Tooling for automated testing of cyber-physical system models

Tim Broenink
Robotics and Mechatronics
University of Twente
Email: t.g.broenink@utwente.nl

Bas Jansen
Robotics and Mechatronics
University of Twente
Email:b.jansen-1@alumnus.utwente.nl

Jan Broenink
Robotics and Mechatronics
University of Twente
Email: j.f.broenink@utwente.nl

***Abstract*—This work presents a tool for automatic testing of cyber-physical systems via simulation. Cyber-physical system design can benefit from this automated testing as it allows for system-level requirements and prevents regression of the design.**

The tool is based on three parts: A testing language, a simulator controller, and a post processor. The testing language is a domain-specific language based on a Gherkin style syntax and can define test for multiple models and simulators. The domain specific language also defines algebraic, logical, and linear temporal logic transformations for outputs to define testing conditions. The tool can perform different sub-sets of tests based on a graphical or command line interface.

The tool is demonstrated using an example where a motor is selected for a winch system. Here it is shown that the tool can verify component- and system-level requirements, and can detect regression. The tool is basis for a method supporting the design of cyber-physical systems.

I. Introduction

This work aims to improve the process of designing Cyber-Physical Systems (CPS). It does so by developing a tool to automate the testing when designing these, using different simulators. As stated by Lee[1], there are challenges when designing cyber-physical systems. These challenges are based on the interaction between different domains, and the issues that can crop up based on timing and boundary conditions. A way to tackle these design challenges has been proposed in earlier work[2], based on an iterative design approach. This required clear requirements for the final system under development (SUD), and the different stages of the design. Verifying these requirements is an essential part of this development method. During the design process these can be validated in simulations based on models of the final product[3]. These requirements might require a large amount of tests to verify or might require the complete system to verify.

Automated testing can be used to enable more tests to be run, with less effort. This can give some of the benefits of Test-Driven-Development (TDD) as applied in software engineering[4], namely, more reliable products, and the prevention of regression in the design. The automated testing of CPS needs more extensive tooling than a conventional software testing tool. For simulation of CPS, models spanning different domains, time standards or models of computation (continuous-time, discrete-time, event-driven, etc.) are used. Furthermore, the outputs of the simulations (e.g. traces, time-series, boolean) need to be converted to the same form to be compared.

So to automatically test the designs, a tool is needed to define tests, run these tests on multiple different simulators, and to process and collate the results. This work aims to provide these capabilities by defining a domain-specific language (DSL) which can be used to define these tests, and a tool to run and interpret these tests using different simulators. As concluded by Grimm, Anders and Wang [5], in order to be useful for the development of complex CPS, this tool will have to support a wide range of models and simulation methods. This would allow the tool to be applicable for different viewpoints and levels of detail of the same product. Thus the tool is developed to be extendible to different simulators.

II. Problem Analysis

To describe the design and development of the automated testing tool, it is important to have a clear distinction on what problem we are trying to solve. To do this three aspects of the overall design and testing process are discussed here. First the workflow used to design CPS is discussed, which contains the expected steps taken to design the CPS and the tests. Then the requirements for the automatic simulations is discussed. This discusses the tooling required for the simulations, but also the required information en specifications. Finally the test definitions are discussed. What needs to be defined in the tests, and what is required to be able to correctly interpret the results.

A. Workflow

The workflow discussed in this sections is based on previous work on how to design CPS[2]. This way of working is used as it allows for multiple levels of detail in the design, which allows for testing during different stages of development. To define the requirements for individual stages of the development, and for the different subsystems, it is important that the function and requirements are clear. These requirements should be made on system level using a structured way of creating requirements, for example EARS[6]. The system-level requirements can then be used to create system-level tests. They can also be used to define the requirements for subsystems. As the automatic testing tool is based on simulations, models need to be available. Not only models of the final design, but also of less detailed stages of the design process need to be available. The detail used in different parts of the system under development can be varied according to the requirements or subsystem being

978-1-7281-6389-5/20/$31.00 ©2020 IEEE

tested. But the models should be described in such a way that this detail can be varied without changing the structure [7]. This allows for testing of subsystems before adding them to the complete SUD. With the specifications, tests, and models, the (sub)-system can then be tested. After the tests are passed, the tests can be added to the total set of system tests, giving the advantages of Test-Driven Development[4]. One can now move on to the next stage of the design, either a new part, or adding a new level of detail.

