Masterarbeit

# The Introduction of Variability Management to Automotive Co-Simulation

Philipp Töglhofer

Institut für Technische Informatik
Technische Universität Graz


Kompetenzzentrum - Das Virtuelle Fahrzeug
Forschungsgesellschaft mbH Graz


Betreuerin: Dipl.-Ing. Andrea Leitner
Begutachter: Dipl.-Ing. Dr. techn. Christian Kreiner

Graz, Mai 2012

## Abstract

Simulation is a well-known approach to supporting the development of complex systems at an early stage in the development process. The increased interrelationship between engineering disciplines leads to the demand for the integration of subsystems from various domains. Co-simulation accommodates this demand by enabling the coupling of subsystems, which are simulated in their domain-specific simulation tools. The independent co-simulation platform ICOS, developed at the Virtual Vehicle Competence Center, aims to couple subsystems from different areas of expertise of the automotive domain.

Variability has an inherent nature in the automotive industry. Vehicle manufacturers supply customers with a tremendous number of variants. Customer needs and differing legal constraints are just two reasons for the need of vehicle variability. This drives the need for managing variability across all processes in automotive engineering, including co-simulation.

This thesis aims to investigate the sources of variability that affect co-simulation environments. Items or properties that are commonly variable in co-simulations, so called variation points, are identified. Based on the common variation points, the requirements of a tool that introduces variability management to co-simulation environments are analysed.

As part of the work a tool to describe and manage variability in ICOS was implemented. The main goal is to make variability in ICOS co-simulation projects explicit. This is done by describing the variation points of a co-simulation project in a variability model, which is decoupled from the co-simulation. At a later point in time, the introduced variation points are bound to specific values, resulting in concrete co-simulations. Having an explicit, decoupled description of a co-simulation project's variability improves communication and traceability of variability. It further enables the integration of this variability model into higher level models or approaches, which span over the entire development process of the system.

The tool was applied in a case study of a hybrid electric vehicle co-simulation. Several variation points were described for the co-simulation, resulting in a variability model of the co-simulation. This variability model was then used to automatically simulate different vehicle variants as well as to optimise parameter values. The main benefits that were observed were (1) efficient development and maintenance of a large number of product variants and (2) having a single point of control and a dedicated description of the co-simulation's variability.



**Keywords:** co-simulation, cosimulation, independent co-simulation, automotive co-simulation, ICOS, variability management, variant management, product line engineering, software product lines, SPL

# Kurzfassung

Simulation ist ein vielversprechendes Werkzeug bei der Entwicklung komplexer Systeme. Die zunehmende Verflechtung unterschiedlicher Ingenieurdisziplinen macht eine gemeinsame Simulation von Teilsystemen dieser Disziplinen erforderlich. Co-Simulation ermöglicht die Zusammenführung domänenspezifischer Teilsysteme in einer gemeinsamen Simulation. Die Co-Simulationsplattform ICOS, welche am Kompetenzzentrum - Das virtuelle Fahrzeug entwickelt wurde, ermöglicht die Kopplung von Teilsystemen aus unterschiedlichen Fachgebieten. Dabei werden bereits etablierte Simulationsprogramme für die Modellierung und Simulation der Teilsysteme verwendet.

Variantenvielfalt ist seit Jahren ein fester Bestandteil der Automobilindustrie. Nahezu alle Hersteller bieten eine große Anzahl von Produktvarianten an. Die Gründe dafür sind unter anderem unterschiedliche Kundenwünsche und gesetzliche Bestimmungen in den Absatzmärkten. Diese Entwicklung macht es erforderlich, dass Varianten über den gesamten Entwicklungsprozess hinweg verwaltet werden. Dies umfasst auch die Co-Simulation.

Die vorliegende Arbeit untersucht die Auswirkungen von Variabilität auf Co-Simulation. Dabei werden Anforderungen an das Variantenmanagement im Bereich der Co-Simulation analysiert. Teile und Eigenschaften einer Co-Simulation, die in unterschiedlichen Varianten voneinander abweichen, sogenannte Variationspunkte, werden identifiziert. Teil dieser Arbeit ist die Implementierung eines Variantenmanagementtools für die ICOS Co-Simulationsplattform. Hauptziel der Implementierung ist die explizite Beschreibung von Variabilität in einem, vom Co-Simulationsprojekt getrennten, Variantenmodell. Das Variantenmodell beschreibt die Variabilität der Co-Simulation mit Hilfe der Definition von Variationspunkten. Zu einem späteren Zeitpunkt können Produkte erstellt und simuliert werden, indem konkrete Werte für die Variationspunkte gewählt werden. Die explizite Repräsentation der Variabilität ermöglicht unter anderem eine verbesserte Kommunikation und Dokumentation von Variabilität in einer Co-Simulationsumgebung. Außerdem ermöglicht es die Integration des Co-Simulation-Variantenmodells in das Variantenmanagement einer höheren Ebene, unter anderem des Gesamtsystems.

Das implementierte Tool wurde daraufhin verwendet, um konkrete Beispielsimulationen aus dem Bereich der Hybridfahrzeugtechnik durchzuführen. Die Variabilität der Co-Simulation wurde in einem Variantenmodell beschrieben, welches im Anschluss zur automatisierten Simulation von Fahrzeugvarianten und zur Optimierung von Simulationsparametern verwendet wurde. Dabei wurden mehrere Vorteile der expliziten Repräsentation der Variabilität beobachtet, unter anderem die effiziente Entwicklung einer großen Anzahl von Produktvarianten und die Möglichkeit, Variabilität von einem zentralen Punkt verwalten zu können.

**Stichwörter:** Co-Simulation, unabhängige Co-Simulation, ICOS, Variantenmanagement, Variabilitätsmanagement, Software Product Lines, Produktlinienentwicklung

## Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, May 10th, 2012 .....

(signature)

## Acknowledgment

This master thesis was written at the Institute for Technical Informatics (ITI), Graz University of Technology and the Virtual Vehicle Competence Center (ViF) Graz. I'd like to thank the Area E at the Virtual Vehicle Competence Center for the project funding and provision of software licences for the ICOS co-simulation platform. Furthermore, I'd like to acknowledge the financial support of the "COMET K2 - Competence Centres for Excellent Technologies Programme" of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economy, Family and Youth (BMWFJ), the Austrian Research Promotion Agency (FFG), the Province of Styria and the Styrian Business Promotion Agency (SFG).

I thank my supervisors Andrea Leitner and Christian Kreiner (ITI) for their help and guidance throughout the whole project. Special thanks go to Josef Zehetner, Norbert Thek, and Wenpu Lu (ViF) for their support.

Last but not least I'd like to thank my friends and family: Markus, Matthias and the Georgs for all the discussions during our projects, Anna for her patience and understanding during the last couple of years, my brother and sister for their encouragement and my parents for their support and belief in me.

Graz, May 2012 Philipp Töglhofer

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **AE** | Application Engineering |
| **AM** | Application Model |
| **API** | Application Programming Interface |
| **AUTOSAR** | AUTomotive Open System ARchitecture |
| **BCS** | Boundary Condition Server |
| **CAE** | Computer Aided Engineering |
| **DM** | Domain Model |
| **DE** | Domain Engineering |
| **DOM** | Document Object Model |
| **ECU** | Electronic Control Unit |
| **EDLC** | Electric Double-Layer Capacitor |
| **EMS** | Energy Management System |
| **FOH** | First-Order-Hold |
| **GUI** | Graphical User Interface |
| **HEV** | Hybrid Electric Vehicle |
| **HW/SW** | HardWare/SoftWare |
| **ICOS** | Independent CO-Simulation |
| **ICOS Bat** | Independent CO-Simulation batch mode |
| **ICOS VM** | Independent CO-Simulation - Variability Management |
| **IDE** | Integrated Development Environment |
| **JAXB** | Java Architecture for Xml Binding |
| **NEPCE** | Nearly Energy-Preserving Coupling Element |
| **OEM** | Original Equipment Manufacturers |
| **OO** | Object Oriented |
| **OS** | Operating System |
| **PHEV** | Plug-in Hybrid Electric Vehicle |
| **POJO** | Plain Old Java Object |
| **SISO** | Single-Input Single-Output |
| **SOC** | State Of Charge |
| **SOH** | Second-Order-Hold |
| **UDC** | Urban Driving Cycle |
| **UI** | User Interface |
| **Var** | Variant |

| | |
|---|---|
| **VM** | Variability Management |
| **VP** | Variation Point |
| **XML** | eXtensible Markup Language |
| **XPath** | XML Path Language |
| **XSD** | XML Schema Definition |
| **ZOH** | Zero-Order-Hold |

# Chapter 1

# Introduction

## 1.1 Problem Description and Scope

Simulation is a well-known approach to support the development of complex systems. Time and money can be saved if a model of the system is simulated before the real system is actually built. The increased interdependence of engineering disciplines led to the demand for the integration of several domain-specific models into a single simulation. This demand can be satisfied by the application of *co-simulation* [GKL06].

The main task of co-simulation is the holistic simulation of a system to determine its global characteristics. The overall system consists of several subsystems, which are simulated in their domain-specific simulation tools. The co-simulation platform ensures the interaction of the subsystems and thus enables the simulation of the overall system.

Particularly in automotive engineering the overall system consists of several subsystems from various engineering disciplines, such as mechanical engineering, electrical engineering and many others. Figure 1.1 shows an example of an automotive co-simulation environment. ICOS (independent co-simulation) is a co-simulation platform developed at the Virtual Vehicle Competence Center[1]. It enables cross-domain co-simulation for a wide range of engineering disciplines in the field of automotive engineering. The main goal of ICOS is to establish a universal co-simulation environment to integrate different subsystems into the overall system called vehicle [Pun07].

## 1.2 Motivation

Variability is the ability of a system to support the production of a set of artefacts that differ from each other in a preplanned fashion [BC05]. Variability has an inherent nature in the automotive industry. The main reasons for variability in automotive engineering are [NSL09]:

- Different customer needs
- Differences in regulations and legal constraints
- Different required functionality between body variants and drivetrain-variants

[1] http://www.v2c2.at

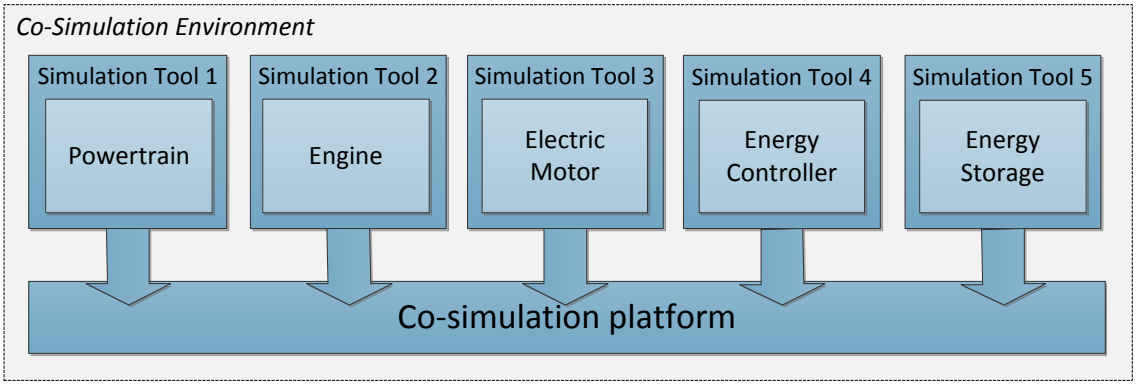


Figure 1.1: Example of an automotive co-simulation environment

- New customer expectations and standards

This drives the need for managing the variability across all processes in automotive engineering, including co-simulation. As automotive co-simulation unites subsystems from various engineering disciplines, the variability that comes from these disciplines must be handled in co-simulation too. Therefore, three different scopes of variability in co-simulation need to be considered:

- Variability of the overall system
- Variability of subsystems that are part of a co-simulation
- Variability of the co-simulation environment

## 1.3 Goals

The main goal of variability management is to *make variability explicit* and to handle its impact on all artefacts for their entire lifecycle. Thus, co-simulation variability management aims to make the variability of its included subsystems and the co-simulation platform explicit. Furthermore, it is required to handle the impact of this variability on all artefacts that are related to co-simulation.

So far reflecting the presence of several alternative representations of a subsystem could be handled implicitly, e.g. using approaches such as copy and change: A co-simulation project including a particular subsystem is copied and the reference to the subsystem is changed to one of its alternatives.

The goal of this thesis is to create a concept of how to make variability in co-simulation explicit. Furthermore, it aims to integrate this explicit description of co-simulation variants into a global (vehicle wide) variant management approach.

The concept of variability in co-simulation environments serves as a base for the implementation of a variability management tool for the ICOS independent co-simulation platform. The implementation is a proof of concept. It aims to enable explicit variability management in ICOS. Moreover, it supports the generation of product variants based on a

variable co-simulation project and the explicit description of the variability (variability model).

This thesis has been funded by the Virtual Vehicle Competence Center and has been performed at the Inititute for Technical Informatics, Graz University of Technology.

## 1.4 Outline

**Chapter 2** gives an overview of the related work. It first defines variability and describes concepts of variability management. Moreover, it explores how to manage variability within a software product line. Section 2.2 presents the basic concept of co-simulation in the automotive domain. Finally the chapter is completed with a description of the ICOS independent co-simulation platform in Section 2.3.

**Chapter 3** investigates variability management in co-simulation environments in general. The results of these investigations will further be used to define the requirements for the introduction of variability management to the ICOS independent co-simulation platform. **Chapter 4** describes details of the implementation of the ICOS variability management tool. The implementation is evaluated in **Chapter 5** using an automotive co-simulation case study. An overview of the lessons learned and an outlook of future work is presented in **Chapter 6**.

# Chapter 2

# Related Work

## 2.1 Managing Variability

### 2.1.1 Design for Reuse

The concept of *reuse* is not at all new in the software industry. Software developers have reused existing code using copy-and-paste for decades. Subsequently more *systematic* approaches such as architectures, patterns, components, libraries and frameworks were established. As reuse is greatly more cost effective than development, this is a promising way to reduce development time and cost, improve software quality and leverage existing effort [Sch06, LSR07]. Some positive examples of systematic reuse are the C++ standard template library, framework-based middleware technologies (such as CORBA and J2EE) and object oriented techniques (such as design patterns).

However, component reuse is often reduced to the integration of third-party libraries and is far too often *''not an integral part of an organisation's software development processes''* [Sch06].

One systematic reuse strategy that has been established is *software product line* engineering (SPL). In contrast to the approaches mentioned earlier, software product lines use a predictive reuse strategy, i.e. software artefacts[1] are only built if their future use in the product line is predicted [Kru12a].

The overall goal of all the presented approaches is the reuse of existing artefacts in some way or the other. In order to make use of these existing artefacts, they might need to support some kind of *variability*. However, the degree of variability, which is possible to achieve, depends on the specific artefact. For instance, using a third-party library supports little variability in the artefact itself (the library). On the other hand, templates are highly variable.

Software product lines try to handle variability in a way that maximises return on investment for building and maintaining specific products. Thus, it attempts to find a good trade-off between variability on the one hand and the costs for introducing and maintaining variability on the other hand [BC05].

[1] In software development artefacts are, but are not limited to, code, tests, requirements...

Many concepts that will be presented in this chapter are not only applicable to software engineering. Other engineering disciplines or techniques can benefit from approaches such as variability management and product line management. In fact, Chapter 3 attempts to determine an approach to introduce and manage variability in co-simulation. Moreover, the possibility of systematic reuse in co-simulation will be studied.

### 2.1.2 Terminology

Until 1996 two independent groups worked on software product line engineering. Therefore each of the groups established their own terminology. Table 2.1 gives the most important terms from both groups [Lei09]. Note that for the course of this work the terms in the first column will be used. However, citations may contain terms from the second column too.

| **Used terminology** | **Alternative terminology** |
|---|---|
| product line | system family, product family |
| software product line | software family |
| core assets, core artefacts | software artefacts |
| domain engineering | core asset development |
| application engineering | product development |
| product, product assets | application |

Table 2.1: Software Product Line Terminology [Lei09]

### 2.1.3 Variability

In common language, variability refers to the ability or tendency to change [PBL05]. This is a rather wide ranging definition and therefore not enough to support the arguments made later.

Bachmann and Clements [BC05] define variability as *"the ability of a system, an asset, or a development environment to support the production of a set of artefacts that differ from each other in a preplanned fashion"*.

The definition requires variability to be supported in a *preplanned fashion*. This describes variation as something that does not occur by chance, but is anticipated. Developers of core assets think about the consequences of different variations. They explicitly define and constrain variations in a way that takes into account limited development time and budget [BC05].

### 2.1.4 Variability Management

*''Variability Management (VM) encompasses the activities of explicitly representing variability in software artefacts throughout the lifecycle, managing dependencies among different variabilities, and supporting the instantiations of those variabilities''* [SJ04]. In other words the main goal of variability management is to *make variability explicit* and handle its impact on all artefacts for their entire lifecycle.

Variability management supports the development and reuse of variable artefacts. In order to plan for variability, four issues have to be addressed [PBL05]:

1. *Defining variability*: identify and document variability.
2. *Managing variable artefacts*: manage the variability of code, requirements...
3. *Resolving variability*: support the resolution of variability at particular points in time (binding time, Section 2.1.5)
4. *Support traceability*: Collect, store and manage trace information between artefacts.

### 2.1.5 Making Variability Explicit

As stated above it is of major importance to make variability explicit. When the variability of artefacts is defined, it is important to ask oneself three questions [PBL05]:

1. ***What*** varies?
2. ***Why*** does it vary?
3. ***How*** does it vary?

As an example, consider a car. Only the colour of a certain type of car is defined variable, while every other part or property of the car is constant. Thus, the first question is already answered. The reason for cars of different colours might be the different tastes of people. How can it vary? Obviously, it can be red, green, blue or any other colour.

Explicit documentation of variability improves decision making. It forces an engineer to document the motivations for the introduction of variability. Furthermore, explicitly documented variability helps other engineers in binding[2] a variable part. Moreover, explicit documentation improves communication about and the traceability of variability [PBL05].

The remainder of this section describes concepts that help to explicitly define variability.

#### Variability Objects and Subjects

The first question to be asked when variability is made explicit is ''What varies?''. This question can be answered by precisely identifying the item or property which is variable. Such a variable item is called a ***variability subject.*** Hence, a variability subject can be defined as *''a variable item of the real world or a variable property of such an item''* [PBL05].

[2]Binding can be explained as deciding about something that is variable or choosing a particular variant. Binding is described in more detail later in this section.

The question ‘‘How does it vary?’’ deals with the different kind of shapes or values a variability subject can have. This is referred to as ***variability object*** and can be defined as *‘‘a particular instance of a variability subject’’* [PBL05].

For example the colour can be a variability subject of a real world item. Thus blue, red, green or any other possible colour can be a variability object connected to the variability subject ‘‘colour’’.

### Variation Points and Variants

While variability objects and subjects are defined for real world items and properties, variation points and variants are terms in the context of software product lines and are therefore related to a specific domain.

A ***variation point*** (VP) can be defined as *‘‘a variability subject within domain artefacts enriched by contextual information’’* [PBL05]. *‘‘The variation point describes where differences exist in the final application’’* [LSR07].

Respectively, a ***variant*** is defined as *‘‘a representation of a variability object within domain artefacts’’*.

In correspondence with the explanation of variability objects and subjects, it can be concluded that variation points answer the question ‘‘What varies?’’, while variants give an answer to ‘‘How does it vary?’’. However, both questions are bound to a certain context.

Taking the example from above, the colour of a car can be identified as a variation point in the automotive domain. The colours red and green can be defined as variants for this variation point and therefore no other colour is allowed in this context.

Some authors use the term *variable parts* instead of variation point. The explanation is, that a variable part is more than just a point or a location within the core assets that needs adaptations or configuration, but it is an organising container for artefacts that are used for product-specific adaptations [BC05].

### Resolution/Binding

***Binding*** is the process of selecting from the options available for each variation point [Kru12b]. When binding is completed the behaviour of the variation point is fully specified.

*‘‘The binding time describes the point in time, when the decision upon selection of a variant must be made’’* [LSR07].

Variation points can be seen as delayed design decisions [BFG+02]. Delaying a decision to a later point in time enables a certain degree of flexibility. The choice of the appropriate binding time is crucial: If a variation point is bound too early, the flexibility that was achieved by introducing the variation point in the first place, is lost. However, binding variation points at a late stage in development is expensive [JB02].

Obviously there are a lot of different binding times in the software development cycle, such as compile time or after-build-time. It is possible to have multiple binding times for variation in a single software product line. Table 2.2 gives a non-exhaustive list of binding times and mechanisms that support those binding times [Kru04].

| **Binding Time** | **Examples** |
|---|---|
| Source reuse time | Specialisation, glue, frameworks, binary components |
| Development time | Clone-and-own, independent development |
| Build time | Preprocessors, script variants, application generators, templates, aspects |
| Package time | Assembling collections of binaries/executables/resources |
| Customer customisations | Code modifications, glue code, frameworks |
| Install time | Install config file, wizard choices, licensing |
| Startup time | Config files, database settings, licensing |
| Runtime | Dynamic settings, user preferences |

Table 2.2: Binding time examples [Kru04]

### Variants/Variation Point Relationships

Until now, variation point and variant relationships have been ignored. Nevertheless, the introduction of relationships may be useful or relationships may also be implicitly present between variants. Pohl et al. [PBL05] distinguish between variability dependencies and constraints. A dependency is a relationship between a variation point and its associated variants. It states whether a variant has to be selected (*mandatory variant*) or it can be selected or not (*optional variant*). If variants are optional, *alternative choices* can be specified. This is done by specifying a minimum and maximum number of variations that need to be selected from a group of optional variants.

Variability constraints encompass two different types of relationships: *exclusion* and *requirement*. If a variant $V_1$ *requires* another variant $V_2$, every product that selects variant $V_1$ needs to select $V_2$. In contrast, if a variant $V_1$ *excludes* another variant $V_2$, every application that selects variant $V_1$ must not select $V_2$. [PBL05, NSL09, BC05]

Figure 2.2 shows a graphical notation of variation points and variants and its relationships. The details of the graphical notation are presented in the next section.

Relationships can originate from various sources [NSL09]:

- *Logical facts:* e.g. a car without a rear window does not need an ECU that controls the rear wiper
- *Design and implementation:* e.g. choosing a particular framework has far-reaching consequences to the functionality offered

- *Management decisions*

### Graphical Notation

Pohl et al. [PBL05] created a graphical notation for variability models. Figure 2.1 shows the basic elements of the graphical notation. The authors distinguish between variation point relationships and variant relationships. This aspect was ignored when relationships were presented in the previous section, because there is no semantic difference between these relationships.

Figure 2.2 shows an example variability model using the graphical notation.

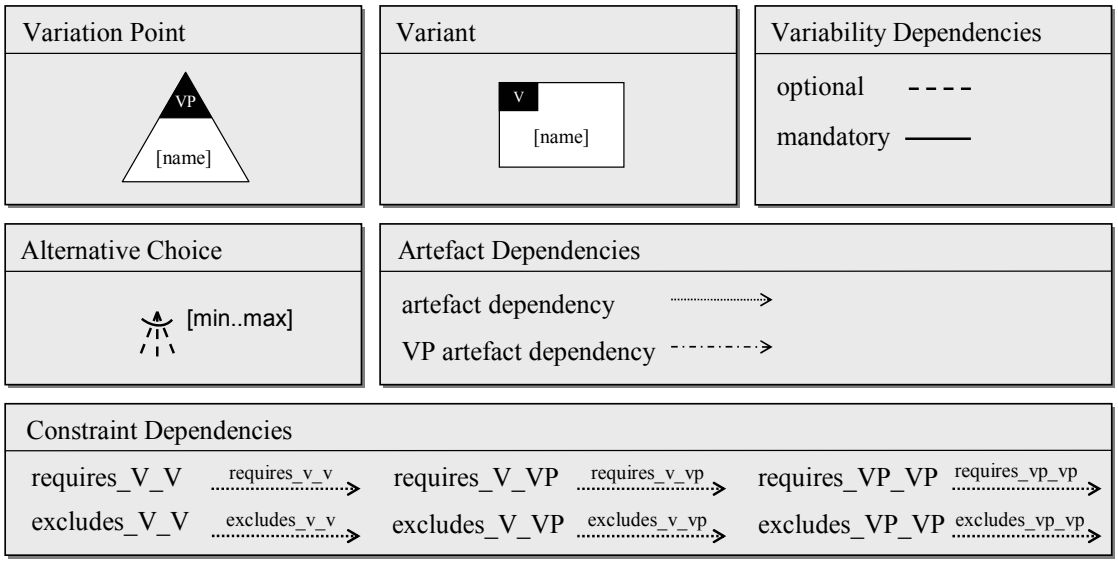


Figure 2.1: Graphical notation of variation points, variants and their relationships [PBL05]

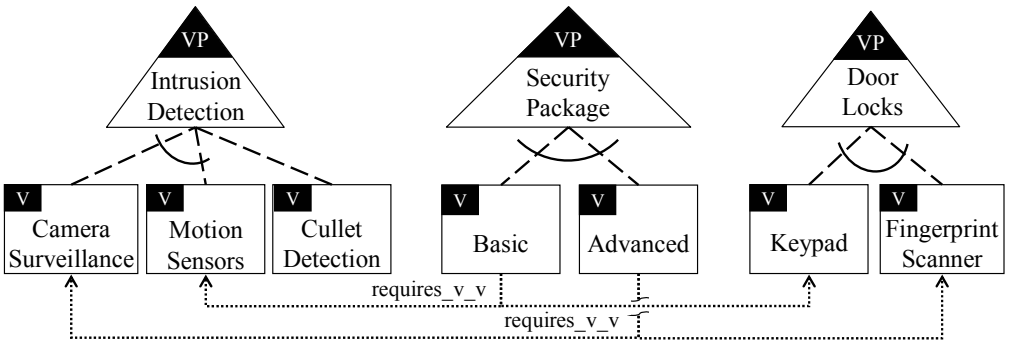


Figure 2.2: Example of a variability model using a graphical notation [PBL05]

### Internal vs. External Variability

A way to classify variation points is by its visibility to the customer. *External variability* is present, if the variability is visible to the customer, while *internal variability* is hidden from the customer [PBL05].

