`formalSpec` — Semi–automatic Formalization of System Requirements for Formal Verification (Tool Presentation)

Axel Busboom, Simone Schuler, and Alexander Walsch*

Controls and Embedded Systems Lab, GE Global Research Europe, Garching, Germany
{Axel.Busboom, Simone.Schuler, Alexander.Walsch}@ge.com

Abstract

We present the proof–of–concept tool `formalSpec` for semi–automatic translation of system requirements from controlled natural language into hybrid automata. These can be automatically integrated as monitor automata with an existing SpaceEx system model.

1 Introduction

In industry, system requirements are usually formulated in natural language or controlled natural language (CNL). A CNL is a subset of a natural language with a constrained grammar such as to reduce or eliminate ambiguity. Requirements written in a CNL can be understood across different professional disciplines (e.g. technical, legal, marketing), but can also be translated into a formal representation and hence be used along a system engineering tool chain.

In the case of hybrid systems, the translation from CNL into a formal representation needs to be done manually which introduces the risk of translation errors. In particular, requirements which explicitly mention time or uncertainties in the environment models, may lead to complex expressions.

We are proposing a template–based, semi–automated translation of system requirements. The `formalSpec` tool is a graphical front–end to SpaceEx [1, 2] which translates system requirements based on specification templates into hybrid automata. These can be imported into SpaceEx and coupled to existing SpaceEx system models. We use SpaceEx as the target tool since it provides the de–facto standard format for machine readable representation of hybrid automata. Further, a translation tool [3] to other major verification tools exists [4].

This paper focuses on the software tool and attempts to show conceptually how such tools can help separate the requirements engineering process from the formal verification domain, as these are typically performed by different organizations with different skill sets in industry. A more theoretical paper formally proving the equivalence between the specification patterns and the hybrid automata generated by the tool is currently in preparation.

We briefly describe the state of the art and theoretical background of the requirement formalization in Section 2. In Section 3, we explain the graphical user interface, and an example is presented in Section 4.

2 Requirement formalization

In order to reduce manual intervention in the formalisation of system requirements, we use requirement templates that are mapped to standard monitor automata. These are hybrid automata which express requirements by means of forbidden states. If the system were to enter a forbidden state, this would constitute a violation of the corresponding requirement [5].

The use of a CNL notation employing templates for the specification of system requirements and their (semi–)automatic translation for verification has been proposed earlier. Specification

---

Table 1: Specification pattern templates for hybrid systems as used in the GUI.

Start	01: property	::=	$scope$ ”,” $specification$ ”.”
Scope	02: scope	::=	”Globally” | ”After q”
General	03: specification	::=	qualitativeType | realtimeType
Qualitative	04: qualitativeType	::=	$absencePattern$ | $universalityPattern$
	05: absencePattern	::=	”it is never the case that” p ”holds”
	06: universalityPattern	::=	”it is always the case that” p ”holds
Real-time	07: realtimeType	::=	”it is always the case that” ($minDurationPattern$ | $maxDurationPattern$ | $periodicPattern$ | $boundedResponsePattern$ | $boundedInvariancePattern$)
	08: minDurationPattern	::=	”once” p becomes satisfied, it holds for at least” c ”time unit(s)”
	09: maxDurationPattern	::=	”once” p becomes satisfied, it holds for less than” c ”time unit(s)”
	10: periodicPattern	::=	p ”holds at least every” c ”time unit(s)”
	11: boundedResponsePattern	::=	”if” p ”holds, then” s ”holds after at most” c ”time unit(s)
	12: boundedInvariancePattern	::=	”if” p ”holds, then” s ”holds for at least” c ”time unit(s)”

templates contain predicates which are to be substituted by logical expressions in order to formulate the actual requirements. Dwyer et al. [6] were among the first to introduce qualitative specification pattern templates and their translation into different logic expressions (e.g. linear temporal logic). Among others, Konrad and Cheng [7] extended Dwyer's original patterns to the real–time domain. Post et al. [8] applied and extended those patterns for the automotive industry. A generalisation to probabilistic specification patterns was introduced in [9].

For discrete–time systems, tools already exist that allow for the input of CNL expressions (e.g. in the form of the above mentioned specification templates) and automatically translate them into formal expressions. Examples for such tools are Stimulus [10], Embedded Specifier [11], AutoFocus3 [12, 13, 14] and SpeAR [15, 16]. The output of these tools, however, is in the form of finite state machines, Büchi–automata or general ω–automata, and is not applicable to hybrid systems, i.e. systems with a combination of discrete and continuous states.

