Context-Oriented Modelica for Advanced Variability Management

Zizhe Wang[1,2] Christian Gutsche[1,2] Uwe Aßmann[2]

[1]Boysen-TU Dresden-Research Training Group, Dresden, Germany
[2]Software Technology Group, Technische Universität Dresden, Germany
zizhe.wang, christian.gutsche, uwe.assmann@tu-dresden.de

Abstract

Context-aware systems are crucial in modern cyber-physical systems (CPS), enabling dynamic adaptation to changing conditions. Many such systems involve complex variability. Modelica, a leading equation-based language for modeling and simulating physical systems, struggles to manage this complexity. As variability management becomes more complex, traditional Modelica constructs such as conditional statements, state machines, and state graphs become increasingly inadequate and difficult to maintain. This work introduces a **context-oriented modeling paradigm for Modelica** by integrating Petri Net-based context control with FMI-based Variable Structure Systems (VSS). Specifically, we present Context Petri Nets (CoPN) as a formal mechanism for representing and managing contextual dependencies. By combining CoPN with VSS, our approach enables advanced variability management, supporting the modeling and simulation of complex, context-aware CPS.

Keywords: Modelica, variable structure systems, FMI, Context Petri Nets, context-aware systems, variability management, context-oriented modeling and simulation

1 Introduction

Context-aware systems exist everywhere in daily life. According to Dey and Abowd 2000, the context refers to conditions that influence a system's behavior or state. A typical example is the smartphone, where different operation modes (e.g., `do not disturb`, `power saving`, `gaming`) dynamically adjust software and hardware configurations based on active contexts (e.g., battery level). More importantly, these modes often have contextual relationships. For example, when the battery level drops below 20%, `power saving` mode activates, reducing the screen refresh rate and limiting CPU/GPU performance. If `gaming` mode attempts to activate at that moment, it fails, which demonstrates an **exclusion** relationship between the two modes. Another example is the iPhone's crash detection feature. When a car crash occurs, the iPhone detects the event and activates the `emergency` context, automatically dialing emergency services. In this case, `crash` and `emergency` have an **inclusion** relationship, as one triggers the other. More specifically, this represents a **weak inclusion** relationship, where `crash` triggers `emergency`, but not vice versa.

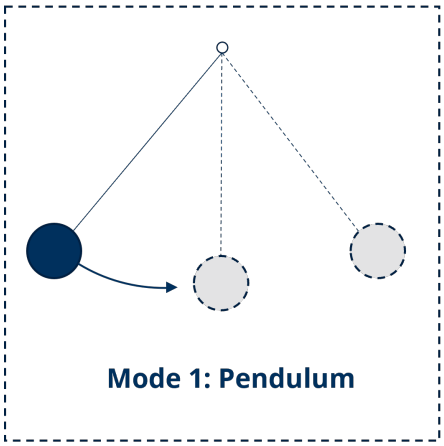
Modelica, a state-of-the-art equation-based modeling language, along with its various tools and libraries, is widely used in industry for modeling and simulating multi-domain and multi-physical systems. For simple context expression and variability management, conditional statements (e.g., `if-else` and `when` clauses) are sufficient. However, for more advanced context modeling and management, especially when dealing with complex contextual relationships, traditional conditional statements become inadequate. They not only lack expressiveness but also lead to unreadable and hard-to-maintain code. Addressing advanced contextual relationships in Modelica remains an open research challenge. This work addresses this challenge by introducing **context-oriented Modelica**, an approach inspired by context-oriented programming (Hirschfeld, Costanza, and Nierstrasz 2008), tailored to fit the Modelica modeling paradigm.

The structure of this work is as follows: Section 2 introduces the foundational concepts. Section 3 explores a novel and efficient solution for enabling VSS through FMI. Section 4 focuses on Context Petri Nets (CoPN) and their integration into Modelica for advanced variability management. Section 5 brings together CoPN and VSS into a more generalized approach. A proof of concept demonstrates how this enables the development of context-aware systems. Finally, Section 6 concludes the work and highlights future directions.

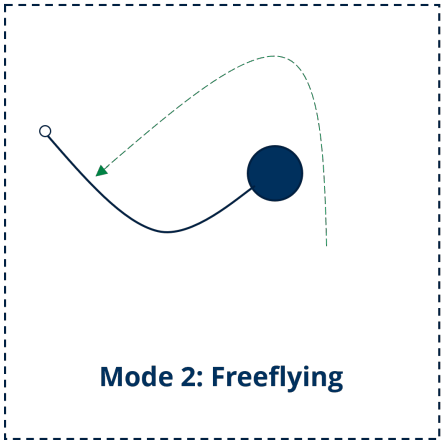
2 Background

2.1 Variable Structure Systems in Modelica

One significant challenge for realizing advanced context relationships is Variable Structure Systems (VSS), which was first introduced by Utkin 1977. VSS consist of various subsystems with proper switching logic for switching between these subsystems. In the context of Modelica, VSS mean to switch between different submodels and equation systems based on conditions, as Figure 1 shows. The challenge is that, because of Modelica's static nature (everything must be defined before simulation), the switching of submodels and equation systems is limited. Wang et al. 2024 details the conditions under which VSS works and where it fails. In a summary, when equation systems have different variables and/or differential indexes, the structural transition will fail.




```
model Pendulum
  Real phi(start=0); // Pendulum angle
  Real dphi(start=-2); // Angular velocity
  parameter Real m = 1; // Bob mass
  parameter Real g = 9.81; // Gravity
  parameter Real L = 2; // Pendulum length
  Real x, y, dx, dy, F;
equation
  x = L * sin(phi);
  y = -L * cos(phi);
  dx = der(x); dy = der(y);
  dphi = der(phi);
  der(dphi) = -g/L * sin(phi);
  F = m * (g * cos(phi) + L * dphi^2);
end Pendulum;
```




```
model Freeflying
  Real x;
  Real y;
  Real vx;
  Real vy;
  parameter Real m = 1;
  parameter Real g = 9.81;
  parameter Real L = 2;
equation
  vx = der(x);
  vy = der(y);
  m * der(vx) = 0;
  m * der(vy) = -m * g;
  r = sqrt(x^2 + y^2);
end Freeflying;
```

Figure 1. Two modes with their corresponding equation systems. Modelica cannot handle the transition, thus, at the moment of switching from mode 1 to mode 2, the simulation fails.