The colour of a car, for instance, is definitely customer-visible and therefore an external variation point. On the other hand the communication protocol between two electronic control units (ECUs) in a car is not visible to the customer and can be classified as an internal variation point.

To determine if a variation point is customer-visible or not is rather important for the support of different views on variability [NSL09]. While external variability might be of interest to customers, sales or marketing-people, internal variability in many cases is only relevant to developers.

### Variability Dimensions

Variability can be separated into two dimensions: space and time. These dimensions are interrelated.

*''Variability in time is the existence of different versions of an artefact that are valid at different times''* [PBL05]. This dimension is sometimes referred to as software evolution and is targeted by configuration management. Contrarily *''variability in space is the existence of an artefact in different shapes at the same time''* [PBL05].

Variability management in conventional software engineering only deals with software variation over time. On the other hand variability management in software product line engineering is multi-dimensional. It deals with variation in both time and space. This makes variability management the key discriminator, which distinguishes product line engineering from conventional software engineering [Kru02].

### Handling Complexity

One issue that arises when variability is modelled using variation points and variants is how to model complex systems. For instance, a variability model of a vehicle can easily consist of several hundred variants. One approach to tackle this problem is the introduction of ***abstract variation points***. Abstract VPs combine concrete VPs and predefine the bindings of its variation points. Figure 2.2 shows an abstract variation point, the security package. If the basic security package is chosen from the security package VP, for instance, the variants motion sensor and keypad are selected due to the requirement constraints.

At first glance it might seem that the additional abstract VPs add complexity to the variability model. However, it enables the creation of different views of the variability [PBL05]. For example it is possible to provide a view that only contains the security package and its variants for customers, managers or other stakeholders. The complex relationship between the security package and the other variants is hidden in such a view.

## 2.1.6 Software Product Line Essentials

Navet and Simonot-Lion [NSL09] define a software product line as a

- set of software products, that

- share a certain degree of commonality,
- while sharing substantial differences and
- are derived from a single, variable product definition in a well-defined, prescribed way.

Software product line engineering aims to embody *mass customisation* of software products. Mass customisation is the ability to efficiently create many variations of a product. The process of product creation is called production and is depicted in Figure 2.3. **Production** uses a set of existing **core assets** to derive the **products**. Core assets[3] are assets that can be configured and composed in different ways to create products [Kru12c]. The process of production can be fully automated, partially automated or completely manual [Kru12d].

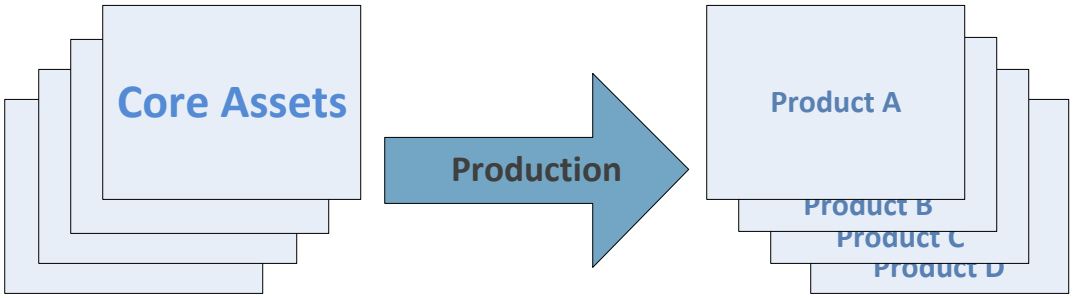


Figure 2.3: Production in a software product line

In order to build a software product line in an organisation, different approaches can be used [BC05]: The SPL can be built *proactively*. This means that core assets are created first and products are derived from these assets. Another way to build an SPL is *reactively*. In contrast to the former approach, products are built first and the core assets are created by analysing the variabilities and commonalities of the products. Of course any combination of these two approaches could be suitable in some situations.

### 2.1.7 Software Product Line Processes

Software product line engineering can be separated into two main processes [LSR07]: *domain engineering* and *application engineering* . The main goal of domain engineering is the definition of the variability and commonality of the SPL.

The process of application engineering is responsible for deriving products from the variable product definition that was established in the domain engineering process.

The split into these two processes leads to a separation of concerns in SPL engineering:

- Building a robust platform
- To be able to generate customer-specific products in short time

[3] Core assets are sometimes referred to as common assets or software artefacts

In order to benefit from this separation, the two processes have to interact with each other.

Figure 2.4 shows an SPL engineering framework that is built around the processes of domain and application engineering.

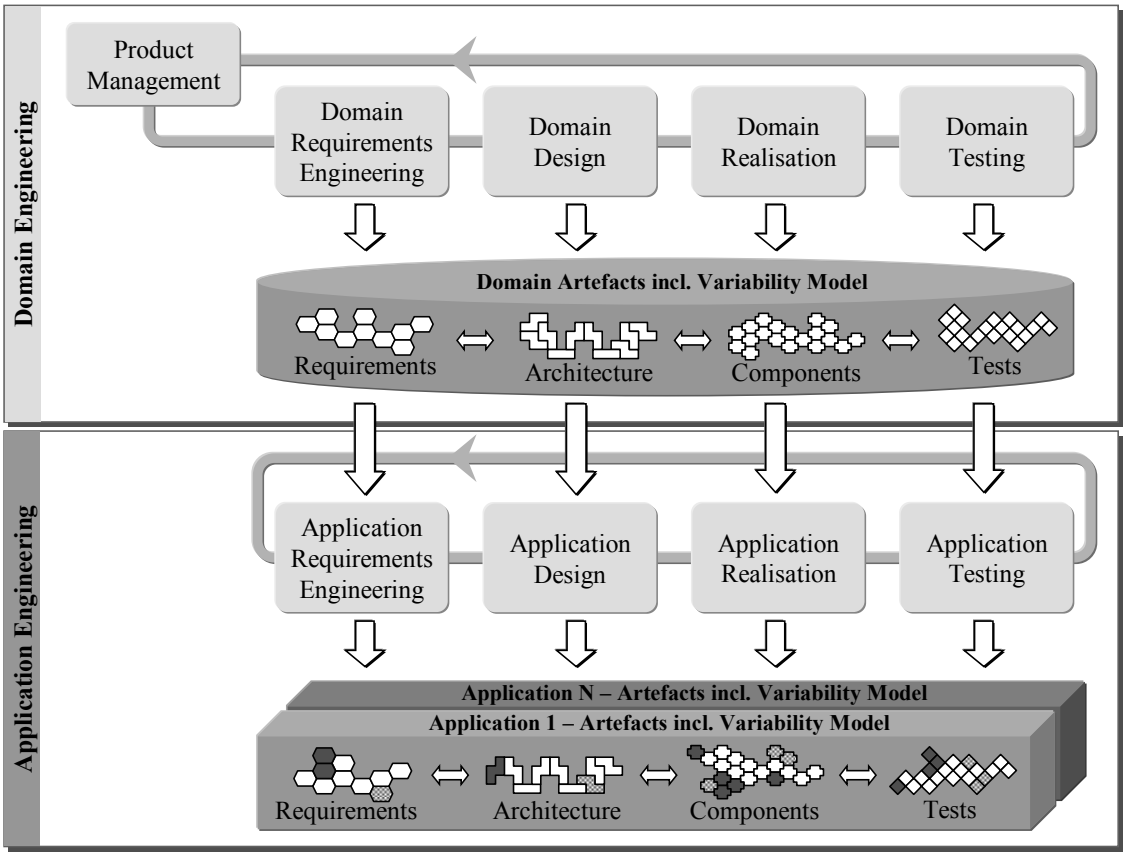


Figure 2.4: SPL engineering framework [LSR07]

### Domain Engineering

*"Domain engineering is the life-cycle that results in the core assets that together form the product line's platform"* [LSR07]. Recalling the definition of SPLs (. . . derived from a single, variable product definition. . . ), it can be stated that domain engineering is responsible for establishing this variable product definition. In other words, it is responsible for building the *core assets*.

The main task of the domain engineering process is the definition and realisation of the variability and the commonality in an SPL [PBL05]. The questions "What varies?" and "Why/how does it vary?" (Section 2.1.5) have to be answered during the domain engineering process. Hence, in domain engineering the variation points and variants are defined. This is done using five subprocesses, depicted in Figure 2.4: product management, domain requirements engineering, domain design, domain realisation and domain testing. All of these subprocesses result in core assets.

It can now be concluded that the key goals of domain engineering are to [PBL05]:

- Define the commonality and variability of the SPL.
- Define the set of products, the SPL is planned for (define the SPL's scope).
- Define and construct reusable core assets that attain the desired variability.

#### Application Engineering

Application engineering uses the core assets of the SPL to derive products [LSR07]. During the process of application engineering the variabilities of core assets are bound. This results in concrete product assets that are fit for the products being developed. These derived product assets can be combined with product-specific assets that are developed in the course of application engineering.

Similar to domain engineering, application engineering consists of subprocesses (Figure 2.4). In contrast to the domain engineering processes, these subprocesses result in concrete product assets, which are combined to a specific product [PBL05].

Therefore, the main goals of application engineering are to [PBL05]:

- Achieve an as high as possible reuse of the core assets.
- Exploit the commonality and variability of the software product line
- Document the product assets and relate them to the core assets.
- Bind the variability according to the application's needs.
- Analyse the impact of the differences between application and domain requirements.

### 2.1.8 Problem vs. Solution Space

In software product line engineering the different activities of development (such as requirements engineering, implementation and testing) are divided into two distinct groups [BBM05]:

- *Problem Space:* In problem space, domain analysis and requirements engineering are done and results in a specification for the system to build. The resulting specification is independent from any technical realisation. The development activities that are part of the problem space are requirements engineering and domain analysis.
- *Solution Space:* In solution space, concrete systems are built according to the (problem space) specifications. The development activities that are part of the solution space are system architecture/design, implementation and testing.

Problem and solution spaces can be described in a number of ways. The problem space can, for instance, be described in a feature model (Section 2.1.10) or with a domain specific language (DSL) [BPSP04].

It can be seen that by the separation of software activities into problem and solution spaces, application as well as domain engineering each have an impact on both problem and solution space. Therefore the essential software product line activities can be divided into four quadrants, as shown in Figure 2.5.

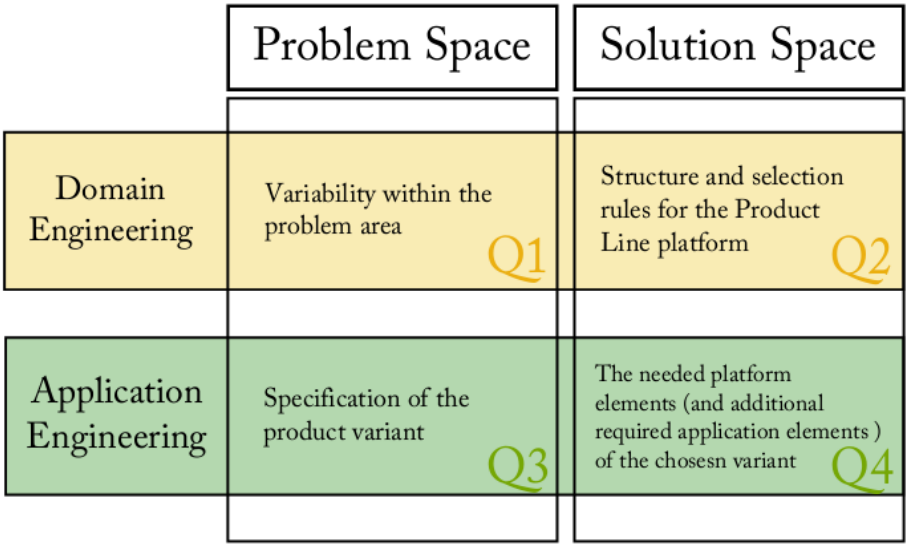


Figure 2.5: Combining process and development space [BPSP04]

### 2.1.9 Variation Mechanisms

At some time in the lifecycle of an SPL, variants of core assets are generated. In other words variation points are bound. A ***variation mechanism*** is the mechanism that is used to produce variants (product assets) of core assets in a controlled way [BC05]. Product assets are produced by either selection and modification, or creation.

With *selection and modification* a core asset is selected and then modified or configured in a preplanned way. Templates are a good example for this kind of mechanism.

With *creation*, on the other hand, a core asset is used to produce a new product asset. After the creation there is no more connection between the core and the product asset. A generator is a good example for this kind of mechanisms.

Examples of variation mechanisms include, but are not limited to, plug-ins, inheritance, component substitution, parameters, aspects, configurators, templates and generators.

Choosing the appropriate variation mechanism is crucial for the success of an SPL. Therefore it is necessary to take into account some of the properties of the variation mechanisms when choosing properties for particular product assets [BC05]:

- The skills and costs required to *implement the mechanism* (e.g. using inheritance requires a programmer).
- The skills, cost and time to *exercise the mechanism* (by exercise the actual application of the mechanism is meant).
- The impact of the variation mechanism on quality (e.g. performance penalties because of parameters that are read at runtime)
- The impact on the mechanisms maintainability

- For which of the core assets is the mechanism appropriate.

Additionally, a core asset developer needs to have *product information* to choose the right variability mechanism. Product information contain details about the products that will be produced using the core assets. With respect to variability it is particularly important to know about how the products vary from each other. In other words, the three questions to make variability explicit (Section 2.1.5) need to be answered.

### 2.1.10 Feature Models

In general, variability modelling aims to present an overview of a product line's variability. Feature models are simple, hierarchical models that capture the commonality and variability on a rather high level. A feature is a system characteristic that is relevant to a particular stakeholder. Features are organised in a tree structure. Many graphical notations for feature models have been introduced, but no standard notation has been established yet. Relationships between features can be introduced explicitly or are defined by the child-parent relationship in the tree structure [NSL09, BPSP04]. If a parent feature is contained in a variant [BPSP04]

- all its mandatory child features must be also contained (n from n)
- any number of optional features can be included (m from n, 0 < = m<=n)
- exactly one feature must be selected from a group of alternative features (1 from n)
- at least one feature must be selected from a group of or features (m from n, m<1)

### 2.1.11 Variability Management in Automotive Engineering

The main motivation to introduce a product line to automotive engineering is the inherent nature of variability in the automotive industry. Automotive software has to support variability for a number of reasons, such as customer expectations and differing legal constraints. Software product line engineering tries to manage the huge impact of variability on the company and its processes [NSL09].

The main sources of variation in the automotive industry are [NSL09]:

1. Different customer needs (different target groups)
2. Differences in regulations and legal constraints (U.S. law vs. EU regulations. . . )
3. Different required functionality between body variants (limousine, station wagon, . . . ) and drive-train-variants (e.g. automatic vs. manual transmission)
4. New customer expectations and new standards (e.g. innovation in telematics and entertainment systems)

Software product line engineering in the automotive domain has a number of characteristics compared to conventional SPLs. Some of these characteristics can make it quite hard to introduce and maintain SPLs in this field [NSL09]:

**Complex dependencies between artefacts**
In the automotive domain it is particularly important to support dependencies, such as "requires" and "excludes" between variants. Dependencies orginate either from management decisions (e.g. cars sold in Austria are either red or white), or from logical facts (e.g. a car without a rear window does not need a rear wiper). These dependencies have to be defined and managed and possible conflicts (e.g. A requires B requires C, C excludes A) have to be resolved as early as possible.

**Cooperation of heterogeneous systems**
In the automotive domain there is a need for various parties (such as developers, management) to view or manage variability. Therefore it is important that all systems and tools in an organisation (from developer tools to after-sales systems) support variability.

**Different view on variability**
As stated before, variability has an impact on various stakeholders, such as developers, marketing people, management or sales people. Obviously not all stakeholders have the same requirements when it comes to viewing the SPL. Developers might need a much more fine-grained view than e.g. sales people, who might only be interested in customer-visible variants.

**Complex configurations**
Product configuration does not only occur for the final solution of a product, but also at intermediate steps, such as prototypes. The configuration of the product is of course subject to change during development.

**No clean separation of domain and application engineering**
Due to many reasons, such as hard deadlines and often used stopgap solutions, there is no clean separation of domain and application engineering in automotive engineering. This can be a show stopper when SPLs are introduced in an organisation.

**Synchronisation with supplier strategies**
OEM (original equipment manufacturers) must integrate their strategy with a number of suppliers. On the other hand a supplier who cooperates with more than one OEM or other suppliers might have to integrate several SPL strategies. Tischer et al. [TMKG07] introduce a product line approach in the development of engine control units. They defined the synchronisation of the product line with customer needs as one of the major challenges of the introduction.

**Difficult incremental introduction**
A step-by-step introduction is of great importance in the automotive domain. The long product life cycle of cars is an obvious reason for this. Therefore, a bottom-up approach to the introduction of an SPL is more realistic. A small, local SPL for a subsystem can be introduced initially and expanded to the whole organisation.

## 2.2 Co-Simulation

*Simulation* is a well-known approach to support the development of complex systems. Time and money can be saved if a model of the system is simulated before the real system is actually built. Sometimes it is even infeasible to build a system before simulating it. In that case, simulation *''enables the study, analysis and evaluation of situations that would not be otherwise possible''* [Sha98].

The increased interdependence of engineering disciplines led to the demand for the integration of several domain-specific models into a single simulation [GKL06]. This demand can be satisfied by the application of *co-simulation*.

The main task of co-simulation is the holistic simulation of an overall system to determine the global characteristics of the system [BZWB11]. The overall system consists of several subsystems, which are simulated in their domain-specific simulation tools. Thus, the co-simulation platform is responsible for assembling the subsystems by connecting their inputs and outputs [BZWB11]. This ensures the interaction of the subsystems and thus constitutes the simulation of the overall system.

Section 2.2.2 provides a definition of co-simulation. Section 2.2.2 presents the main tasks of a co-simulation platform. One of these tasks is the data exchange between connected subsystems. This task is called coupling. Section 2.2.3 analyses the challanges of coupling. Section 2.2.4 and 2.2.5 present some applications of co-simulation. Finally, Section 2.2.6 shows the importance of a co-simulation platform's support for distributed environments.

### 2.2.1 Terminology

#### The Co-Simulation

Sometimes the phrase ''the co-simulation'' is used. ''A co-simulation'' is a synonym for ''a co-simulation project'' or ''a co-simulation setup''. In other words ''a co-simulation'' is a concrete application which is executed on a co-simulation platform.

#### Models

The American Heritage Dictionary defines a model as *''a schematic description of a system (...) that accounts for its known or inferred properties and may be used for further study of its characteristics''* [Lan03].

In the course of this work it is sometimes stated that a ''system or subsystem is simulated''. These kinds of phrases appear regularly in current research. Testing or simulating a system, in the case of co-simulation, is equivalent to testing/simulating a model of the system.

### 2.2.2 Towards a Definition of Co-Simulation

Simulation can be defined as *''the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behaviour of the system and/or evaluating various strategies for the operation of the system''* [Sha98].

It is often not possible to design a single, holistic model of a real system. Therefore a system is divided into several subsystems which can be modelled and simulated in their domain-specific tools. Hence, the *''main task of co-simulation is the holistic simulation of an overall system to determine the global characteristics of the system''* [BZWB11].

For instance, it is not feasible to create a holistic model of a vehicle. However the system ''vehicle'' can be divided into several subsystems, which can be modelled and simulated separately. Co-simulation can be used to *couple* simulation tools, in order to simulate the system ''vehicle'' as a whole.

The reasons for a split-up process like this are manifold. Some of them are listed below [Pun07, LPPFK06]:

- *Different modelling depths* of the subsystems: sometimes it is not applicable to have the same degree of detail for all subsystems.
- *Short simulation times* can be ensured by dividing the overall system into subsystems in a way that results in a short simulation time.
- *Use of existing simulation tools* is the most obvious reason in cross-domain co-simulation.
- *Logical decomposition:* sometimes a single, comprehensive model exists, which is actually composed of several partial models. The model can be decomposed into several loosely coupled partial models. These partial models can then be co-simulated. For instance Lang et. at. [LPPFK06] split-up a heat model into the two partial models, the ''cabin model'' and the ''energy flow model''.

A lot of approaches to coupling simulation tools have been developed in current research. Some of them will be described in Section 2.2.3. This led to the use of *various terms for the same approach* as well as the use of *the same term for various approaches.*

Geimer et al. [GKL06] try to define co-simulation by two criteria:

- The number of integrators and
- the number of modelling tools

Approaches that only use one tool to create a model of the whole system are called *closed modelling*. Those which use two or more tools to create models of subsystems, are called *distributed modelling*.

The same applies to the number of integrators that are used to couple subsystems. While the usage of a single integrator is called a *closed simulation*, using multiple integrators refers to *distributed simulation* [GKL06].

Amory et al. [AMO+02] also distinguish co-simulation platforms using a similar criterion, the number of simulators. However, their terminology differs as they define single simulator co-simulation as *homogeneous* and multiple simulator co-simulation as *heterogeneous co-simulation.*

Combining the two criteria from above obviously results in four different approaches. Two of these approaches have been shown to be particularly useful for the simulation of interdisciplinary systems [Dro04]: distributed modelling with closed simulation and distributed modelling with distributed simulation.

Geier et al. [GKL06] define the latter of the two to be **co-simulation**. Hence, co-simulation is the coupling of models, which were created in more than one modelling tool, using more than one integrator for simulation.

#### The Co-Simulation Platform

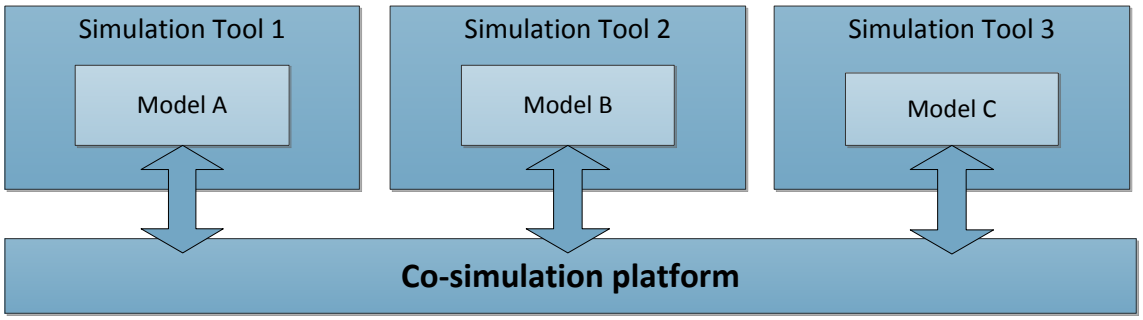


Figure 2.6: The co-simulation platform

The term *co-simulation platform* is not defined consistently. Therefore, the main tasks of a co-simulation platform will be defined, which should be sufficient for the course of this work.

Benedikt et al. [BZWB11] state that the task of a co-simulation platform *''is to take the complex interactions of the various simulation models in a suitable and correct way into account''*. Further they state that *''the platform has to enable the precise co-working of different simulation tools''*. The co-simulation platform is responsible for defining an effective scheduling for the simulation tools and handling the two-way communication between the simulation tools at specific points in time (Section 2.2.3) [AHLAO+07]. In other words, the co-simulation platform is responsible for the initialisation, scheduling, and communication of the various simulation tools and the subsystems, which are simulated in these tools.

