RAPID DEVELOPMENT OF EMBEDDED CONTROL SOFTWARE USING VARIABLE-DETAIL MODELLING AND MODEL-TO-CODE TRANSFORMATION

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

KEYWORDS

systems, variable-detail, rapid prototyping, agile development, embedded control software, segway, model structure, model-to-code transformations.

ABSTRACT

This paper shows a method for the development of embedded control software of a cyber-physical system. The approach consists of two parts, a cycle for the rapid development of a set of features based on agile software development, and a variable-detail approach using model-driven development to develop and test single features. The method is used to develop the control system of a mini-segway, which is able to balance, steer and drive. This structured method gives effective results and a large set of models for future development.

INTRODUCTION

The development of cyber-physical systems comes with a set of challenges (Lee 2008). These can be tackled using a combination of model-driven development, (co-)simulation, and physical prototypes. A structured way to apply these different techniques when designing a cyber-physical system is required. This research focusses on the implementation of the cyber part of the system and assumes the physical part to be known. More specifically it focusses on the embedded control software. The physical part is made from off-the-shelf components, with known specifications and limitations. However this does not mean that the same techniques cannot be applied to the physical part of the system.

In this paper a structured way of developing and testing embedded control software is described. The design of these control systems should take into account both the limitations of the cyber and physical parts. The aim of this approach is to reduce the development of these systems to a series of short cycles based on rapid development principles and modelling techniques (Jensen et al. 2011). Rapid development techniques take their inspiration from software development strategies (Chandra 2015; Krishnan 2015). The aim of this is to break up the development into steps that can be individually tested. These immediate tests make it possible to receive feedback about the performance of the system and to spot errors earlier. This allows errors to be fixed early in the development process, thus requiring less effort. The short cycles make it possible to create simulations and prototypes early in the development process. When combining this with techniques to rapidly build a prototype, an early working version of the system can be made.

The approach outlined in this paper is based on two different cycles, a inner and an outer cycle (shown in Figure 1). The outer cycle details how to add features to the design at hand and how to come to a finished product. The inner cycle uses a variable-model detail approach (Broenink and Broenink 2018) to develop and test a feature.

The approach is demonstrated using a case study of a mini-segway (shown in Figure 5), a piece of hardware used for education in dynamic systems and control theory, where it is used to develop the control software.

This document first outlines the overall methodology. Then the inner and outer cycle of this development method are detailed. Then the example case is shown using a mini-segway, and the example case is evaluated. Finally the conclusions and recommendations for future research are discussed.

METHODOLOGY

The development method combines two different cycles, the rapid development cycle to develop embedded control software, and the variable-detail approach to implement and test the different features. The requirements of the control software are divided into different features, which are then individually implemented and tested.

When combining the two cycles, we get the following series of steps (see Figure 1):

- Order and split the features and levels of detail as

Communications of the ECMS, Volume 33, Issue 1,
Proceedings, ©ECMS Mauro Iacono, Francesco Palmieri,
Marco Gribaudo, Massimo Ficco (Editors)
ISBN: 978-3-937436-65-4/978-3-937436-66-1(CD) ISSN 2522-2414

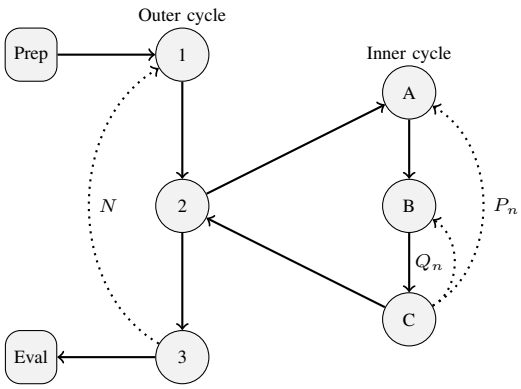

Fig. 1. The different cycles of the development process, of N features. whereby the amount of internal cycles for feature n are denoted as P_n and Q_n

preparation

1) Design new feature and tests.
2) Implement and test a new feature.
   A) **Design** a feature based on an ideal model
   B) Combine the feature with a more **detail**ed version of the rest of the system, adding more detail when needed.
   C) **Determine** if the tests pass if so, add more detail (go to B.). If it fails, redesign the feature (go to A).
3) Continue to next feature, until all features are implemented.

