller.

correct one (having mistakes wrt. the given specification). For instance, in the case when the system is specified to have its current water level be between 200 and 800 gallons, the wrong implementation does not satisfy this property and instead have the the current level below 200 and above 800. The wrong example also include the failure of the pump to turn on or off. Table 4 shows number of invocations for every API call in each test run. Both translated TLA+ and FOCUSST specifications have similar numbers since the schematic translation from TLA+ and FOCUSST to Scala is similar in both cases. The extended ScalaCheck implementation that we developed does not shrink the test case to generate minimal failing test cases (which would make the code easier to debug). The future work will include the shrinking behaviour that is inspired by *QuickCheck* library.

Table 5 contrast the performance of permutations and PBT test runs between the schematic translation of TLA+ and FOCUSST. There are no observable differences between the performance of TLA+ and FOCUSST in almost all of the phases of the workflow. This is expected since both TLA+ and FOCUSST has similar syntax and the translation is similar in most cases. For the same reason, there is no considerable difference between lines of code after translation from TLA+ to Scala which was 70 lines of code and the translation from FOCUSST to Scala was 75 lines. All tests were carried out on two machines:

```
Intel Core i5 2.6 GHz,  RAM  8 GB
Intel Core-i7 360QM 2.0 GHz, RAM  4GB
```

To evaluate the performance of the scripts using to support the framework, we used a number of further

Table 4: Number of API Invocations in test cases.

API Code	TLA+	FOCUSST
startSystem()	1	1
endSystem()	1	1
pumpDidOpen()	27	27
openPump()	11	11
pumpDidClose()	17	17
closePump()	47	47
waterLevelDidChange(amount: Int)	21	21
checkWaterLevel()	20	20
controlSignalDidChange(val: Int)	26	26

Table 5: Translated TLA+ and FOCUSST statistics (time in seconds).

	TLA+	FOCUSST
API permutations	10-11	10-11
Behaviour Generating	7-8	7-8
Single Test run	0.5	0.5
Total Test run time 100 test cases	23-25	23-25

problems commonly used in the TLA+ community:

- *One Bit Clock* simply alternates between 0 and 1. Such a clock is used to control any modern computer. Its times being displayed as the voltage on a wire. Therefore, there are only two states; the *0 state* and the *1 state*.
- The *DieHard problem* from the movie*Die Hard 3*, the heroes had to solve the problem of obtaining exactly 4 gallons of water using a 5 gallon jug, a 3 gallon jug, and a water faucet.
- Euclid‘s algorithm for computing the greatest common divisor of two positive integers.
- Therac-25, a radiation therapy machine used in curing cancer, led to deaths and serious injuries of patients which received thousand times the normal dose of radiation (Miller, 1987; Leveson and Turner, 1993). The causes of these accidents were software failures as well as problems with the system interface. The machine included VT-100 terminal which controlled the PDP-11 computer, where the sequence of user actions leading to the accidents was as follows: user selects 25 MeV photon mode, enters *cursor up*, select 25 MeV Electron mode, previous commands have to take place in eight seconds.

Table 6 shows the statistics on the applied behaviour generator. *Diameter* column is the number of states in the longest path of the graph in which no state appears twice. *States Found* column is the total number of states it examined in the first step of the algorithm or as successor states in the second step. *Distinct States* column is the number of states that form the set of nodes of the graph. For instance, in case of *One Bit Clock*, model checker found two distinct states.

Table 6: Behaviour Generator Statistics.

Example	Diameter	State Found	Distinct States
DieHard	9	97	16
One Bit Clock	1	4	2
Euclid Algorithm	3	22	8
Therac25	9	97	16

4 CONCLUSION

We have presented our framework for application of the property-based testing (PBT) concepts on top of temporal formal models. This allows us to have an executable Scala code that corresponds to the formal model, as well as to perform PBT of the models functionality. The framework is aiming on reduction of the impedance mismatch between formal methods and practitioners through the combining of formal methods with property-based testing. We introduced the core ideas on how the framework can be applied to particular formal languages, such as TLA+ and FocusST.

REFERENCES

Alzahrani, N., Spichkova, M., and Blech, J. O. (2016). *Spatio-Temporal Models for Formal Analysis and Property-Based Testing*, pages 196–206. Springer.

Blech, J. O. and Schmidt, H. (2014). BeSpaceD: Towards a tool framework and methodology for the specification and verification of spatial behavior of distributed software component systems. *CoRR*.

Bowen, J. P. and Hinchey, M. G. (1995). Seven more myths of formal methods. *IEEE software*, 12(4):34.

Broy, M. and Stølen, K. (2001). *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer.

Claessen, K. and Hughes, J. (2011). QuickCheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64.

Gerdes, A., Hughes, J., Smallbone, N., and Wang, M. (2015). Linking unit tests and properties. In *SIGPLAN Workshop*, pages 19–26. ACM.

Hinchey, M. G. (2003). Confessions of a formal methodist. In *Safety Critical Systems and Software*, pages 17–20. ACS.

Hughes, J. (2010). Software testing with quickcheck. In *Central European Functional Programming School*, pages 183–223. Springer.

Kühnel, C. and Spichkova, M. (2007). Fault-tolerant communication for distributed embedded systems. In *Software Engineering of Fault Tolerance Systems*, volume 19, page 175. World Scientific Publishing.

Lamport, L. (1993). Hybrid systems in TLA+. In Grossman, R. L., Nerode, A., Ravn, A. P., and Rischel, H., editors, *Hybrid Systems*, number 736 in LNCS, pages 77–102. Springer.

Lamport, L. (1994). The temporal logic of actions. 16(3):872–923.

Leveson, N. G. and Turner, C. S. (1993). An investigation of the therac-25 accidents. *Computer*, 26(7):18–41.

Miller, E. (1987). The Therac-25 Experience. In *Conf. State Radiation Control Program Directors*.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon Web Services Uses Formal Methods. *CACM*, 58(4):66–73.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media.

Spichkova, M. (2007). *Specification and seamless verification of embedded real-time systems: FOCUS on Isabelle*. PhD thesis, Technical University Munich.

Spichkova, M. (2012). Human Factors of Formal Methods. In *IADIS Interfaces and Human Computer Interaction 2012*.

Spichkova, M. (2013). *Design of formal languages and interfaces: "Formal" does not mean "unreadable"*. IGI Global.

Spichkova, M. (2016). Spatio-temporal features of FocusST. *CoRR*.

Spichkova, M., Blech, J. O., Herrmann, P., and Schmidt, H. W. (2014). Modeling Spatial Aspects of Safety-Critical Systems with FocusST. In *MoDeVVa*, pages 49–58.

Spichkova, M., Liu, H., Laali, M., and Schmidt, H. W. (2015). Human factors in software reliability engineering. *Workshop on Applications of Human Error Research to Improve Software Engineering*.

Yu, Y., Manolios, P., and Lamport, L. (1999). Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer.

Zamansky, A., Rodriguez-Navas, G., Adams, M., and Spichkova, M. (2016). Formal methods in collaborative projects. In *11th International Conference on Evaluation of Novel Approaches to Software Engineering*. IEEE.