### 2.2.3 Simulation Tool Coupling

*Coupling* is the process of simulation tool integration. *''The main tasks of a coupling methodology is on the one hand to handle the input and output data of subsystems and on the other hand to define an efficient schedule''* [BSW10]. In other words coupling handles the data exchange between simulation tools, or more precisely between the subsystems that are simulated in these tools. Apparently this is not a trivial task, especially if little or no knowledge of the subsystems, which are coupled, is present.

#### One step at a time

One of the main tasks of a co-simulation platform is the coupling of data between simulation tools at *specific points in time* [BZWB11]. Each simulation tool involved in a co-simulation project uses a specific numerical solver to solve a subsystem. The step-size which they use to advance the simulation in order to solve the subsystem is called a *micro time step*

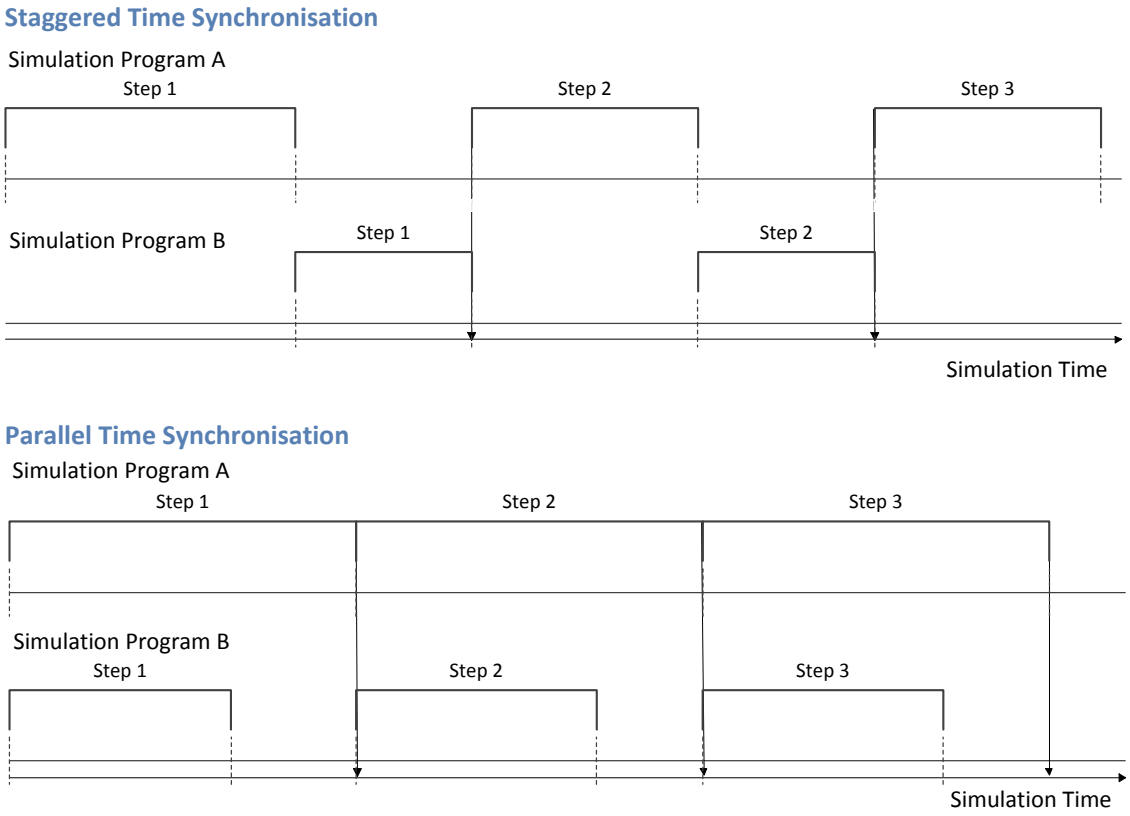


Figure 2.7: The difference between staggered and parallel time synchronisation

($\delta T$). In addition to the micro time steps, the co-simulation platform uses *macro time steps* ($\Delta T$). The macro time step defines the interval in which data exchange between the interconnected subsystems is performed.

The use of several different time steps is called *multi-rate co-simulation* [GNLG11, OV04]. This kind of co-simulation reduces simulation time by an order of magnitude without an increased error.

In order to couple the simulation of several subsystems, two approaches of synchronisation can be used: staggered and parallel time synchronisation.

In *staggered time synchronisation* each subsystem is simulated step by step in an alternating manner. So after step 1 of subsystem A is done, step 1 of subsystem B is performed and so on.

In contrast, a *parallel time synchronisation* approach handles each step of all subsystems at the same time [Pun07]. So, step 1 of subsystem A and B are performed at the same time. When the simulation of both subsystems has finished, synchronisation (coupling) is performed and the co-simulation proceeds to step 2 of both subsystems. Figure 2.7 shows the difference between the two synchronisation schemes.

### Internal loops

When two or more subsystems are present in a co-simulation system it is likely that an internal loop exists. In the simplest form an internal loop is present if the input of a subsystem depends on another subsystem's output and vice versa. This scenario is shown in Figure 2.8. It seems obvious that the co-simulation platform has to be able to deal with this situation, by providing a way of letting the simulation of both subsystems proceed [BZWB11].

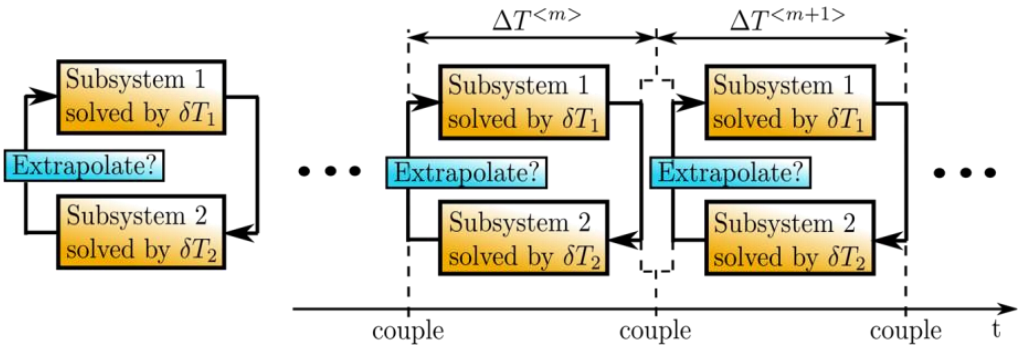


Figure 2.8: A simple internal loop in a co-simulation [BZWB11]

To solve the problem of internal loops, co-simulation platforms use *extrapolation* to predict the output of a subsystem [BSW10]. The following actions can be done to solve an internal loop like the one in Figure 2.8 [BZWB11]:

1. *Extrapolate the output of subsystem 1:* The extrapolated value is then used as input for subsystem 2.
2. *Extrapolate the output of subsystem 2:* The extrapolated value is then used as input for subsystem 1.
3. *Extrapolate the output of both subsystem 1 and subsystem 2:* The extrapolated output of subsystem 1 is used as input for subsystem 2 and vice versa.

Obviously when parallel time synchronisation is used, only the third approach (extrapolating both output values) works.

Additionally, the extrapolation techniques can be divided into iterative and non-iterative approaches. In *non-iterative coupling* (or weak coupling) past results are used to extrapolate coupling values for the current macro time step $\Delta T$. Each subsystem is *solved exactly once* for each macro time step.

In *iterative coupling* (or strong coupling) the subsystems are *solved several times* for each macro time step. Usually the iteration is performed until a certain error threshold is reached.

As the subsystem is solved several times for a particular time step, the states of the subsystem have to be reinitialised for each iteration. Many simulation tools do not support

this [BZWB11]. Thus, if a co-simulation tool is required to be independent from the simulation tools it uses, the non-iterative approach is better suited.

#### Extrapolation techniques

*''Extrapolation is the prediction of simulation results (output of the subsystems) over the subsequent macro time steps $\Delta T$ at defined coupling time instances''* [BZWB11]. Several extrapolation techniques exist for both iterative and non-iterative coupling. The most often used non-iterative extrapolation techniques are polynomial approaches of different degrees: Zero-Order-Hold (ZOH), Frist-Order-Hold (FOH) and Second-Order-Hold (SOH). While ZOH uses only the result from the macro time step $\Delta T$-1 to predict a value at time step $\Delta T$, FOH and SOH use two and three previous values respectively.

All of these approaches only use values at macro time steps. To further improve the results of extrapolation, values at intermediate time steps have to be used.

Extrapolation predicts values, thus an extrapolation error is introduced. The order of the extrapolation error depends on the coupling approach, the homogeneity of the time scales in the subsystems and of course on the extrapolation technique that is used [GNLG11, BSW10].

#### The perfect step size

Until now constant macro time steps $\Delta T$ were assumed, hence the size of the macro time step could not change over simulation time. Nevertheless, the macro time step is an important factor of influence for [BSW10]

- *simulation time:* A small macro time step size, leads to a high simulation time. This is due to the synchronisation overhead for each macro time step.
- *error:* If a co-simulation system contains internal loops, extrapolation has to be used to solve them. The overall error that is introduced by the extrapolation is heavily dependent on the macro time step size.

Therefore it is crucial to find a macro step size that is suitable for a specific co-simulation [BSW10]. Suitable means a good trade-off between simulation time and simulation error. More often than not, such a macro time step is defined by a domain expert or determined by a trial and error approach.

Another approach to finding a well-suited macro step size is called *adaptive macro time step*. Benedikt et al. [BSW10] presented adaptive time step approaches for iterative and non-iterative coupling.

### 2.2.4 Automotive Cross-Domain Co-Simulation

As the name suggests cross-domain co-simulation is the integration of subsystems from different domains/disciplines into an overall system which is simulated as a whole. Every subsystem is modelled by a domain expert and the resulting subsystems are connected and simulated using a cross-domain co-simulation platform [GKL06].

The involvement of several domains is inherent to modern automotive engineering. Neither a mechanical nor an electrical engineer can build a modern vehicle on his own. Further to that, the modelling and simulation of dynamic systems is crucial at an early stage of the vehicle development process [Cen12a]. Cross-domain co-simulation supports the simulation of the overall system or several parts of the system at an early stage. This leads to early concept decisions for the end product.

### 2.2.5 Hardware/Software Co-Simulation

*''Hardware/software co-simulation refers to verifying that hardware and software function correctly together''* [Row94].

Hardware/software (HW/SW) co-simulation approaches can be classified into three different groups [ASB99]:

1. *Sequential approach:* is the coupling of two simulators without support for feedback.
2. *Standalone approach:* uses one complex program that handles all types of models.
3. *Paired approach:* This approach is also called the *backplane approach*. The backplane is a part of the HW/SW co-simulation platform similar to the co-simulation platform introduced in Section 2.2.2. Atef et al. [ASB99] describe the backplane as the backbone that connects different simulators. It is responsible for the data exchange between simulation tools.

Not all approaches above conform to the definition of co-simulation, given in Section 2.2.2. The definition requires that several tools are used to model subsystems (hardware and software components) and several simulation tools are used to perform the simulation of these subsystems. Obviously, the second approach (standalone) does not satisfy the requirements of the definition. The sequential as well as the paired approach use more than one simulation tool. Hence, they satisfy the second requirement of the definition. However, they do neither prohibit nor encourage the use of more than one modelling tool.

### 2.2.6 Distributed (Co-)Simulation

Please note that distributed co-simulation refers to simulation on geographically distributed computers. The term has been used in Section 2.2.2 in another context to be able to define co-simulation.

*''Distributed simulation is concerned with the execution of simulations on geographically distributed computers interconnected via a local area and/or wide area network''* [Fuj99]. The reasons why co-simulation is sometimes executed in a distributed environment are manifold [Fuj99, AMO+02]:

1. *Reduced execution time*
2. *Geographic distribution* of simulation tools. For instance, Faruque et al. [FSSD09] implement a *''combined electrical and thermal simulation carried out using two real-time digital simulators located approximately 3500 km from each other''*.

3. *Integrating a great number of simulators* that execute on machines from different manufacturers or exclusively on different platforms. This is especially true for co-simulation environments: cross-domain co-simulation systems usually consist of subsystems that are simulated in several different simulation tools.

4. *Simulator's license management:* Simulators can be installed only on a core set of machines.

5. *Fault tolerance* can be achieved by the replication of hosts.

6. *Resource sharing*

7. *Intellectual property management:* a core provider may allow the simulation of a component without giving out its description (e.g. source code).

8. *Project decentralisation*

### 2.2.7 Reusing Existing Models

Section 2.1.1 states the importance of reuse. *''Co-simulation enables the reuse and combination of already existing and validated subsystem models without re-entering model data''* [BHR+07].

The reuse of existing models can be supported by using a model database within an organisation. The model database is a collection of existing models including a description of the model and instructions how to use the model within a co-simulation. Wang et al. [WWLZ09] describe the implementation of a model database for vehicle components. Models can be store and categorise and later retrieved from the database. This enables systematic reuse of the models within an organisation.

## 2.3 ICOS - Independent Co-Simulation

ICOS (Independent CO-Simulation) is an independent co-simulation platform, developed at the Virtual Vehicle Competence Center[4]. It enables cross-domain co-simulation for a wide range of engineering disciplines in the field of automotive engineering [Cen12a].

This section gives an overview of the ICOS - Independent Co-Simulation environment and a motivation for the development of this tool. Furthermore some basic concepts of ICOS are explained. Finally, some elements of ICOS, such as the project file format, will be studied in detail. The degree of detail which is used to describe certain parts of ICOS corresponds to the importance of these parts for the concept and implementation presented in the subsequent chapters.

### 2.3.1 Motivation and Objectives

In the automotive industry many specific simulation tools have been used in the past. Most of these tools specialise on a single area or discipline of automotive engineering. Hence, there is very little support for a heterogeneous simulation environment which is inherent to the automotive industry.

Typical co-simulation platforms try to overcome this limitation by supporting coupling of various simulation tools. Nevertheless, most of them focus on a single, specific area of expertise. One example is a co-simulation tool for the *''design of a thermal management system with a heterogeneous tool landscape''* [BZWB11].

*''The development of modern, mechatronic systems requires a much broader approach. The interactions between sub-systems from different areas have to be taken into account''* [BZWB11].

ICOS supports the coupling of existing domain/area-specific simulation tools and models that were developed using these tools. The coupling of models from different areas of expertise represents a promising way to *''establish a universal independent coupling approach to integrate different subsystem to the overall system called vehicle''* [Pun07].

### 2.3.2 Models and Wrappers

As stated in Section 2.2, the main task of co-simulation is the holistic simulation of an overall system consisting of several subsystems. However, not the system itself but a model of the system is simulated. ICOS supports cross-domain co-simulation and, therefore, the interaction of simulation tools of various engineering disciplines [Cen12a]. Currently, ICOS supports the simulation tools listed in Table 2.3.

One of ICOS' main design goals was to separate the co-simulation platform and its coupling algorithm from the simulation tools that are part of the co-simulation environment. In other words, the co-simulation platform must be **independent** from the simulation tools it uses. One of the advantages resulting from this design is the minimised effort to support new simulation tools [Pun07].

[4] http://www.v2c2.at

| | |
|---|---|
| **Abaqus:** | finite element analysis tool |
| **Adams:** | multibody dynamics and motion analysis software |
| **AVL Boost:** | advanced and fully integrated ‘‘Virtual Engine Simulation Tool’’ |
| **AVL Cruise:** | adaptable tool for vehicle system and driveline analysis |
| **Dymola:** | modelling and simulation environment based on the open Modelica modelling language |
| **Flowmaster:** | tool to simulate thermo-fluid systems |
| **Kuli:** | tool for simulating and optimizing the thermal management system for automotive applications |
| **Labview:** | visual programming system |
| **LSDyna:** | general-purpose finite element program |
| **Matlab/Simulink:** | tool for modelling, simulating and analysing multidomain dynamic systems |
| **Simpack:** | multi-body simulation tool to aid engineers in the analysis and design of mechanical and mechatronic systems |
| **User-Defined:** | ICOS enables developers to include custom $C++-$programs into a co-simulation project |

Table 2.3: Simulation tools currently support by ICOS [Cen12b]

In order to separate the concerns of the co-simulation platform and the simulation tools, ICOS is built using a three-tier architecture [Pun07]:

- *Application layer:* This layer consists of all simulation tools that are used in the co-simulation environment (e.g. Adams, KULI, Matlab/Simulink).
- *Wrapper layer:* This is the intermediate layer which hides the application specific details from the co-simulation layer. It consists of application-specific (simulation tool- specific) interfaces and a global interface. Any changes in an API (Application Programming Interface) in the application layer lead only to changes in an application-specific interface. Hence, changes in the application layer are hidden from the co-simulation layer.
- *Co-simulation layer:* The co-simulation layer is an independent control unit that handles the coupling process.

Figure 2.9 shows the three-tier architecture of ICOS.

### 2.3.3 Parameters

The provision of input and output parameters of subsystem models is supported by many simulation tools. This way the ICOS platform is able to exchange data between subsystem

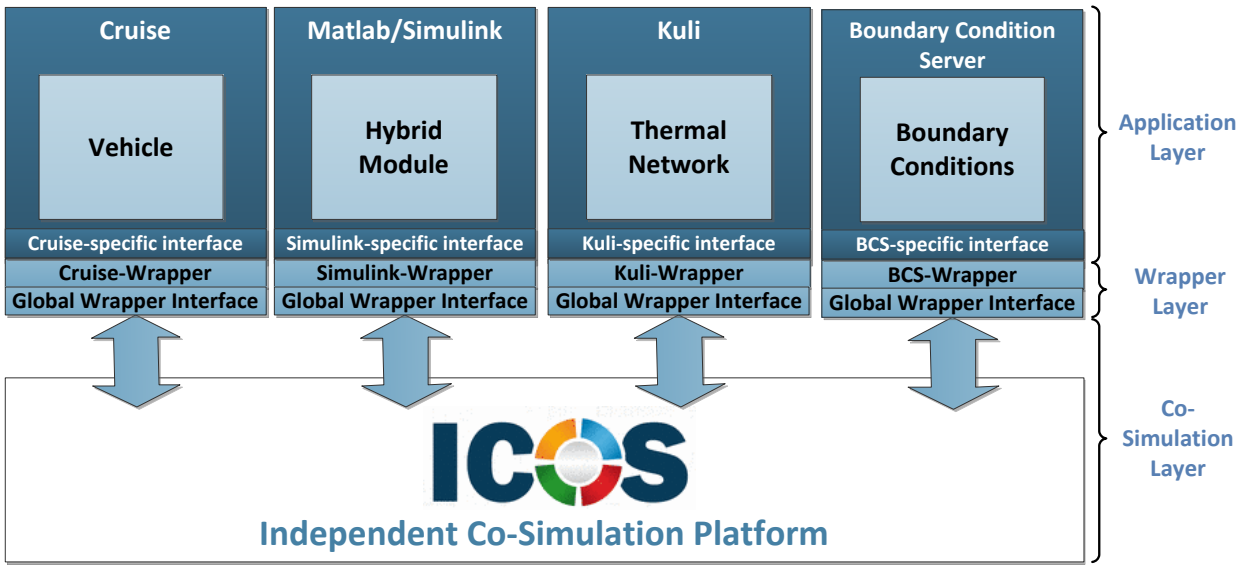


Figure 2.9: The ICOS co-simulation environment's layered architecture [Pun07]

models. In the case of Matlab/Simulink, for instance, there are two ways to connect Simulink component models to ICOS. Either the standard in- and outport components of Simulink can be used or special "ICOS InPort" and "ICOS OutPort" components can be imported into a Simulink model [Cen12b].

For every model in the co-simulation, its parameters have to be specified in the co-simulation project. In order for ICOS to be able to use a parameter, the following properties have to be provided [Cen12b]:

- *In/out:* tells ICOS whether the parameter is an input parameter or an output parameter of the model.
- *Name in simulation tool:* Most simulation tools use names or numbers for its input and output parameters/ports. Giving the name of the parameter in the simulation tool enables ICOS to connect to the model parameter with the given name.
- *Name in ICOS:* is an ICOS-internal name of the parameter.
- *Unit:* ICOS enables the specification of the unit (such as meter, ms, Fahrenheit...) to protect from connecting parameters with different units.
- *Data type and dimensions:* Currently ICOS supports parameters of type integer, string and double. Values can be scalars as well as vectors.

The distinction between parameter names in the model and the name in ICOS is in accordance with the layered architecture of ICOS. Figure 2.10 shows the mapping from parameter names in the model to the according names in ICOS. This separation enables one to refer to the parameter in ICOS independently from the parameter's name in the simulation tool. Therefore, changing the name in a model requires only a minor change in

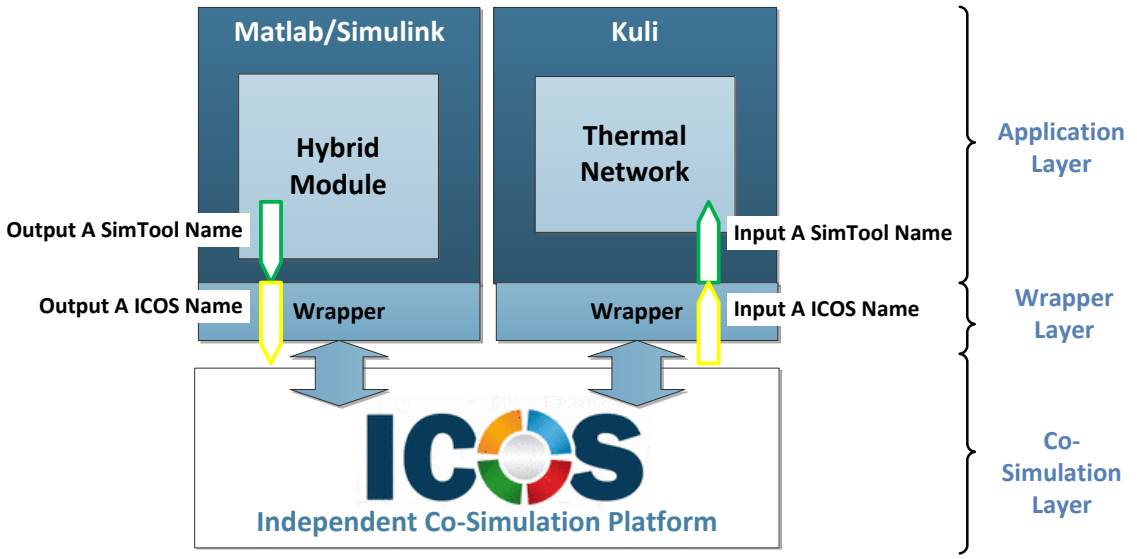


Figure 2.10: Parameters in the ICOS co-simulation environment's layered structure

the co-simulation project. Among other tasks, the parameter name in ICOS is used for the linking of parameters (Section 2.3.6) and for subscribing to parameter results.

In addition to parameters that are provided by the subsystem models, parameters can be provided by a boundary condition server, which is described in the subsequent section. In order to distinguish the different types of parameters the term ''model parameter'' will be used to explicitly refer to a parameter of a subsystem model, and the term ''BCS parameter'' to refer to boundary condition server parameters. Thus, the term ''parameter'' will refer to both model and BCS parameters.

### 2.3.4 The Boundary Condition Server

The *boundary condition server* (BCS) is a component of the ICOS co-simulation platform that provides boundary conditions and initialisation values for other subsystem models [Cen11]. For example a temperature model in KULI[5] might require the ambient temperature as input. The initial temperature as well as the change of the temperature in time can be provided using the BCS. Figure 2.11 shows a BCS example configuration in the ICOS user interface.

In general a boundary condition server is just another component of an ICOS co-simulation, just like a Matlab/Simulink or KULI model. The only difference is that a BCS only has output parameters. Thus, BCS parameter values have to be configured before the co-simulation is started and the BCS cannot react on the output of any other model. Per definition, every ICOS project contains exactly one boundary condition server [Cen11]. This does not imply any constraints as the number of output parameters for this unique instance is unlimited.