To automatically detect undesired system behaviour in SpaceEx, observers can be added to the system model. An observer, following the same syntax and semantics as the system model, is also called monitor automaton. Since the monitor automaton encodes forbidden states, which must not be reachable by the system, it is semantically equivalent to a system requirement. The verification of a given set of requirements combined with a given system is successful if the intersection of the reachable states and the forbidden states is empty. While monitor automata can be manually added in SpaceEx, we are interested in a format which is closer to natural language, does not require expert knowledge in SpaceEx when formulating the requirements and lends itself to industrial system engineering processes.

We adopt a subset of the grammar introduced by Konrad and Cheng [7] since it is widely used in academia. While we have implemented the pattern shown in Table 1, it would certainly be possible to extend these by additional specification patterns depending on application needs. The specification templates for hybrid systems as such do not differ from those for discrete systems. The novelty of our approach is in the formalization as hybrid automata and their integration with SpaceEx as a verification tool. We have cross–checked the specification templates by Konrad and Cheng against a number of hybrid system requirements from different industrial applications to convince ourselves that they cover the vast majority of requirements encountered in practice. For details on this analysis, the reader is referred to [17].

3 Graphical User Interface

The `formalSpec` tool provides functionality to enter and edit requirements based on the specification templates explained above. The user can choose one of 14 patterns to create a new requirement, and is then required to replace the predicates p, q and s (cf. Table 1) by actual logical expressions, and the variable c by an arithmetic expression. The set of requirements being

edited may be linked to an exisiting SpaceEx model.

SpaceEx provides the SpaceEx Model Editor as a graphical front end for visually editing hybrid system models. A SpaceEx model consists of any number of 'base components' and 'network components'. A base component specifies a hybrid automaton by its locations with invariants and flows, and its transitions with guards and assignments. A network component is built from one or more base components or other network components that can be linked to each other. Since network components can be nested, they can be used to build hierarchical system models of arbitrary complexity. Network components can be re–used, e.g. an integrator component once specified as a base component can be used any number of times when assembling a model. When performing an analysis in SpaceEx, one component must be specified as the 'system' to be verified. In a hierarchical model, one would typically select the top–level network component, representing the overall system, as the system, even though there is no requirement to do so.

When specifying invariants, flows, guards and assignments in the SpaceEx Model Editor, one can use any type of (typically real–valued) parameters. Parameters can be specified as global or local, depending on whether or not they should be visible to a superordinate network component. Note that in a superordinate network component, any parameter linked to a subordinate component can again be specified as global or local, i.e. visible or invisible to network components at the next higher level, and so forth.

In the following, we highlight some features of the `formalSpec` tool:

- The user may open a SpaceEx model file and select a component from this model as the system. The `formalSpec` tool will inspect the model file to be opened and automatically propose the most plausible top–level component as the system, even though the user can override this choice.

- The tool will then open a tree view of the entire SpaceEx system hierarchy, starting at the top–level system and descending down the hierarchy of components. Note that in this tree view, any component specified in the SpaceEx model may appear once, multiple times, or never, depending on its use in the system hierarchy. The tree view shows all parameters for each component, with parameters local to the respective component being greyed out.

- In formulating the logical and arithmetic expressions in the requirements, one may use any parameters used in the SpaceEx system, regardless of whether they are visible to the top–level system or local to some subordinate component. This is an important feature as in many cases we will need to specify requirements on some subsystem and using quantities that will not be accessible at the system level.

- In order to disambiguate parameter names, in analogy to SpaceEx we use the naming convention `system.component.subcomponent.  ...  .parameter`. A postfix (e.g. `subcomponent.parameter`) can be used if there is no potential naming conflict. The user may browse through the tree view of the system by expanding and collapsing components. By double–clicking on any parameter in the tree view, it will automatically be copied to the requirement that is being edited.

- The tool provides syntax highlighting to indicate whether the parameters used in the specifications are unambiguous within the SpaceEx system (green), ambiguous (yellow), or non–existing (red).

- When finished editing, the results can be exported back to a SpaceEx model which consists of (a) the original SpaceEx model, (b) one base component for each specified requirement, implementing the corresponding monitor automaton, and (c) one or more 'system–with–monitor' components. A system–with–monitor is a network component linking one or more monitor automata to the SpaceEx system to be analysed. In the generated monitor automata, the predicates and variables from the template automata are replaced by the actual logical and arthmetic expressions specified in the `formalSpec` tool.

