Context-Oriented Equation-based Modeling in ModelingToolkit.jl

Christian Gutsche[1, 2] Christoph Seidl[2] Volodymyr Prokopets[2]
Sebastian Götz[2] Zizhe Wang[1, 2] Uwe Aßmann[2]

[1]Boysen–TU Dresden–Research Training Group, Dresden, Germany
[2]Chair of Software Technology, Technische Universität Dresden, Dresden, Germany
{christian.gutsche, christoph.seidl, volodymyr.prokopets, sebastian.goetz1,
zizhe.wang, uwe.assmann}@tu-dresden.de

Abstract

Cyber-physical systems are self-adaptive, changing their behavior at run time to adapt to their context. Hence, their simulations must also handle variability at run time. The lack of support for variability in industrial equation-based modeling languages, such as Modelica, causes problems when simulating self-adaptive systems, e.g., they only limitedly support structural variability, and state transitions are based on if-then-else conditions that can cause conflicts, especially for complex control mechanisms.

We present a modeling technique for equation-based models containing variability by implementing concise and dedicated language constructs to express state space and transitions via contextual modeling. Contextual modeling abstracts the modeled world and allows the definition of constraints that reduce the risk of reaching conflicting states. We demonstrate the feasibility of our approach on a case study, presenting the advantages of our modeling technique regarding the definition of state control and the reduction of risk for reaching conflicting states.

Keywords: Cyber-physical systems, Self-Adaptive Systems, Contexts, VSS, Julia, Simulation

1 Introduction

The 5C architecture of cyber-physical systems (CPSs) (Lee, Bagheri, and Kao 2015) includes a configuration level that implies system behavior variability. Therefore, when simulating CPSs, their simulations must also handle variability. Variability at simulation time is supported only partially by the equation-based modeling (EBM) language Modelica (Fritzson and Engelson 1998), e.g., Modelica does not support conditional definitions of components during simulation (Zimmer 2010) and has limitations with switching equations at simulation time if the DAE index changes (Benveniste, Caillaud, and Malandain 2020). Even if these limitations are bypassed, Modelica supports only if-then-else or when conditions based on Boolean expressions. These expressions become complex and incomprehensible for complex state spaces and transitions. This lack of expressiveness results in an inconvenient definition of variability in simulation models.

In this paper, we present context-oriented equation-based modeling (COEBM), a modeling technique for EBM containing variability by implementing concise and dedicated language constructs to express state space and transitions via contextual modeling. Contexts abstract the simulated world and possible control actions. Constraints between contexts are checked during runtime, preventing reaching conflicting states, e.g., simultaneously charging and discharging a battery. During simulation, context changes imply transitions between models to achieve structural variability. We further develop *Contexts.jl*, presented in our previous work (Gutsche et al. 2024), to achieve better applicability of context-oriented programming (COP) and context modeling in EBM.

For our implementation, we use the EBM language *ModelingToolkit.jl* (MTK) (Ma et al. 2021). MTK is implemented as a library-based embedded domain-specific language (DSL) for the general-purpose programming language Julia (Bezanson et al. 2015; The Julia Project 2023). This enables utilizing Julia's features for extending the modeling language (Rackauckas and Nie 2019) and using the expressiveness of a programming language in callbacks during simulation time. We present how to extend MTK with syntax suitable for COEBM.

The main contributions of this paper are:

1. Integrating COP in EBM for concise syntax for variable reinitialization
2. Integrating *Contexts.jl* language constructs in MTK to define structural variability
3. Using context modeling with constraints to prevent conflicting states in EBM and implementing state control mechanism

We present a case study to evaluate the contribution's feasibility for simulating self-adaptive systems. Therefore, we model an energy park for green hydrogen production with COEBM. We port Modelica models to MTK and implement the control logic with contextual modeling. Simulation speed is crucial when using simulations to support CPS operation. Therefore, we also provide performance measurements for our COEBM implementation.

In section 2, we present background information on Julia and context modeling, followed by an overview of related work about variable structure system (VSS) simulations in EBM in section 3. In section 4, we present the concept of COEBM and how to simulate systems containing variability with COEBM. The evaluation, consisting

of the mentioned energy park case study and performance measurements, is presented in section 5. To conclude, we provide a discussion in section 6 and a conclusion and outlook for future work in section 7.

2 Background

2.1 Julia and ModelingToolkit.jl

Julia (Bezanson et al. 2015; The Julia Project 2023) is a just-in-time compiled general-purpose programming language. A major difference from other commonly used programming languages is the multiple dispatch paradigm. Instead of defining functions within classes and dispatching over the function's object, the dispatch relies on the types of the function's arguments. Some of the advantages of this design choice are discussed later. Besides this, Julia has powerful metaprogramming capabilities.

In addition, Julia provides the rich ecosystem of scientific machine learning (SciML) containing libraries for mathematical computation and optimization (Lubin et al. 2023), numerical equation solvers (Rackauckas and Nie 2017), and EBM (Ma et al. 2021). The implementation in Julia enables utilizing multiple dispatch, which allows developers to, thanks to design choices of *DifferentialEquations.jl*, extend the functionality of the SciML ecosystem (Rackauckas and Nie 2019).

MTK is an embedded DSL implementing Modelica-like object-oriented EBM in Julia. MTK components consist of variables, parameters, equations, sub-components, callbacks, and Julia code. Models defined with the macro `@mtkmodel` are translated to equation systems. Depending on the mathematical structure, MTK provides, among others, `ODESystem` and `PDESystem`. These systems are used to define `ODEProblem`s containing initial values and a time span. `ODEProblem`s are solved with an implemented `solve` function where different solver algorithms are available. MTK benefits from support for extensibility of the SciML ecosystem (Rackauckas 2021). We implement our modeling technique in MTK because of this support for extensibility and the possibilities for combining Julia language features with EBM.

2.2 Context Modeling

Contexts describe the situation and environment in which a program is executed as discrete categories. Context-awareness (Abowd et al. 1999) enables software to achieve dynamic adaptation of its behavior to the current situation by specifying behavior for different contexts. Contexts are activated and deactivated based on the global state of the program or its environment. We call this state activeness. A programming paradigm based on this principle is called COP (Keays and Rakotonirainy 2003; Hirschfeld, Costanza, and Nierstrasz 2008). In COP, developers can specify contextual variants of a function. During the program's run time, the current activeness of contexts is evaluated, calling the function defined for this context. Therefore, dynamic adaptivity is achieved.