[5] http://www.kuli.at/

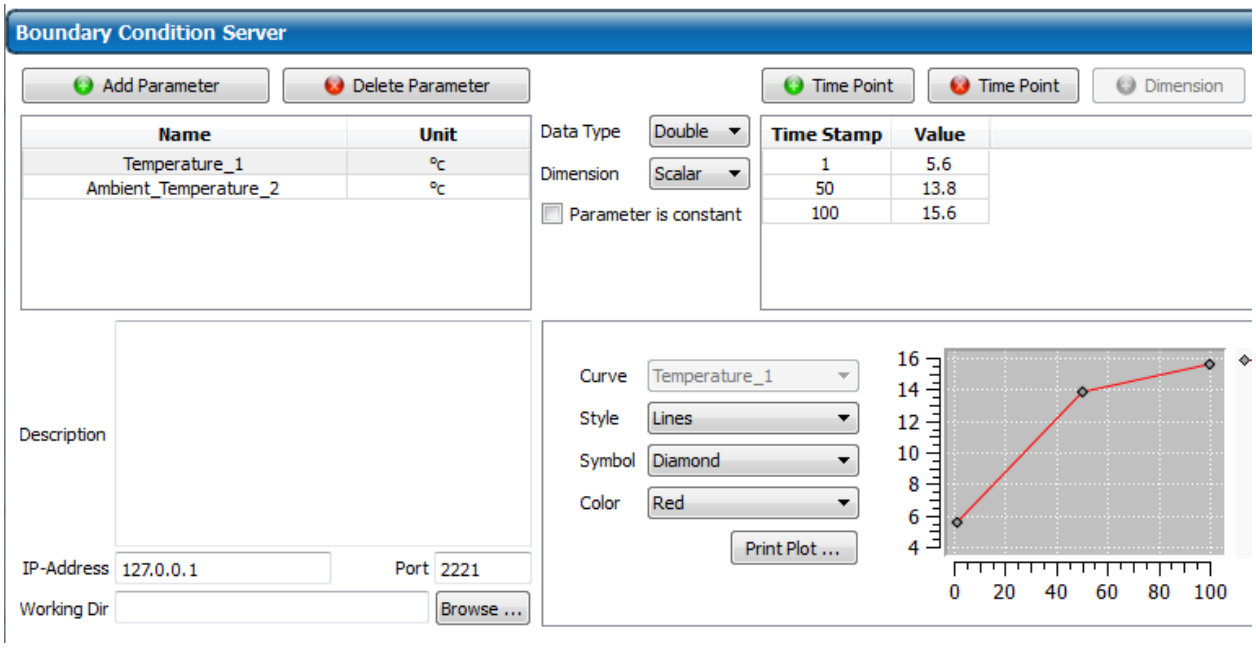


Figure 2.11: ICOS GUI - Configuration of a linear boundary condition server

As stated above, BCS parameters differ from model parameters (Section 2.3.3) in two ways. First, BCS parameters are always output parameters. Second, the values of BCS parameters must be configured in the ICOS co-simulation project before the co-simulation is started. There are two different types of BCS parameter values [Cen12b]:

- *Constant values* don't change over time. Only one value needs to be specified which will be the output value of the BCS parameter during the whole run of the co-simulation.
- *Linear values* can change over time. Values can be specified for any point in time of the co-simulation. The values for simulation steps between any two specified points in time will be linearly interpolated. This can be seen in the plot in the bottom right of Figure 2.11.

### 2.3.5 Coupling Strategies in ICOS

ICOS lets the user choose between parallel and sequential coupling and to optinally use an adaptive time step algorithm. Further, ICOS provides a coupling approach called NEPCE (Nearly Energy-Preserving Coupling Element) [Cen12b]. This is a novel, non-iterative coupling approach that is to significantly reduce the introduced extrapolation error.

### 2.3.6 Parameter Linking

In order to exchange data between models, ICOS needs linking information. This information has to be provided by the user. In other words, the user connects input and output parameters. Obviously two connected parameters need to have the same data type.

Further it is possible to connect one and the same output parameter to more than one input parameter [Cen12b].

### 2.3.7 Distributed Co-Simulation in ICOS

Section 2.2.6 provides reasons for the distribution of a (co-)simulation environment. ICOS allows simulation tools to be distributed [BZWB11]. This means that the simulation tools, which are used during co-simulation, may run on a number of different physically distributed hosts and different platforms. The ICOS co-simulation framework communicates via TCP/IP with the *ICOS remote servers*, which are running on the same hosts as the simulation tools. The remote server uses the wrapper interface (Section 2.3.2) to communicate with the simulation tool. Figure 2.12 shows the setup and different components of the distributed co-simulation.

In practice, the co-simulation project file (Section 2.3.9), along with some other files, is sent from the host that is running the ICOS framework to the hosts that are running the simulation tools. The remote server examines the project file and starts the specific simulation tool. During the course of the co-simulation the remote server reacts on the simulation tool's input and output.

### 2.3.8 ICOS Co-Simulation Work Flow

The ICOS co-simulation workflow consists of three steps in the given order [Cen12b]:

1. *Configure* co-simulation settings and define all models and its parameters.
2. *Link* input and output parameters.
3. *Simulate* the ICOS co-simulation.

Obviously, after step 3 a user evaluates the results and then reconsiders his co-simulation settings, thus going back to step 1. Further, it should be mentioned that these steps only cover the process of setting up and running a co-simulation. Before this can be done, the subsystem models that are used in the co-simulation have to be developed.

### 2.3.9 ICOS Co-Simulation Project Files

All information about the ICOS co-simulation project is stored in a project file, represented in XML. The file stores several kinds of information:

- *General information* about the co-simulation project. For instance, the name, description and version of the project file.
- *Co-simulation settings* define important properties of the co-simulation. Examples include the physical simulation duration, the coupling mode and the time step mode.
- *Wrappers/models:* Every model and its wrapper (Section 2.3.2) is defined in an XML tag with the name ''wrapper''. Among other things, the location of the model file on the remote host, the simulation tool to be used, the macro step size and the location of the init file (Section 2.3.10) are specified.

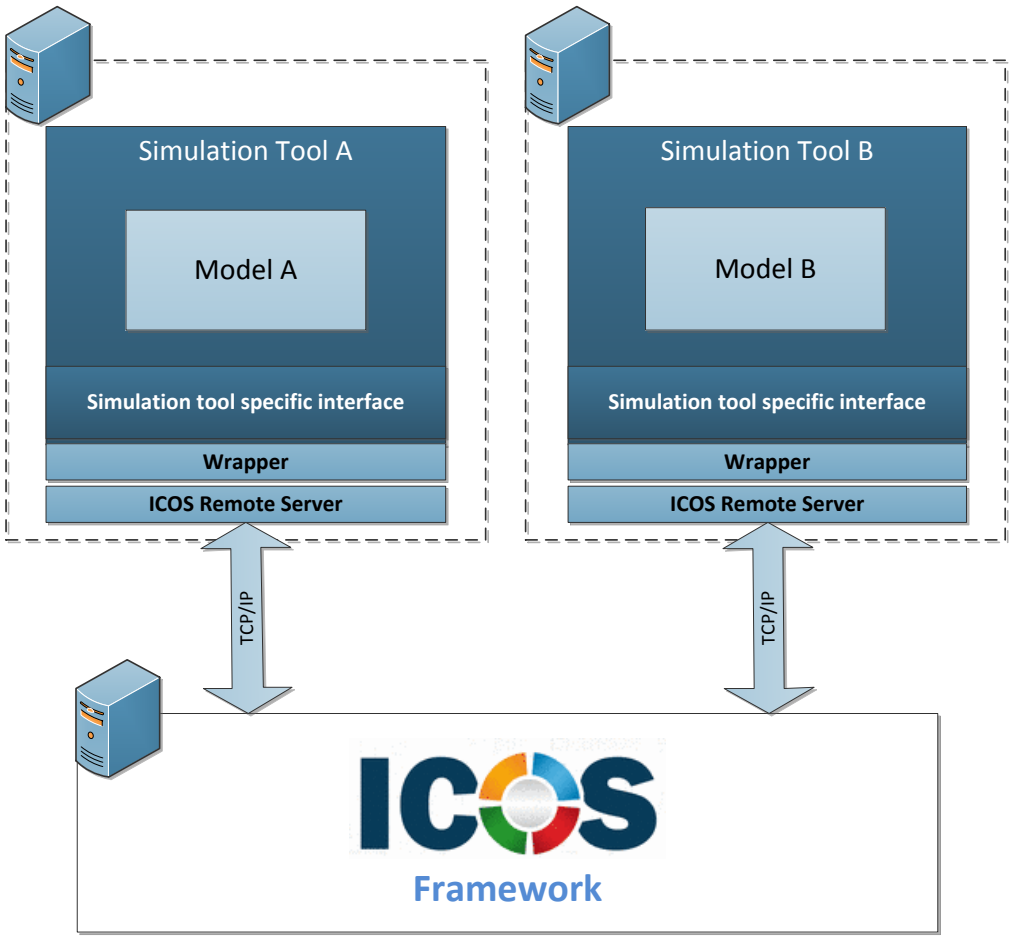


Figure 2.12: Distributed co-simulation in the ICOS co-simulation environment

- *Parameters* are defined as parts of the models. This means that the input and output parameters of a model are defined within the wrapper's XML tag. The name of the parameter in ICOS and in the simulation tools as well as its data type and unit is defined in the parameter's XML tag.
- *Links* between input and output parameters are stored in the input parameter's XML tag. This tag simply has a child tag with the name ''inputAssociatedWith''. For instance, if the parameter XML tag with the name''MyInputParameter'' has an XML child tag with the name ''inputAssociatedWith'' and the value ''MyOutputParameter'', these parameters are linked together.
- *Boundary Condition Server* describes general settings of the BCS, such as the network location and port of the host that runs the BCS. Furthermore, all BCS parameters are defined. This definition includes their name, data type, unit and their constant or linear values (Section 2.3.4).
- *GUI Data:* The name of this section is confusing. On the one hand, it actually

defines GUI-related data, such as a list of output parameters that should be plotted on the GUI after the co-simulation is executed. On the other hand, it also defines the ''subscribed parameters''. These are the parameters of which the ICOS framework collects data during a co-simulation execution. This data is used in the ICOS GUI, but is also available to the user as .dat file.

### 2.3.10 Initialisation Files

An initialisation file can be used to initialise a subsystem model [Cen12b]. It can be configured separately for each model. However, not all simulation tools support the use of init files.

## 2.4 Hypothesis

This chapter presents two different fields of research: variability management and co-simulation. Variability management, its importance and its relationship to software product lines were described. Additionally, the main goal of variability management, to make variability explicit, was defined.

On the other hand the concept of co-simulation, the details of a co-simulation platform and its importance in automotive engineering were presented.

The remainder of this work aims to introduce variability management to automotive co-simulation. First, variability in co-simulation environments in general is investigated. The results of this investigation will be used to implement an approach to manage variability in ICOS co-simulation projects. The concept and implementation that will be presented aim to prove two assumptions:

- introducing explicit variability management to co-simulation enables systematic reuse.
- making variability explicit enables the integration into a global (system-wide) variability management approach.

# Chapter 3

# Variability Management in Co-Simulation

This chapter describes variability management in independent co-simulation. First, the key motivation for identifying and managing variability in co-simulation environments is investigated. Further, the common variation points in many co-simulation scenarios are studied. Based on this study the requirements for an implementation of a variability management tool for the ICOS co-simulation platform are derived and a basic architecture is presented.

## 3.1 Application Scenarios

During the course of the project two different application scenarios are identified:

1. Standalone variability management in co-simulation environments

2. Integration in a (software) product line

The first scenario (standalone) is particularly useful if no product line is established in an organisation. Variability management makes existing variability explicit (variants exist, but are hidden in artefacts) or can be used to support optimisation or calibration tasks (finding optimal co-simulation settings).

In the second scenario (SPL integration) an existing product line is used or a new one is established, in which the process of co-simulation is integrated. The pure::variants[1] variability management tool is used to integrate ICOS variability managment into a product line.

The remainder of this work focuses on the first scenario. It presents the basic architecture of a standalone variability management tool for ICOS. Nevertheless, Section 3.5 gives an outlook on the integration of co-simulation variability management into an SPL using the pure::variants variant management tool.

[1] http://www.pure-systems.com

## 3.2 Motivation

One of the main goals of introducing variability management in co-simulation environments is to elevate systematic reuse of existing models and co-simulations. Another goal is to make variability explicit.

Co-simulation is a powerful approach to verifying a system design early in the development process. Existing models of subsystems can be coupled and different setups can be evaluated. New models of subsystems can be simulated within a holistic model of the system. Furthermore co-simulation can be used throughout the whole development cycle of a vehicle to support other development steps.

Dealing with variants is an integral part of the automotive development process. Many of these variants lead to multiple models of subsystems. For instance, if a hybrid electrical vehicle is sold with two different types of electric engines, two different models of the engine have to be developed and simulated. At best only one variable model of the engine exists. This model can be bound to the different variants of the engine.

Variability management (VM) in co-simulation environments enables the simulation of these variable models within a holistic simulation of the resulting vehicle variants. This allows early detection of errors and the evaluation of consequences that are caused by different variants. For instance, a vehicle's thermo dynamical energy flow heavily depends on the engine [Pun07]. Consequences of different engines can be evaluated using co-simulation.

But VM in co-simulation environments does not only enable the support for variability in subsystem models. Sometimes variability does not affect the subsystems of vehicles directly, but through their environment. For instance, the simulation of the heat flow of a car that is sold on the Russian market might need to be simulated with completely different ambient temperatures than one that is only sold in Australia. This can be solved by varying inputs from the vehicle's environment (such as the ambient temperature).

Another task that is supported by the explicit description of variability in co-simulations is optimisation. This is done by defining variation points for co-simulation parameters, such as the macro time step $\Delta T$, or boundary condition parameters. These parameters can then be bound to a range of values and optimised by evaluating simulation results.

A main goal of variability management is to make variability explicit. Therefore, instead of hiding variants inside subsystem models and co-simulation projects, VM in co-simulation needs to provide a distinct view of the ***variability model*** of the co-simulation.

## 3.3 Variability Management in Co-Simulation

The setup of a co-simulation environment and the essentials of a co-simulation platform were discussed in Section 2.2. Now the kind of variability to be introduced to co-simulation and how this is done is explored. Recalling the three questions from Section 2.1.5, variability management should give an answer to:

- *What* varies?

- *How* does it vary?
- *Why* does it vary?

These questions need to be answered for concrete co-simulation projects. However, in order to investigate co-simulation variability management in general, more generic questions have to be answered:

- What is commonly expected to be variable in various co-simulations? In other words, what are the possible types of variation points?
- How is it expected to vary? In other words, what kinds of variants are possible?
- What is the motivation for the support of the given variability?

The remainder of this section presents an answer to these questions. Items and properties of a co-simulation project are presented, which are expected to be variable - **variation points of a co-simulation**. Furthermore, possible variants that result from this variability will be discussed. Finally, examples for all different kinds of variation points are given.

To allow a better overview, variation points of co-simulation are separated into four groups:

- *Model-Related Variation Points*
- *Linking Variation Points*
- *Environment Variation Points*
- *Coupling Variation Points*

For the remainder of this chapter the presence of an *independent* co-simulation environment is assumed. Therefore, no simulation tool-specific properties or information can be used. In other words the models of the subsystems are treated as black boxes. The only available information are the models' interfaces (the names and data types of the input and output parameters).

### 3.3.1 Model-Related Variation Points

Model-related variation points define the variability of a subsystem model's implementation or the model itself. As these kinds of variation points handle the variability of subsystems (e.g. of a vehicle), they are probably the most obvious variation points in co-simulation.

For instance, let's assume two variants of an electric engine (engine A and B). Both are part of some distinct variants of a hybrid electric vehicle. At this point introducing variability enables one to simulate every variant of the hybrid electrical vehicle with the appropriate model of the electric engine.

This example shows one of the main motivations for model-realted variability, which is variability of the overall system (vehicle). Another motivation is the use of different modelling depths depending on the co-simulation scenario. For some co-simulation scenarios it might be sufficient to have a rather high level model of a subsystem, while others need a more detailed (but slower to simulate) model of the same subsystem.

How can this kind of variability be established in a co-simulation:

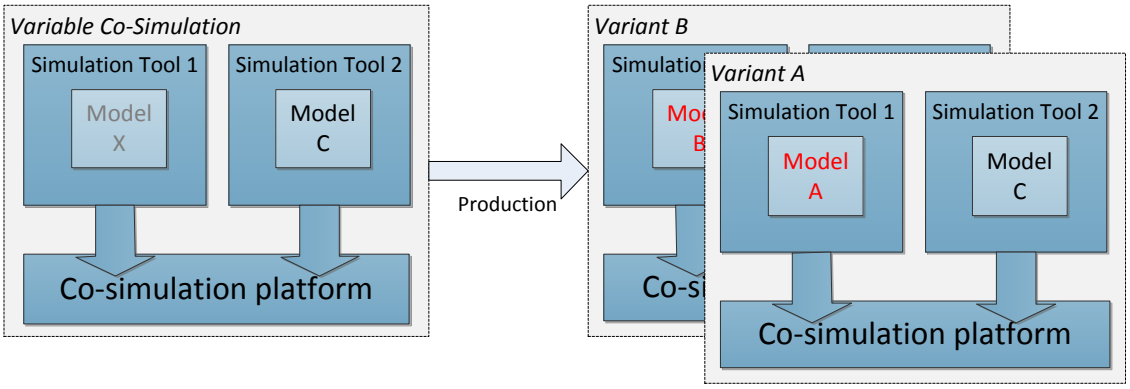


Figure 3.1: Model substitution in co-simulation environments; two models A and B of the same subsystem exist and can be used in different product variants of the co-simulation

- **Substitute models:** If both engines are represented in two distinct models, these models can be substituted. This means that the co-simulation project is configured to allow both models as an alternative. This scenario is depicted in Figure 3.1. Obviously, in order for two or more models to be substitutable, their interfaces need to be compatible, i.e. the number of input and output parameter and their data types must match.

- **Substitute models and simulation tools:** When two models of a subsystem need to be simulated in different simulation tools, the simulation tool has to be substitutable too. This scenario is depicted in Figure 3.2. Independent co-simulation abstracts the use of different co-simulation tools. Therefore, substituting models including their simulation tool is similar to substituting models in the same tool. Section 3.4 describes how this is done using the ICOS independent co-simulation environment.

- **Exploit model variability:** Models might provide some kind of variability. This variability can be used to change the behaviour of a model in the co-simulation. As stated before, the models are black boxes, i.e. only their input and output parameters are known. Thus, it is only possible to exploit the variability provided by a subsystem model through an input parameter. To give an example, Figure 3.3 shows a Simulink component model which contains a variant switch. The variant switch chooses a subcomponent for a calculation (add or multiply) according to the input of a configuration port. The input of the configuration port can be provided by an input parameter.

### 3.3.2 Linking Variation Points

In addition to model-realted variability, it might be desirable to change links between model input and output parameters. This is particularly true in the case of model substitution (see previous section). It was stated that two substitutable models must provide compatible

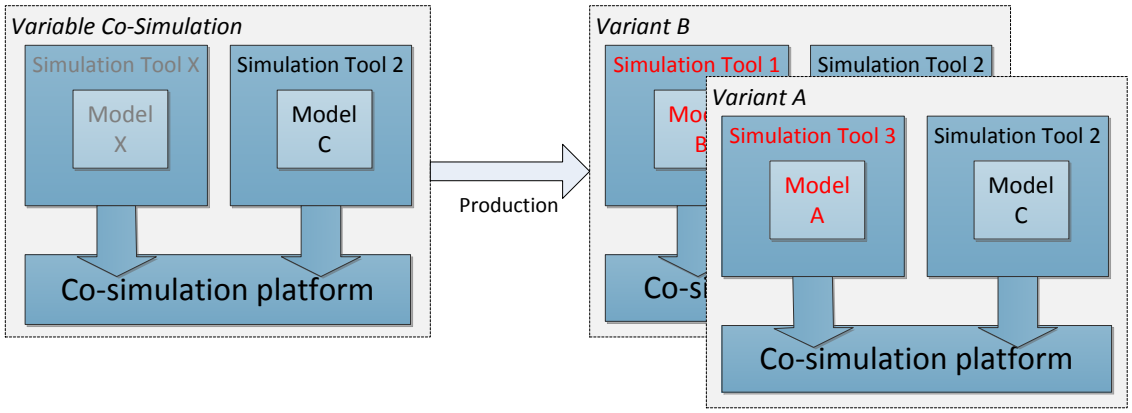


Figure 3.2: Model and simulation tools substitution in co-simulation environments; two models A and B that are modelled and simulated in different simulation tools exist; they can be used in different product variants of the co-simulation

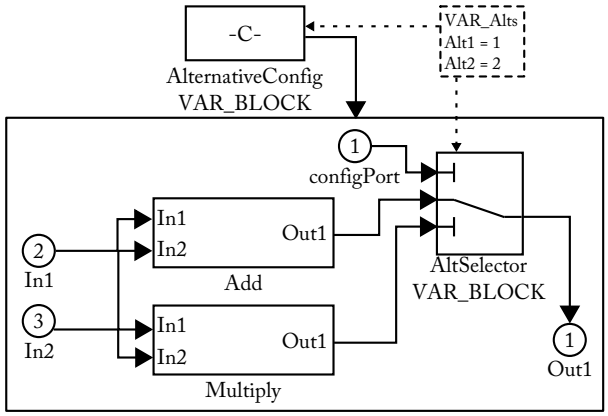


Figure 3.3: Modelling alternatives in Simulink with the Variant Block Set [Pav11]

interfaces (the same number of parameters and compatible data types). However, if a co-simulation's linking is variable, the different interfaces can be handled. Figure 3.4 shows an example of how changing the linking can solve the problem of incompatible interfaces.

Variable links in a co-simulation can make sense even if no models are substituted. Take for example a single-input single-output subsystem (SISO), which does some calculation or conversion of the input and provides the converted data as output. There might be product variants, where the conversion is not desired. In this case the linking can be changed to skip this conversion/calculation.

### 3.3.3 Environment Variation Points

Section 3.3.1 explained how models can exploit variability by providing input parameters that change the model. Instead, input parameters can be used for initialisation values or

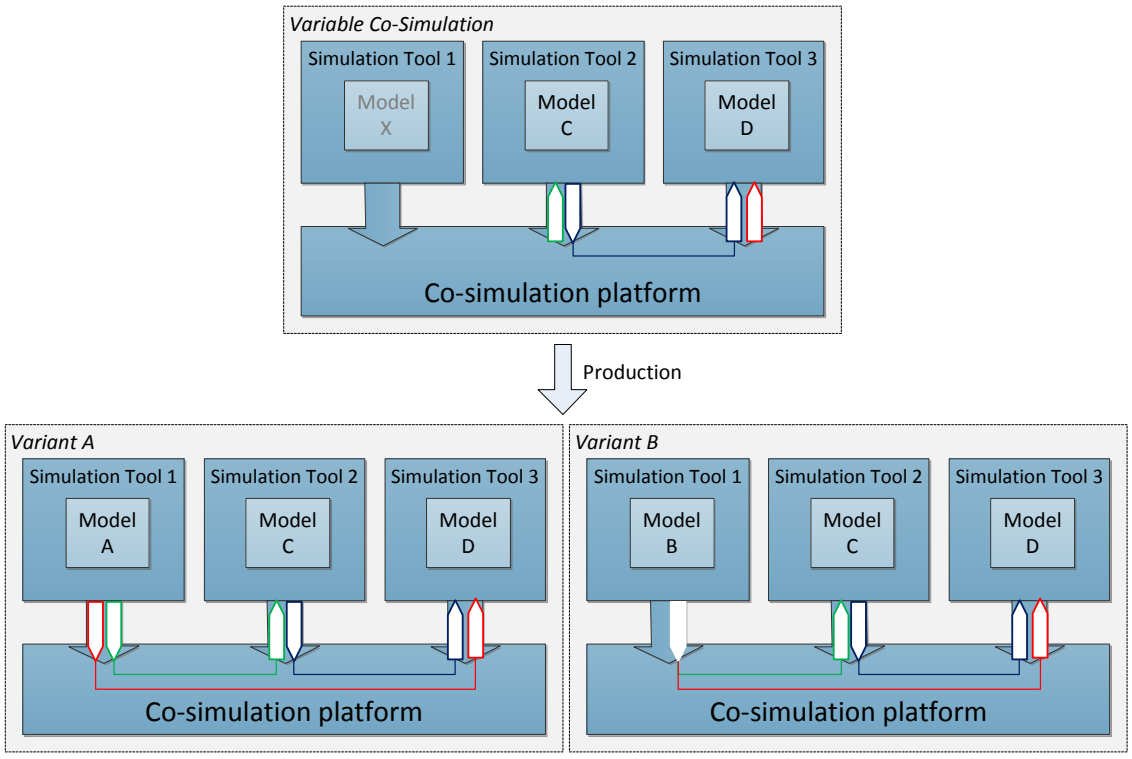


Figure 3.4: Variability in the co-simulation linking; links between model parameters can vary between several product variants and thus adapt to different model interfaces

boundary conditions. In this case the input does not change the model itself. An example that has already been used above is the ambient temperature which can vary for different co-simulation scenarios.