The complete workflow can be summarized in four steps:

1) The specifications of the CPS or parts are made. These are translated into specific requirements (EARS).
2) Tests are created based on the specifications. These tests apply to the newly designed part, and/or the total CPS.
3) Models required for these tests are acquired/created.
4) The new design is tested using the new tests made in step 2 and using the global tests available for the CPS to confirm that everything is still within specification.

Step two and four, the definition and running of the tests are the steps that are supported by this work.

B. Requirements for the Automation

It is important to have a clear view of all information that is required for the automation of the tool, as these things will have to be created during the design process. For the automation, a certain amount of information about the system under test (SUT) is needed, and some capabilities are required of the tooling used. The information that is required for automated tests includes, but is not be limited to:

- Models that can competently describe the behaviour of the SUT during the tests. The might not be the same for all the tests, or even cover the whole system.
- Configurations and parameters of the SUT. This does not only include system parameters, but also the state the system should be in at the beginning of the tests, for example the resting position, or the environment.
- Scenarios that the SUT is used in, so all inputs that are received during the test. This can be the planned trajectories, set-points, received commands, etc.
- Conditions that the tests need pass must be clear. What exactly constitutes a pass or a fail?
- Simulators that are used to execute the tests.

Furthermore the simulators and tools used to run the tests need to have certain functions that are required for automation. They need to be capable of simulating the given models and scenarios, but that is not all. The simulator needs an interface through which the simulation can be controlled automatically. This interface should be able to:

- Start and stop the simulation
- Open new models
- Set model parameters/inputs

The simulator also needs a way to export the data of the simulator, this can be any machine readable format, depending on the support of the tooling. With these definitions and capabilities an automatic simulation can be possible.

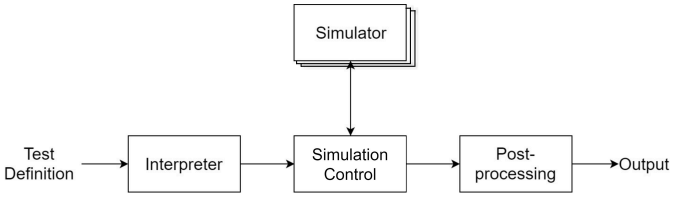

Fig. 1. A responsibility overview of the testing tool. The simulator is an external tool and not implemented by the testing tool.

The testing tool that was developed can then, given this information, run all the models in their respective tests and receive the data from these tests. The data from these tests then needs to be processed to determine test results. These results can then be combined by the tool to give a final pass or fail for the SUT. Depending on the design goals, it might not be required to test the whole CPS every time. This is why the tool needs a certain flexibility to run only a subset of all tests to save time. It also needs to be able to run different simulators for the different types of models present in the design.

C. Test definitions

The final requirements are based on what the tests should contain. For a single test, definitions are needed. Furthermore, the conditions specified in the tests should be capable of handling the required types of signals. The conditions should be able to translate the output of the simulation to a final pass or fail results. The output of the simulator can be of many forms, it can be time based or a single value, it can be numeric or boolean. Conversions from all these output types to a single boolean value, the test verdict, must be available. Furthermore a test needs to specify all information required for a single simulations run, which as mentioned before should contain the model, simulator, parameters, and the scenario that is run.

All these conditions and definitions should be contained in a self-contained unit that is easy to share and run.

D. Tool requirements

If we combine the requirements on the workflow, automation and tests, we get the following requirements for the complete tool:

- The tool can combine and show the results of all the run tests.
- The tool can run single, all or a selection of tests.
- Tests can run on different models and simulators.
- Tests can convert different signals (time, scalar, boolean) to a final conclusion.
- Tests can change parameters/inputs of the simulation.