2.2 Variability Management in Modelica

Modelica supports several constructs for managing variability, the most basic being conditional statements like **`if-else`** and **`when`**. While suitable for simple systems, these constructs quickly become unwieldy and error-prone in complex systems, making the logic hard to maintain and debug. Modelica also offers **state machines** (as defined in the *Modelica Language Specification*) and **state graphs** (`Modelica.StateGraph`, as defined in the *Modelica Standard Library*). These introduce hierarchy and parallelism, improving clarity compared to raw conditionals. However, their reliance on directed transitions limits flexibility, since state activation typically follows predefined, sequential paths, making it difficult to handle highly dynamic, condition-driven context changes. For more advanced variability management in complex systems, a more flexible and expressive mechanism is needed. Petri Nets offer a powerful alternative. Their token-based, condition-driven structure allows non-directional context activation, supporting complex, concurrent, and non-sequential behaviors without rigid state progression. This enables precise control over context interdependencies while maintaining system reliability through strong formal analysis, which is essential for robust orchestration of complex, dynamic behaviors.

2.3 Petri Nets

Petri 1962 introduced Petri Nets as a formal modeling approach in his PhD dissertation *Kommunikation mit Automaten* (English: Communication with Automata). A Petri Net consists of four fundamental components: **places** (P), which denote system states and are typically represented as circles; **transitions** (T), which represent events or actions that drive state changes and are depicted as rectangles or bars; **tokens**, shown as dots inside places, which indicate the current system state and move between places; and **arcs**, drawn as arrows, which connect places and transitions, directing token flow. The token distribution across all places is called a marking and represents the current state of the Petri Net. The system's behavior is determined by firing rules, where a transition is enabled if all its input places contain the necessary number of tokens. Upon firing, the transition consumes tokens from its input places and generates tokens in its output places, updating the system state accordingly. Their later-introduced graphical nature also makes them intuitive for visualizing complex dynamic behaviors in distributed or reactive systems. Figure 2 shows how a Petri Net works. This Petri Net consists of three places, P_1, P_2, and P_3, and one transition T. The diagram shows the system both **before** and **after** firing.

DOI
10.3384/ecp218165

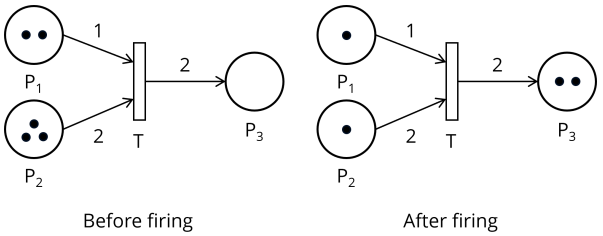


Figure 2. Example of a Petri Net

2.4 Functional Mock-Up Interface

Functional Mock-up Interface (FMI) (Blochwitz et al. 2011) is an open standard for the exchange and co-simulation of dynamic models across different modeling languages and platforms. It defines a container and a set of APIs that allow models, packaged as Functional Mock-up Units (FMUs), to be shared and executed consistently in various simulation environments. FMI is widely adopted in cyber-physical systems engineering for enabling interoperability. FMI 1.0, 2.0, and 3.0 were released in 2010, 2014, and 2022, respectively. FMI supports three FMU types: Model Exchange, Co-Simulation, and the newly introduced Scheduled Execution in FMI 3.0.

3 Advancing VSS with FMI

3.1 State of the Art

Many approaches have been developed to realize VSS in equation-based modeling environments. Based on timeline and design concepts, these can be broadly categorized into four groups. The **first group** includes early works based on new languages/frameworks, such as Mosilab (Nytsch-Geusen et al. 2005), Sol (Zimmer 2010), Hydra (Giorgidze 2012), and Modelyze (Broman and Siek 2012). These support true structural changes and enable VSS modeling but are incompatible with the Modelica ecosystem and cannot reuse existing Modelica libraries. The **second group** focuses on integrating VSS support directly into Dymola, as proposed by Elmqvist, Mattsson, and Otter 2014 and Mattsson, Otter, and Elmquvist 2015. These extensions were only validated with simple examples and have not been included in stable Dymola releases due to limited robustness and generality. The **third group** involves frameworks that orchestrate VSS simulations externally, such as DySMo (Mehlhase 2015), MoVaSim (Gómez Esperón 2017), and PyVSM (Stüber 2017). These do not support true structural changes, and thus only simulate VSS. However, since the models remain in Modelica, modelers benefit from the Modelica ecosystem. The **fourth group** features recent Julia-based approaches, such as Modia (Elmqvist, Neumayr, and Otter 2018), OM.jl (Tinnerholm et al. 2020), and ModelingToolkit (Ma et al. 2021). Julia's dynamic type system and multiple dispatch enhance VSS support. However, like the first group, they require porting Modelica libraries and generally lack the graphical interfaces.

The goal is to remain within the Modelica ecosystem. In DySMo, conditions must be defined within each model. If conditions are managed externally, DySMo can only evaluate them after each time step. For example, if the time step is 1 second and a condition is met at 0.5 seconds, it will still be evaluated at 1 second, not immediately at 0.5 seconds when the condition occurs. This limitation makes DySMo unsuitable for our work. MoVaSim, while more flexible, is implemented on the JModelica compiler. Since JModelica has not been maintained since late 2019, MoVaSim is effectively obsolete unless it is ported to another Modelica compiler. PyVSM lacks sufficient public documentation, with only a single mention available, making detailed evaluation infeasible. To overcome these limitations, we propose an approach for realizing VSS using FMI. This approach resolves DySMo's conceptual issues, improves performance, and leverages FMI's standardized APIs for simulation and management of VSS.

3.2 Concept

The concept, inspired by DySMo, coordinates different simulations by detecting transition points, recording necessary variable values, and using them to initialize the subsequent modes. Simulation results from different submodels are then combined. Using FMI streamlines this process, with Python libraries like FMPy (Sommer 2017) and PyFMI (Andersson, Åkesson, and Führer 2016) providing efficient APIs that avoid costly data serialization and external tool calls, significantly speeding up execution.