In our previous work (Gutsche et al. 2024), we presented a COP implementation in Julia. We introduced the library *Contexts.jl*, which provides concise syntax based on metaprogramming to define contexts and achieve context-dependent behavior for functions by utilizing Julia's multiple dispatch. In (Gutsche et al. 2024), we also already showed that combining *Contexts.jl* and MTK to implement variability in equation-based models with contexts is possible.

As presented in (Gutsche et al. 2024), context modeling in *Contexts.jl* is based on singleton objects representing contexts. The activeness of contexts can be constrained by an approach based on context Petri nets (Cardozo et al. 2012). The Petri nets defined in *Contexts.jl* are Dynamic Feature Petri nets (Muschevici, Clarke, and Proença 2010) that represent the constraints from (Cardozo et al. 2012). For example, an exclusion constraint prevents two specific contexts from being active simultaneously, and weak inclusion constraints imply that the activation of one context causes the activation of another context. This reduces the risk of reaching conflicting states.

3 Related Work

Several approaches aim to increase functionalities for specifying and simulating variability in EBM. As (Wang et al. 2025) show, these approaches focus on enabling EBM to simulate models containing structural variability, called VSSs (Utkin 1977).

Several new languages were designed, heavily inspired by Modelica, including Sol (Zimmer 2010), Hydra (Giorgidze 2012), and Modelyze (Broman and Siek 2012). While they support structural variability, advanced modeling techniques to specify state spaces and control mechanisms are not implemented. The already mentioned MTK and *Modia.jl* (Elmqvist, Neumayr, and Otter 2018) are new EBM implementations, implemented as embedded DSLs in Julia. In contrast to MTK, *Modia.jl* has dedicated support for VSS simulation (Neumayr and Otter 2023) but also lacks advanced modeling techniques.

DySMo (Mehlhase 2014), MoVaSe (Esperon, Mehlhase, and Karbe 2015), and PyVSM (Stüber 2017) rely on scripts that orchestrate the simulation. DySMo and PyVSM use Python to simulate different models, defining variants, e.g., modeled with Modelica. While they enable VSS simulation, they lack modeling techniques for constructing state space and transitions. MoVaSe, later called MoVaSim, is implemented in the Eclipse ecosystem and includes modeling techniques to specify model variants. However, the only available references to MoVaSim are a Work-in-progress paper (Esperon, Mehlhase, and Karbe 2015) and a dissertation (Gómez Esperón 2017). The editor itself is not available. In these approaches, variability modeling and state control are performed in non-EBM technical spaces, namely Python and Eclipse, creating a hurdle for users.

OM.jl (Tinnerholm, Pop, and Sjölund 2022) is a Mod-

DOI 10.3384/ecp218155

elica compiler written in Julia. *OM.jl* adds syntax for VSS allowing specifying different modes and their transitions. However, also *OM.jl* lacks advanced modeling to specify mode transitions. An approach that aims to improve this modeling is presented in (Wang et al. 2025), where a preprocessor is used to implement syntax for specifying contexts that control the modes. The preprocessed code is then compiled by *OM.jl*. However, only one context can be active at a time, and context activeness is controlled by Boolean conditions based on the time value. Therefore, the advantage of using contexts is lost.

We identify a gap: none of them supports both VSS simulation and advanced modeling of the state space and control mechanisms with dedicated syntax.

4 Context-Oriented Equation-based Modeling

We use contexts in COEBM to achieve behavior variability. Therefore, a well-defined context model is the base of COEBM. This section first describes the modeling of such context models. Afterwards, we present two ways to use contexts and COP for variability: variable reinitialization and structure change.

4.1 Contexts in Equation-based Modeling

We classify contexts into measurable and control contexts. Measurable contexts change their activeness based on measured values of observables, e.g., a temperature. Those observables cannot be directly influenced. Control contexts describe states or variables controlled by operators, e.g., a variable voltage. By changing the context, the operational mode changes.

In (Gutsche et al. 2024), we presented how *Contexts.jl* enables the definition of contexts with concise syntax based on metaprogramming and how constraints based on (Cardozo et al. 2012) are used to describe the state space. Using primarily measurable contexts to imply the activation and deactivation of control contexts via constraints enables the definition of control mechanisms while minimizing the risk of reaching conflicting states.

We introduce a new constraint for contexts, the `alternative`. This constraint implies that out of N contexts, exactly 1 context must be active at any time. This constraint occurs frequently in modeling but was not described by (Cardozo et al. 2012) or (Gutsche et al. 2024). We implemented the constraint in *Contexts.jl* using a Petri net like the library's other constraints. The constraint checks two things when a context's activeness is changed. If the currently only active context gets deactivated, this deactivation will be undone, and if a context $C1$ gets activated, but an alternative context $C2$ was active before, $C2$ becomes deactivated.

Another concept that we introduce is context groups. A context group implies an `alternative` constraint between the contexts that compose the group. Calling the group object will return the currently active context of this group. Context groups should describe orthogonal context dimensions to prevent conflicts. We also implemented context groups in *Contexts.jl* to improve the applicability for COEBM. An example of how to define contexts within a context group is shown in Listing 1.

```
1 @newContext Cold, Medium, Warm
2 activateContext(Medium)
3 Temp = ContextGroup(Cold, Medium, Warm)
4 Temp() # returns Medium
```

Listing 1. Example for defining contexts `Cold`, `Medium`, and `Warm` as well as a context group `Temp`. These contexts describe the Temperature as a discrete classification.

Modeling the context space should consist of four steps. First, identify relevant contexts. Second, cluster contexts to context groups. Third, classify contexts as measurable or control contexts. Fourth, define constraints between contexts of different groups. The activation and deactivation of measurable contexts may imply the activation and deactivation of control contexts, but not vice versa.

An example is a temperature control system. For the sake of simplicity, we assume that the system either activates heating or cools the system by activating the air conditioning (AC). These two possible operations create four contexts: `HeatingOn` and `HeatingOff`, composing the group `Heating`, and `ACOn` and `ACOff`, composing the group `AirConditioning`. For a basic control mechanism, the activeness of heating and AC might only automatically depend on the temperature. We define a context group `Temperature` consisting of the contexts `Cold`, `Medium`, and `Warm`.