### 3.3.4 Coupling Variation Points

Section 2.2.3 described the process of coupling and properties of different coupling strategies. When introducing variability to co-simulation, it might be desirable to make settings, such as the macro time step or the coupling mode, variable. Two motivations for this are:

1. variability in coupling settings can be used to ***adapt the resulting co-simulation (the product).*** For instance, some simulation tools might require sequential simulation. Therefore, all resulting co-simulations that include models for these simulation tools, need to run sequentially.

2. variability of coupling settings can be used to ***optimise*** these settings or to ***find an appropriate value.*** This scenario is actually not associated with variability management as such, but with optimisation. In Section 2.2.3 it was stated that finding a step size that is a good trade-off between accuracy and simulation time is crucial for the success of co-simulation. If no appropriate step size is known by

experience, trial and error or optimisation can be done by varying the step size and evaluating the results.

### 3.3.5 Summary of Variation Point Groups

The four different groups of variation points (model-related, linking, environment and coupling) are classified by their consequences for the co-simulation environment. Thus, variation points from two different groups might be implemented in the same way, but their purpose for the co-simulation differs. Take for instance the variation points (1) changing the ambient temperature (environment VP) and (2) choosing an engine model's internal implementation (model-related VP). Both variation points are implemented using a model input parameter and connecting it to a boundary condition parameter. However, the latter VP (2) actually changes a model. Contrarily VP (1) provides a parameter from the system's environment to a model.

### 3.3.6 Separation of Concerns - Domain vs. Application Engineering

Software product line engineering distinguishes between two subprocesses (Section 2.1.7). The same separation can be used here: While domain engineering is responsible for establishing a variable co-simulation definition, application engineering is responsible for deriving concrete co-simulations (products) from the variable definition. Figure 3.5 shows the separation into domain and application engineering and its implications on co-simulation.

#### Domain Engineering

Domain engineering is the process responsible for the creation of a ***variable co-simulation environment.*** A variable co-simulation environment consists of a co-simulation project with explicitly defined variabilities and a set of subsystem models. In other words in domain engineering the core assets are developed, which together form a variable co-simulation. Therefore the core assets are:

- Static/invariable subsystem models that can be selected for resulting products.
- Variable subsystem models, which are bound during application engineering .
- A variable definition of the co-simulation consisting of links, coupling properties (coupling mode, step size. . . ), and boundary conditions.

#### Application Engineering

During application engineering the variability that was introduced before is bound and concrete products are derived. A concrete product in this context is a co-simulation project. The variability of the subsystem models that are included in the co-simulation project is bound.

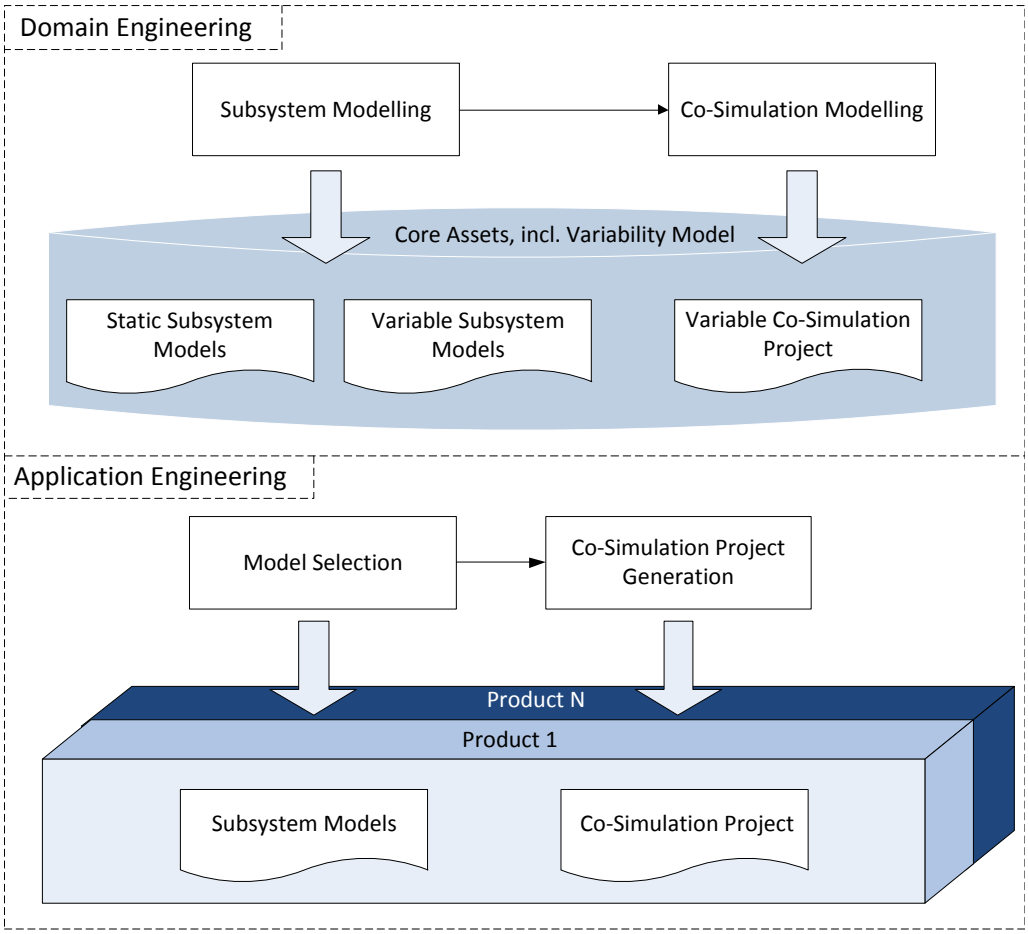


Figure 3.5: Domain and application engineering in co-simulation variability management

## 3.4 Requirements for Variability Management in ICOS

So far variability in co-simulation in general and how to manage the variability has been discussed. These observations are now applied to identify requirements for the support of variability in the ICOS co-simulation platform. Later on these requirements are used to design a variability management tool for ICOS.

### 3.4.1 Independent Co-Simulation

ICOS is an independent co-simulation tool. Hence, the co-simulation platform is independent from the simulation tools it uses. Therefore, variability management in ICOS is required to be independent too.

**Requirement 1** *(Independence): The variability model of an ICOS co-simulation has to be independent from the simulation tools that are used within the co-simulation.*

### 3.4.2 Decoupled Variability Model

The variability model describes the variability of a co-simulation. Obviously, this description depends on the co-simulation and its models. Contrarily, the co-simulation and its subsystem models are required to be decoupled from the variability model. Therefore, references from the co-simulation project to the variability model must not exist. Furthermore, the co-simulation and its models may exist on their own without the presence of the variability model.

**Requirement 2** *(Decoupled Variability Model): The co-simulation project and all its affiliated models are independent from the co-simulation variability model.*

### 3.4.3 Core Assets

It was stated that a co-simulation project is one of the core assets. In the case of ICOS variability management this co-simulation project is a complete, executable co-simulation. Therefore, an abstract model as a placeholder cannot be used, but a concrete model for the variable co-simulation project has to be specified. For the example of model substitution this means: instead of having an abstract variable model (Model X, Figure 3.1) which is a placeholder for concrete existing models, a particular concrete model (Model A, Figure 3.7) is substituted with another concrete model (Model B, Figure 3.7).

**Requirement 3** *(Executable Co-Simulation): The co-simulation project that is used in domain engineering has to be a complete, executable co-simulation project.*

### 3.4.4 Variation Points and Modifiers

Section 3.3 introduces four groups of variation points (model-related, linking, environment, coupling). The fact that some of these types are interrelated was ignored. This means that a variation point does not make sense without the existence of another one. Take, for instance, a variation point A that defines two models to be substitutable. The interfaces of the models are not compatible. Thus, they require a variation point B that changes the linking in an acceptable way.

One way to solve this is to define an abstract variation point and add constraints between variants of variation point VP 1 and VP 2 (Figure 3.6a). However, looking at this scenario from a higher level, substituting a model and changing its links can be seen as ***one single variation point***. None of these two variation points makes sense on its own.

To resolve this issue, another level of abstraction was introduced: modifiers. A single variant consists of one or more modifiers (Figure 3.6b). In the example above there would be one variation point, which consists of two variants Variant A and Variant B. Each of the variants consists of two modifiers; one for substituting the model and one for adapting the links. Figure 3.6 shows the realisation using an abstract VP and the require-relationship on the one side (a) and the introduction of modifiers on the other side (b).

Both approaches are possible in the proposed variability model. What are the advantages of modifiers and when is the use of modifiers preferable?

- Modifiers should be used if a product must not exist, in which these modifiers are separated from each other. Referring to the example in Figure 3.6b, one should use

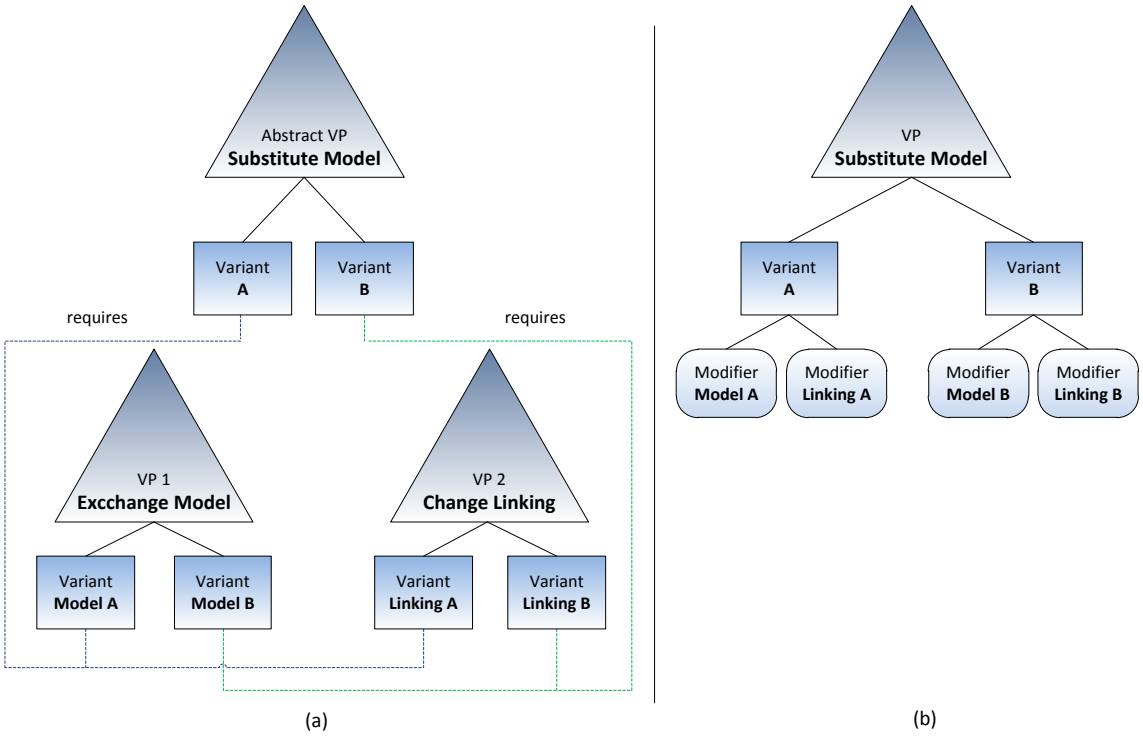


Figure 3.6: Solving the problem of interrelated variation points (a) with the introduction of modifiers (b)

modifiers if Modifier Model A cannot exist without Modifier Linking A in a product and vice versa.

- An obvious advantage of the use of modifiers is the creation of a simpler variability model.

- Looking at the variability model with modifiers (Figure 3.6b), a domain engineer immediately sees the dependency between Modifier Model A and Modifier Linking A. This is not the case for Figure 3.6a.

In the subsequent sections all modifiers that will be available in the ICOS variability management tool are described. Furthermore, their relationship to the variation point groups of Section 3.3 (model-related, linking, coupling and environment variation points) will be explored.

### Model Substitution Modifier

Models that are part of a variable co-simulation should be substitutable. Thus, a model can be replaced by another compatible model, as described in Section 3.3.1. The approach was separated into the substitution of models that use the same tool and those which use different tools for simulation. Due to implementation details, covered in Chapter 4,

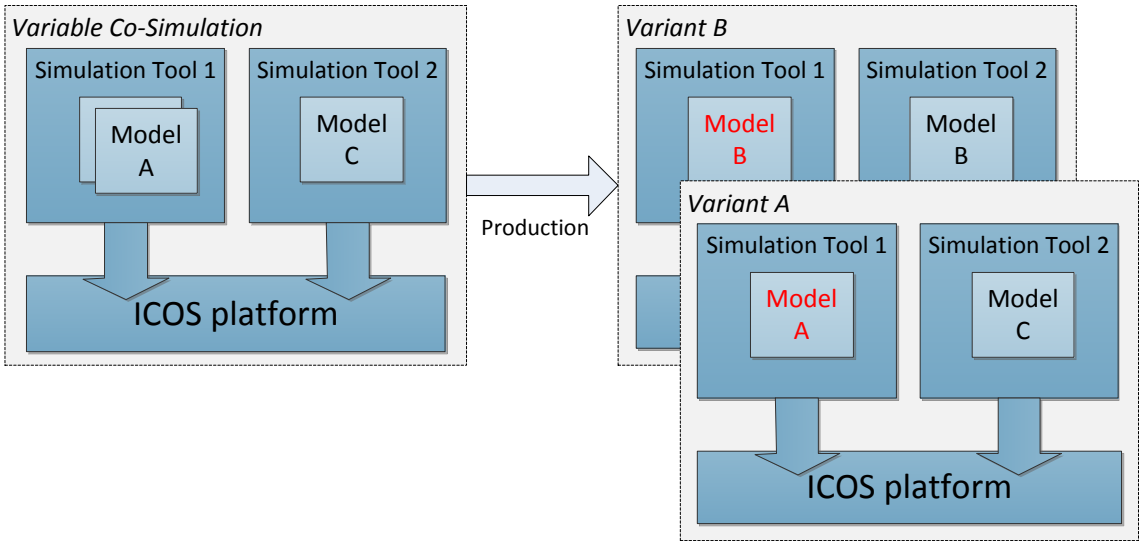


Figure 3.7: Model substitution in ICOS VM; Model A is substituted by Model B in one of the product variants

model substitution in ICOS VM is restricted to models that are simulated in the same tool. Exchanging models simulated using various tools is accomplished by model alternatives.

Figure 3.7 depicts model substitution of two models A and B that are simulated using the same simulation tool.

**Requirement 4** *(Model Substitution): A model that is part of the variable co-simulation project can be substituted by another model that is simulated in the same simulation tool.*

### Model Alternatives Modifier

As stated in the previous section, due to implementation details, the substitution of models from various simulation tools is accomplished by introducing model alternatives. Several models can be declared as alternatives and at binding time, one model is chosen. This scenario can be seen in Figure 3.8.

**Requirement 5** *(Model Alternatives): Several models that are part of the variable co-simulation project can be made substitutable by specifying model alternatives.*

### Linking Modifiers

The linking variation points described in Section 3.3.2 are transferable to our requirements for ICOS VM. One constraint has to be considered: an ICOS co-simulation project is only valid, if all input parameters are connected to a single output parameter. However, output parameters can be connected to several input parameters. Therefore, it has to be ensured that this constraint is fulfilled for every generated product.

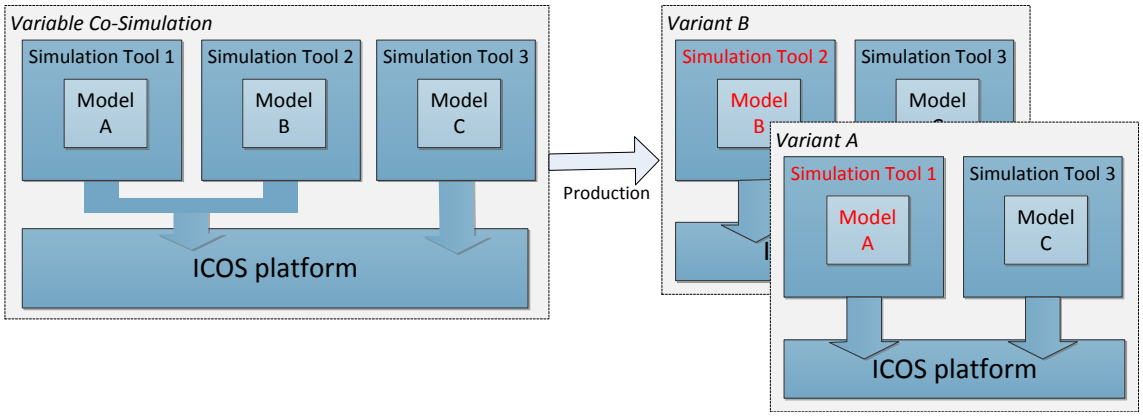


Figure 3.8: Model alternatives in ICOS VM; the alternative models Model A and Model B are both part of the core co-simulation environment; one of them is chosen for each product variant

**Requirement 6** *(Variable Linking): Links between input and output parameters can be variable. Their variability has to be bound in a way that in every product each input parameter is linked to exactly one output parameter.*

### Parameter Modifiers

Section 2.3.3 explained the use of two distinct names for a parameter. On the one hand, the name of the parameter in the simulation tool and on the other hand, the name of the parameter in ICOS. In the case of model substitution, it is required to adapt differing model internal parameter names to a single parameter name in ICOS. This is depicted in Figure 3.9.

**Requirement 7** *(Adopting Parameter Names): Internal model parameter names need to be variable in order to adapt to differences in substitutable models.*

### Boundary Condition Modifiers

The boundary condition server (BCS, Section 2.3.4) acts like any other model in an ICOS co-simulation. The only difference is that it is configured within the ICOS user interface.

The BCS offers boundary conditions (e.g. the ambient temperature) over output parameters. One way to make boundary conditions variable is to provide different values on several different output parameters and change the linking of those output parameters.

However, this way the different values of the BCS parameter are specified in the co-simulation project. A more convenient solution is the specification of different BCS values in the variability model. Therefore, boundary condition modifiers are introduced. With the help of boundary condition modifiers the value of a boundary condition can vary in several ways:

- *Single value:* The boundary condition parameter takes a single, constant value.

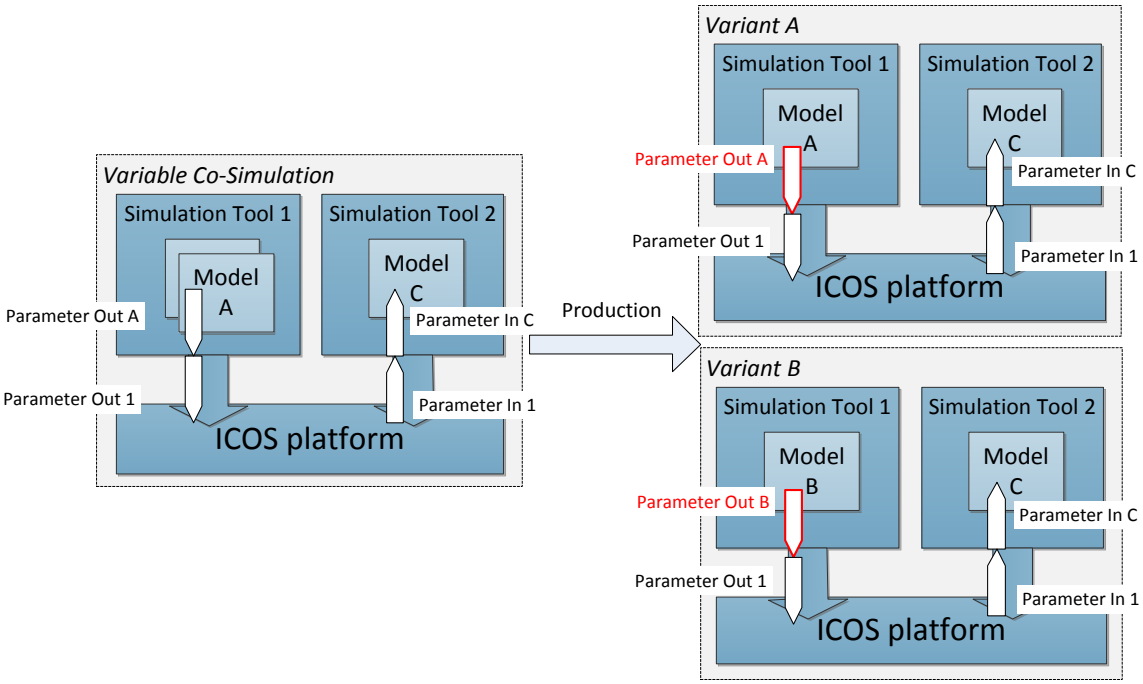


Figure 3.9: Adopting parameter names of substitutable models in ICOS VM

- *Multiple values:* The boundary condition parameter takes multiple, variable values.
- *Range of values:* This is a special case of multiple values, where a start and an end value as well as a step size is defined. E.g. from 1 to 3 (step 0.5) results in 1, 1.5, 2, 2.5 and 3.

**Requirement 8** *(Variable boundary conditions): Boundary conditions can be variable by changing their value to a single or multiple values without changing the parameter linking.*

The boundary condition modifier is connected to the model-related as well as the environment variation points.

### 3.4.5 Extensibility

There is a huge amount of possible parameters, properties or items that one might want to make variable. Any group of variation points might include some scenarios that are not covered by the requirements stated so far. Some examples:

- Different model initialisation files for different variants (model-related variation point)
- Variable step sizes (coupling variation points)
- Varying coupling mechanism (coupling variation points)

Therefore the set of available modifiers is required to be extensible.

**Requirement 9** *(Extensibility): The effort to implement new types of modifiers has to be relatively small and the skills needed should not require extensive training.*

### 3.4.6 Domain and Application Engineering

In accordance to the separation of concerns that were described in Section 3.3.6, the same separation for ICOS variability management is required. While *domain engineering* encompasses the development of core assets, *application engineering* concerns the development of particular products. The processes of domain and application engineering, as well as the models that result from these activities, are described in more detail in Chapter 4.

#### Domain Engineering

The main task of *domain engineering* is the development of core assets, in this case (1) the ICOS co-simulation project, including domain-specific subsystem models, and (2) the co-simulation variability model. The ICOS *co-simulation variability model* describes the variability of a co-simulation project. It contains the variation points, variants and modifiers of a variable co-simulation project.

#### Application Engineering

The application engineer configures a particular product by selecting a set of variants from the co-simulation variability model, called the application model. The *application model* describes products (product variants) that are derived from the core assets. Commonly application engineering refers to building one particular product. However, certain ICOS VM tasks, such as optimisation, require the production of several product variants at the same time. Therefore, the ICOS VM application model can consist of several configurations. Each configuration describes exactly one product variant.

### 3.4.7 Constraints

Dependencies between variants and variation points are an integral part of variability models as described in Section 2.1.5. The require- and exclude-constraints as well as optional and mandatory variants were described. Those constraints are a core feature of variability and therefore required for ICOS variability management. However, due to the introduction of modifiers in the previous section, there is no need for dependencies between variation points. Hence the introduction of dependencies between variants in the co-simulation variability model seems sufficient.

**Requirement 10** *(Constraints): A variant can require or exclude another variant. Moreover the selection of variants can be optional or mandatory.*

### 3.4.8 Generating Application Models

An application model consists of several product configurations. Sometimes it is desirable to have an application model that contains all valid configurations. A valid configuration is a selection of variants, that does not violate any constraint (require, exclude).

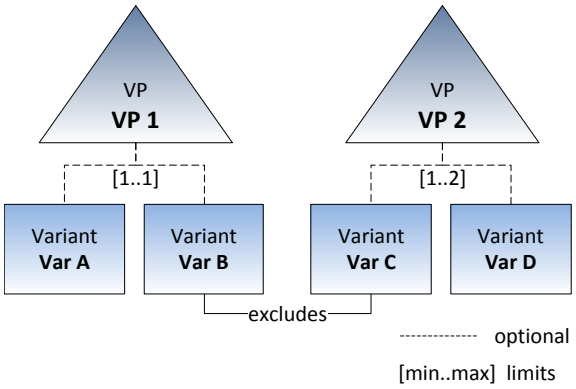


Figure 3.10: Example of a co-simulation variability model consisting of two variation points, four variants and several constraints between the variants; modifiers have been omitted for clarity

Take the rather simple variability model in Figure 3.10 as an example. Any combination of variants, containing Var B, as well as Var C is not valid, as this violates the exclusion constraint. Furthermore, according to the limit constraints, each valid configuration needs to select one variant from variation point VP 1, and either one or two variants from VP 2. Hence, all valid combinations of variants consist of the sets {Var A, Var C}, {Var A, Var D}, {Var A, Var C, Var D} and {Var B, Var D}.