This concept was implemented with examples such as the *Pendulum-Freeflying* example mentioned earlier in Figure 1. The simulation result is shown in Figure 3. The first mode (*Pendulum*) will stop when the string loses force on the bob. The system detects this, records the stop time and variable values, and then starts the second mode (*Freeflying*) at the stop time of the first mode, initializing the model with the recorded values. The second mode stops when the string holds the bob again. Other examples can be found at GitHub | FMUVSS.

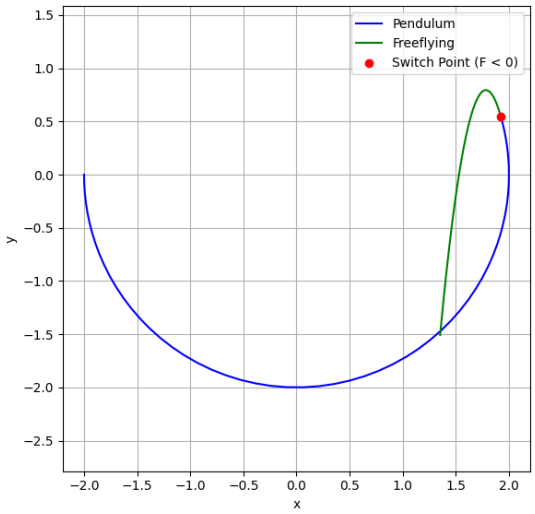


Figure 3. Trajectory of the *Pendulum-Freeflying* example

3.3 The Framework FMUVSS

Based on this concept, the FMUVSS framework was developed to automate the process and provide a user-friendly interface, eliminating the need for manual prototyping. It streamlines the simulation of VSS and supports both FMI 2.0 and 3.0 by leveraging their low-level APIs for finer control over the simulation process (e.g., retrieving specific variable values during simulation). Stop conditions can be defined directly without the need to modify the original models, ensuring a clean separation of concerns: model design remains intact while variability management and switching logic are handled externally. Compared to FMI 2.0, FMI 3.0 offers significant advantages by introducing **event mode** and **early return** in Co-Simulation FMUs, enabling the master algorithm to respond to zero-crossings or mode switches precisely when they occur. In contrast, FMI 2.0 only evaluates events after each simulation step, preventing events from being triggered at their exact time of occurrence. In critical systems, such timing inaccuracies can lead to unstable or incorrect simulation outcomes.

Listing 1. Example of mode configuration in FMUVSS

```
'SlidingBall': {
    'fmu_path': './Ball.fmu',
    'monitored_vars': ['y'],
    'outputs': ['x', 'y', 'vx', 'vy'],
    'stop_condition': lambda vars:
        vars['y'] < 10,
    'transition_mapping': {
        'FlyingBall': {
            'x' : 'x', 'y' : 'h',
            'vx' : 'vx', 'vy' : 'vy'}
            },
    'next_mode': 'FlyingBall'},

'FlyingBall': {
    'fmu_path': './FlyingBall.fmu',
    'monitored_vars': ['h', 'r'],
    'outputs': ['x', 'h', 'vx', 'vy'],
    'stop_condition': lambda vars:
        vars['h'] < vars['r'],
    'transition_mapping': {
        'BouncingBall': {
            'x' : 'x', 'h' : 'h',
            'vx' : 'vx', 'vy' : 'v'}
            },
    'next_mode': 'BouncingBall'},

'BouncingBall': {
    'fmu_path': './ContactBall.fmu',
    'monitored_vars': ['mass.s', 'r'],
    'outputs': ['x', 'h', 'vx', 'v'],
    'stop_condition': lambda vars:
        vars['mass.s'] > vars['r'],
    'transition_mapping': {
        'FlyingBall': {
            'x' : 'x', 'h' : 'h',
            'vx' : 'vx', 'v' : 'vy'}
            },
    'next_mode': 'FlyingBall'}
```

To illustrate the framework in practice, Listing 1 shows part of the configuration interface used to define modes for a *SlidingBall–FlyingBall–BouncingBall* system: The ball begins by sliding down an inclined surface in the `SlidingBall` mode. After reaching a certain height, it leaves the surface and transitions into the `FlyingBall` mode. When it hits the ground, it switches to the `BouncingBall` mode, where damping reduces its bounce height. The system continues alternating between flying and bouncing until the simulation ends.

Compared to DySMo, FMUVSS eliminates the need for modifications to the Modelica models. In DySMo, stop conditions and a `transitionID` must be inserted into the original models to trigger the transition logic. The `transitionID` is initialized to 0, and when the transition condition is met, it changes to 1, signaling the framework to stop the current simulation and switch to the next mode. In contrast, FMUVSS allows developers to export unmodified Modelica models as FMUs. Transition conditions are defined externally and managed centrally, enabling advanced variability handling such as nested conditions and inter-condition relationships.

Experiments demonstrate that FMUVSS runs significantly faster than DySMo, as shown in Table 1. This comparison was conducted using the same models, the DASSL solver proposed by Petzold 1982, a tolerance of `1e-4`, and identical hardware for three VSS models: the Pendulum and SlidingBall models, as well as a clutch model from Benveniste et al. 2022, where two clutches engage and disengage. Compilation time for compiling each model with DySMo was excluded.

Table 1. Performance comparison: FMUVSS vs DySMo

VSS Model	Framework	Time (s)
Pendulum	FMUVSS	0.53
	DySMo	18.69
SlidingBall	FMUVSS	3.34
	DySMo	81.08
Clutch	FMUVSS	2.43
	DySMo	339.4

Profiling with Python tools, such as `cProfile`, reveals that DySMo's slow performance is due to its heavy reliance on Python interpretation, text-based file parsing, and high-overhead interprocess communication (IPC) for executing Modelica simulations. Additionally, DySMo uses dynamic list growth, leading to memory allocation inefficiencies. In contrast, FMUVSS achieves superior performance by using direct binary access through the FMI standard, which avoids the overhead of text parsing and IPC. By leveraging preallocated arrays and zero-copy buffers, FMUVSS minimizes both Python overhead and memory allocation delays, resulting in faster and more efficient execution. More details are available at: GitHub | FMUVSS | Performance Evaluation.