- The user may choose between creating a single system–with–monitor component for all requirements, or creating a separate system–with–monitor for each requirement individually. The former option has the advantage that all requirements can be simultaneously verified in

a single SpaceEx run, but may cause performance issues when dealing with complex system models and/or large numbers of requirements. The latter option mitigates these performance issues, but requires a separate SpaceEx analysis for each requirement.

- An important feature is related to parameters which are local to a subordinate component and not visible at the system level in the original SpaceEx model: When exporting back to SpaceEx, the `formalSpec` tool will modify the system model such that these parameters are automatically 'pulled up'. By that we mean that the parameters will be modified to global parameters in the component that they used to be local to. Further, in all superordinate network components, the parameter will be linked to and made accessible as a global parameter to the next higher component. Unique names will be assigned at all levels. Hence, the exported model will be structurally identical to the original model, but will have additional parameters that are accessible at the top–level system and which are linked to the respective monitor automata in the system–with–monitor components. Note that while the pulling up of parameters could of course also be done manually, this would in practice make the manual generation of monitor automata in SpaceEx very cumbersome and prone to error.

4 Example

To illustrate the functionality of the GUI, we will take the reader through an example: As the hybrid system to be considered we use a strongly simplified model of a wind turbine and its controller. This system has been submitted as a benchmark problem to ARCH 2016 [18]. The hybrid nature of this system mainly stems from the switching behaviour of the controller which, depending on wind speed, will switch between torque–control (below rated power), pitch–control (above rated power) and a transitional control regime between these two regions. While in [18] a Simulink implementation of the model is used, we have also generated a SpaceEx representation of the system, a screenshot of which in the SpaceEx Model Editor is shown in Figure 1. When we open the same SpaceEx model in the `formalSpec` tool, it provides us with a browsable tree view of all parameters used in the SpaceEx model (Figure 2).

As a simple example of a requirement to be validated, let us assume we want to ensure that the pitch controller works properly in the sense that the actual (measured) pitch angle closely follows the commanded pitch angle. We obviously need to allow for some tracking error as well as measurement error. We may decide to quantify one of the requirements in natural language as follows: 'The absolute difference between the commanded pitch angle and the measured pitch angle must never exceed 1° for longer than 0.5 seconds.' Note that this would not be the only requirement needed to validate the pitch controller: In practice, one would also need to specify an absolute bound on the instantaneous error, a bound on the steady–state error, on allowable maximum and minimum pitch angles, maximum allowable pitch rates, and others.

We now need to map this requirement to one of the 14 specification patterns available in `formalSpec`. The pattern suitable for this particular type of requirement is a maximum duration pattern with global scope, one of the real–time patterns described in [7]. Reformulating it using the templates presented in Table 1, this corresponds to the sequence (01,02,03,08) or — in textual form — to the pattern: Globally, it is always the case that once p becomes satisfied, it holds for less than c time units.

The main window of the `formalSpec` tool is essentially a table in which each row corresponds to one requirement. It is shown in Figure 3 where we have already created one skeleton requirement based on the pattern above. Next, we need to replace the placeholders in the template by the actual expressions, the result of which is shown in Figure 4. When doing this, parameters from the parameter browser can be copied to the requirements table by double–clicking. As can be seen from the figure, the tool also allows to assign names and comments to each requirement.