- Evaluate and reflect on the cycle process.

The details of these different cycles are explained in the following sections. An overview of these cycles is given in Figure 1.

Preparatory work

For this method, models with different levels of detail of the physical system are used. It is assumed that the development process is started with an ideal, or more abstract model. During the development process richer models are then used, until a point is reached where the model is competent to describe the behaviour of the prototype. The detail that is added represents the different aspects of the physical system or prototype, which can be included or abstracted away. When the model is competent, all aspects that influence the behaviour of the system to a significant degree have been added.

The order of adding these aspects should be determined before the cycles are started, as they influence the structures of the models that are required. The order in which the different aspects are added should be chosen carefully, as this can have a large impact in how quickly errors are found. Aspects of the design that are expected to be critical should be implemented first. As these aspects have a large chance of invalidating the current design (Chance of Failure). Furthermore steps that take a lot of effort to implement should be done as late as possible, as to change these again and again takes lots of time (Cost of Change). Taking into account these two qualifications one can try to minimize the amount of time a single iteration takes before finding an error, and reduce the time it takes to redo.

Cycle evaluation

Given a single Feature n, there are a certain amount of iterations of P_n and Q_n. These describe the amount of design variations of a feature, and the amount of different detail levels used respectively. Q_n contains the detail steps for all design variations. P_n is dependent on how well the initial implementation is designed. Q_n can be at most the amount of aspects that need to be added, for a single p_n. If one would plot the different iterations of step B, as shown in Figure 2, the efficiency of the method is indicated by how empty the matrix is. As there is a minimum amount of iterations needed, the best distribution of step B would only fill the top row and the rightmost column. In the ideal situation, which requires no redesigns, this matrix would be only a single filled column. In an situation where there are some design errors, requiring redesigns, this would look like the matrix shown in Figure 3. This is the minimal amount of B cycles required for a certain amount of redesigns (P_n).

When there are a lot more B cycles, it would suggest that the order in which the details were implemented was sub optimal. For example when most iterations reach $B_{p,3}$ it might be useful to test the aspect embodied by 3 first.

$$\begin{bmatrix} B_{1,1} & B_{2,1} & \cdots & B_{p,1} \\ B_{1,2} & B_{2,2} & \cdots & B_{p,2} \\ \vdots & \vdots & \ddots & \vdots \\ B_{1,q} & B_{2,q} & \cdots & B_{p,q} \end{bmatrix}$$

Fig. 2. A matrix plotting the instances of Step B for p and q.

$$\begin{bmatrix} B_{1,1} & B_{2,1} & \cdots & B_{p,1} \\ & & & B_{p,2} \\ & & & \vdots \\ & & & B_{p,q} \end{bmatrix}$$

Fig. 3. A matrix plotting the instances of Step B for p and q, in a fairly ideal situation

Set of models

Both of these cycles results in a set of models and descriptions of the complete system that can be used for further development. An overview of the set of models is shown in Figure 4.

It should be noted that subsequent features are not only designed and tested with the model of the physical system, but also with the models of the previous features. Thus the set of models representing the system is expanded at every detail level. As the physical part of the system was made

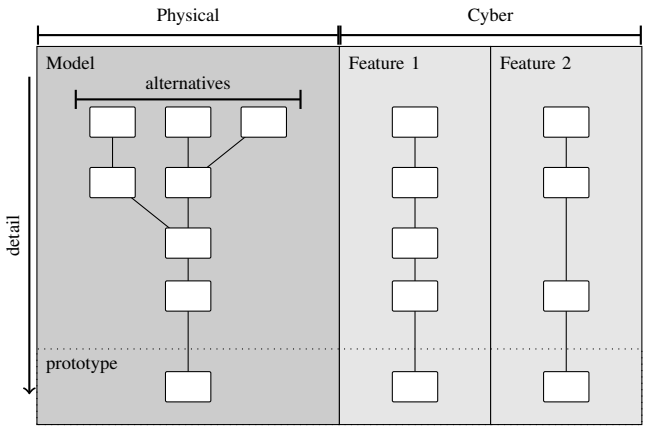

Fig. 4. The structured model set build with this methodology. Models at the same detail level are compatible and can be simulated together.

beforehand, an assumption is made that there are models available of the system.