DOI
10.3384/ecp218165

4 Context-Oriented Petri Nets

4.1 Petri Nets for Context-Aware Systems

Gutsche, Wang, et al. 2023 presents using Petri Nets as a control mechanism for state transitions in Modelica. However, if the contexts become more complicated, e.g., if some of the contexts are dependent, such as activating one context (de)activates the other, or activating one requires the other to be active. Using original Petri Nets is not easy for describing these dependencies. Thus a more powerful form of Petri Nets is needed. Petri Nets have many extensions and they have also been extended in different forms to model context-aware systems. Gutsche, Prokopets, et al. 2024 summarized four main extensions in this field: Contextual Petri Nets from Baldan, Corradini, and Montanari 2001; Feature Petri Nets from Muschevici, Clarke, and Proenca 2010; Context Petri Nets from Cardozo et al. 2012; Adaptive Petri Nets from Mai et al. 2018. Context Petri Nets are explicitly designed for context-aware systems, where different contexts (de)activate parts of the model. For this reason, CoPN were chosen for this work.

4.2 Context Petri Nets in Modelica

In CoPN, contexts are represented as places, each guarded by its activation and deactivation conditions. When a condition is met, the corresponding transition fires, adding/removing a token to/from the place. A token count of **one** indicates an active context; zero means inactive. This token-based representation makes dependencies clear and supports modular control logic and formal verification.

If no VSS is involved, CoPN can be directly applied to existing Modelica models without modifying the original structure. The separation of concerns ensures that the control layer and system layer remain separate, allowing for better maintenance. To achieve it, we can utilize the Petri Nets library *PNlib* (Proß and Bachmann 2012)[1]. Through careful design with the use of the inhibitor arc (which inhibits a transition from firing if the connected place has tokens), CoPN and their four types of context relationships can be effectively represented in Modelica. As shown in Figure 4, different dependencies can also be composed together. However, due to the nature of PNlib, the number of input and output ports for each place and transition must be explicitly defined, and connections between them must be unique: each port can be connected only once, and all ports must be used. Otherwise, the model becomes unbalanced. As the number of context places and their relationships increases, modeling and debugging CoPN becomes increasingly complex. Even with only a few contexts and relationships, debugging can be highly challenging, as demonstrated in Figure 4.

The original CoPN work proposed four relationships between contexts: **weak/strong inclusion**, **exclusion**, and **requirement**. These are explained in detail in Figure 5.

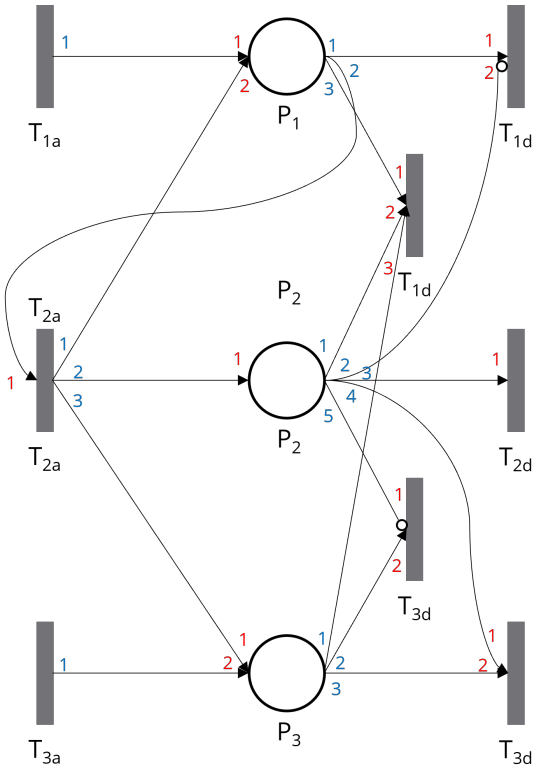


Figure 4. A CoPN containing both requirement and strong inclusion relations. P_2 requires P_1, meaning P_2 can only be active if P_1 is active. P_2 and P_3 are connected by a strong inclusion relation, so they are always activated and deactivated together.

4.3 Composition with CoPNCompositor

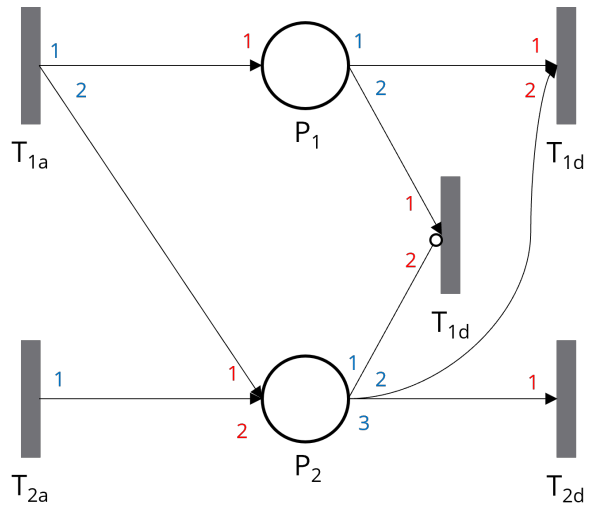
To address this growing complexity with context relationships, the Python-based tool `CoPNCompositor`[2] was developed. It utilizes the Python template engine `Jinja`, where basic components such as places, transitions, and arcs (normal or inhibitor arcs) are defined as templates. Based on specific goals, components are automatically created and connected. Developers can then define contexts, their (de)activation conditions (time-based or event-based), and any context relationships in natural language, as shown in Listing 2. The tool automatically composes the CoPN and generates the corresponding Modelica code.