While the temperature contexts are measurable, the contexts for heating and AC are control contexts. We model the control contexts' activeness based on the activeness of the temperature contexts. Therefore, we specify the following constraints:

- `Cold` weakly includes `HeatingOn`
- `Medium` weakly includes both, `HeatingOff` and `ACOff`
- `Warm` weakly includes `ACOn`

Note that because `Cold` and `Warm` exclude each other, `ACOn` and `HeatingOn` will never be activated automatically at the same time. To also prevent manual simultaneous activation of `ACOn` and `HeatingOn`, we define an exclusion rule between those two contexts. A visualization of the specified context model is shown in Figure 1.

We use context models within EBM in MTK to control simulation variability. MTK allows the definition of discrete and continuous events. We use events to call callback functions that include the functionalities of *Contexts.jl* to activate or deactivate contexts. Based on the defined constraints, other contexts change their activeness, and the simulation model adapts based on these changes. Hereby, adaptation can be implemented by variable (or parameter) reinitialization or structural changes.

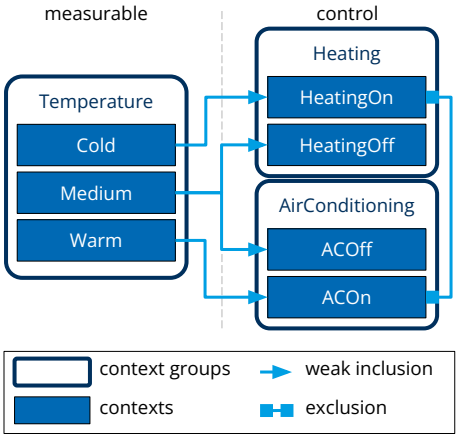


Figure 1. Context model for a heating system consisting of three contextual groups. The implicit alternative constraints between contexts of one group are not separately visualized.

4.2 Context-Oriented Variable Reinitialization

MTK supports variable reinitialization to achieve discrete changes of variable values. We extend this by using the COP features of *Contexts.jl* to control the reinitialization based on changes in the context activeness. Again, we explain this mechanism with the example from above. Therefore, we assume heating can be simulated as on or off by reinitializing a voltage value. We model possible context transitions within one context group as a state machine. For example, the temperature context transitions from `Cold` to `Medium` if $T \geq 15\,°\text{C}$ and from `Medium` to `Cold` if $T \leq 14\,°\text{C}$. The difference between temperature thresholds prevents cyclic event calls. Accordingly, transitions between `Warm` and `Medium` are defined, but transitions between `Warm` and `Cold` are not foreseen. Callback functions in MTK are Julia functions, i.e., Julia's complete expressiveness is available. In Listing 2, a callback function is defined on lines 9 to 26, and computational logic representing the state machine is shown on lines 10 to 24. In line 25, a context-oriented function is called to reinitialize the voltage value if the previously performed context changes influenced the activeness of the heating context. The two context-specific definitions for the function are shown in lines 1 to 7.

In this explanatory example, the voltage values could be reinitialized directly based on temperature values instead of activating the temperature contexts. However, for spaces containing multiple measurable context groups or when activating control contexts also depends on the activeness of other control contexts, separating the state machines for measurable contexts from the activation of control contexts is advantageous. Such an example will be presented in section 5.

4.3 Contextual Views for Structural Changes

A case not natively supported in MTK (or Modelica) is variability due to changing, adding, or deleting variables,

```
1  @context HeatingOn function setV(integ, u)
2    integ.u[u.V] = 12
3  end
4
5  @context HeatingOff function setV(integ, u)
6    integ.u[u.V] = 0
7  end
8
9  function tempChange!(integ, u, p, ctx)
10   if isActive(Warm)
11     if integ.u[u.T] <= 24
12       activateContext(Medium)
13     end
14   elseif isActive(Medium)
15     if integ.u[u.T] <= 14
16       activateContext(Cold)
17     elseif integ.u[u.T] >= 25
18       activateContext(Warm)
19     end
20   elseif isActive(Cold)
21     if integ.u[u.T] >= 15
22       activateContext(Medium)
23     end
24   end
25   @context Heating() setV(integ, u)
26 end
```

Listing 2. Example for a callback function. The computational logic from lines 10 to 24 represents a state machine changing the temperature context based on the current temperature value. The function call on line 25 is a context-oriented call of the function defined on lines 1 to 7, performing a variable reinitialization.

equations, and whole components. We realize the simulation of such VSS by defining distinct differential equation systems that are represented by `ODESystem` objects in MTK. When simulating the whole system, the solving algorithm must switch between those `ODESystem` objects and perform state migration. The algorithm must transfer the end values of all variables to the initial values for the respective variables in the new `ODESystem`. This workflow, realized in Python and not for MTK, was already applied in DySMo (Mehlhase 2014) to simulate VSS. We implement an adapted version of this workflow where contexts decide which model is simulated next.

In our modeling technique, when to switch the model and which model should be simulated next is decided based on the context model, called contextual VSS (CVSS). Therefore, every model variant is assigned to a context or multiple contexts linked with logical operators. Every context must be unambiguously mapped to a model to ensure systems have a well-defined behavior.

To simulate CVSS in MTK without performing the state migration manually, we implemented multiple features. As described in (Rackauckas and Nie 2019), we implement this extension based on Julia's type system and multiple dispatch paradigm.

1. We defined a new problem type for contextual VSS, `CVSSProblem`, as a Julia `struct`. Instead of a single `ODESystem` object, the `struct` contains a dictionary, here called `f`, for the perviously men-

DOI
10.3384/ecp218155

tioned mapping of contexts to `ODESystems`.