After every rapid development cycle a new set of models is added, detailing the development and implementation of a feature. This feature can then be implemented and tested, either by using a model-to-code transformation (Hemel et al. 2008) or by implementing these models as code manually. The next feature can then be developed using the models of the physical system and models of all previous features. The method results in a set of models detailing the design and testing of all different features and changes in the embedded control software. Thus allowing later revisions and checks.

OUTER CYCLE

The outer cycle is a rapid development cycle based on short development cycles reminiscent of agile (Martin 2002) or spiral (Boehm 1988) software development techniques. To utilize these cycles properly it is important that intermediate stages of the design can be tested. Both to validate if it even works at all, and to test the performance of the system.

The testing of these intermediate stages of the design can be based on simulations or based on a prototype. The simulations requires that a simulation method is available to simulate the system competently enough to be sure of the results. It also requires a more extensive model of the physical system. If a prototype is used it requires that the intermediate stages of the development process can be quickly deployed on the prototype. This would typically require some form of model-to-code transformation or hardware-in-the-loop simulation. Usually a combination of the two is the most practical.

The cycles of this process are based on the features of the embedded control software. The functionality of this control software needs to be broken down into different features that can be individually implemented. It is important to note that these features should be split in such a way that the results of the individual cycles is testable. A cycle of the rapid development cycle contains the following steps:

1) Design new feature and the corresponding tests.
2) Implement and test a new feature.
3) Continue to the next feature, or finish the product.

The tests mentioned in these steps are an important part of the development process. When defining a feature one should have a clear image of how the feature should perform and how to test this performance. This clear definition allows one to quickly determine if a simulation or prototype performs as expected. It should also describe how to test the system and what to expect. This also gives the advantages inherent in a test-driven development approach (Janzen and Saiedian 2005). Step 2, the implementation and testing of the features is implemented using the variable-detail approach detailed in the following section.

INNER CYCLE

The inner cycle is a variable-detail approach, which is used to design and test the different features as selected by the rapid development cycle. The approach goes from a very low-detailed model, for example a control law, to a very detailed model that can be synthesised into code. This is done by adding all the aspects, as decided in the preparation phase, to the design. Within the design of this feature the following steps are used:

A) **Design** a feature based on an ideal model
B) Combine the feature with a more **detail**ed version of the rest of the system, adding more detail when needed.
C) **Determine** if the tests pass if so, add more detail (go to B.). If it fails, redesign the feature (go to A).

The large set of different models is used to leverage the advantages of model-based design of cyber-physical systems (Jensen et al. 2011) to a large degree. The variable-detail method assumes that there is not only a single model of the plant available, but that the plant model is available in multiple levels of detail. If these versions of the plant model do not exist, but a detailed model is available, these less detailed models can be made. As this paper focusses on embedded control software, the approach is explained using a control perspective.

The least detailed version of the controller is the control law. A single transfer function designed based on a linear representation of the plant, is the most abstract representation of the plant relevant to the problem. This control law can be designed using PID tuning, pole placement, or a compensator, etc.

If a controller is designed that is sufficient for the linearised plant it can be then tested on the non-linearised model. If the controller is still able to control the physical system, one can continue to the next step. If not, the controller has to be redesigned.

The following steps are to keep adding more detail and/or constraints to the plant model, and updating the controller accordingly. Examples of this could be to include limitations of IO, sensor processing, or discrete time. Every time detail is added, the new system can be tested using the more detailed implementation of the controller and the more detailed model of the plant.