When done editing the requirements, the project can be saved in an XML–format proprietary to `formalSpec`. In addition, it can be re–exported to SpaceEx, the result of which can be seen in Figure 5. The exported model contains an additional network component called `system_with_monitor_automata` which, as subordinate components, contains the original SpaceEx
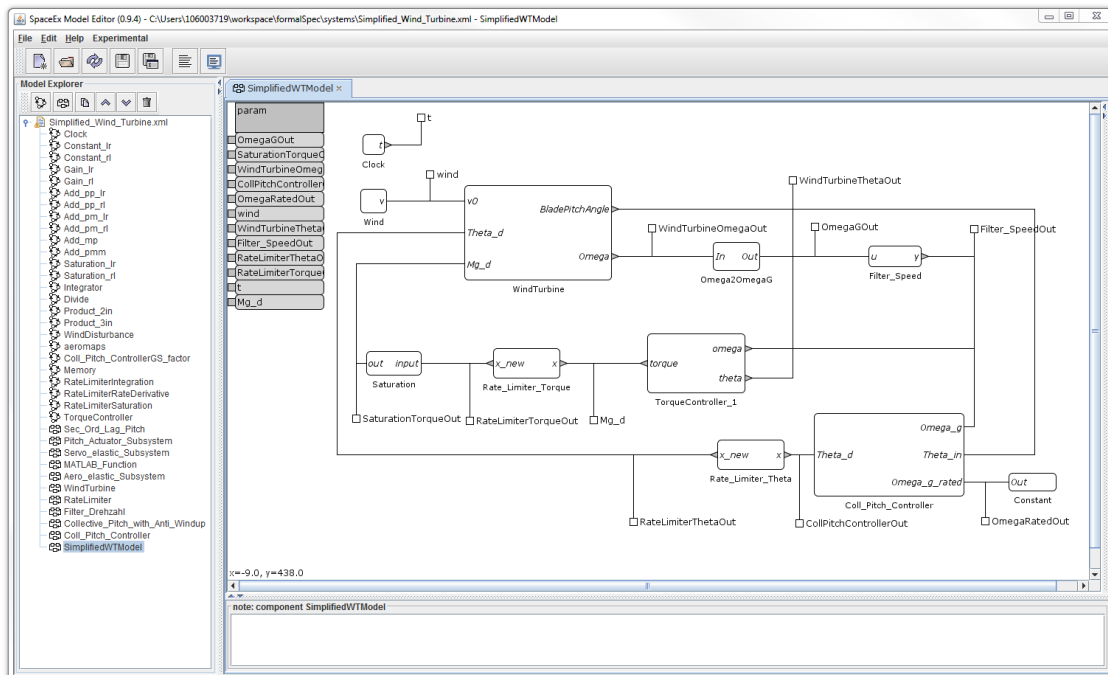
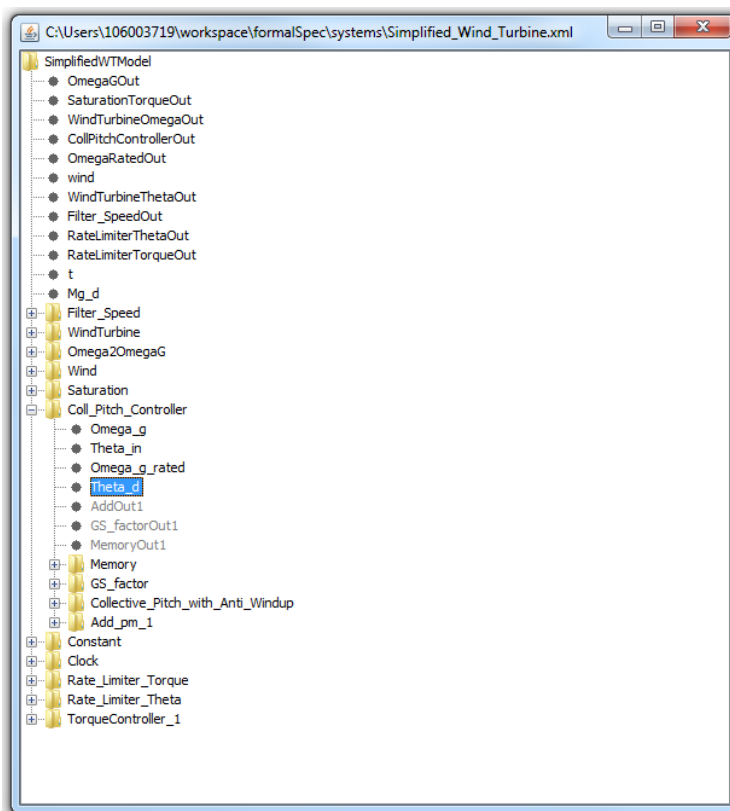Figure 1: Simplified model of wind turbine and controller in the SpaceEx Model Editor.