2. We specify a new implementation of the `solve` function of *DifferentialEquations.jl* for `CVSSProblem` types with multiple dispatch. This `solve` function checks the keys of `f`, which are the contexts `ODESystems` are mapped to. If a context is active, the respective `ODESystem` will be used for simulation. Therefore, an `ODEProblem` and a respective `Integrator` object are created. The initial values `u0` and simulation start time `t0` are set to either the original start values or the end values of the previous partial solving process. Then, the `ODEProblem` is solved, and its `ODESolution` is saved. If the simulation is restarted, the process is repeated. The simulation ends if the specified ending time is reached, the function `terminate!` is called, or an error occurs. The workflow is visualized in Figure 2
3. We define a new function `restart!`. It calls the `terminate!` function to stop the solving process. If a termination with `restart!` is detected by the `solve` function, the activeness of contexts is checked again, followed by solving the respective `ODESystem`.
4. We define `VSSSolution` as a subtype of `SciMLBase.AbstractODESolution`. The `VSSSolution` contains all the attributes of ODE solution `structs` but also contains a vector of the `ODESolution` objects obtained during the solving. We implement the `getindex` function for the `VSSSolution` objects so that the solutions are obtained as usually done in MTK.

```
1  function tempChange!(integ, u, p, ctx)
2    ACContext = AirConditioning()
3    ## State machine as shown in
4    ## Listing 1, lines 10 to 25
5    if ACContext != AirConditioning()
6      restart!(integ)
7    end
8  end
9
10 f = Dict(ACOn => system_acOn,
11           ACOFF => system_acOff)
12 prob = CVSSProblem(f, [], (0, 60))
13 sol = solve(prob, RadauIIA5())
```

Listing 3. Example for a callback function (lines 1 to 8) and the code to simulate the system (lines 10 to 13). The function `restart!` (line 6) is used to stop the simulation and restart it with a new mode based on the context activeness of the AC context. In lines 10 and 12, the dictionary `f` is defined that maps context to `ODESystem` objects that are used for simulation. With `f`, we define a `CVSSProblem` (line 12) and simulated by solving the `CVSSProblem` (line 13).

We assume the heating example from above is modeled as a VSS system where there is one model for active AC and one model for inactive AC. Accordingly, we create two `ODESystems`, that we call `system_acOn` and `system_acOff`. Both `ODESystems` contain events that call the callback function `tempChange!` as we defined in Listing 2, but we extend it with the model switching logic as shown in Listing 3 on lines 1 to 8. In line 2 in Listing 3, we save the AC context before changing the temperature context. If the AC context was changed due to changes in the temperature context, the `restart!` function is called to perform the model switch. The definition of the context to feature map `f` is shown on lines 10 and 11. Line 12 shows `CVSS` definition and line 13 the solve call, recreating MTK syntax.

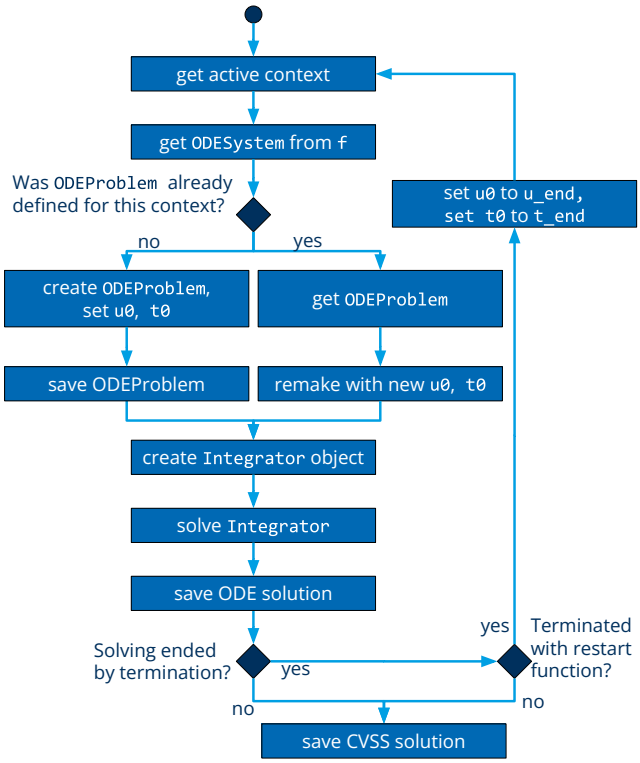


Figure 2. Workflow for solving a CVSS problem. The algorithm checks which contexts mapped to `ODESystems` are active. If this `ODESystem` is solved for the first time, a respective `ODEProblem` is created with the initial values `u0` and start time `t0` and saved. If the `ODESystem` is used another time during solving the `CVSSProblem`, this `ODEProblem` is obtained again and newly initialized with `remake` using the new `u0` and `t0`. An `Integrator` object is created from `ODEProblem` and the `ODESystem` is solved, and the solution is saved. If the simulation was stopped with the `restart!` function, the end values of variables are used to restart the simulation within another context. In the end, the solution is returned as a `VSSSolution` object.

5 Evaluation

We show the feasibility of our modeling process by presenting a simulation of an energy park where hydrogen is produced from renewable energy sources as a case study. After presenting the modeling process and simulation results, performance measurements are provided.

5.1 Case Study: Modeling of an Energy Park

The energy park model consists of wind turbines and photovoltaic (PV) modules for producing electricity, a battery, and an electrolyzer to produce green hydrogen. Also, the park is connected to the electricity grid. Because only green hydrogen should be produced, hydrogen production must be stopped when renewable electricity is unavailable. Also, excess electricity should be stored in a battery to power the electrolyzer with green electricity, even if the weather conditions would not allow it. The energy park should adapt its behavior based on its state, Hence, it is self-adaptive, and context awareness is advantageous.

One of the disadvantages of MTK compared to Modelica is the relatively low availability of libraries. Therefore, most of the needed components have been modeled by us. The models are inspired by Modelica libraries like the Modelica Standard Library, PhotoVoltaics (Brkic et al. 2019), and Wind Power Plants (Eberhart et al. 2015). The electrolyzer model and system design are inspired by (Migoni et al. 2016). A scheme of the system is shown in Figure 3.

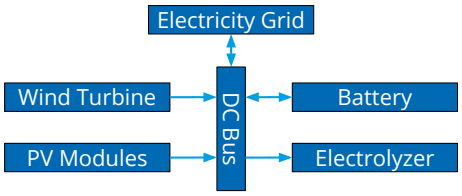


Figure 3. Scheme of the simulation model. Wind turbines and PV modules supply electricity, and an electrolyzer consumes electricity. A battery is used for storing excess electricity. The system can also exchange electricity with the grid.