III. Tool Design

The tool has been split into three different parts based on separate responsibilities: The interpretation of tests, the running of tests, and the interpretation of data. This is shown in Figure 1. Further details on the implementation of this tool are given in subsection III-C.

The interpreter is based on a domain-specific language created for the definition of tests. This DSL will be presented further in the upcoming section.

The running of tests is done by a simulation control subsystem. This subsystem is responsible for connecting to different simulators. This means that the simulation control contains different interfaces depending on the capabilities of the different simulators. Thus if support for a new simulator is required it is added to the simulation control as a new interface definition. The simulation control receives information from the interpreter on which models to run with what parameter to obtain the test data. This test data is then processed by the post processor, which tests the conditions that are specified in the tests to get a final result. The post processor is further discussed in the upcoming sections.

A. Test DSL

To give a clear structure to define tests in, a domain-specific language is created. This language handles the definitions of: models, parameters, inputs, and test conditions. It can also be used to structure multiple tests in a logical structure. The DSL for the tests is inspired by the Gherkin syntax used in behaviour and test-driven development[8], [9]. This syntax uses a set of keywords to describe the desired behaviour of a system under test. As there is a difference between tests for CPS and for pure software, two extra keywords are introduced to control the simulations: *With* and *For*, next to the two core keywords of Gherkin, *Given* and *Then*. *Given* is used to specify which model is used for the simulation. *With* is added as extra keyword to specify parameters and settings for the model that is used. *For* is added to specify the simulated time of the model, as some simulations might not have a clear ending. *Then* is added to specify conditions which must hold during the simulation. These conditions are specified using a mix of names, functions and equations, which is where the tool deviates from the Gherkin syntax. To specify the conditions of the simulations, multiple different syntaxes and equations can be used. More on this in the next section on post processing. A single test can contain multiple conditions. These must all be true for the test to pass.

Furthermore the DSL implements *Feature* and *Scenario* as keywords, as used by Gherkin. *Feature* is used to label a group of tests as belonging to a certain feature. These will be grouped in the output of the tool. *Scenario* is used to label a test, to give a human readable name to a test. A overview of the DSL is given in Listing 1

B. Post processing

The post processor is responsible for processing the data from the simulations. This data must be processed to check the constraints specified in the tests. Depending on the type of model and simulation, the output data can take different forms. An output can vary over time, or be one value. The data type of the output can also be different. The different types of outputs are classified in Table I

Listing 1
A STRUCTURAL OVERVIEW OF THE TEST DSL

```
Feature "feature name"
    method name(argument1, argument2, ...):
        constraint
    ...
    Scenario "scenario name 1"
        Given "model name"
            with "option1" = X
            with "option2" = Y
            ...
            include "file1"
            include "file2"
            ...
            in simulator
            for X time unit

        Then constraint
        Then constraint
        ...
    Scenario "scenario name 2"
    ...
```

Data type	Time-varying	Constant
Real	Trace	Scalar
Boolean	Boolean-time-series	Boolean

TABLE I
AN OVERVIEW OF THE TERMS USED FOR THE DIFFERENT OUTPUT TYPES

The post processor is capable of doing different kinds of manipulations to manipulate the output. It can change the value of signals without changing the output type. This can be algebraic, for example multiplication or addition, or boolean, e.g. negation. It can also transform between output types. It can transform from Real to Boolean using comparisons. Or it can transform from Time-varying to constant using either special functions for Traces, for example *min* or *max*, or Linear Temporal Logic (LTL)[10] to tranform Boolean time series to Boolean. With these transformations, it is possible to convert all output types to Boolean output. These boolean outputs can then be combined to a test verdict by logically and-ing them. In this situation, the LTL syntax has been slightly modified. The neXt operation (X) normally takes the next step. But to properly work with dynamic systems, the neXt operation has been changed to accept a time span as a parameter. It will then take the value after that amount of time. An overview of the transformations between the Output types is shown in Figure 2

C. Implementation

For the implementation of the tool some specific choices have been made. First the programming language for the tool is Python 3.7. Furthermore antler is used to create the specifications of the DSL. Python was chosen for easy of use and availability of a large amount of libraries for the processing of data.