Listing 2. Configuration interface of CoPNCompositor

```
contexts = ["A", "B", "C", "D", "E"]
weak_inclusions = [("A", "B")]
strong_inclusions = [("B", "C")]
exclusions = [["A", "E"]]
requirements = [("D", "E")]
event_definitions = {
    "A":{"activation": "",
         "deactivation": ""}
    "B":{"activation": "",
         "deactivation": ""}
    ...}
```

[1] https://github.com/AMIT-HSBI/PNlib

[2] https://github.com/wangzizhe/CoPN



Weak Inclusion

P: Discrete Place

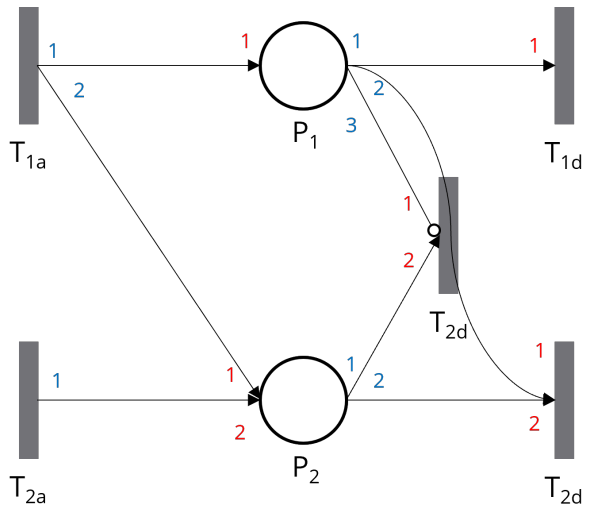
T_a: Activation Transition

T_d: Deactivation Transition

Normal / Inhibitor Arc

1, 2... / 1, 2... Input / Output Ports

(a) Contexts P_1 and P_2 are weakly included (from P_1 to P_2), meaning the (de)activation of P_1 also (de)activates P_2.



Strong Inclusion

P: Discrete Place

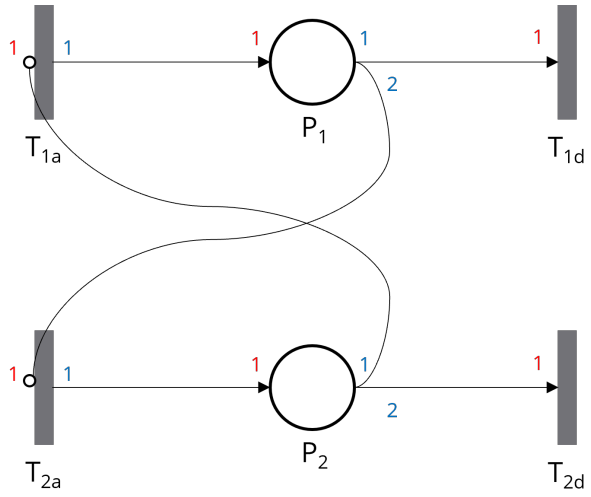
T_a: Activation Transition

T_d: Deactivation Transition

Normal / Inhibitor Arc

1, 2... / 1, 2... Input / Output Ports

(b) Contexts P_1 and P_2 are strongly included (bidirectionally), meaning the (de)activation of one also (de)activates the other.



Exclusion

P: Discrete Place

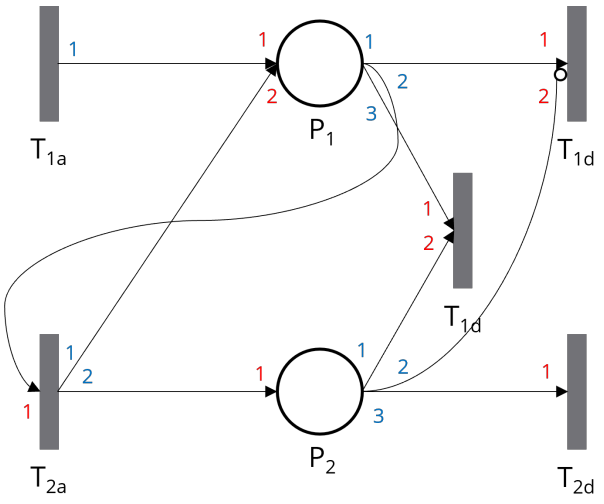
T_a: Activation Transition

T_d: Deactivation Transition

Normal / Inhibitor Arc

1, 2... / 1, 2... Input / Output Ports

(c) Contexts P_1 and P_2 are mutually exclusive, meaning one can only be activated if the other is not active.



Requirement

P: Discrete Place

T_a: Activation Transition

T_d: Deactivation Transition

Normal / Inhibitor Arc

1, 2... / 1, 2... Input / Output Ports

(d) Contexts P_1 and P_2, where P_1 is a prerequisite for P_2; activating P_2 requires P_1 to be active, and deactivating P_1 also deactivates P_2.

Figure 5. Four context relationships of CoPN modeled with PNlib in Modelica

DOI 10.3384/ecp218165

5 Context-Oriented Modelica

In Section 3, we introduced how to realize VSS with FMI. In Section 4, we discussed how CoPN can be applied in Modelica. By combining these two approaches, we can achieve advanced variability management in a more generalized manner and realize context-oriented Modelica.

5.1 ContextModelica

In systems without structural change, CoPN can be directly integrated into the model, as described in the previous section. For systems involving structural change, CoPN act as the central context manager, while FMUVSS handles VSS. As shown in Figure 6, each mode or context (i.e., submodel) is represented as a place in the CoPN. The token count of each place indicates whether the mode/context is active (1) or inactive (0). Each place is guarded by activation and deactivation transitions, and may include context dependencies spanning the entire net. These (de)activation conditions are typically linked to variable values during simulation. CoPN observe and evaluate these values at runtime. When all conditions and dependencies are satisfied, the corresponding transition fires, thereby (de)activating the relevant mode/context.

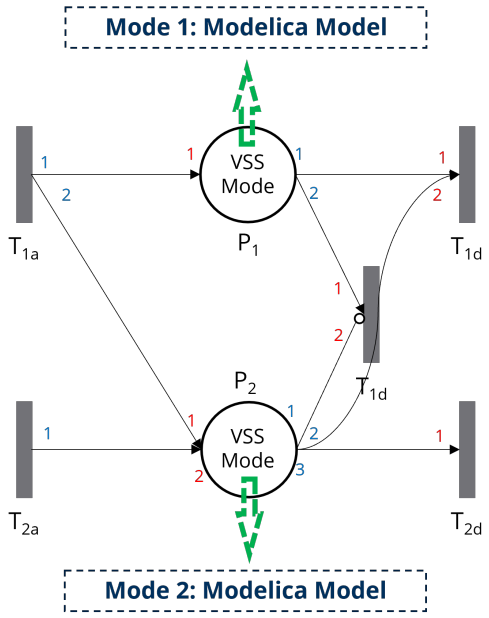


Figure 6. The corresponding Modelica submodels are represented as places in the CoPN. Activating a place triggers FMUVSS to coordinate the simulation of the associated submodel. As a result, the overall model dynamically activates specific submodels based on the currently active contexts. The final output reflects the complete simulation result of the entire system.