As this simulation does not aim to give detailed insight into the technical properties of an energy park, the model contains numerous simplifications. We simplified the components themselves compared to the implementations in Modelica. We modeled the system completely in DC with a constant voltage, and converters are neglected. The hydrogen has no physical properties besides mass. Nevertheless, the electricity and hydrogen output are in the correct order of magnitude, and their behavior is modeled reasonably. Because these properties are significant for the presented control via COEBM, the model is suited to show the feasibility of COEBM.

To only produce green hydrogen, two variation points can be controlled: First, the battery can be charged and discharged, and second, the electrolyzer can be put into active and standby mode. Battery charging and discharging is controlled by variable reinitialization, while the electrolyzer operation mode is controlled with a structural change. The active electrolyzer is modeled inspired by (Migoni et al. 2016), while we model standby mode as a resistor representing a constant electrical load. When to switch modes depends on different variable values and the system's state itself.

The variation points are directly translated to contexts. The control contexts `BatteryCharging`, `BatteryDischarging`, and `BatteryIdle` form the context group `BatteryChargingMode`. The contexts `ElectrolyzerActive` and `ElectrolyzerStandby` form the context group `ElectrolyzerOperation`. The measurable contexts represent three properties. First, the power that is fed into or drawn from the grid. We define 4 contexts, `GridOutput` when power is drawn from the grid and `GridInputLow`, `GridInputHigh`, and `GridInputHighWithElectrolyzer`, representing different quantities of power supply to the grid. `GridInputHighWithElectrolyzer` got its name because it can only be reached if the electrolyzer is operated with maximal power. The second context group represents the state of charge (SoC) of the battery containing the contexts `BatterySoCMin`, `BatterySoCMedium`, and `BatterySoCMax`. And third, we define two contexts `ElectrolyzerMaxPower` and `ElectrolyzerMinPower` that are activated based on the voltage with which the electrolyzer is operated. Note that those contexts are not part of a context group because they can be deactivated simultaneously. The context model is visualized in Figure 4.

The lower part of Figure 4 represents the weak inclusion constraints, which implement the control logic. We define seven constraints. Every constraint realizes a specific control step. If the electrolyzer is not operated with maximal power and the battery is charging, charging is stopped (C1). If the electrolyzer is not operated with maximal power and the battery is in idle mode and not empty, the battery gets discharged (C2). Those constraints increase hydrogen production. If the battery discharges but becomes empty, it is set to idle mode (C3). Respectively, if the battery charges but becomes fully charged, it is set to idle mode (C4). If the electrolyzer is active and at minimum power, but grid input is low or we have grid output, the electrolyzer is set to standby mode (C5). This constraint prevents producing hydrogen from grid electricity. If the electrolyzer is in standby mode and the grid input is high, the electrolyzer is set to active mode (C6). If the electrolyzer is operated with maximal power and a certain amount of power is supplied to the grid, the battery is charged if it is not already fully charged (C7).

We define events and callback functions to activate measurable contexts during simulation time. For each of the two context groups `GridState` and `BatterySoC`, we define one callback similarly designed as the one shown in Listing 2. Again, we implement a state machine to control the activation of measurable contexts, and constraints imply the activation of control contexts. We use the `restart!` function and context-oriented functions for variability control. We also define two callback functions for each context: `ElectrolyzerMaxPower` and `ElectrolyzerMinPower`. One function activates the context, and one deactivates it. One of those callback

DOI
10.3384/ecp218155

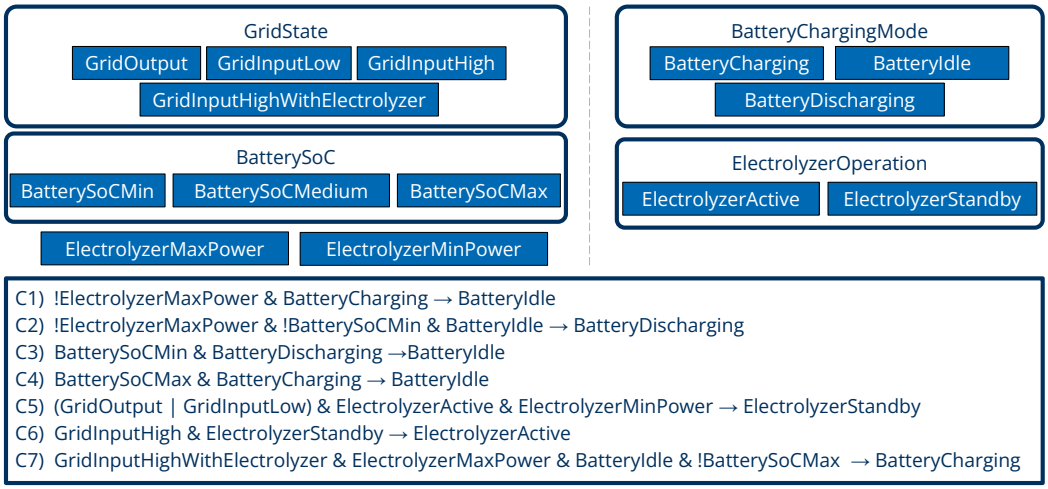


Figure 4. Context model of the energy park simulation. It defines a total of 14 contexts and 4 context groups. The grid state, battery SoC, and electrolyzer power are used for measurable contexts. The battery's charging mode and electrolyzer activeness are implemented as control contexts. The seven constraints that control the system are shown in the lower part of the scheme.

functions is shown in Listing 4, and the rest can be viewed in the code example in the Git repository[1]. The events that call these functions are also shown in the repository. The call on lines 5 and 6 in Listing 4 changes the battery's activation mode. Here, the advantage of using COP is noticeable. Without COP, if-then-else conditions would be needed to check the wanted operation mode. This would result in 7 lines instead of the one presented. In line 8, the restart function is called. The `solve` function automatically chooses the correct next simulation model. It is not needed to specify explicitly to which model to switch.

```
1  function electrolyzerMin!(integ, u, p, ctx)
2    oldContext = ElectrolyzerOperation()
3    if !(isActive(ElectrolyzerMinPower))
4      activateContext(ElectrolyzerMinPower)
5      @context BatteryChargingMode() (
6              batteryMode(integ, u))
7      if oldContext != ElectrolyzerOperation()
8        restart!(integ)
9      end
10   end
11 end
```

Listing 4. Example of a callback function that checks if the context `ElectrolyzerMinPower` must be activated. If the control contexts change after that, variable reinitialization and model switches are performed.