The tests have been defined using *.test* files that contain a group of test defined as a feature in plain text. The tool has 2 user interfaces. The first one is a simple GUI based on TKinter that allows to select tests to run or edit. Furthermore, there is a Command line interface. This interface is more suitable for an automated workflow and can run multiple tests.

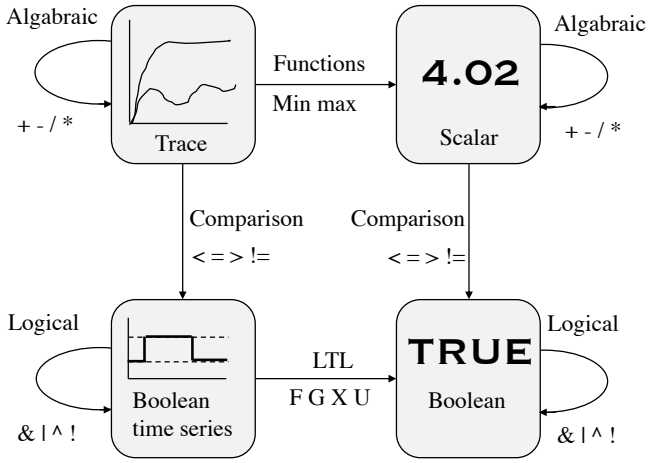

Fig. 2. The transformations between different Output types performed by the post processor. All data types end up as a single boolean.

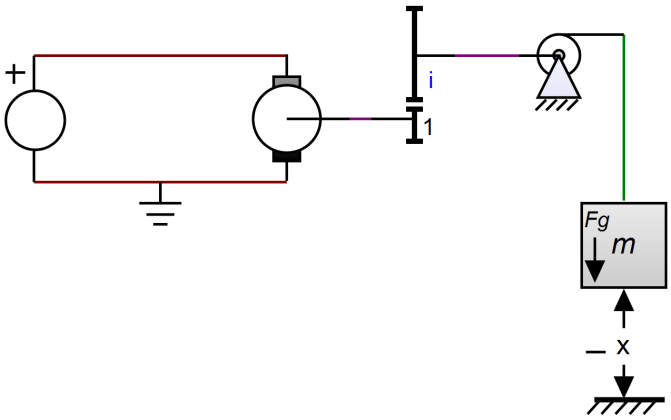

Fig. 3. A schematic overview of the Winch system used in the usage example, modelled in 20-sim.

Support for two different simulators is implemented in the tool, to show the capabilities of working with multiple simulators: 20-Sim[11], a simulator for dynamic systems based on bond-graphs and block diagrams, as it is used a lot in our research; VREP[12], a physics-based 3D robot simulator. This not only demonstrates the use of different simulators, but also the tool's ability to interface in different programming languages, namely, Python and Java.

IV. Usage

To demonstrate the usage of the testing tool, and to give an example of the envisioned workflow, a case study is presented. The workflow as presented in subsection II-A is followed. The example is based on a situation where a winch is designed to lift a certain mass in a short amount of time. To keep the example compact, it only shows the physical part of this system. The System Under Development (SUD) is also limited by power use, both peak and continuous. A schematic overview of the SUD is given in Figure 3.

Power supply voltage	16 V
Gear transmission ratio	1:3
Drum radius	18 cm

TABLE II
The fixed parameters for the winch.

To limit the scope of the example, only the choice of motor will be considered. The transmission and drum, as well as the power supply voltage have been selected. These components will all be assumed ideal. The mass of the load is known. The goal of the design questions is: To select which motor to use to conform to all system requirements.