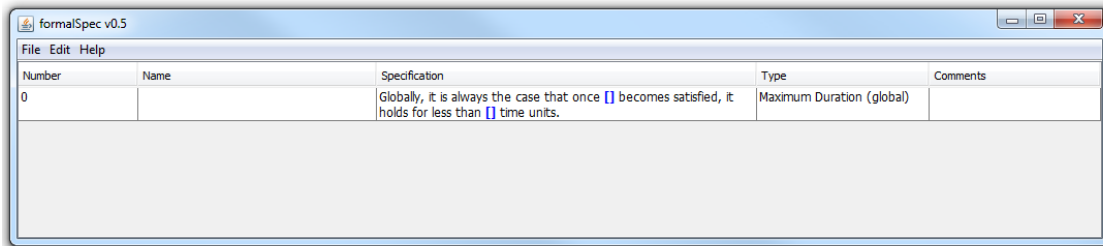Figure 2: Hiearchical view of parameters of the same system model in `formalSpec`.

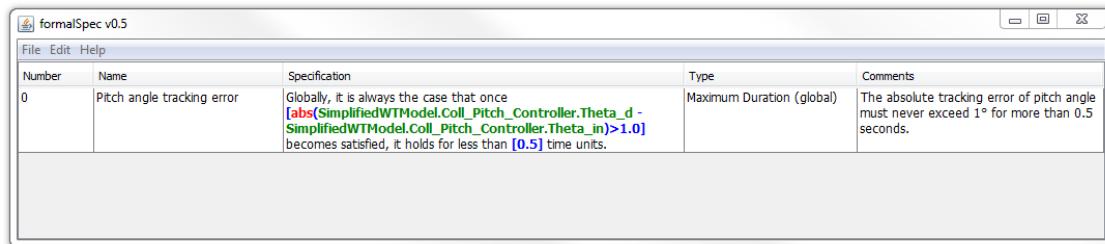Figure 3: Main window of `formalSpec`.

Figure 4: Main window of `formalSpec` with fully specified requirement.

system and in this case a single monitor automaton corresponding to the requirement that we have specified. Figure 6 shows a view of the generated monitor automaton in which the predicates have been substituted with the expressions specified in `formalSpec`.

Note that in this particular example the two parameters happened to be already available at the top–level system; so the tool simply determined the equivalence between the parameter names in the subsystem and the top–level system. Otherwise, the tool would have automatically pulled up any needed parameters and assigned unambiguous names such that they can be accessed by the system–with–monitor component.

5   Conclusion

We have presented a rudimentary proof–of–concept tool to show how requirements for hybrid systems can be captured in Controlled Natural Language, automatically translated into hybrid monitor automata, and combined with a hybrid system model for validation in a tool like SpaceEx. This should provide a step towards the applicability of hybrid system verification tools in an industrial systems engineering setting.
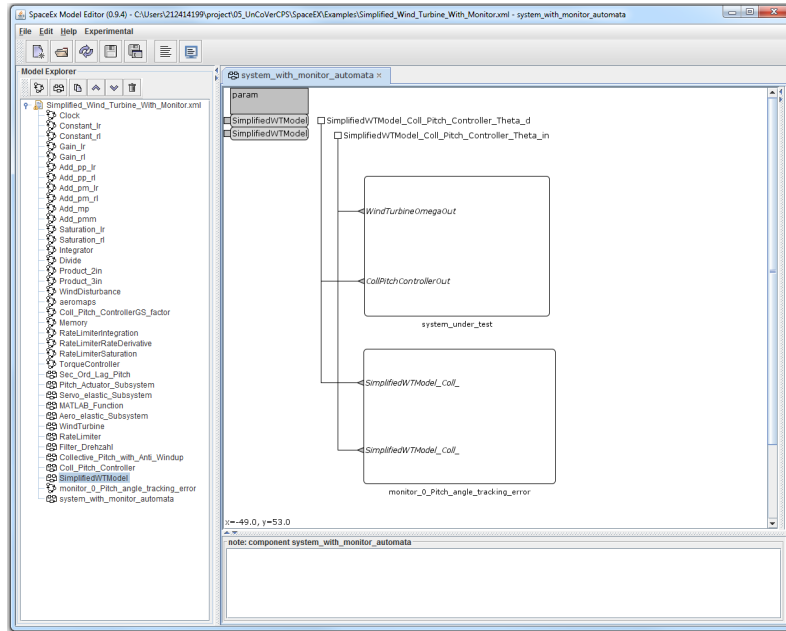
Figure 5: SpeceEx Model Editor view of the 'system–with–monitor' as a network component.
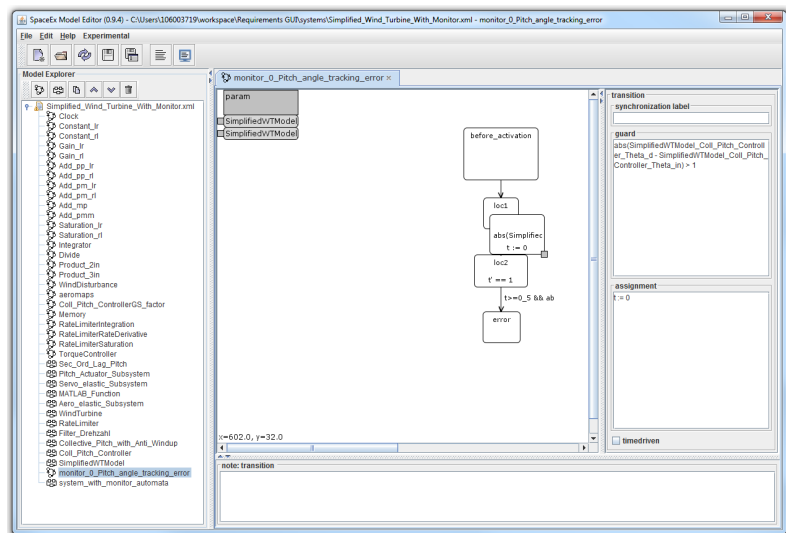