In order to easily switch between these detail levels it is imporant that both the model and the controller are structured in a way to allow this. One way to do this is proposed in Broenink and Broenink (2018), but any method that allows rapid switching of detail will do.

When a level of detail does not conform to the specifications or tests, one must take a step back and look at the design decisions made in this step and in the previous steps. However the cause of the problem should be easy to determine, as only one step of detail was added.

When the feature is finished it should be detailed enough that it can be implemented on the prototype. This would be the case when all sensor processing, limitation and timing behaviour reflecting the real system are implemented.

MINI-SEGWAY EXAMPLE

To test it, the overall method is applied on a mini-segway (shown in Figure 5). This mini-segway is used for education of students in system dynamics and control theory. A model of the mini-segway is known. The mini-segway contains a raspberry-pi 3 and electronics to control the motors and sensors. It is also supported by the 20-sim4C (www.20sim.com/features/20sim4c.html) model-to-code tooling.

A complete model of the physical part of the mini-segway is available. This model is first abstracted into a few simpler models. The most basic model is the linearisation of the complete mini-segway around an equilibrium point. A more detailed abstraction is the non-linear model of the segway without any limitations and ideal sensors. This is followed by a model which takes into account the limitations of the motors. Then, there is a model with sensors, but still in continuous time. The last simulation is done with a model which also contains the discrete-time behaviour of the system. Finally the physical prototype of the mini-segway model is used. An overview of all models and steps used in this process is shown in Figure 6.

The embedded control software of this mini-segway has to implement three tasks:

Fig. 5. The mini-segway used in this example, motor drivers, and Inertail measurement unit (IMU) and absolute wheel encoders.

1) Balancing the mini-segway upright
2) Steering the mini-segway
3) Driving forward and backward

These three tasks are implemented using the posed methodology, as three different features. They are implemented in the order shown.

Before the method can be used the order of detail that have to be added has to be decided. Taking into account the limitations of the mini-segway there are three elements that need to be added to the ideal system:

- The maximum power of the motors
- The sensor behaviour and limits
- The sampling frequency of the raspberry-pi

The most important limit in this case is probably the limitations of the motor, with the sensors in second place. As the control frequency is about a 100 times faster then the natural frequency of the system (500 Hz vs 5Hz), the time discretization is expected to not be a problem.

For every feature first the feature and tests are described (step 1). Then the inner cycle can be started (step 2). For the inner cycle the different steps are be denoted as A_p, $B_{p,q}$, and $C_{p,q}$. Where p indicated the iteration of the design, and q the current level of detail.

The aspects added to the model are:

1) Non-linear model
2) Motor limitations
3) Sensor models including limits on readouts
4) Discrete time models
5) The prototype

Balancing

As a first feature the balancing controller is implemented. This has to only keep the mini-segway upright.

Step 1:

The assumptions are made that there is no steering, and that driving away does not mater. Even if this causes the mini-segway to fall over due to maximum velocities being reached. A linearised model is used that related the balancing angle to the motor voltage input.

Tests: The test for this feature consist of starting the mini-segway with (slightly) different initial balancing angles. It should be able to handle an error of at least 5°. The mini-segway should stabilize, but does not have to remain in a single position. It is acceptable if it falls over eventually due to increasing speed.

Step2:

A_1: The control law is designed by using pole placement based on the linearised model, to control the balancing angle of the segway.

$B_{1,1}$: This control law is then tested against the more detailed model which contains the non-linear behaviour of the mini-segway.

$C_{1,1}$: With the designed controller it has been observed that the mini-segway remains stable, and can stabilize from a unstable starting position. What is noted however is that the mini-segway starts driving away and requiring more and more motor power to keep uptight.

$B_{1,2}$:The first limitation is then added. The motor voltage is limited to the maximum voltage the driver can provide. In this case 24V.