Building CoPN outside of Modelica is more flexible, as it avoids the need to manually define the number of ports and their exact connections. Any Python-based Petri Net library that supports inhibitor arcs can be utilized for building the CoPN. In this work, the SNAKES library (Pommereau 2015) has been used to construct the CoPN, as it offers a compact syntax and support for inhibitor arcs.

The framework `ContextModelica` was developed based on this concept to unify the functionality and configuration interfaces of the CoPN compositor and FMUVSS described earlier. Through this framework, developers can define configurations such as various VSS modes, context (de)activation conditions, and their interdependencies. In essence, ContextModelica enables context-oriented modeling in Modelica, allowing for flexible and modular simulation of systems with structural variability.

5.2 Proof of Concept

A simplified IT system model has been applied. To focus on demonstrating the underlying mechanism, the complexity of each submodel has been intentionally reduced. The goal is to validate the proposed concept through a minimal working example.

The system includes two energy supply modes: `greenSupply`, powered solely by green hydrogen, and `hybridSupply`, which combines hydrogen and grid electricity. "H_2ProducedPower" and "loadDemand" are input variables that change over time. Depending on load demand and energy availability, the system switches between three operation modes: `energySaving`, `normal`, and `highPerformance`. Each mode adjusts CPU core count and frequency to reflect the desired performance. (De)activation conditions are based on variable thresholds during simulation, as summarized in Table 2.

Table 2. (De)activation conditions of different modes

Mode	(De)activation Condition
greenSupply	H_2ProducedPower $\geq$ loadDemand
hybridSupply	H_2ProducedPower $<$ loadDemand
energySaving	loadDemand $<$ 150 kW
normal	150 kW $\leq$ loadDemand $<$ 200 kW
highPerformance	loadDemand $\geq$ 200 kW

In addition to conditions, the system enforces **contextual relationships** between modes, as illustrated in Figure 7. Both energy supply modes and operation modes are *mutually exclusive*, ensuring that only one mode from each category is active at any given time. Moreover, some modes depend on others: for instance, `energySaving` requires `greenSupply` to be active, while `highPerformance` depends on `hybridSupply`. Therefore, even if the activation condition is satisfied, these modes will not activate unless their required contexts are also active.

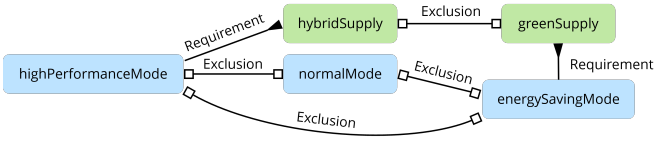


Figure 7. Context relationships of the IT system model

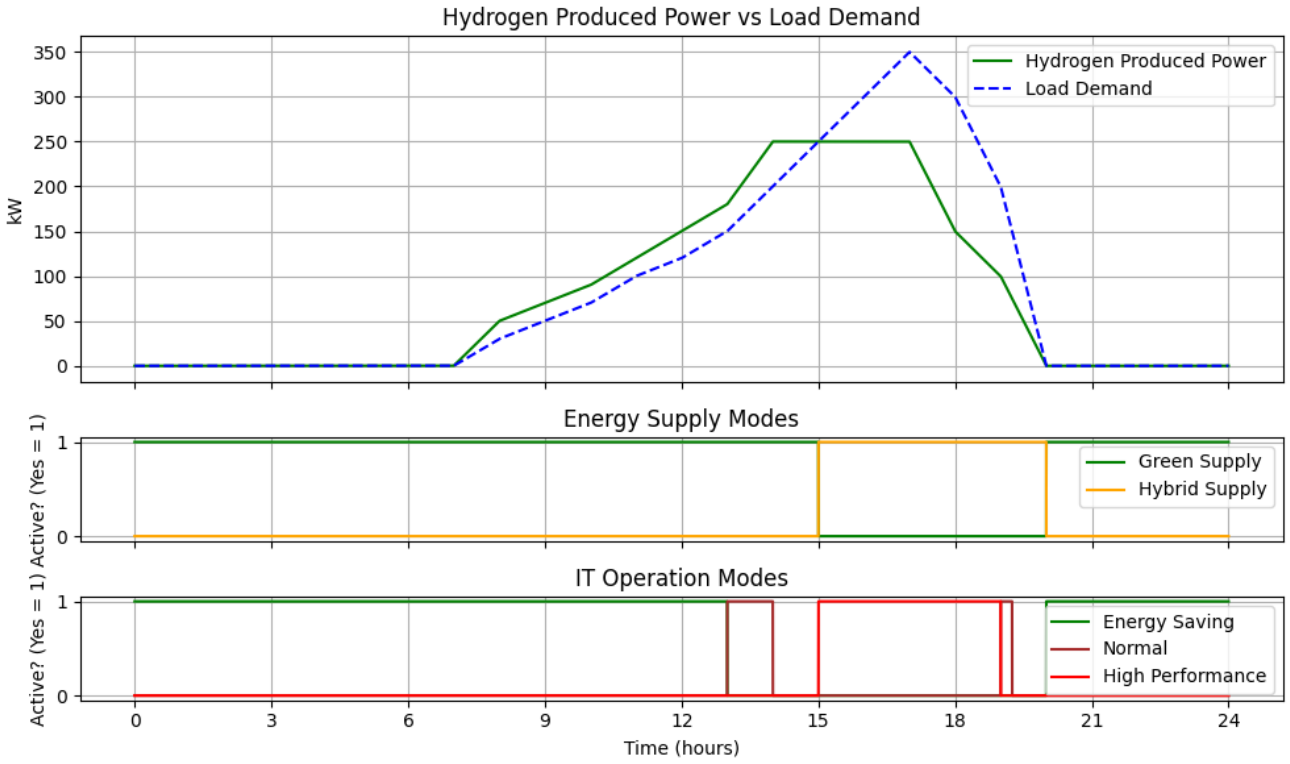


Figure 8. Simulation result of the IT system example

Each mode maps to a specific Modelica submodel. During simulation, the CoPN monitors relevant variables and manages transitions based on defined relationships and guard conditions. The IT system model was simulated over a 24-hour period. Figure 8 illustrates the results. At hour 13, `loadDemand` reached 200 kW, satisfying the activation condition for `highPerformance`. However, since it depends on `hybridSupply`, it only became active at hour 15 when `hybridSupply` was also activated. Likewise, `energySaving` met its condition at hour 19.25 but could not activate until `greenSupply` was active. The delayed activation of `energySaving` and `highPerformance` highlights the importance of context-aware coordination, showcasing the net's ability to synchronize mode interactions safely.