Figure 6: SpeceEx Model Editor view of the monitor automaton as a base component.

References

[1] G. Frehse et al. *Computer Aided Verification*, chapter SpaceEx: Scalable Verification of Hybrid Systems, pages 379–395. Springer, Berlin, Heidelberg, 2011.

[2] G. Frehse. An introduction to SpaceEX v0.8. http://spaceex.imag.fr/documentation/user-documentation/introduction-spaceex-27, accessed on Sep. 9, 2015.

[3] S. Bak, S. Bogomolov, and T. T. Johnson. HyST: A source transformation and translation

tool for hybrid automaton models. `http://verivital.uta.edu/hyst`, accessed on Sep. 30, 2015.

[4] S. Bak, S. Bogomolv, and T. T. Johnson. HyST: a source transformation and translation tool for hybrid automaton models. In *Proc. Conf. Hybrid Systems Computation and Control (HSCC)*, pages 36–42, 2015.

[5] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *3rd Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*. Springer Verlag, 1993.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21th Int. Conf. Software Engineering*, pages 411–420, 1999.

[7] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proc. 27th Int. Conf. Software Engineering*, pages 372–381, 2005.

[8] A. Post, I. Menzel, and A. Podelski. *Requirements Engineering: Foundation for Software Quality*, chapter Applying Restricted English Grammar on Automotive Requirements—Does it Work? A Case Study, pages 166–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[9] L. Grunske. Specification patterns for probabilistic quality properties. In *Proc. 30th Int. Conf. Software Engineering*, pages 31–40, 2008.

[10] Argosim. Stimulus. `http://argosim.com`, 2015.

[11] BTC. Embedded specifier. `http://www.btc-es.de`, 2015.

[12] Manfred Broy, Franz Huber, and Bernard Schätz. AUTOFOCUS — ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. In *Informatik Forschung und Entwicklung*, pages 121–134, 1999.

[13] F. Hölzl and M. Feilkas. *Model–Based Engineering of Embedded Real–Time Systems*, chapter 13 AUTOFOCUS 3 — A Scientific Tool Prototype for Model–Based Development of Component–Based, Reactive, Distributed Systems, pages 317–322. Springer, Berlin, Heidelberg, 2010.

[14] Fortiss. AUTOFOCUS3. `http://af3.fortiss.org`, 2015.

[15] SpeAR — specification and analysis for requirements tool. `https://github.com/AFifarek/SpeAR`, accessed on March 11, 2016.

[16] L. Wagner et al. SpeAR: Specification and analysis of requirements. In *to be presented at High Confidence Software and Systems Conference*, 2016.

[17] S. Schuler, A. Walsch, and M. Woehrle. Unifying control and verification of cyber–physical systems (UnCoVerCPS), Deliverable D1.1 — Assessment of languages and tools for the automatic formalisation of system requirements. `http://cps-vo.org/node/24197`, accessed on March 11, 2016.

[18] S. Schuler, F. D. Adegas, and A. Anta. Benchmark problem: hyrid modelling of a wind turbine. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH), submitted*, 2016.