**Requirement 11** *(Application Model Generation): Fully automated generation of an application model, consisting of all valid configurations of products, must be supported. Some means of restricting the generated configurations has to be provided.*

### 3.4.9 Automated Production / Variant Generation

The transformation of variable core assets to products (product variants) is called production. Production can be achieved fully automated, partially automated or completely manual. For the ICOS VM tool, the production is required to be fully automated. Thereby, the consistency of the resulting products has to be ensured. A consistent product is a co-simulation project with valid linking (every input parameter is connected).

**Requirement 12** *(Automated Production): The process of production is fully automated. The product variants that are derived from the core assets need to be consistent, executable co-simulation projects.*

## 3.5 Software Product Line Integration

So far a standalone tool implementation for variability management in ICOS co-simulation projects has been described. Even though this makes variability explicit and is a first step for systematic reuse, the integration of this variability management tool into an automotive (software) product line is desirable.

A possible scenario that emphasises this, is the presence of variants in automotive embedded software components, for example Simulink models. In this case each variant of a Simulink component can be simulated with the appropriate models of other subsystems in a co-simulation. Thus, co-simulation becomes an integral activity of product line engineering.

The SPL integration is described in more detail in Section 4.5 Additionally, the complete concept and description of the implementation is described in another document with the title ‘‘Integration of ICOS Co-Simulation Variability Management in Pure::Variants’’ [Toe12].

# Chapter 4

# Variability Management in ICOS

## 4.1 General Design

This chapter presents the design and implementation of the ICOS Variability Management Tool (ICOS VM). The design is derived from the general concept and requirements given in Chapter 3.

### 4.1.1 Independence

Requirement 2 states that the co-simulation has to be decoupled from the variability model. Thus, ICOS co-simulation projects cannot be extended to support variability. Additionally, ICOS is required to be independent from the ICOS VM tool. This means that the ICOS VM tool must be implemented as a standalone tool and the presence or absence of this tool does not affect ICOS' core features.

On the other hand, it was stated that the co-simulation variability model depends on the co-simulation. Therefore, ICOS VM depends on ICOS core functions. Thus, ICOS VM is able to use ICOS tools such as the ICOS GUI, batch mode or the remote server.

### 4.1.2 Standalone Tool vs. Product Line Integration

The goal of the implementation is to provide a way to define and manage variability in the ICOS co-simulation environment. As stated in the previous chapter two different application scenarios are identified:

1. Standalone variability management

2. VM integrated into a software product line

The remainder of this chapter mainly describes the standalone ICOS variability management tool. However, Section 4.5 briefly presents the integration into an SPL using the pure::variants variant management tool. How the implementation was tested is described in Appendix A.

## 4.2 The ICOS VM Tool Chain

The ICOS Variability Management Tool is a standalone tool which is integrated into the ICOS workflow. Figure 4.1 shows how a conventional ICOS co-simulation (without VM) is created and simulated and which tools are part of this workflow. First, subsystem models are created in modelling tools from various domains. Obviously, existing models can be reused instead of creating new models. These models are then connected in a co-simulation project using the ICOS graphical user interface. The resulting co-simulation is then simulated using ICOS and domain-specific simulation tools. As stated in Section 2.3, each domain-specific subsystem model is thereby simulated in a particular tool and ICOS handles the coupling between these tools.

Note that the terms ‘‘simulation tool’’ and ‘‘modelling tool’’ are used. These terms separate the process of model development from model simulation. Nevertheless, often the same tool is used to create and simulate a particular model.

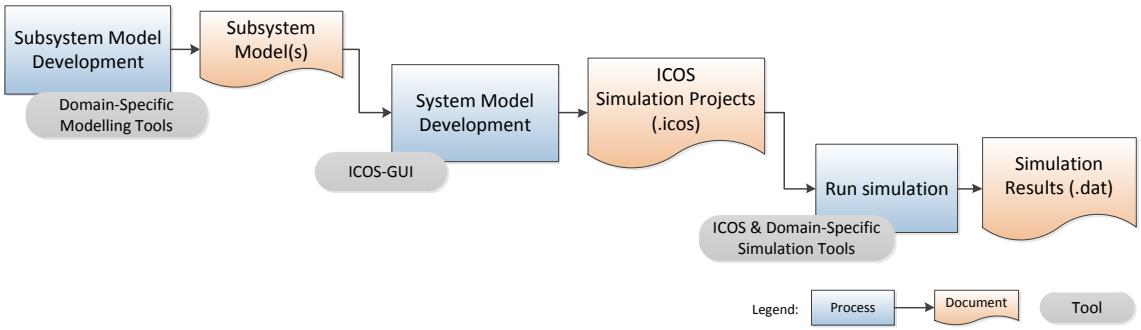


Figure 4.1: Co-simulation workflow in ICOS without variability management support

Figure 4.2 shows the ICOS variability management workflow. As in the conventional co-simulation process described above, the ICOS VM workflow starts with the development of domain-specific subsystem models and an ICOS co-simulation project. Subsequently, the domain engineer describes the variabiliy of the co-simulation project. This is done by defining variation points and variants in a co-simulation variability model. Developing the co-simulation (Figure 4.2, step 1) and defining the variability of the co-simulation (Figure 4.2, step 2) are both part of domain engineering.

The variability that was introduced during domain engineering, is bound during application engineering. Therefore, the user defines product variants in an application model file (Figure 4.2, step 3). Alternatively, the application model can be automatically generated (Section 4.4.4). The product variants are automatically generated (Figure 4.2, step 4), based on their description in the application model. These resulting product variants are ICOS co-simulation projects. Finally, each of these product variants can be simulated either (1) one by one using the ICOS GUI, or (2) all together using the ICOS batch mode (Section 4.2.2).

### 4.2.1 User Roles

Figure 4.2 describes the different tasks in the ICOS VM workflow. In order to complete

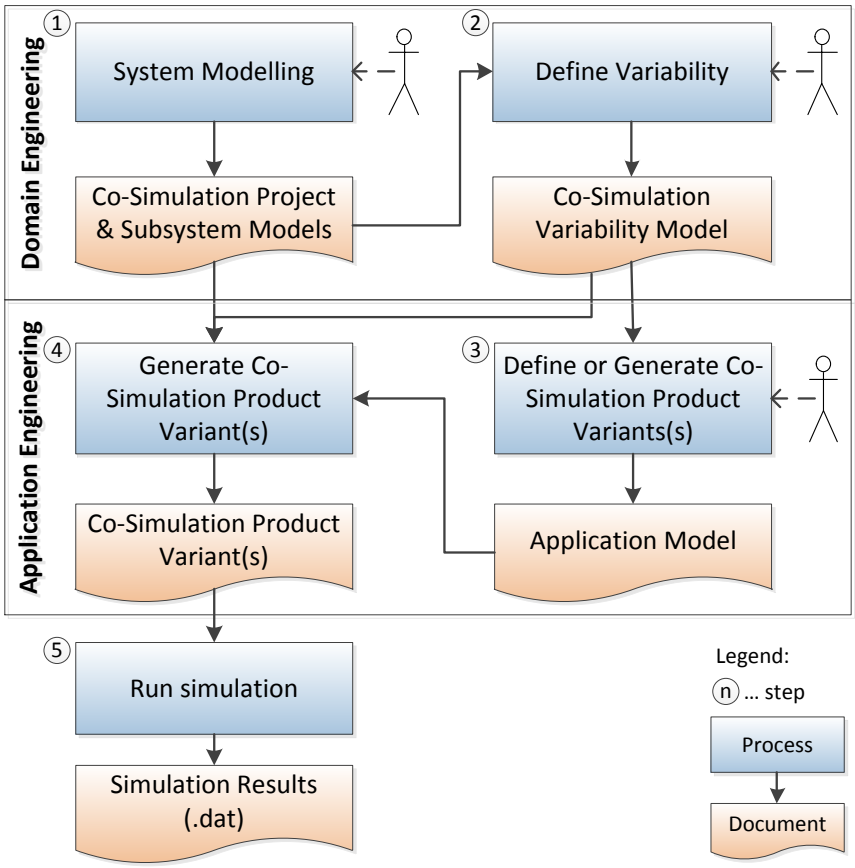


Figure 4.2: ICOS VM workflow, separated into domain and application engineering

each of these tasks, users may need to have different skills and knowledge. Therefore, roles are assigned to users that perform certain tasks. Obviously one and the same person can have more than one role in the workflow.

- A ***subsystem domain-expert*** is responsible for the development of a particular domain-specific model. He needs to have all the knowledge and skills necessary to create and maintain this model. For instance, this could be a mechanical engineer, who creates the model of the engine or power train.
- The ***domain modeller*** takes existing subsystem models (or delegates the task of model development to a subsystem domain-expert) and combines them in a co-simulation project. Further, he defines the variability of the co-simulation. Most often this is done by investigating the commonalities and variabilities of the resulting product variants.
- The ***application engineer*** is responsible for the configuration and generation of the resulting products. Furthermore, he executes the simulation and handles the evaluation of the products.
- A ***manager*** is a person that does not need to know about the internals of the system

to be simulated, neither is he aware of any internal variant. The manager's role might be more important in the SPL integrated ICOS VM, which is covered in Section 4.5. However, even when variability management is introduced to ICOS in absence of an SPL there might be a person in this role. Thus, the variability model should provide high level views for people with little or no knowledge of the system's internals.

### 4.2.2 Key Interfaces to ICOS

In the ICOS VM workflow several components or artefacts of the ICOS co-simulation environment are used:

- ICOS GUI (tool)
- ICOS co-simulation project file (artefact)
- ICOS batch mode (tool)
- ICOS batch file (artefact)

#### ICOS GUI

ICOS provides a graphical user interface (GUI) to create and edit ICOS co-simulation projects. In the ICOS VM workflow the GUI is only used to configure the core co-simulation project.

Figure 4.3 shows the GUI displaying a simple co-simulation project consisting of two KULI models. The user interface is separated into three parts:

- Model, parameter, BCS and co-simulation configuration
- Parameter linking
- Simulation

#### ICOS Co-Simulation Project File

An ICOS co-simulation project is saved in XML format. The details of the file format were discussed in Section 2.3.9.

The ICOS project file format plays a major role in the ICOS VM workflow. On the one hand, the core project file serves as a base for the variability model and is used as an input to the ICOS VM tool. Thereby the variability model references models, links or parameters from the ICOS project.

Additionally, the file format is used as an output format too. Each product that is generated is basically a new ICOS project file.

The investigation of the file format revealed some peculiarities. These peculiarities have to be taken into account for the ICOS VM tool implementation:

1. *XML Schema:* There is no public XML Schema, which describes the XML format in detail. Therefore the specification has to be reverse engineered. Hence, there might be parts of the XML file format that were not yet explored.

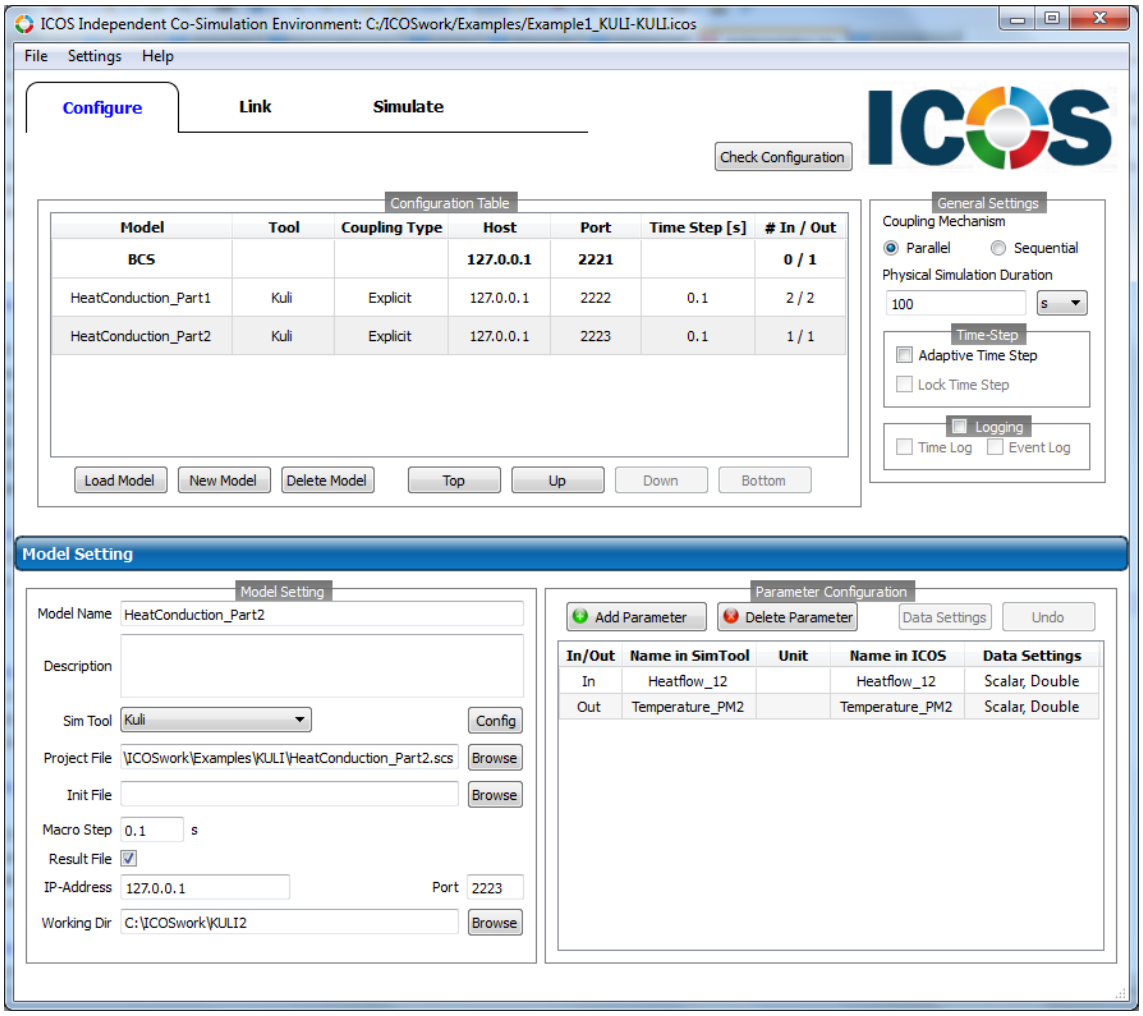


Figure 4.3: ICOS graphical user interface

2. *Frequent changes:* The file format is frequently changed when new features are implemented. Therefore, the ICOS VM tool needs to support changes in the file format to a certain extent.

3. *Ambiguous parameter names:* Parameter names in a co-simulation project are not unique. However, the parameter name and its type (input/output parameter) together serve as a compound key that uniquely identifies a parameter.

4. *Linking table:* As stated in Section 2.3.9, there is no separate table which stores information about the links between input and output parameter. Instead each input parameter references an output parameter by name. Therefore changing the links between parameters is not inconsequential.

The first and second aspect lead to yet another requirement for the implementation.

**Requirement 13** *(Co-Simulation Project File Format): Changes in the co-simulation project file format need to be supported by the ICOS VM tool to some extent. Beyond that extent, the effort to adapt the tool to these changes needs to be minimised.*

#### ICOS Batch Mode

The ICOS batch mode can be used to simulate several co-simulations sequentially. The ICOS batch mode is a console application which takes the path to an ICOS batch file as input. The batch file simply lists all co-simulation project files that ought to be simulated. Additional information, such as remote server addresses and passwords, are also part of the batch file. All co-simulation projects that are referenced in the batch file will be simulated sequentially and the output is written into a log file. A sample batch file can be found in Appendix B.4.

One major drawback of the ICOS batch mode application is the absence of feedback after the execution; i.e. the exit code of the application is the same, no matter if the simulation fails or succeeds. Therefore, it's hard to react to the simulation results after completion.

### 4.2.3 Integration in ICOS VM Tool Chain

The ICOS VM tool chain (Figure 4.2) shows that the ICOS VM tool has two different tasks:

- generation of an application model and
- generation of product variants.

In order to generate an application model, ICOS VM takes a co-simulation project and a co-simulation variability model as input. Product variants are produced from an ICOS co-simulation project, a co-simulation variability model, and an application model as input.

Note that the ICOS co-simulation project can be created using the ICOS GUI, while the other two models cannot. The implementation of a graphical user interface would be beyond the scope of this work. Therefore these models are represented in XML format. Thus, the domain and application modeller need to create these XML files manually. The implementation of a GUI to support this task can be seen as future work.

The ICOS VM tool is a console application implemented in Java. This platform was chosen due to several reasons:

- ***OS-independence***
- The integration into an SPL, described in Section 4.5, is done with a tool called ***pure::variants.*** This tool is an Eclipse plugin which provides extension points (interfaces) for other Eclipse plugins in Java.
- Java is an object oriented programming language, which is in ***widespread use*** and therefore the maintenance of the tool after the completion of this thesis can be ensured more easily.

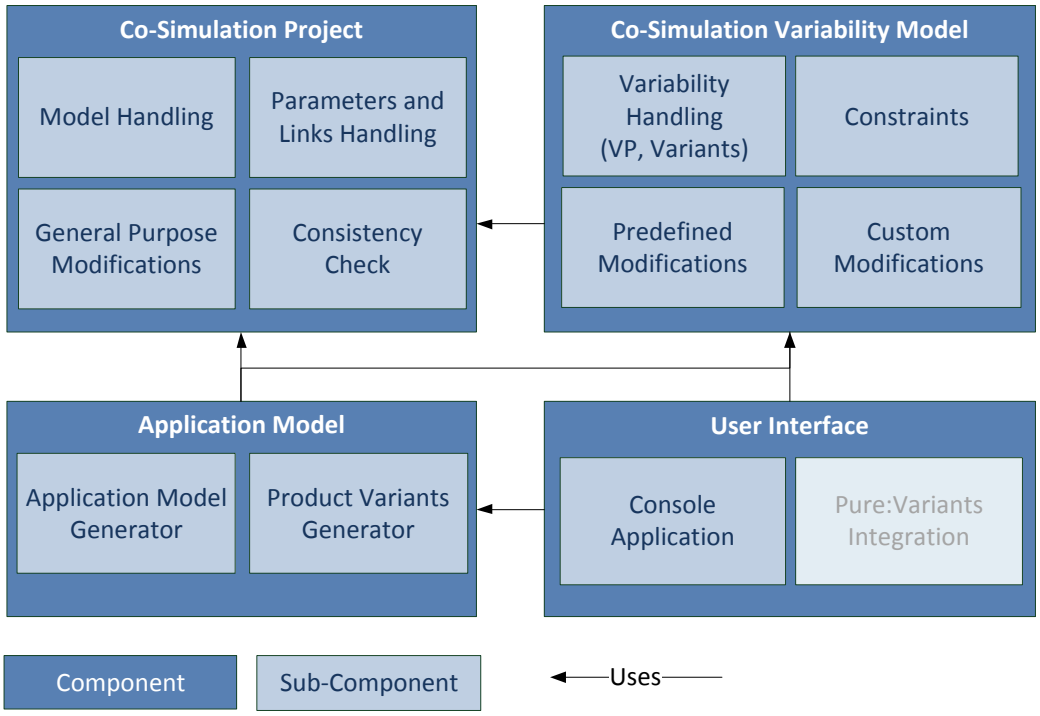


Figure 4.4: ICOS VM standalone - component model

## 4.3 ICOS VM Tool Architecture

### 4.3.1 Components

The ICOS VM tool is composed of four distinct components, which are depicted in Figure 4.4. Each component covers a certain aspect of the variability management tool. The components are loosely coupled and communicate via defined interfaces:

- The ***user interface*** component represents the console application and wrappers of this application which are visible to the user.
- The ***simulation project*** component handles reading, modifying and storing of co-simulation projects. Furthermore, it provides an object oriented representation of the co-simulation's subsystem models, parameters and links.
- The ***co-simulation variability model*** component covers all aspects that are related to the variability model. Variation points, variants, modifiers, and constraints are defined in this model.
- The ***application model*** component handles the generation of the application model itself and the generation of product variants.

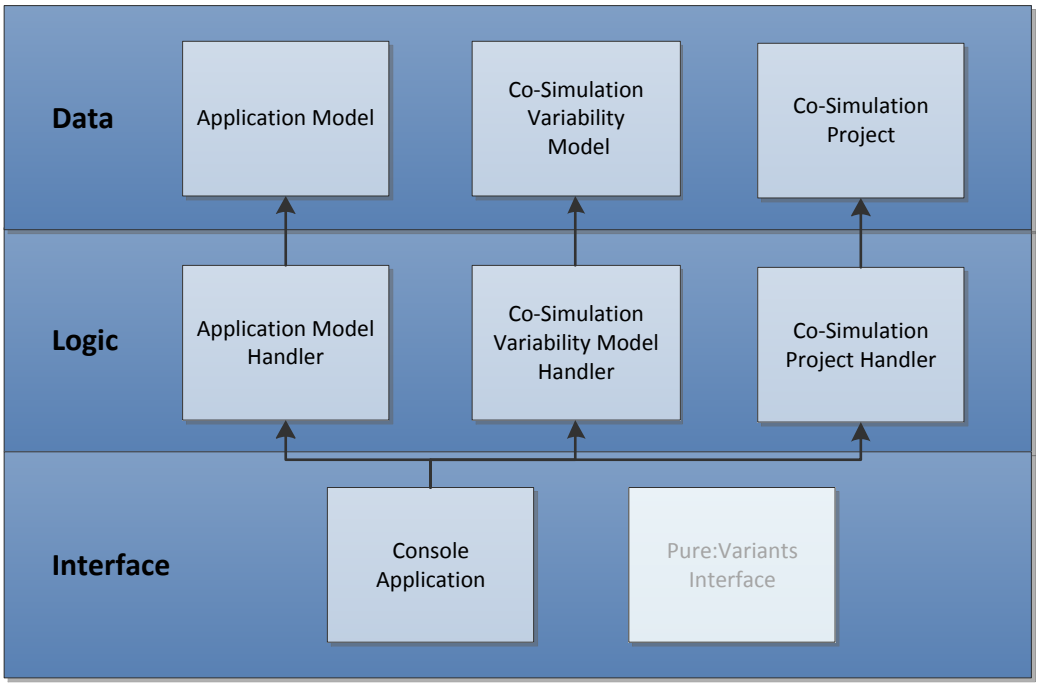


Figure 4.5: ICOS VM standalone - layered architecture

### 4.3.2 Layered Architecture

The separation of the tool into four components distinguishes the components by purpose. This means that every component is responsible for handling one part of the functionality that the tool provides.

However, another abstraction of the tools architecture results in three horizontal layers:

- The ***data layer*** (or persistence layer) handles the representation, reading and writing of co-simulation project, variability domain and application model files.
- The ***logic layer*** is responsible for the modification of each of the three models/projects. It must ensure that the models are consistent and that any modifications do not change their consistency.
- The ***interface layer*** provides the interface to the user and to other tools in the SPL.

### 4.3.3 Sequential Workflow

Before the four components are covered in detail, an example execution of the tool is given. As this example aims to give an overview of how the tool works, it is somewhat simplified. Thus, the focus is on the overall workflow, not on the details of steps taken during execution.

**Example 1: Generating an application model**

This example shows the workflow of the ICOS VM tool for the generation of an application model. It should be remembered that generating an application model means to produce product configurations for all valid combinations of variants. Variants are described in the co-simulation variability model, which is given to the ICOS VM tool as input. The co-simulation project (for which the variability is described) is not required for the generation of the application model. However, it is required as input in order to check the validity of the co-simulation variability model.

Taking the co-simulation project and the variability model as input, the following steps need to be executed:

1. Read the ICOS co-simulation project (.icos file).
2. Check the validity of the ICOS co-simulation project.
3. Read the co-simulation variability model file (.icosdm file).
4. Check the validity of the co-simulation variability model.
5. Create an empty instance of the application model.
6. Generate all valid combinations of variants from the co-simulation variability model.
7. Save the generated combinations to the new instance of the application model.

Step 6 states that all *valid* combinations are generated. The term ''valid'' is used, because some possible combinations are not valid because of a constraint (require, exclude).

**Example 2: Generating product variants**

This example shows the workflow of the ICOS VM tool to generate all product variants that are specified in an application model. It should be remembered that generating product variants means to produce co-simulation projects which differ from each other in the way described in the co-simulation variability model.