In order to determine the requirements for the motor, the requirements for the SUD need to be made explicit first (step 1 of the workflow):

- The winch shall function with loads up to 1 kg.
- When lifting, the winch can lift the load to 4 meters within 2 seconds.
- While lifting the load, the winch cannot use more than 50 W continuously.
- The winch shall never use more than 200 W.

To translate the requirements into motor requirements the other system parameters are needed. These are shown in Table II.

Using the system parameters and requirements, we can do some quick estimations on how the motor should perform. The power supply is the simplest. If we assume a fixed 16 V power supply, the 50 W and 200 W convert to 3.125 A and 12.5 A respectively. To hold the mass level, we need 1 g of force. If we look at the transmission of torque to force in the winch we get:

$$T_{motor} = F_{Load} * r_{Drum} * i_{gearbox}^{-1}$$

Which gives us a required holding torque of:

$$T_{motor} = 9.81\,\mathrm{N} * 0.18\,\mathrm{m}/3 \approx 0.58\,\mathrm{N\,m}$$

To lift the mass 4 meters in 2 seconds, we would need an acceleration of about $4\,\mathrm{m\,s^{-2}}$. This means an extra 40% holding torque. The motor will also need to rotate at a maximum velocity of:

$$4\,\mathrm{m\,s^{-1}} * \frac{1}{0.18} * 3 \approx 67\,\mathrm{rad\,s^{-1}} \approx 4000\,\mathrm{RPM}$$

Based on these calculation we can formulate motor requirements:

- The motor will provide at least 0.58 N of torque at 3.125 A.
- The motor will have a maximum speed of at least 4000 RPM at 16 V

Given these requirements, a few tests are created (step 2). Two tests for the motor and three tests for the complete system. The motor tests are shown in Listing 2 and Listing 3. In these tests the motor parameters are validated with a constant current and voltage input. For these tests, the motor model is required, as it needs to be simulated.

For the total SUT there are three scenarios tested, to check the performance. The first scenario is to lift the load from 0 to

Listing 2
THE MAXIMUM TORQUE TEST

```
Feature "Maximum torque"

Scenario "Constant current"
    Given "TorquetestA.emx" in 20-sim for 2 seconds
    with "CurrentSource.I"=3.125

    Then F("DCmotor.p2.T">0.58)
```

Listing 3
THE MAXIMUM SPEED TEST

```
Feature "Maximum speed"

Scenario "Constant voltage"
    Given "SpeedtestA.emx" in 20-sim for 2 seconds
    with "VoltageSource.U"= 16

    Then F("DCmotor.p2.omega">66.7)
```

4.2 meters. Here the winch is checked on how fast it can raise the load, and power requirements. The 4.2 meters is chosen to be sure the load ends up above 4 meters. The second scenario is the holding situation, this checks whether the winch can keep up when keeping a load at a certain height, without too much position deviation. And the last one checks the power requirements when dropping the load 4.2 meters. These tests are shown in Listing 4.

For the implementation of the system three different motors are suggested (step 3). A separate dynamic model of each motor is made in 20-sim. These motors are characterised by their electrical properties. The parameters of the three motors are found in Table III.

Now the motors are individually tested, using the torque and speed tests (step 4). If the model passes these tests, the motor model is included in the full system. The controller of the system under test is then tuned to the new motor, and the system tests are run. If the motor passes these tests the motor is suitable for the current design.

Listing 4
THE SYSTEM-LEVEL TESTS FOR THE WINCH

```
Feature "Total system test"


Scenario "Lifting test"
    Given "Winch.emx" in 20-sim for 5 seconds
    with "Setpoint.C"= 4.2

    Then G(X 2 seconds("Mass.x">4))
    Then G("VoltageSource.current"<12.5)
    Then F("VoltageSource.meancurrent"<3.125)

Scenario "Holding test"
    Given "Winch.emx" in 20-sim for 5 seconds
    with "Setpoint.C"= 0

    Then G(abs("Mass.x")<0.2)
    Then G("VoltageSource.current"<12.5)
    Then F("VoltageSource.meancurrent"<3.125)

Scenario "Dropping test"
    Given "Winch.emx" in 20-sim for 5 seconds
    with "Setpoint.C"= -4.2

    Then G("VoltageSource.current"<12.5)
    Then F("VoltageSource.meancurrent"<3.125)
```