5.3 Managing Submodels

Each mode in a VSS system corresponds to a distinct submodel. Therefore, effectively managing these submodels is essential. The most conventional approach is to export a separate FMU for each VSS mode. While straightforward, this method can become inefficient in systems with tightly coupled components or a large number of modes. In such cases, developers often have to duplicate parameters, variables, and equations across multiple models, which increases maintenance overhead. To address these challenges, we turn to software composition techniques for developing more scalable and maintainable methodologies for managing submodels.

One approach is **model composition**, where individual FMU components are combined dynamically during the VSS simulation based on the active context. FMI supports **hierarchical FMUs**, which allows communication between FMU components via defined input and output ports. However, a major limitation of this approach is the requirement to modify each submodel to expose appropriate ports, which increases maintenance overhead.

Another approach is to encapsulate all submodels within a single unified model and activate only relevant parts based on the active context. This is known as the **Single Underlying Model (SUM)** methodology, as illustrated in Figure 9.

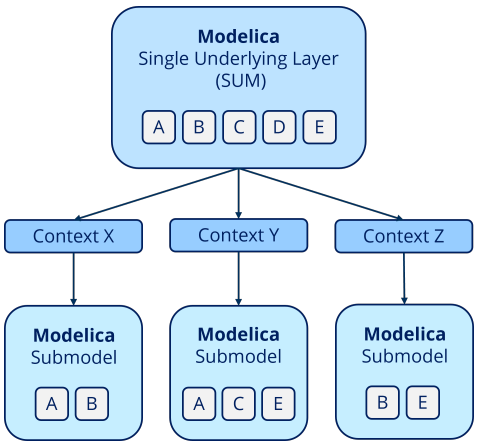


Figure 9. Modelica SUM to submodels based on contexts

DOI
10.3384/ecp218165

In this case, a single model or FMU contains all potential subcomponents, and the active ones are selected at runtime according to the simulation context. The new **Scheduled Execution** FMU type from Gomes et al. 2021 introduced in FMI 3.0 aligns well with this concept. It allows different partitions (representing different modes) within a single FMU to be activated or deactivated based on events. However, as of August 2025, no existing Modelica tools support exporting FMUs using the Scheduled Execution type.

To bridge this gap, the module **ModeGen** was developed as an intermediate solution. ModeGen embraces the core idea of SUM by encapsulating the full system structure in a single Modelica model. It then dynamically generates mode-specific submodels by selectively including or excluding components based on the active context, and exports them as corresponding FMUs. To remain consistent with standard Modelica syntax, a lightweight labeling system/syntax has been introduced to annotate the elements of the Modelica SUM. ModeGen includes a **parser** that reads these labels, interprets them in relation to the active mode or context, and automatically composes the corresponding submodels. A minimal example of a Modelica SUM based on the *Pendulum–Freeflying* scenario is shown in Listing 3. After generation, ModeGen invokes a Modelica compiler to perform model checking for all generated submodels and return the visualized results.

Listing 3. SUM of the *Pendulum-Freeflying* example

```
/*#
  MODEL-METADATA:
    Modes: [pendulum, freeflying]
    Shared: [m, g, y]
#*/

model PendulumFreeflyingSUM
  //# [all]
  parameter Real m = 1;
  parameter Real g = 9.81;
  Real y;

  //# [pendulum]
  parameter Real L = 2;
  Real phi(start=0), dphi(start=2);
  Real x(start=-2), dx, dy, F;

  //# [freeflying]
  Real x, vx, vy;

@#equation
  //# [pendulum]
  x = L * sin(phi); y = -L * cos(phi);
  dx = der(x); dy = der(y);
  dphi = der(phi);
  der(dphi) = -g/L * sin(phi);
  F = m * (g * cos(phi) + L * dphi^2);

  //# [freeflying]
  vx = der(x); vy = der(y);
  m * der(vx) = 0; m * der(vy) = -m * g
```

6 Conclusion and Outlook

6.1 Conclusion

Managing variability in Modelica is challenging due to its static nature and lack of native VSS support, which limits switching between modes and submodels. Constructs like `if-else` and `when` handle only simple cases, while state machines and state graphs offer greater expressiveness but depend on sequential paths, making them less suitable for highly dynamic context-driven changes.

To address these challenges, we introduced CoPN as an advanced variability management pattern for Modelica. CoPN allow for the efficient representation of complex contextual dependencies and dynamic transitions between system states. Additionally, we presented FMUVSS as a framework for realizing VSS within the Modelica ecosystem, offering a comparison with DySMo and identifying DySMo's bottlenecks. With its user-friendly configuration interface, FMUVSS provides a seamless approach to managing variability and dynamic transitions.

By combining CoPN with FMUVSS, we proposed a generalized approach for context-oriented Modelica, where simulations are composed of submodels activated by current contexts. This forms the foundation of a **context-oriented modeling paradigm for Modelica**. The proof of concept demonstrates the potential for advanced variability management in dynamic and adaptive systems. Specifically, we showed that CoPN can be implemented directly in Modelica for systems without structural change and introduced the CoPNCompositor for automated CoPN construction using PNlib. For systems with structural change (VSS), we introduced ContextModelica, which integrates CoPN and FMUVSS.