Taking a co-simulation project, a variability model, and an application model as input, the following steps need to be executed:

1. Read the ICOS co-simulation project (.icos file).
2. Check the validity of the ICOS co-simulation project.
3. Read the co-simulation variability model file (.icosdm file).
4. Check the validity of the co-simulation variability model.
5. Read the application model file (.icosam file).
6. Check the validity of the application model.
7. Take a product configuration A from the application model.
8. Check the validity of configuration A according to the co-simulation variability model (constraints, references).
9. Take all modifiers M from all the variants part of configuration A.

10. Apply the modifiers M to a fake co-simulation project in order to check the validity of configuration A.
11. Create a clone C of the input ICOS co-simulation project.
12. Apply the modifiers to co-simulation project C.
13. Save co-simulation project C.
14. Repeat step 7-13 for all configurations from the app model.

Both step 10 and step 12 apply modifiers to a simulation project. Why this is necessary and how the design of the implementation supports this is covered in more detail in Section 4.4.2.

## 4.4 Main Components

Figure 4.4 showed the separation of the ICOS VM tool into four components. In this section these components are described in detail.

### 4.4.1 User Interface

#### ICOS VM Console Interface

The implementation of a graphical user interface is out of scope for this prototypical implementation. Therefore a console application serves as the main user interface. As stated before, the ICOS VM tool is responsible for

- the generation of an application model and
- the generation of the co-simulation variants (product variants).

Different command line arguments let the user choose to either generate the application model or provide an application model that is used for product variant generation (production). The console interface is a JAVA application that can be executed in the command line.

#### The Wrapper Batch File

The ICOS VM tool chain ranges from the development of subsystem models over several other activities to the simulation of the resulting product variants. The ICOS VM Console (JAVA) Application only handles the generation of an application model and the resulting co-simulation product variants. In addition, the wrapper batch file combines these activities with the actual execution of the simulations. In other words, the wrapper batch file

- calls the ICOS VM JAVA console application in order to generate co-simulation project variants (.icos files) and then
- simulates these output co-simulation projects using the ICOS batch mode.

In order to be able to open the ICOS batch mode, the wrapper batch file needs to know the file location of the ICOS executable and the licence file of the ICOS user. These locations can either be provided as command line arguments or set as environment variables.

#### Installer

Several steps have to be performed to setup the environment of the ICOS VM tool. To support the user, ICOS VM provides an installer for Microsoft Windows Systems that performs the following tasks:

- Create environment variables to point ICOS VM to the ICOS executable and licence file.
- Install the custom modification description files (see Section 4.4.3).
- Install the console application and the wrapper batch file properly.
- Check if the JAVA runtime environment is installed.

### 4.4.2 Co-Simulation Project

The co-simulation project component (sometimes referred to as simulation project component) is the component responsible for loading, verifying, modifying and saving ICOS co-simulation project files. Loading and saving the project file is rather trivial. However, modifying a project and at the same time perceiving the consistency and completeness of the project file is more challenging. First, it is described how the project file is loaded and stored. Further, the modifications the interface to the co-simulation project offers are presented. Finally, it is described how to ensure that there is consistency throughout the project.

#### Persistence

The details of the ICOS file format are described in Section 2.3.9. Additionally, Section 4.2.2 describes some particularities of the file format.

As the file format is XML, the use of an existing marshalling software between XML and JAVA objects would have been desirable. However, most of the marshalling frameworks, such as JAXB (Java Architecture for XML Binding) require the XML file to be described by a schema. Furthermore, frequent changes in the XML schema would create the need to frequently regenerate the JAVA classes.

As stated in Requirement 13, it has to be possible to support frequent changes of the ICOS project file format. At best, these changes are supported without the need for rebuilding the ICOS VM tool. Hence, the use of XML to object marshalling software is not an option for the co-simulation project component.

Therefore, the standard JAVA XML DOM[1] implementation is used to load, save and modify the co-simulation project. The modification is done by retrieving parts of the XML using XPATH[2] expressions and changing the values of the retrieved object.

#### Modifying a co-simulation project

The co-simulation project component needs to provide an interface for modification. Other components, such as the co-simulation variability model component, use this

[1] DOM: Domain Object Model
[2] XML Path Language

interface to modify the simulation project in a desired way. The interface is called *ISimulationProjectModify* and offers methods for

- deleting subsystem models,
- changing subsystem model properties (e.g. the model source file),
- changing parameter mappings/connections,
- changing parameter properties (e.g. their simulation tool-internal name), and
- applying any kind of modification using an XPATH expression (general purpose modifications, see Section 4.4.3).

***Model deletion*** in a co-simulation project is implemented using a two-way deletion procedure: a model can be registered for deletion, but the model is not deleted before either the method *applyDeletions* is called, or the project is saved to memory. This implementation makes it possible to modify models that are already deleted without failing. One could argue, that it does not make sense to modify deleted models. In any case the outcome of a series of modifications has to be the same, no matter which order they are applied in. For instance if Model A is changed and later the same model is deleted, Model A would be modified and later deleted. However, if these modifications are applied in reverse order, changing Model A would fail, because Model A does not exist anymore. Delaying the actual deletion can solve this problem.

Section 4.4.3 shows why this behaviour is desired by the co-simulation variability model component.

The co-simulation project component offers another interface called *ISimulationProjectReferenceCheck*. This interface offers methods to check the existence of models, parameters, and properties of the simulation project. As the co-simulation variability model references models and parameters from the simulation project, this interface provides a way to check these references and query properties of models and parameters.

Figure 4.6 shows a class diagram including the most important classes of the co-simulation project component. The two interfaces at the top of the diagram have already been described. The *XMLProject* class implements XML related functionality that is used by the *SimulationProject* class. The *SimulationProjectHandler* class itself implements the two interfaces previously presented. It actually applies the modifications to the simulation project. The other classes will be presented in the remainder of this section.

### Simulation Project Consistency Check

As stated before, *ISimulationProjectModify* provides an interface that enables to modify co-simulation projects. Among other things the interface offers methods of deleting models as well as changing parameter connections. When a model is deleted all its parameters are deleted as well.

Requirement 12 stated that the production of consistent co-simulation projects has to be supported. An ICOS co-simulation project is only consistent, if all input parameters are connected to a single output parameter. Therefore, the project consistency needs to be checked when modifying the co-simulation project.

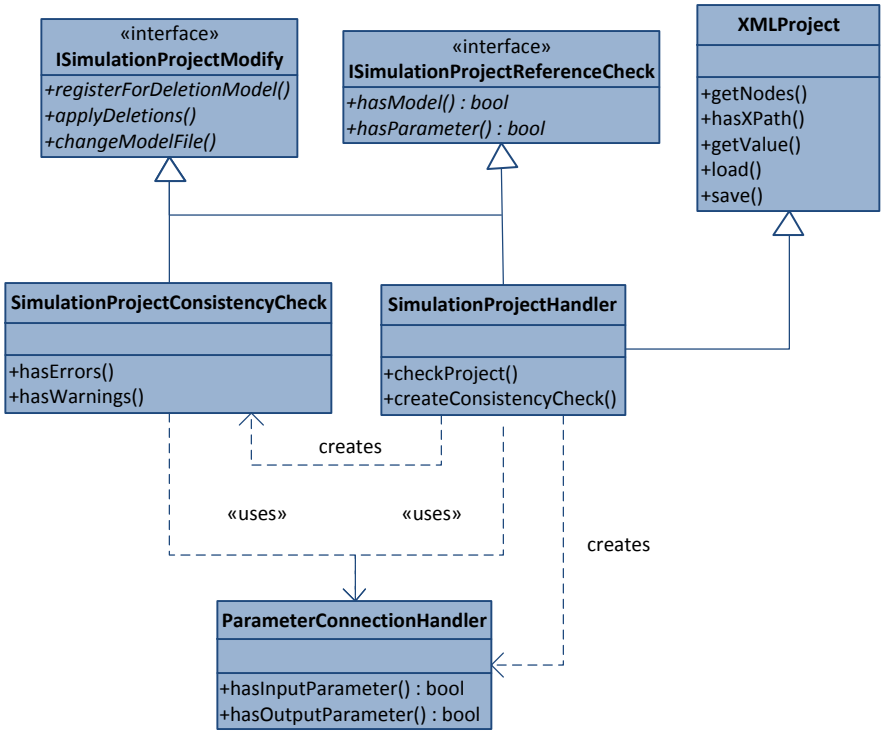


Figure 4.6: Class diagram of the most important classes of the co-simulation project component

This is done by the *SimulationProjectConsistencyCheck* class. It implements the *ISimulationProjectModify* interface, but it does not actually modify the simulation project. Instead it records the applied modifications. The recorded modifications can then be checked and an error message is issued if these modifications result in an inconsistent co-simulation project.

The only real inconsistency occurs if input parameters are not connected. However, warnings are issued if applied modifications, do not have any effects on the simulation project. For instance, if properties of Model A are modified and the same Model A is deleted, the modification of Model A does obviously not have any effects. This is not an error, as the co-simulation project is still consistent. Nevertheless, a warning message is issued for the user to recognise the effectless modification.

### Parameter Connection Handler

The *ParameterConnectionHandler* class is responsible for handling the connections between input and output parameters. It is initialised with the connections that exist in the simulation project file. The *SimulationProjectHandler* class uses it to apply modifications to the parameter connections. Similarly, the consistency check uses the *ParameterConnectionHandler* to check if deleting a model parameter leaves the co-simulation project in an inconsistent state.

### 4.4.3 Co-Simulation Variability Model Component

Section 3.3.6 describes the concept of the co-simulation variability model. The variability model stores information about the variability of a co-simulation project. The co-simulation variability model component encompasses all classes and interfaces that are responsible for

- *loading* the co-simulation variability model from and *saving* it to memory,
- *checking* the model for validity,
- *applying modifiers* to the co-simulation project, and
- *handling constraints* (require, exclude...) specified in the co-simulation variability model.

#### Persistence

Co-simulation variability model files are represented in XML format. The format is specified by a strict XSD (XML Schema Definition) schema. This XSD schema has also been used to generate Java classes that correspond to the XML-element and attribute types specified in the schema. The generation of Java classes from the XSD schema and the marshalling and unmarshalling between JAVA object and XML is done using JAXB (JAVA Xml Binding [Ort12]).

An example co-simulation variability model file can be found in Appendix B.1. The example XML shows the tree structure of the co-simulation variability model. A co-simulation variability model contains variation points, which in turn consist of variants, which consist of modifiers.

Figure 4.7 shows some of the most important classes of the co-simulation variability model component. The *VariabilityModelHandler* class is initialised with the location of the variability model file. It loads the file (unmarshalls the XML file) creates the co-simulation variability model, consisting of variation points, variants and modifiers. It further initiates the validation of the co-simulation variability model.

#### Checking the Co-Simulation Variability Model

When a variability model was loaded from memory, its validity has to be checked. This requires several steps:

1. Checking the XML validity (XML schema check)
2. Checking modifier properties
3. Checking references to the simulation project
4. Checking the constraints for validity (e.g. ‘‘Variant A excludes Variant A’’ is not a valid constraint)

While step 1 can be done using Java built-in functionality, step 2 to 4 will be described in the subsequent sections.

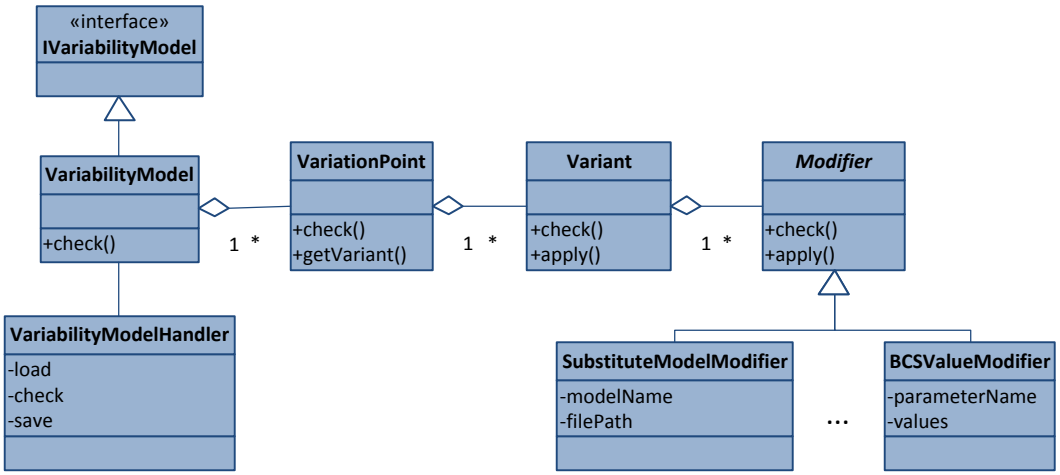


Figure 4.7: Class diagram of the most important classes of the co-simulation variability model component

### Variants and Variation Points

As can be seen in Figure 4.7 a co-simulation variability model is composed of variation points which are composed of variants. Variation points and variants are important elements used to structure the variability of a co-simulation project. For instance, a variation point ‘‘electrical engine’’ with its variants ‘‘simple engine model’’ and ‘‘complex engine model 2’’ express the meaning of this variation point for the simulation project. The lower level modifiers then describe in detail how these variants are applied.

Apart from the purpose of structuring the model, variants and variation points have additional uses. Constraints can be set at variant level (variant A excludes/requires variant B) or variation point level (at least 1, at most 2 variants of variation point A have to be selected).

### Modifiers

Modifiers are elements at the lowest level in the variability model. A modifier describes exactly one modification to the simulation project.

For instance: A variant ‘‘substitute model A’’ might be composed of the following modifiers:

- Substitute the model file of model A,
- change the simulation tool-internal parameter names, and
- change a boundary condition value, because the new model expects a different range of values.

All of these modifiers belong together in the sense that neither of them on their own produces a meaningful, or even valid, co-simulation.

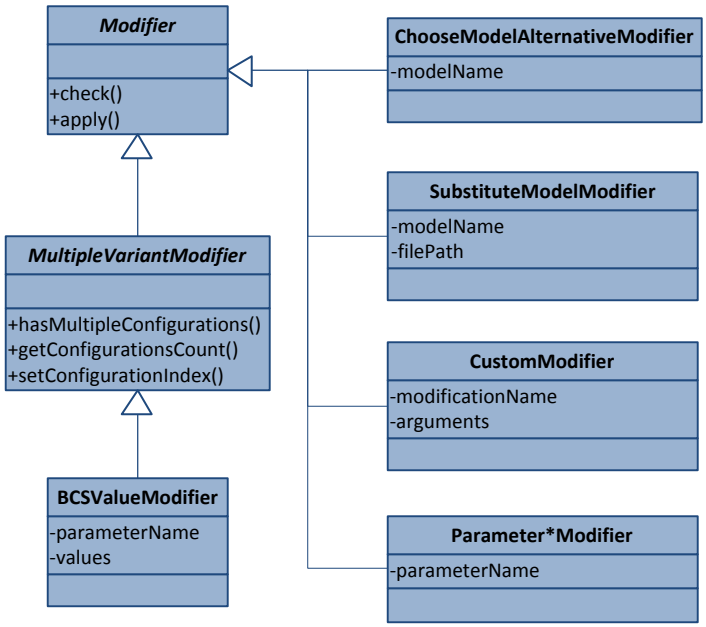


Figure 4.8: Class diagram of modifiers in the co-simulation variability model

Each modifier that is described in a co-simulation variability model file is implemented as a subclass of the abstract *Modifier* class. All modifiers can be seen in Figure 4.8.

### Checking Modifiers

Each modifier implements a *check*-method, which is called when the co-simulation variability model is loaded. The *check*-method is responsible for

- checking the configuration of the modifier (e.g. required attributes) and
- validating *references to simulation project* elements, such as models or parameters.

The method takes an implementation of the interface *ISimulationProjectReferenceCheck* as parameter. As described in the previous section, this interface provides methods for querying the simulation project for subsystem models and parameters.

Take for example the *SubstituteModelModifier*. This modifier has two attributes: the name of the subsystem model to be substituted and the path to the new model file. Using the *ISimulationProjectReferenceCheck* interface, the modifier can check the given model name's existence. Further, it checks that the given file path is a valid path string.

### Applying Modifiers

The main purpose of a modifier is to modify the simulation project. The way in which the simulation project is modified depends on the type of modifier. Each modifier implements a method *apply*, which takes an implementation of the *ISimulationProjectModify* interface as parameter. This interface provides ways to modify a simulation project. The

modifier itself does not need to be concerned about any simulation project-specific details.

It was stated above that the consistency of a simulation project must be ensured. This is done with the help of the *SimulationProjectConsistencyCheck* class. This class implements the *ISimulationProjectModify* interface. Thus, a modifier can be applied to the actual simulation project which performs the modifications. On the other hand, it can be applied to the consistency check class which does not modify the simulation project, but verifies that a series of modifiers do not produce an inconsistent simulation project.

Take for instance a modifier that deletes a model. The modifier is first applied to the consistency check. If the deletion does not jeopardise simulation project consistency, the modifier is applied to the project itself. This step actually deletes the model from the simulation project.

#### Predefined Modifiers

According to the requirement analysis presented in Chapter 3, several modifiers are implemented that can be used in the co-simulation variability model.

- ***Substitute Model Modifier:***
  Substitute the file that stores the domain-specific model of a subsystem.
- ***Choose Model Alternative Modifier:***
  Choose a model (e.g. Model A) out of a defined set of model alternatives (e.g. Model A, Model B and Model C). This is done by deleting all subsystem models from the simulation project that are not chosen (Model B and Model C).
- ***BCS Value Modifier:***
  Change the output value of a boundary condition parameter. The modifier allows multiple values in form of a list of values (e.g. 1, 3, 4, 6, 9) as well as value ranges (e.g. from 1 to 15 in steps of size 0.6).
- ***Input Parameter Mapping Modifiers:***
  Changes the connection of an input parameter to another output parameter. For instance, remapping an input paramter InA to an output parameter OutB, means that InA is now connected to OutB.
- ***Output Parameter Mapping Modifiers:***
  Changes all connections from an output parameter to another output parameter. For instance, remapping an output paramter OutA to another output parameter OutB, means that all input parameters that were connected to OutA, are now connected to OutB.
- ***Parameter Name In Simulation Tool Modifiers:***
  Changes the simulation tool-internal name of a parameter.

#### Custom Modifiers

Requirement 9 states that the effort used to implement new modifiers has to be minimised. This is done by the introduction of custom modifiers. From an abstract point of view a custom modifier is just like all the predefined modifiers described above. The difference

is their implementation. While all the predefined modifiers are implemented in JAVA, custom modifications are described in an XML file called *custom modification description file*. An example for such a file can be found in Appendix B.2. It is suggested to investigate this example before continuing with this section.

The benefit of this design is the quick implementation of simple modifiers without writing or changing any source code. Another benefit is the fact that the introduction of new custom modifiers does not require recompilation and redistribution of the ICOS VM tool. In fact new custom modification description files can be added at runtime by copying those files to a predefined directory.

A custom modification description file is an XML file, which contains the description of several custom modifications. The files have to be stored in a particular directory (the custom modification directory) or their path can be supplied as command line parameters. Each custom modification description consists of

- a **unique name** identifying the modification (the user, who wants to apply a custom modification references this unique name),
- **arguments** provided by the user,
- a list of **modified models and parameters**,
- **boolean expressions** or rules that need to be checked before the modification is applied, and
- the actual **modifications** as XML/XPATH selectors.

A custom modification description can be compared to a function in general programming. It defines a name and arguments (of a particular data type). It further specifies what has to be done with the given arguments.

In analogy using a custom modifier in the variability model can be compared with a function call in programming. The function (custom modification) is referenced by its name and values for defined arguments are supplied.

#### Arguments

Arguments are data items that can be supplied by the user, who wants to apply a custom modification. Each argument that is defined in the custom modification description has a name and a data type. The user references the modification by its unique name and provides a key-value list of arguments. Listing 4.1 shows an example application called ChangeIniFile. The user provides two arguments. The XML shown in the listing is part of a co-simulation variability model file.

The argument values provided by the user can be referenced at various places within a custom modification description. In other words, the custom modification description is a template with placeholders for argument values.

```
<Modification Type="ChangeIniFile">
    <Argument Name="ModelName" Value="Model1" />
    <Argument Name="FilePath" Value="C:\Temp\MyIniFile1.exe" />
</Modification>
```

Listing 4.1: Example custom modification application in a co-simulation variability model file

**List of Modified Parameters and Models**

The creator of a custom modification description has to specify which subsystem models and parameters will be modified. This information is required to check if the custom modifier might jeopardise the consistency of a simulation project.

Obviously, the names of the models to be modified need not to be hard coded, but can reference argument values.

**Boolean Expressions / Rules**

Before a custom modification is applied to a simulation project, the creator might want to evaluate some conditions. The schema for custom modification descriptions provides an extended set of boolean expressions that can be combined. The boolean expression is evaluated before the modification is applied. In case the expression evaluates to false, the modification can either

- fail,
- warn the user, or
- skip the modification.

Obviously, argument values can be used for the creation of such expressions. Listing 4.2 shows an example of a boolean expression. The expression lets the modification fail in case the argument *maxTimeStep* is not greater than the argument *minTimeStep* or if any of the two values is strictly greater than zero.

```
<Expression
    Action="Fail"
    Message="The maximum time step ({maxTimeStep}) can not be
       less than the minimum time step ({minTimeStep}). Both
       values have to be strictly greater than zero.">
     <And>
         <!-- leq = less or equal -->
         <Leq>
             <ArgumentValue ArgumentName="minTimeStep" />
             <ArgumentValue ArgumentName="maxTimeStep" />
         </Leq>
         <!-- gt = greater than -->
         <Gt>
             <ArgumentValue ArgumentName="minTimeStep" />
             <Value Value="0" DataType="double" />
         </Gt>
     </And>
</Expression>
```

Listing 4.2: Example custom modification boolean expression

The XML expression is parsed, the argument placeholders are replaced with the according values and the expression is evaluated. Parsing the expression results in a tree structure. All the available types of nodes in the tree are depicted in Figure 4.9. The class

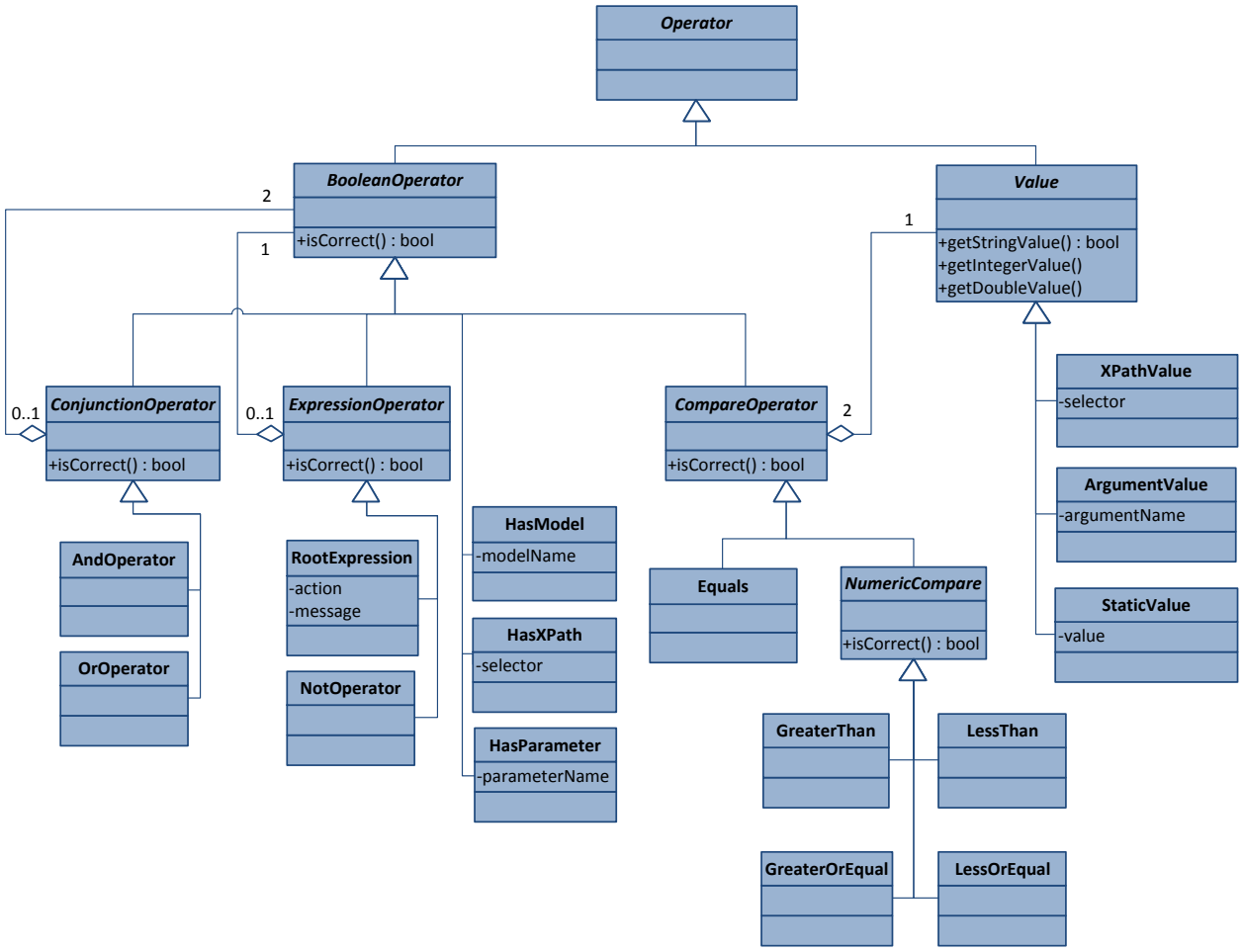


Figure 4.9: Class diagram of a custom modification expression validation

*RootExpression* obviously is the root of the expression tree and corresponds to the root element in the XML expression. Every boolean operator implements the method *isCorrect*. Calling this method for the *RootExpression* will bubble down the request to the leaves of the tree and returns whether the expression is correct or not.