Motor	$K_m(\mathrm{N\,m\,A^{-1}})$	$R_{series}(\Omega)$	$I_{series}(\mathrm{H})$
MotorA	0.1	0.4	0.4
MotorB	0.2	0.8	0.25
MotorC	0.23	1.2	0.3

TABLE III
THE PARAMETERS OF THE DIFFERENT MOTORS SUGGESTED FOR THE WINCH

Listing 5
THE OUTPUT OF THE TEST TOOL WHEN TESTING MOTOR A WHEN RUNNING BOTH MOTOR LEVEL TESTS AT THE SAME TIME

```
speedtest.test : True
  Feature Maximum speed : True
    Scenario Constant voltage : True
      Constraint F("DCmotor.p2.omega">66.7) : True
torquetest.test : False
  Feature Maximum torque : False
    Scenario Constant current : False
      Constraint F("DCmotor.p2.T">0.58) : False
```

Motor A fails both the maximum torque test and the maximum speed test, so it is not even modelled in the compete system. The output of the tests of motor A are shown in Listing 5

Motor B passes both the speed and the torque test. This motor is thus included in the complete model. Then the system-level tests are run. The test results of the system-level tests with motor B are shown in Listing 6. Here we can see that motor B passes all the requirements set. So motor B is a valid choice for the winch, for now.

Motor C has also passed the speed and torque tests. It is subsequently included in the full system model, and the tests are run again. Motor C also passes all system-level tests.

Based on the current requirements, both motor B and motor C appear to be a valid option for the design. Depending on the implementation of the rest of the system under development this might change. But the system-level test should then notice immediately when this is the case. For now motor C is used in the rest of the development process, but it should be kept in mind that B was also valid.

Listing 6
THE TEST OUTPUT FOR MOTOR B FOR THE SYSTEM-LEVEL TESTS

```
systemtest.test : True
  Feature Total system test : True
    Scenario Lifting test : True
      Constraint F("VoltageSource.meancurrent"<3.125) : True
      Constraint G("VoltageSource.current"<12.5) : True
      Constraint G(X2seconds("Mass.x">4)) : True
      Constraint X2seconds("Mass.x">4) : True
    Scenario Holding test : True
      Constraint F("VoltageSource.meancurrent"<3.125) : True
      Constraint G("VoltageSource.current"<12.5) : True
      Constraint G(abs("Mass.x")<0.2) : True
    Scenario Dropping test : True
      Constraint F("VoltageSource.meancurrent"<3.125) : True
      Constraint G("VoltageSource.current"<12.5) : True
```

To show the effect that further design can have on the tests, a small change is made. The power supply of the winch is further developed, and a choice is made for relatively long cables. This means the power supply has a resistance of $1.7\,\Omega$. When this change is modelled, motor C is just a bit too slow and fails the system-level tests. This would be a good moment to remember motor B could also be valid, and run another test with the existing model for B, which would pass the tests.

It is clear that these test can quickly show if a SUD is still within its requirements, by simply running all the available tests on the new system model.

V. Conclusion

The aim of this work was to create a tool that supports the development of cyber-physical system by enabling automatic tests though models simulations. The tool shown here is a basis for a method that can support the design. Combined with the work-flow presented one should have all the elements required to setup automated tests.

The tests are defined based on a domain-specific language that can support multiple simulators and multiple different output types. Thus allowing the tool to support a wide range of models.

In the example, it is shown how the tool can be used to make design decision while selecting components. And it is shown how the tests can be used to verify the requirements of the system under development. It is even capable of detecting regression in the SUD, as all tests can easily be run again.

Thus, we can conclude that this tool is capable of supporting a test-driven design workflow for cyber-physical systems. Thus, enabling a robuster design and increased certainty on the correctness of the design.