We use solar irradiation data from (Krähenmann et al. 2016b) and wind speed data from (Krähenmann et al. 2016a) as input data. We simulate the system for 1000 h. Figure 5a shows the power of the wind turbines, PV, electrolyzer, battery, and the power exchange with the grid. Positive power values indicate power consumption, while negative values indicate power supply. For the grid, positive values indicate power fed into, and negative values indicate power drawn from the grid.

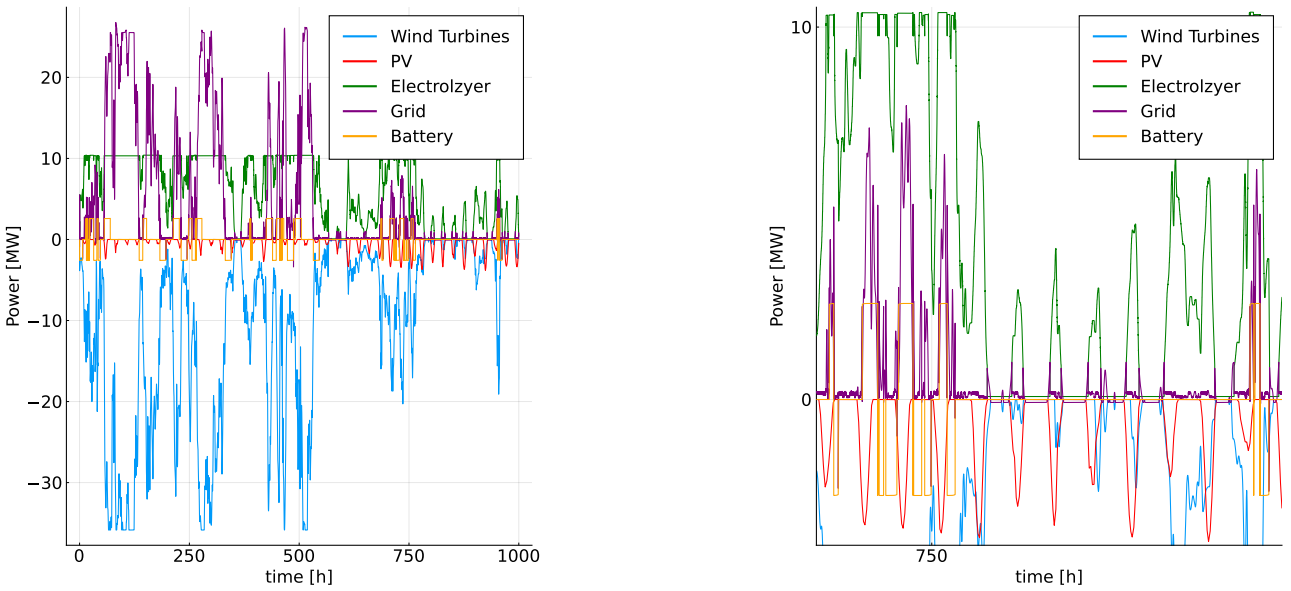
[1] github.com/cgutsche/ContextualEBM.jl

The effects of context control are better visible in Figure 5b. We achieved control of the electrolyzer and battery by activating measurable contexts. The positive values of the battery power curve show that the battery is only charged if the electrolyzer is operated with maximal power. The negative values show that the battery is discharged if the electrolyzer can not be operated with maximum power because the power generated by renewable energy systems is insufficient. The results show that for multiple days, electricity is barely produced by wind power plants. Then, only the PV provides electricity during the day to operate the electrolyzer, while the electrolyzer is set to standby at night. Therefore, by properly defining the context model, we achieved the goals of producing hydrogen only with energy supplied by renewables and increasing the hydrogen output with battery control. In total, the model switched 32 times during the simulation. The electrolyzer was active during 826.3 h from a total 1000 h simulated.

5.2 Performance

One advantage of simulating VSS is to switch between more and less detailed models depending on the needed accuracy under different input conditions. Also, one possible use case for CVSS simulation is to support the operation of CPS where simulation time becomes crucial. However, computing the context control logic, callback functions in events, and the system switches are expected to have a computational overhead. Therefore, we provide performance measurements to evaluate its impact.

We compare 7 different scenarios. First, the CVSS simulation of the energy park as it was presented (*CVSS*). Second, the `restart!` functions are removed from the callbacks, and the system is simulated as an `ODEProblem` (*Active* and *Standby*). Hence, the events are still triggered, and contexts are activated, but the simulation model is not switched. Third, the events are kept, but the callback functions contain no action, i.e., the context activeness is un-



(a) Complete Simulation results.

(b) Detailed view.

Figure 5. Simulation results of the energy park model.

changed (*ActiveEmptyCB* and *StandbyEmptyCB*). Fourth, the events are entirely removed from the simulation models (*ActiveNoCB* and *StandbyNoCB*). The measurements are performed 10 times, and the simulations not based on `CVSSProblem` are performed for the model with the electrolyzer in active and standby mode. The results are shown in Figure 6

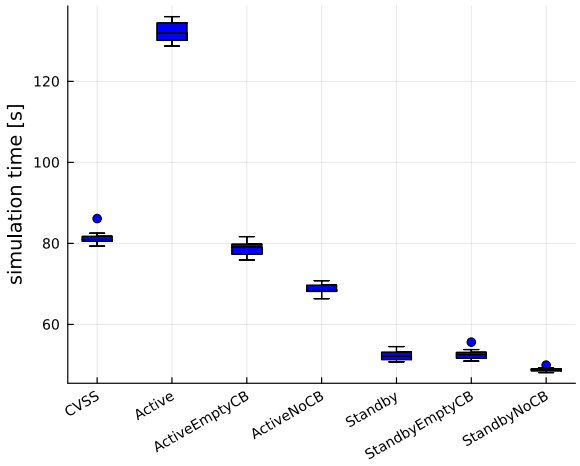


Figure 6. Boxplot that shows the simulation time for seven different simulations. Every model was simulated 10 times.