$C_{1,2}$: When the mini-segway performace is simulated now, it is seen that it does not stabilize completely. There are a lot of oscillations due to the limitations of the motor in combination with the aggressive controller.

A_2: The control law is made less agressive.

$B_{2,1}$: The new control law is tested against the non-linear model.

$C_{2,1}$: The performance is slightly worse, but well within bounds.

$B_{2,2}$: The new control law is tested against the a model that includes the limitations of the motors.

$C_{2,2}$: There are no more oscillations and the mini-segway stabilizes. The maximum disturbance angle from which the mini-segway can recover is also reduced quite severely, but this is as expected. It is still well within bounds. When the mini-segway drives away, it can also be seen that it falls over as soon as it needs more then 24V. This is however within the bounds of the controller.

$B_{2,3}$: Until now, the controller gets the angle of the mini-segway directly from the simulation. This is not realistic, so the sensors need to be modelled. The mini-segway contains a inertial measurement unit with a 3 degree-of-freedom accelerometer and a 3 degree-of-freedom gyroscope. In order to estimate the angle of the mini-segway, the outputs of the accelerometer and gyroscope are combined using a complementary filter. This filter has to be added to the control software. To test the implementation of the processing it is important that the controller is simulated with the mini-segway model that contains the sensor limitations.

$C_{2,3}$:When this system is implemented it is seen that the mini-segway has more difficulty when starting from a unstable angle. However it can still stabilize, just after a few more seconds.

$B_{2,4}$:The final limitation before testing on the physical mini-segway is to convert the control software to discrete time. This is a simple operation in 20-sim.

$C_{2,4}$:This control software can then be simulated with the discrete model of the mini-segway, and it can be seen that there is hardly any difference in performance, as was expected.

$B_{2,5}$:After converting the control model to discrete time, it is then run on the actual mini-segway using 20-sim4C.

$C_{2,5}$:The mini-segway behaves as expected: It stabilizes, and starts driving away, before falling over due to the motor limitations. It does not drive straight though.

Step 3:

The feature behaves as expected. We continue to the steering of the mini-segway. All the different detail steps used for this controller are shown in Figure 6. The amount of cycles is shown in the leftmost branch of Figure 8.

Steering

The second feature is to keep the mini-segway straight while standing/driving.

Step 1:

The assumption for the initial linearised model is that the mini-segway does not fall over. This model shows the relation between the different wheel inputs and the steering velocity of the mini-segway. This model is shown as an alternative in Figure 6.

Tests: The test for this feature is to give the mini-segway a desired steering angle, in relation to the starting angle. It should reach this angle and not fall over.

Step 2:

A_1: Based on this model a controller is created using pole placement to control the steering and steering velocity of the mini-segway.

$B_{1,1}$: This controller is then tested against the non-linear model of the mini-segway that can fall over.

$C_{1,1}$: The mini-segway can turn stably and reaches the desired angle.

$B_{1,2}$: Motor limitations are added to the simulation.

$C_{1,2}$: The motor limits have no effect on the turning speed of the mini-segway, as the controller is slow enough to never reach these limits.

$B_{1,3}$, $C_{1,3}$, $B_{1,4}$, $C_{1,4}$: In the same way the sensor models and the discrete time are added. The performance remains approximately the same. The sensor implementation is trivial, as the mini-segway has absolute wheel angle encoders.

$B_{1,5}$: The new control software is implemented on the physical mini-segway.

$C_{1,5}$: The system behaves as expected, the mini-segway still drives away, but it drives away in a straight line.

Step 3:

The combined features behave as expected. We now continue to the position control of the mini-segway. All the different detail steps used for this controller are shown in Figure 6. The amount of cycles is shown in the middle branch of Figure 8.

Position control

The final feature is to keep the Forward velocity and position of the mini-segway.

Step 1:

To implement the position control, the assumption is made that the mini-segway does not fall over or turn, thus a linearised model is made where the angle of the mini-segway is assumed upright and does not turn. This model is shown as an alternative in Figure 6.