We further explored submodel management, discussed how hierarchical FMUs can assist in managing submodels, and highlighted the potential of the new Scheduled Execution FMU type introduced in FMI 3.0. This FMU type aligns well with the SUM concept by enabling event-driven mode transitions within a unified FMU. However, since this functionality is not yet widely supported, we proposed an intermediate solution: managing VSS submodels through the SUM, which encapsulates all model information and can be dynamically composed into submodels based on active contexts and modes.

Overall, we demonstrated how context-oriented modeling and programming can be integrated into Modelica, presenting a scalable framework with a unified configuration interface. Its key modules are summarized in Table 3.

Table 3. Key modules for context-oriented Modelica

Module 1: FMUVSS
Module 2: ContextModelica
Module 3: ModeGen
Supporting Tool: CoPNCompositor

6.2 Outlook

Future work aims to further refine and expand the framework, focusing particularly on improving scalability and usability. We plan to integrate the Scheduled Execution FMUs into the framework, once they become widely supported in Modelica tools. Additionally, we plan to explore further collaborations to apply our framework to real-world models, particularly in industries where variability management and dynamic system behavior are critical. These collaborations will provide valuable feedback to help improve the framework.

Acknowledgements

The authors would like to thank the Boysen–TU Dresden–Research Training Group for the financial and general support that has made this contribution possible. The Research Training Group is co-financed by the Friedrich and Elisabeth Boysen Foundation and TU Dresden.

References

Andersson, Christian, Johan Åkesson, and Claus Führer (2016). "PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface". In: *Technical Report in Mathematical Sciences* 2.

Baldan, Paolo, Andrea Corradini, and Ugo Montanari (2001). "Contextual Petri Nets, Asymmetric Event Structures, and Processes". In: *Information and Computation* 171.1, pp. 1–49.

Benveniste, Albert et al. (2022). "Algorithms for the Structural Analysis of Multimode Modelica Models". In: *Electronics* 11.17, p. 2755.

Blochwitz, Torsten et al. (2011). "The Functional Mockup Interface for Tool independent Exchange of Simulation Models". In: *8th international Modelica Conference*, pp. 105–114.

Broman, David and Jeremy G Siek (2012). *Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages*. Tech. rep. UCB/EECS-2012-173.

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

Dey, Anind K. and Gregory D. Abowd (2000). "The Context Toolkit: Aiding the Development of Context-Aware Applications". In: *Workshop on Software Engineering for Wearable and Pervasive Computing*, pp. 431–441.

Elmqvist, Hilding, Sven Erik Mattsson, and Martin Otter (2014). "Modelica extensions for multi-mode DAE systems". In: *10th international Modelica conference*, pp. 183–193.

Elmqvist, Hilding, Andrea Neumayr, and Martin Otter (2018). "Modia - Dynamic Modeling and Simulation with Julia". In: JuliaCon.

Giorgidze, George (2012). "First-Class Models: On a Noncausal Language for Higher-order and Structurally Dynamic Modelling and Simulation". PhD thesis. University of Nottingham.

Gomes, Cláudio et al. (2021). "The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations". In: *14th International Modelica Conference*, pp. 27–36.

Gómez Esperón, Daniel (2017). "Strukturvariabilität in der objektorientierten Modellierung und Simulation durch Generierung von Varianten". PhD thesis. TU Berlin.

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

Gutsche, Christian, Zizhe Wang, et al. (2023). "PNRG - A Library for Modeling Variable Structure Energy Grids in Modelica using Energetic Petri Nets". In: *15th International Modelica Conference*, pp. 727–736.

Hirschfeld, Robert, Pascal Costanza, and Oscar Nierstrasz (2008). "Context-Oriented Programming". In: *Journal of Object technology* 7.3, pp. 125–151.

Ma, Yingbo et al. (2021). "Modelingtoolkit: A Composable Graph Transformation System For Equation-Based Modeling". In: *arXiv preprint: 2103.05244*.

Mai, Carl et al. (2018). "Adaptive Petri Nets - A Petri Net Extension for Reconfigurable Structures". In: *10th International Conference on Adaptive and Self-Adaptive Systems and Applications*. Vol. 2.

Mattsson, Sven Erik, Martin Otter, and Hilding Elmquvist (2015). "Multi-mode DAE systems with varying index". In: *11th International Modelica Conference*, pp. 89–98.

Mehlhase, Alexandra (2015). "Konzepte für die Modellierung und Simulation strukturvariabler Modelle". PhD thesis. TU Berlin.

Muschevici, Radu, Dave Clarke, and Jose Proenca (2010). "Feature Petri Nets". In: *14th International Software Product Line Conference (SPLC 2010)*. Vol. 2. Lancaster University, United Kingdom, pp. 99–106.

Nytsch-Geusen, Christoph et al. (2005). "MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics". In: *4th International Modelica Conference*. Vol. 2. Citeseer.

Petri, Carl Adam (1962). "Kommunikation mit Automaten". PhD thesis. Universität Hamburg.

Petzold, Linda R (1982). *Description of DASSL: A Differential/Algebraic System Solver*. Tech. rep. Sandia National Labs., Livermore, CA (USA).

Pommereau, Franck (2015). "SNAKES: A Flexible High-Level Petri Nets Library (Tool Paper)". In: *Application and Theory of Petri Nets and Concurrency: 36th International Conference, PETRI NETS 2015*. Springer, pp. 254–265.

Proß, Sabrina and Bernhard Bachmann (2012). "PNlib - An Advanced Petri Net Library for Hybrid Process Modeling". In: *9th International Modelica Conference*, pp. 3–5.

Sommer, Torsten (2017). *FMPy: A Python Library to simulate Functional Mock-up Units (FMUs)*.

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

Tinnerholm, John et al. (2020). "Towards an Open-Source Modelica Compiler in Julia". In: *Asian Modelica Conference 2020*, pp. 143–151.

Utkin, Vadim (1977). "Variable Structure Systems with Sliding Modes". In: *IEEE Transactions on Automatic control* 22.2, pp. 212–222.

Wang, Zizhe et al. (2024). "Proposal for A Context-Oriented Modelica Contributing to Variable Structure Systems". In: *American Modelica Conference 2024*, pp. 53–62.

Zimmer, Dirk (2010). "Equation-Based Modeling of Variable-Structure Systems". PhD thesis. ETH Zurich.