The comparison of the models that do not contain any events shows that the active electrolyzer model increases the simulation time, which is explainable by the model's increased complexity. Adding empty callbacks also increases simulation time. This effect is stronger for the model with an active electrolyzer, which is explainable by a larger number of events. Adding the context control logic to the callbacks significantly increases the simulation time for the model with an active electrolyzer. Using profiling showed that computing the constraints between contexts is the main reason for this increase.

To estimate the additional computation time of the COEBM simulation, we compare the mean simulation time of the CVSS simulation with a weighted sum of the mean simulation time for the active and standby models with empty callbacks that do not activate contexts. The weights are calculated as $826.3\,\mathrm{h}/1000\,\mathrm{h}$ for the active model and $1 - 826.3\,\mathrm{h}/1000\,\mathrm{h}$ for the standby model, so that they represent the share of the respective models in the total simulation time. The mean CVSS simulation took 81.4 s, and the weighted sum of the active and standby simulation results in 74.2 s. Hence, the increase in simulation time is approximately 10 %.

6 Discussion

The presented approach of COEBM allows simulating VSS that can not be simulated in standard Modelica or MTK without external tools or manually concatenating simulations and their results. COEBM enables specifying control mechanisms based on contextual modeling with concise syntax provided by *Contexts.jl*. Instead of manually (de-)activating components or reinitializing variables based only on if-then-else conditions, constraints between contexts can be used to control the system, and context-adaptive behavior is achieved. Defining constraints within the context model prevents reaching conflicting states. As discussed in (Gutsche et al. 2024), context control in *Contexts.jl* provides the basis for model-checking.

The case study shows that using the context model sim-

DOI
10.3384/ecp218155

ulation control is feasible and advantageous. COP reduces the code complexity. Instead of if-then-else-cascades, only a context-oriented function call is used to check how to operate the battery. The remaining if-then-else conditions are comprehensible as they represent state machines and contain Boolean expressions that depend on one or a few variables. The goals of the variability were achieved.

Contexts discretize continuous variables and, therefore, are activated based on threshold values. Therefore, the choice of threshold values influences the control. In the case study, the threshold values were defined meaningfully. However, changing them can, e.g., change when the electrolyzer gets activated and influences the amount of produced hydrogen. Future research could investigate obtaining optimal values via optimization.

The performance measurements show that computing context control is time-consuming. However, *Contexts.jl* was not optimized for performance. Julia gives developers many tools to improve performance. The context control computation can be optimized. For example, the library uses normal Julia arrays for computing constraints, but static arrays can improve the performance [2].

Context activeness must be initialized before the simulation. If initial activeness does not fit the initial variable values, the control mechanisms could work faulty. MTK allows the calculation of initial variable values, which can be used to initialize the context activeness correctly.

The presented case study has several simplifications. Some simplifications are only relevant for getting technical insights and could be resolved by adding or replacing components in the model. However, other simplifications would also influence the control logic. The model does not include an intermediate state between active electrolyzer and standby, resulting in discrete jumps within the simulation. Also, shutdown is not implemented. Adding models for deactivated and ramp-up states would create additional models and contexts and, therefore, more complexity, but the modeling technique is still applicable.

While we focused on modeling variable behavior, another interesting use case for variability in simulation is the use of surrogate models. If the needed accuracy allows surrogates, contexts can switch between more and less detailed models or data-driven components. Using neural networks within EBM models is supported by MTK.

7 Conclusion and Outlook

We presented how COEBM enables simulating equation-based models containing variability. Therefore, we described a modeling technique based on contexts. Contextual modeling is used to abstract the simulated world and specify control mechanisms. Constraints in the contextual model prevent reaching conflicting states. This is advantageous when modeling self-adaptive systems, e.g., cyber-physical systems, with complex state spaces.

We presented a case study of an energy park for hydrogen production, showing the feasibility of the modeling process in more complex scenarios. We achieve VSS simulation by switching between different simulation models and performing state migration. Performance measurements show that the context control negatively impacts simulation time but can be improved by changing the *Contexts.jl* implementation of computing context control.

We are currently working on implementing additional modeling tools in Julia to simplify the generation of model variants using a single underlying model (Atkinson, Stoll, and Bostan 2010; Seifert 2011). This will improve code reuse and maintainability. Additionally, concise syntax for defining variability not located in the simulation model's top-level would improve modeling.

Another interesting topic for future research is investigating the combination of the presented approach and *OM.jl* to achieve COEBM in Modelica. Additionally, MTK provides an experimental feature to use Functional Mock-up Units (FMUs) within MTK models. This could pave the way for coupling FMUs and Modelica models to COEBM.

Acknowledgements

The authors would like to thank the Boysen–TU Dresden–Research Training Group for the financial support that has made this contribution possible. The Research Training Group is co-financed by the Friedrich and Elisabeth Boysen Foundation and the TU Dresden. The authors would also like to thank the German Research Foundation (DFG) for financially supporting the presented research via SFB/TRR 339 (Project ID 453596084).

Disclaimer

To improve the use of the English language, the authors used the AI-based tools DeepL and Grammarly.

References

Abowd, Gregory D. et al. (1999). "Towards a Better Understanding of Context and Context-Awareness". In: *Handheld and Ubiquitous Computing*. Ed. by Hans-W. Gellersen. Berlin, Heidelberg: Springer. ISBN: 978-3-540-48157-7.

Atkinson, Colin, Dietmar Stoll, and Philipp Bostan (2010). "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 206–219. ISBN: 978-3-642-14819-4.

Benveniste, Albert, Benoit Caillaud, and Mathias Malandain (2020). "The mathematical foundations of physical systems modeling languages". In: *Annual Reviews in Control* 50, pp. 72–118. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2020.08.001. URL: https://www.sciencedirect.com/science/article/pii/S1367578820300547.

Bezanson, Jeff et al. (2015). *Julia: A Fresh Approach to Numerical Computing*. arXiv: 1411.1607 [cs.MS].

[2] github.com/JuliaArrays/StaticArrays.jl

Brkic, Jovan et al. (2019-03). "Open Source PhotoVoltaics Library for Systemic Investigations". In: *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*. Linköping Electronic Conference Proceedings. DOI: 10.3384/ecp1915741.