Tests: The test for this feature is to start the mini-segway at a position with a slight disturbance, and after a time let it drive forward a fixed distance. It should reach the final position with a small error.

Step 2:

A_1: Based on the linearised model a controller is designed using PID tuning, to control the forward velocity and position.

$B_{1,1}$: The model is tested against the non-linear model of the mini-segway, including the balancing and steering controller.

$C_{1,1}$: The mini-segway does not stay at the final position, but oscillates around it. The angle controller cannot handle the fast changes created by the position controller.

A_2: A slower controller is designed, with a lag compensator to make sure the steady state error is minimal.

$B_{2,1}$: The model is tested again with the non-linear model of the mini-segway.

$C_{2,1}$: The position controller behaves as expected. Not as fast as the initial control law, but at least it is stable.

$B_{2,2}$, $C_{2,2}$, $B_{2,3}$, $C_{2,3}$: The motor limits and the sensors are added. There is no significant difference in the behaviour of the mini-segway, however the overshoot is slightly larger.

$B_{2,4}$: The controller is transferred to discrete time.

$C_{2,4}$: There is no significant difference.

$B_{2,5}$: The complete control system is implemented on the physical mini-segway

$C_{2,5}$: The mini-segway can remain upright, and keep position, by changing the set-point it can drive around.

Step 3:

The complete controller behaves as expected. All features are now finished and the control software is done. New features could now be thought of and the whole cycle could be started again. The complete set of models made for the design is shown in Figure 6. The amount of cycles is shown in the rightmost branch of Figure 8.

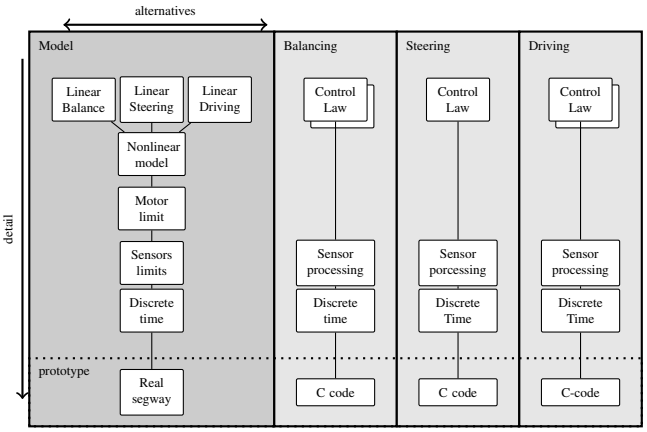

Fig. 6. The complete model structure used in controlling the mini-segway. The models contain the key aspect or limitation added. Models on the same detail level can be simulated together.

Evaluation

The total design flow is visualized in Figure 8. In this figure the different features and cycles are shown. However this does not visualize the distribution of the iterations of step B over the different design iterations. This is why the same matrix plot is used as described in the method, to plot the cycles of the different features. These matrices are shown in Figure 7.

Balance

$$\begin{bmatrix} B_{1,1} & B_{2,1} \\ B_{1,2} & B_{2,2} \\ & B_{2,3} \\ & B_{2,4} \\ & B_{2,5} \end{bmatrix}$$

Steering

$$\begin{bmatrix} B_{1,1} \\ B_{1,2} \\ B_{1,3} \\ B_{1,4} \\ B_{1,5} \end{bmatrix}$$

Position

$$\begin{bmatrix} B_{1,1} & B_{2,1} \\ & B_{2,2} \\ & B_{2,3} \\ & B_{2,4} \\ & B_{2,5} \end{bmatrix}$$

Fig. 7. A matrix plotting the instances of Step B for the different features

When comparing these matrices to the ones described in the method it appears that the assumptions of the order of detail were reasonably correct. Design errors were found reasonably early in the process.