As this tool is an initial development in the field of automated testing, there is still room for improvement. The tool can only select models, and change parameters of these models. When the design of a CPS get larger, it might be usefull to be able to swap out parts of a model. This depends of course on the capabilities of the simulator and model to support these kind of changes, but the testing language could be extended to at least support the possibility. Furthermore, depending on the simulator, the signal names might get very long. So it could be benificial to create support in the testing language to give aliases to specific signals. These aliases could then be imported at the beginning of every test file, thus allowing for shorter and more readable tests and test results.

There are two other suggestions, which are not based on the functionality of the tool, but on the ease of use. The first one is a way to easily quantify why a test has failed. Either by storing the complete data file and showing the traces, or by allowing a tester to immediately open that model/simulator to inspect the scenario. A second improvement is the ability to generate reports that can be saved and later referenced. These would include versions of models and tests used. This would allow for nighly tests, or at least a more hands-off approach. Furthermore the test DSL could be extended to include the use of units when defining constants. This would benefit the readability of the tests, and can be used as a sanity check during simulations.

We are currently already working on the first improvement, the capabilities to switch parts of a model.

References

[1] E. A. Lee, "Cyber physical systems: Design challenges," IEEE, May 2008, pp. 363–369, ISBN: 978-0-7695-3132-8. DOI: 10.1109/ISORC.2008.25.

[2] T. Broenink and J. Broenink, "Rapid development of embedded control software using variable-detail modelling and model-to-code transformation," in *Communications of the ECMS*, ser. 1, vol. 33, Jun. 11, 2019, pp. 151–157, ISBN: 978-3-937436-6.

[3] M. M. Bezemer, *Cyber-physical systems software development: way of working and tool suite*. Enschede: University of Twente, 2013, ISBN: 978-90-365-1879-6.

[4] B. George and L. Williams, "A structured experiment of test-driven development," *Information and Software Technology*, vol. 46, no. 5, pp. 337–342, Apr. 2004, ISSN: 09505849. DOI: 10.1016/j.infsof.2003.09.011.

[5] M. Grimm, R. Anderl, and Y. Wang, "Conceptual approach for multi-disciplinary cyber physical systems design and engineering," in *Proceedings of TMCE*, 2014.

[6] A. Mavin, P. Wilkinson, A. Harwood, *et al.*, "Easy approach to requirements syntax (EARS)," in *2009 17th IEEE International Requirements Engineering Conference*, Aug. 2009, pp. 317–322. DOI: 10.1109/RE.2009.9.

[7] T. Broenink and J. Broenink, "A variable detail model simulation methodology for cyber-physical systems," in *ECMS 2018 Proceedings edited by Lars Nolle, Alexandra Burger, Christoph Tholen, Jens Werner, Jens Wellhausen*, ECMS, May 25, 2018, pp. 219–225, ISBN: 978-0-9932440-6-3. DOI: 10.7148/2018-0219.

[8] (). "Cucumber — tools & techniques that elevate teams to greatness," [Online]. Available: https://cucumber.io/ (visited on 01/15/2020).

[9] G. Adzic, "Specification by example," *Book your training with Díaz & Hilterscheid!*, p. 20, 2011.

[10] E. A. Emerson, "CHAPTER 16 - temporal and modal logic," in *Formal Models and Semantics*, ser. Handbook of Theoretical Computer Science, J. Van leeuwen, Ed., Amsterdam: Elsevier, Jan. 1, 1990, pp. 995–1072, ISBN: 978-0-444-88074-1. DOI: 10.1016/B978-0-444-88074-1.50021-4.

[11] Controllab products. (2020). "20-sim," [Online]. Available: https://www.20sim.com/ (visited on 01/15/2020).

[12] Coppelia Robotics. (2020). "Robot simulator CoppeliaSim: Create, compose, simulate, any robot.," [Online]. Available: http://www.coppeliarobotics.com/ (visited on 01/15/2020).