Broman, David and Jeremy G Siek (2012). "Modelyze: a gradually typed host language for embedding equation-based modeling languages". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-173*.

Cardozo, Nicolas et al. (2012). *Context Petri Nets: Definition and Manipulation*. English. Vrije Universiteit Brussel.

Eberhart, Philip et al. (2015-09). "Open Source Library for the Simulation ofWind Power Plants". In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. Linköping Electronic Conference Proceedings, pp. 929–936. DOI: 10.3384/ecp15118929.

Elmqvist, Hilding, Andrea Neumayr, and Martin Otter (2018). "Modia-dynamic modeling and simulation with julia". In: Juliacon, University College London, UK.

Esperon, Daniel Gomez, Alexandra Mehlhase, and Thomas Karbe (2015). "Appending variable-structure to modelica models (WIP)". In: *Proceedings of the Conference on Summer Computer Simulation*, pp. 1–6.

Fritzson, Peter and Vadim Engelson (1998). "Modelica—A unified object-oriented language for system modeling and simulation". In: *ECOOP'98—Object-Oriented Programming: 12th European Conference Brussels, Belgium, July 20–24, 1998 Proceedings 12*. Springer, pp. 67–90.

Giorgidze, George (2012). "First-class models: On a noncausal language for higher-order and structurally dynamic modelling and simulation". PhD thesis. University of Nottingham.

Gómez Esperón, Daniel (2017). "Strukturvariabilität in der objektorientierten Modellierung und Simulation durch Generierung von Varianten". PhD thesis. Technische Universitaet Berlin (Germany). DOI: 10.14279/depositonce-5805.

Gutsche, Christian et al. (2024). "Context-Oriented Programming and Modeling in Julia with Context Petri Nets". In: *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 1–9. DOI: 10.1109/SEAA64295.2024.00011.

Hirschfeld, Robert, Pascal Costanza, and Oscar Nierstrasz (2008). "Context-oriented Programming". In: *Journal of Object Technology*.

Keays, Roger and Andry Rakotonirainy (2003). "Context-oriented programming". In: *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*. MobiDe '03. San Diego, CA, USA: Association for Computing Machinery, pp. 9–16. ISBN: 1581137672. DOI: 10.1145/940923.940926. URL: https://doi.org/10.1145/940923.940926.

Krähenmann, S. et al. (2016a). *Stündliche Raster der Globalstrahlung für Deutschland (Projekt TRY-Weiterentwicklung)*. DWD Climate Data Center (CDC). DOI: DOI : 10 . 5676 / DWD_CDC/TRY_Basis_v001.

Krähenmann, S. et al. (2016b). *Stündliche Raster der Windgeschwindigkeit für Deutschland (Projekt TRY-Weiterentwicklung)*. DWD Climate Data Center (CDC). DOI: DOI:10.5676/DWD_CDC/TRY_Basis_v001.

Lee, Jay, Behrad Bagheri, and Hung-An Kao (2015). "A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems". In: *Manufacturing Letters* 3, pp. 18–23. ISSN: 2213-8463. DOI: https://doi.org/10.1016/j.mfglet.2014.12.001.

Lubin, Miles et al. (2023). "JuMP 1.0: Recent improvements to a modeling language for mathematical optimization". In: *Mathematical Programming Computation*. DOI: 10 . 1007 / s12532-023-00239-3.

Ma, Yingbo et al. (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. arXiv: 2103.05244 [cs.MS].

Mehlhase, Alexandra (2014). "A Python framework to create and simulate models with variable structure in common simulation environments". In: *Mathematical and Computer Modelling of Dynamical Systems* 20.6, pp. 566–583. DOI: 10 . 1080/13873954.2013.861854.

Migoni, G. et al. (2016). "Efficient simulation of Hybrid Renewable Energy Systems". In: *International Journal of Hydrogen Energy* 41.32, pp. 13934–13949. ISSN: 0360-3199. DOI: https://doi.org/10.1016/j.ijhydene.2016.06.019.

Muschevici, Radu, Dave Clarke, and José Proença (2010-01). "Feature Petri Nets." In: *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)* 2, pp. 99–106.

Neumayr, Andrea and Martin Otter (2023). "Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom". In: *Electronics* 12.3. ISSN: 2079-9292. DOI: 10.3390/electronics12030500. URL: https://www.mdpi.com/2079-9292/12/3/500.

Rackauckas, Christopher (2021). *Modelingtoolkit, modelica, and modia: The composable modeling future in julia*. The Winnower. DOI: 10.15200/winn.162133.39054.

Rackauckas, Christopher and Qing Nie (2017). "DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia". In: *Journal of Open Research Software* 5.1.

Rackauckas, Christopher and Qing Nie (2019). "Confederated modular differential equation APIs for accelerated algorithm development and benchmarking". In: *Advances in Engineering Software* 132, pp. 1–6. ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2019.03.009. URL: https://www.sciencedirect.com/science/article/pii/S0965997818310251.

Seifert, Mirko (2011). *Designing Round-Trip Systems by Change Propagation and Model Partitioning*. Technische Universität Dresden. URL: https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-71098.

Stüber, Moritz (2017). "Simulating a Variable-structure Model of an Electric Vehicle for Battery Life Estimation Using Modelica/Dymola and Python." In: *Proceedings of the 12th international Modelica conference*, pp. 132–031.

The Julia Project (2023). *Julia 1.10 Documentation*. https://docs.julialang.org/en/v1/. Accessed: 2024-01-25.

Tinnerholm, John, Adrian Pop, and Martin Sjölund (2022). "A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability". In: *Electronics* 11.11. ISSN: 2079-9292. DOI: 10.3390/electronics11111772.

Utkin, Vadim (1977). "Variable structure systems with sliding modes". In: *IEEE Transactions on Automatic control* 22.2, pp. 212–222.

Wang, Zizhe et al. (2025-01). "Proposal for A Context-oriented Modelica Contributing to Variable Structure Systems". In: *Modelica Conferences*, pp. 53–62. DOI: 10.3384/ecp20753.

Zimmer, Dirk (2010). *Equation-based modeling of variable-structure systems*. Swiss Federal Institute of Technology, Zürich. URL: https://people.inf.ethz.ch/fcellier/PhD/zimmer_phd.pdf.