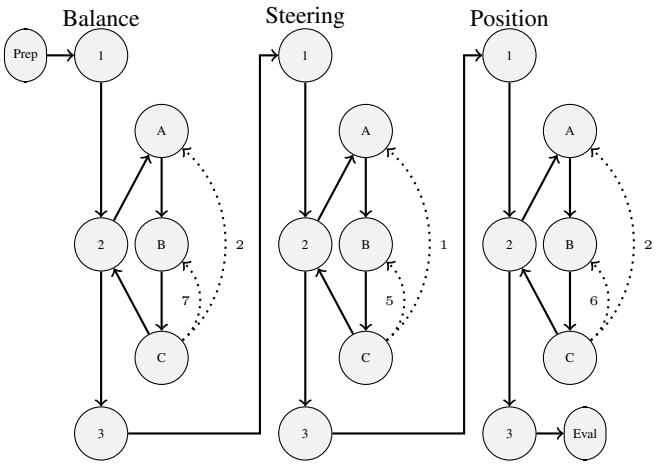

Fig. 8. The different cycles of the development process, of the controller of the mini-segway., including the amount of cycles performed.

CONCLUSION

The methodology proposed has been successfully applied to developing control software for the cyber-physical system at hand. The structured approach allows for rapid development of the embedded control software. The iterative inner cycle makes it straightforward to spot small errors while they are still easy to fix. It also allows for staged development of the features and quickly testing on a prototype.

The evaluation of the process gives extra insight in the order of importance for different details of the model. This would allow the improvement of the efficiency of the development process by switching these around for new features. One would have to take into account that this switch would require extra work, as the models need to be remade.

The model structure that is obtained by this approach is quite extensive. There are models of the cyber and physical part of the system at all levels of detail. These models facilitate adding extra features or testing the system. The different tests at different levels of detail give a lot of insight on where certain errors or constraints originate.

FUTURE WORK

The current method could be improved by using more data on its use. Different design challenges can be undertaken with the same method, which can then be compared.

To further improve this method a way should be chosen of formalizing and automating the testing of different models during the development phase. If tests are created when writing specifications of features, one can prevent regressions and quickly determine whether or not a certain controller design passes al requirements. The automation would also allow for more extensive tests to either compare different designs or to do design space exploration.

The current method uses model-to-code transformations for prototyping. This is still based on an external connection via 20-sim4C. The possibility of generated code for a stand-alone production environment should be investigated.

REFERENCES

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5):61–72.

Broenink and Broenink (2018). 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*, pages 219–225. ECMS.

Chandra, V. (2015). Comparison between various software development methodologies. *International Journal of Computer Applications*, 131(9):7–10.

Hemel, Z., Kats, L. C., and Visser, E. (2008). Code generation by model transformation. In *International Conference on Theory and Practice of Model Transformations*, pages 183–198. Springer.

Janzen, D. and Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50.

Jensen, J. C., Chang, D. H., and Lee, E. A. (2011). A model-based design methodology for cyber-physical systems. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pages 1666–1671. IEEE.

Krishnan, M. S. (2015). Software development risk aspects and success frequency on spiral and agile model. *International Journal of Innovative research in computer and communication Engineering*, (3):1.

Lee, E. A. (2008). Cyber Physical Systems: Design Challenges. pages 363–369. IEEE.

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

AUTHOR BIOGRAPHIES

Tim Broenink (MSc 2016) is a PhD student at the Robotics and Mechatronics Lab of EE at the University of Twente. He is working on a project for a NWO-TTW perspective program regarding cyber physical systems. His track within this project relates to robust motion control for cyber-physical systems. His research interest includes behavior-driven development of hardware and software, automated testing, and co-simulation.

Jan Broenink (MSc 1984; PhD 1990) is Associate Professor in Embedded Control Systems at the Robotics and Mechatronics Lab of EE at the University of Twente. His current research interests are on cyber-physical systems, embedded control systems (realization of control schemes on mostly networked computers) and software architectures for robotics. For that, he is interested in: model-driven design and meta-modeling of robot software architectures; designing software tools including (co)-simulation; concurrent and systems engineering. Since July 2017 he is chairman of the Robotics and Mechatronics group, together with prof. Stefano Stramigioli, who is Scientific Director.