For instance the *isCorrect* method of the *AndOperator* calls the *isCorrect* method of its two child operators. If both child operators return true, the *AndOperator* will return true, otherwise it will return false.

### XML/XPath Modifications

So far it has been described how the custom modification is identified (unique name), how arguments are provided and how these arguments or other conditions are validated. After those steps an actual modification of the simulation project is performed. This is done with the help of an XPath (XML Path Language) expression. An XPath expression addresses parts of an XML document [JC99].

In this case an XPath expression can address one or more elements or attributes of the co-simulation project XML file. The addressed element or attribute will then be set to a

given value. Listing 4.3 shows an example of two modifications. The first modification selects *a model* with a name that is given by a user argument and sets the attribute *timeStamp* to a given value. The second modification selects *all models* and sets their working directory to a given value.

```
<!-- note that {argumentName} are placeholders for argument
    values -->
<!-- the @-sign selects an attribute -->
<Modification
            Selector="//wrapper[@name='{modelName}']/@timeStep"
            Value="{timeStep}"
            CreateIfNotExists="true" />

<Modification
    Selector="//wrapper/workDir"
    Value="{newWorkDir}"
    CreateIfNotExists="false" />
```

Listing 4.3: Example custom modification xpath expression

It can be concluded that custom modifications offer a flexible way to extend the core ICOS VM functionality. New modifiers can be implemented with a relatively small set of skills. The only knowledge the creator of a custom modification description needs to have is how XPATH works and how a simulation project file is represented in XML.

#### Built-In Custom Modifiers

It was stated that several predefined modifiers exist, which were implemented in JAVA. Additionally, several predefined/built-in custom modifiers exist:

- *ChangeIniFile:* substitute the ini file.
- *ChangeModelTimeStep:* change the minimum and maximum time step of a particular model.
- *ChangeAllTimeSteps:* change the minimum and maximum time steps of a model.

#### Multiple Variant Modifiers

Usually, a single modifier changes a certain property of a co-simulation project in one way. For instance, the *SubstituteModelModifier* substitutes the underlying model file. Thus, one input simulation project results in a single output simulation project (one product variant).

However, Figure 4.8 shows that the *BCSValueModifier* is derived from a class called *MultipleVariantModifier*. This means that one input simulation project results in one *or more* output simulation projects (one or more product variants).

This is caused by the nature of the *BCSValueModifier*. It is possible to specify more than one value for a single BCS parameter. For instance, a *BCSValueModifier* can specify the boundary condition parameter called *BCS123* to take values in the range of 5 to 10 (inclusive) with a step width of 1. Obviously, the parameter value cannot be set to all of

these values at the same time. Therefore, the input co-simulation project will result in 6 output simulation projects. In the first project BCS123 has the value 5, in the second project 6 and so on.

In case several multiple variant modifiers are applied to the same co-simulation project, the number of output projects is the algebraic product of the number of output variants per modifier. For instance, take a *BCSValueModifier A* that sets parameter *BCS1* to 3 different values (3 output variants) and a *BCSValueModifier B* that sets the parameter *BCS2* to 4 different values. Obviously, combining those two modifiers results in the production of 12 output simulation projects.

#### Constraints

The co-simulation variability model allows the creator to specify three different kinds of constraints:

1. *Require constraints* specify that a variant requires another variant. Take for instance a variant A, which requires variant B. There must not exist a configuration of a product variant, in which A is selected but B is not.
2. *Exclude constraints* specify that two variants exclude each other. Take for instance a variant A and variant B that exclude each other. No product variant containing A and B can exist.
3. *Limit constraints (cardinality)* specify the number of variants that have to be selected from a specific variation point. A limit constraint specifies an upper and lower bound of the number of selected variants. For instance, a limit constraint can state that for a particular Variation Point A, at least 1 and at most 2 variants are selected. The upper bound can be unbound, meaning that all variants can be selected. The default limit for all variation points is exactly one variant (min. 1, max. 1).

### 4.4.4 Application Model

In Section 3.3.6 the concept of the application model was described. The application model stores information about the resulting product variants. The application model component encompasses all classes and interfaces that are responsible for

- *loading* the application model from and *saving* it to memory,
- *checking* the model for validity,
- *generating* product variants, and
- *generating* an application model from a given co-simulation variability model.

#### Persistence

Just like the co-simulation variability model, the application model file is an XML file that is described by an XSD schema. JAXB is used for marshalling and unmarshalling.

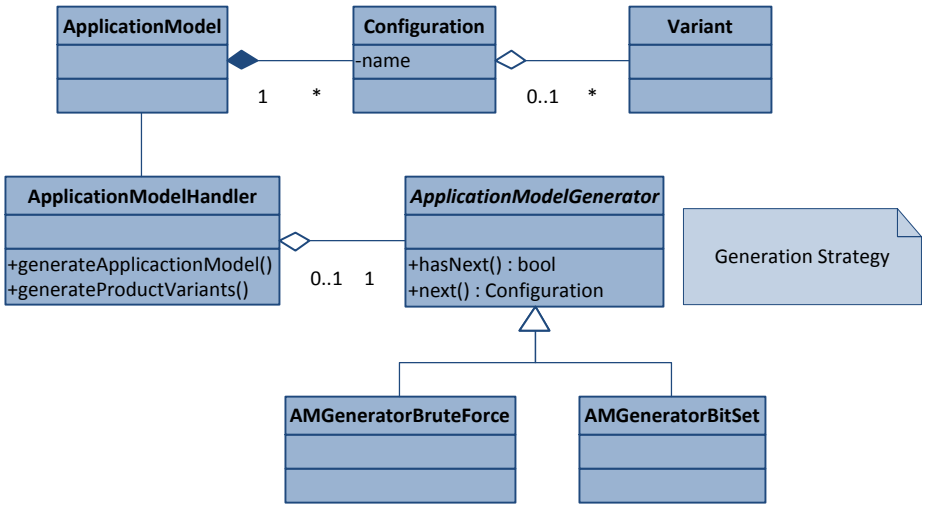


Figure 4.10: Class diagram of the most important classes in the application model component

An example application model file can be found in Appendix B.3. The application model consists of several configurations. Each configuration results in a single product variant (output co-simulation project). Note that this is not true if multiple variant modifiers (Section 4.4.3) are present. Each configuration consists of a collection of variants, so called selected variants. All modifiers that are part of those selected variants will be applied to produce one output co-simulation project.

### Generating Product Variants

Generating product variants is one of the two major tasks of the ICOS VM tool. In this step for every configuration in the application model a product is generated. The configuration consists of references to a collection of selected variants from the co-simulation variability model. In other words, an input simulation project is transformed to an output simulation project. The output differs from the input project in the way that is described by the selected variants.

The *ApplicationModelHandler* (depicted in Figure 4.10) generates the product variants. In order to do this, it needs

- an input simulation project (*ISimulationProjectModify*),
- a reference to the variants defined in the co-simulation variability model (provided by the interface *IVariabilityModel*), and
- a reference to the constraints of the co-simulation variability model (provided by the interface *IVariabilityModel*).

The details of this process are shown in an activity diagram in Figure 4.11. For the purpose of clarity the presence of multiple variant modifiers (Section 4.4.3) is not taken into account.

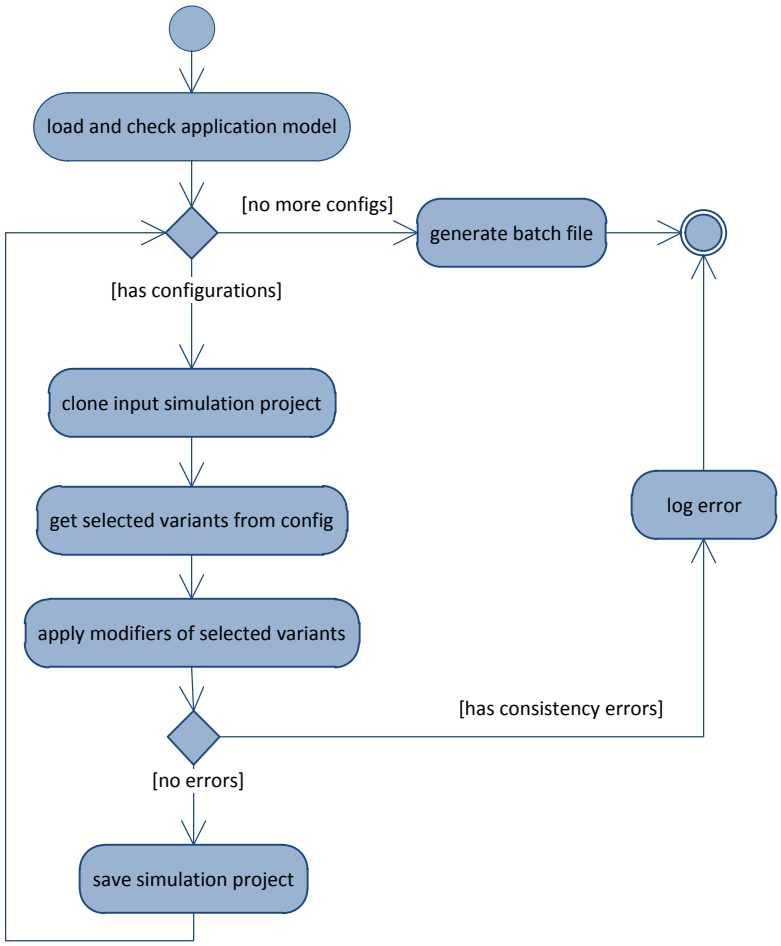


Figure 4.11: Activity diagram showing the process of product variant generation

### Generating the Application Model

As stated before, the application model consists of several configurations that contain references to variants in the co-simulation variability model. This model can be manually created by a user. On the other hand, an application model containing all valid combinations of variants can be generated automatically.

The main problem when generating all combinations of variants is the number of resulting combinations. Assume a co-simulation variability model with n variants without constraints. The number of configurations that would be generated is $2^n$. This is due to the fact, that every variant can be either present or not present in a configuration.

Take for example the co-simulation variability model in Figure 4.12. The model consists of 6 variants. Thus $2^6 = 64$ configurations would be generated, if no constraints existed.

One way to reduce the number of configurations is the introduction of constraints in the co-simulation variability model. If limit constraints are introduced (as can be seen in Figure 4.12), the number of generated configurations can be reduced to 18. From variation

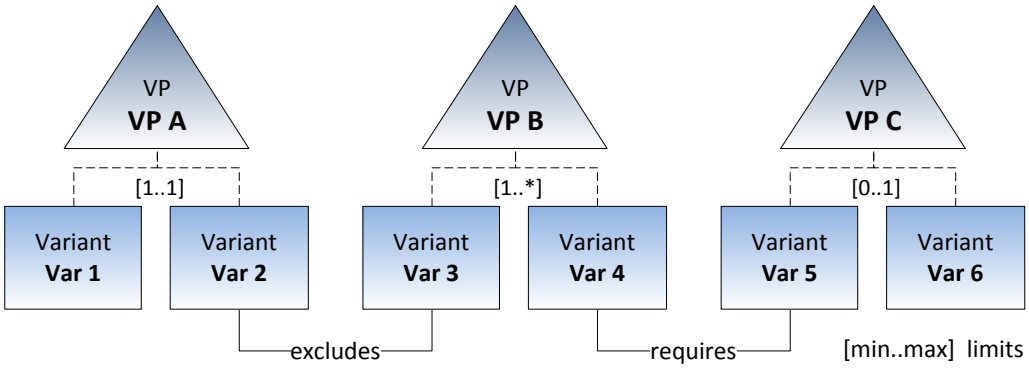


Figure 4.12: Example co-simulation variability model containing 3 variation points, 6 variants and require, exclude, and limit constraints

point VP A exactly one variant has to be selected. Thus, either variant Var 1 or Var 2 needs to be selected resulting in 2 possible configurations. One or two variants need to be selected from variation point VP B resulting in 3 possibilities (Var 3, Var 4, Var 3 and Var 4). Variation point VP C has 3 possible configurations as well (Var 5, Var 6, none). Therefore, the number of configurations is $2 * 3 * 3 = 18$.

Furthermore, requires- and exclude-relationships are introduced (as can be seen in Figure 4.12). This further reduces the number of configurations resulting in 6 configurations. All valid configurations are shown in Table 4.1.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1. | 1 | | 3 | | | |
| 2. | 1 | | 3 | | 5 | |
| 3. | 1 | | 3 | | | 6 |
| 4. | 1 | | 3 | 4 | 5 | |
| 5. | 1 | | | 4 | 5 | |
| 6. | | 2 | | 4 | 5 | |

Table 4.1: Resulting configurations of an example application model generation based on the co-simulation variability model depicted in 4.12

### Application Model Generation Constraints

The constraints presented so far (require, exclude and limit) are part of the co-simulation variability model. In addition to co-simulation variability model constraints, *generation constraints* can be defined. These constraints aim to further reduce the number of generated configurations. They give the user the possibility of only generating specific configurations. A user might only be interested in all configurations that contain variant Var 1. Similarly he could only be interested in those that do not include variant Var 1.

Therefore, two kinds of generating constraints are introduced:

- **Fixed** variant constraints: All configurations that will be generated include all variants that are said to be fixed.

- **Omitted** variant constraints: All configurations that will be generated do not include any variant that is said to be omitted.

Note that generating constraints are only taken into account during application model generation. Contrarily, co-simulation variability model constraints are also taken into account when product variants are generated.

### Application Model Generation Strategies

Generating all configurations of selected variants is implemented with the help of the strategy pattern [GHJV95]. The abstract class *ApplicationModelGenerator* is the interface to the generator, shown in Figure 4.10. Currently, two concrete implementations for the generating strategy exist. The main reason for the dual implementation of the strategy is testing, as described in Appendix A. While the first implementation is rather simple and straight forward the second implementation is more complex and tries to optimise the number of steps taken to generate all valid configurations.

As stated above, the application model generation strategy takes all variants from a co-simulation variability model and generates all valid configurations of variants. A valid configuration is a combination of variants that does not violate any variability model constraints or generating constraints.

#### Strategy 1: Generate and Check

A simple, kind of brute force approach to generate all possible configurations consists of the following steps:

1. Initialise a set of selectable variants. Selectable variants are all variants that are neither fixed nor omitted.
2. Generate all possible combinations of selectable variants.
3. Check each combination for whether it violates any constraint.
4. If it does not violate any constraint, add the configuration to the result set.
5. Add all fixed variants to all configurations in the result sets.

The main step (step 2) is a well-known problem referred to as ''generating all combinations''. Many generic algorithms that solve this problem exist. Knuth [Knu05] presents several algorithms and compares their performance and order in which combinations are generated. Ruskey and Williams [RW09] describe another approach to generate all combinations.

The implementation in the *ApplicationModelGeneratorBruteForce* class implements an algorithm described in [Knu05]. The algorithm generates combinations in lexographical order[3]. Here, the lexographical order refers to the position of a variant in the co-simulation variability model. Table 4.1 gives an example of lexographical order where variant A occurs before variant B in the variability model and so on.

[3] lexicographic order, also known as lexical order, dictionary order, alphabetical order

In order to improve the performance of this algorithm, only combinations of all non-fixed or omitted variants are generated. This is possible because fixed variants are part of all resulting combinations, while omitted variants must not occur in any resulting configuration. Hence, omitted variants are ignored and fixed variants are added to all resulting configurations.

**Strategy 2: Check While Generating**

Strategy 1 has a major drawback: all combinations of variants that result from step 1-2 have to be checked for constraint (require, exclude, limit) violation (step 3). Often, this is not necessary, as one constraint eliminates a series of combinations. Take for instance the example from Figure 4.12. By looking at the variability model, all combinations that include both variants Var 1 and Var 3 can immediately be eliminated, because they exclude each other. The same is true for combinations which include variant Var 4, but do not include variant Var 5, because of the require constraint. But even the limit constraints can be used to omit combinations. For instance, as variation point VP B requires at least 1 variant to be selected, it is known that as soon as variant Var 2 is selected, Var 4 must be selected as well.

Strategy 2 tries to make use of some of this information in order to eliminate invalid combinations as early as possible. The algorithm works based on bit operations and is therefore called *ApplicationModelGeneratorBitSet*. It works as follows:

1. Create a bit set (the Selected bit set) of n bits, where n is the number of variants in the co-simulation variability model.
2. Reserve the first $n_f$ bits for the fixed variants, where $n_f$ is the number of fixed variants.
3. Set the first $n_f$ bits in the Selected bit set to 1 (1 means selected, fixed variants are always selected).
4. Reserve $n_{vp}$ bits for each variation point, where $n_{vp}$ is the number of variants in variation point vp.
5. Prepare the constraint bit sets (explained in detail later).
6. For each variation point vp:
7. For each combination of variants of vp (each possible combination according to the limit constraints):
8. Set at least $limit_{min}$ and at most $limit_{max}$ bits in the bit vector from the bits that were previously reserved for variation point vp, where $limit_{min}$ and $limit_{max}$ are the limit constraints for vp.
9. Check if the combination violates constraints (using the constraint bit sets).
10. If constraints are violated, go to the next combination.

11. If no constraint is violated and vp is not the last variation point, go to the next variation point.

12. If no constraint is violated and vp is *the last* variation point → a valid combination has been found. Add all selected variants to the result set. Selected variants are recognised by a 1 in the Selected bit set.

13. If all combinations of variants in vp are processed, reset vp and go to the previous variation point. Reset means, if vp is visited again, all combinations of selected variants in vp are processed again.

14. If vp is *the last* variation point and there is no more combinations in any other variation point → terminate.

The main difference compared to Strategy 1 is the time and way how constraint checks are performed. After the variants of a variation point are added to the Selected bit set (according to the limit constraint for the variation point), the exclude and require constraints are checked. If any constraint is violated, the current combination is ignored. Along with this combination any other combination that has the same variants selected is ignored too.

So, how does the **bit-wise constraint check** work? The key is the preparation of a number of constraint bit sets (step 5). For each variant, two bit sets of length n are created. One contains bits that represent a variant's require constraints, the other one contains the exclude constraints. Take for instance the require bit set of variant 1 $Req_1$. If bit 4 in $Req_1$ is set, this means that variant 1 requires variant 4. The same applies to the exclude bit set.

Using these bit sets for each variant constraint violations can easily be detected at any time. In order to check if any combination of variants violates constraints, three global bit sets are required. These bit sets are built for a particular combination (selection of variants):

- The **Selected**-bit set (created in step 1): 0 in the bit set means that a variant is not part of a particular selection. Contrarily, 1 means that a variant is part of a configuration.

- The **Required**-bit set: this bit set is an aggregation of the required bit sets $Req_n$ for all variants that are selected. This bit set can be built by combining the required bit sets $Req_n$ of all selected variants with a bitwise OR.

- The **Excluded**-bit set: this bit set is an aggregation of the excluded bit sets $Exc_n$ for all variants that are selected. This bit set can be built by combining the excluded bit sets $Exc_n$ of all selected variants with a bitwise OR.

In the end, the **Selected** bit set has a 1 for variants that **are** part of the current configuration, **Required** has a 1 for all variants that **must be** part of the configuration and **Excluded** has a 1 for all variants that **must not be** part of the configuration.

Using this information one can create the truth table depicted in Table 4.2. The third column is 1 if a constraint is violated. The bold rows indicate the cases where constraints

| **Require** | | | **Exclude** | | |
|---|---|---|---|---|---|
| Selected | Required | $(!Sel\&Reg) \Rightarrow fail$ | Selected | Excluded | $(Sel\&Exc) \Rightarrow fail$ |
| 0 | 0 | 0 | 0 | 0 | 0 |
| **0** | **1** | **1** | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | **1** | **1** | **1** |

Table 4.2: Truth table for exclude and require bit operations

are violated. By evaluating the two expressions from Table 4.2, constraint violations can easily be detected. Evaluating the expression $(!Selected\&Required)$, using the bit vectors described above, a constraint is violated if any bit in the bit set that results from this formula is set. The same applies to the expression for excluded variants $(Selected\&Excluded)$.

The main advantage of this approach is the lower number of combinations that has to be considered. Table 4.3 shows the beginning of a sample execution of the algorithm. The first constraint violation is detected in line 2. In this line many combinations that include variant B and C are eliminated at once.

Note that bit operations are only performed for the first $n_p$ bits, where $n_p$ is the number of variants of all the variation points that have already been processed. All other bits are marked with an x in the Selected bit set in Table 4.3. This is done in order to prevent the detection of constraint violations, even though there are none present. Take for example line 3 in Table 4.3. The Require bit set has a 1 at the $5^{th}$ bit. Thus, variant Var 5 must be present in a valid configuration. However, this is not a constraint violation, as the variation point containing variant Var 5 has not yet been processed.

## 4.5 SPL Integration of ICOS Variability Management

Section 2.1.8 describes the separation of development activities into problem and solution space. Furthermore, it was stated that domain as well as application engineering span over both problem and solution space.

All the activities, models and artefacts that have been described for the standalone ICOS VM tool, only impact the solution space. Therefore, the tool chain in Section 4.2 did not take into account the separation of the tools and processes into these dimensions. As the tool should be integrated in an SPL, the activities need to be distinguished between problem and solution space.

Figure 4.13 shows the basic concept of the ICOS VM product line integration. The activities that are performed in various tools are separated horizontally in application and domain engineering and vertically in problem and solution space. For modelling the problem space as well as the relationships between problem and solution space, a tool called pure::variants is used. Pure::variants provides several different types of models that span over all four quadrants.

| | Step | Selected | Require | Exclude | Note |
|---|---|---|---|---|---|
| 1 | select variant 2 from VP 1 | 01xxxx | 000000 | 001000 | Var 2 is selected (selected bit), which excludes Var 3 (excluded bit) |
| 2 | select variant 3 from VP 2 | 0110xx | 000000 | 001000 | the exclude expression ($Sel\&Exc$) evaluates to true, skip this combination. |
| 3 | select variant 4 from VP 2 | 0101xx | 000010 | 001000 | |
| 4 | select no variant from VP 3 | 010100 | 000010 | 001000 | the require expression ($!Sel\&Req$) evaluates to true, skip this combination |
| 5 | select variant 5 from VP 3 | 010110 | 000010 | 001000 | all VPs processed, no constraint violated → first valid configuration found |
| 6 | select variant 6 from VP 3 | 010101 | 000010 | 001000 | the require expression ($!Sel\&Req$) evaluates to true, skip this combination |
| 7 | ... | | | | |

Table 4.3: Sample execution of application model generation strategy 2

### 4.5.1 Pure::Variants Interface

Pure::variants has to be extended to integrate ICOS VM. At this point, only the implications of the extension to the ICOS VM tool are covered.

The ICOS VM pure::variants extensions communicates with the standalone ICOS VM tool over a defined interface called *IPureVariantsInterface*. This interface offers basic functionality that is required by the extension. Examples for the offered functionality include the creation of modifiers, loading simulation projects and so on. Additionally to the *IPureVariantsInterface* the ICOS VM pure::variants extensions uses the interfaces *ISimulationProject* and *IVariabilityModel*.

Details about the implementation are described in another document with the title ''Integration of ICOS Co-Simulation Variability Management in Pure::Variants'' [Toe12].

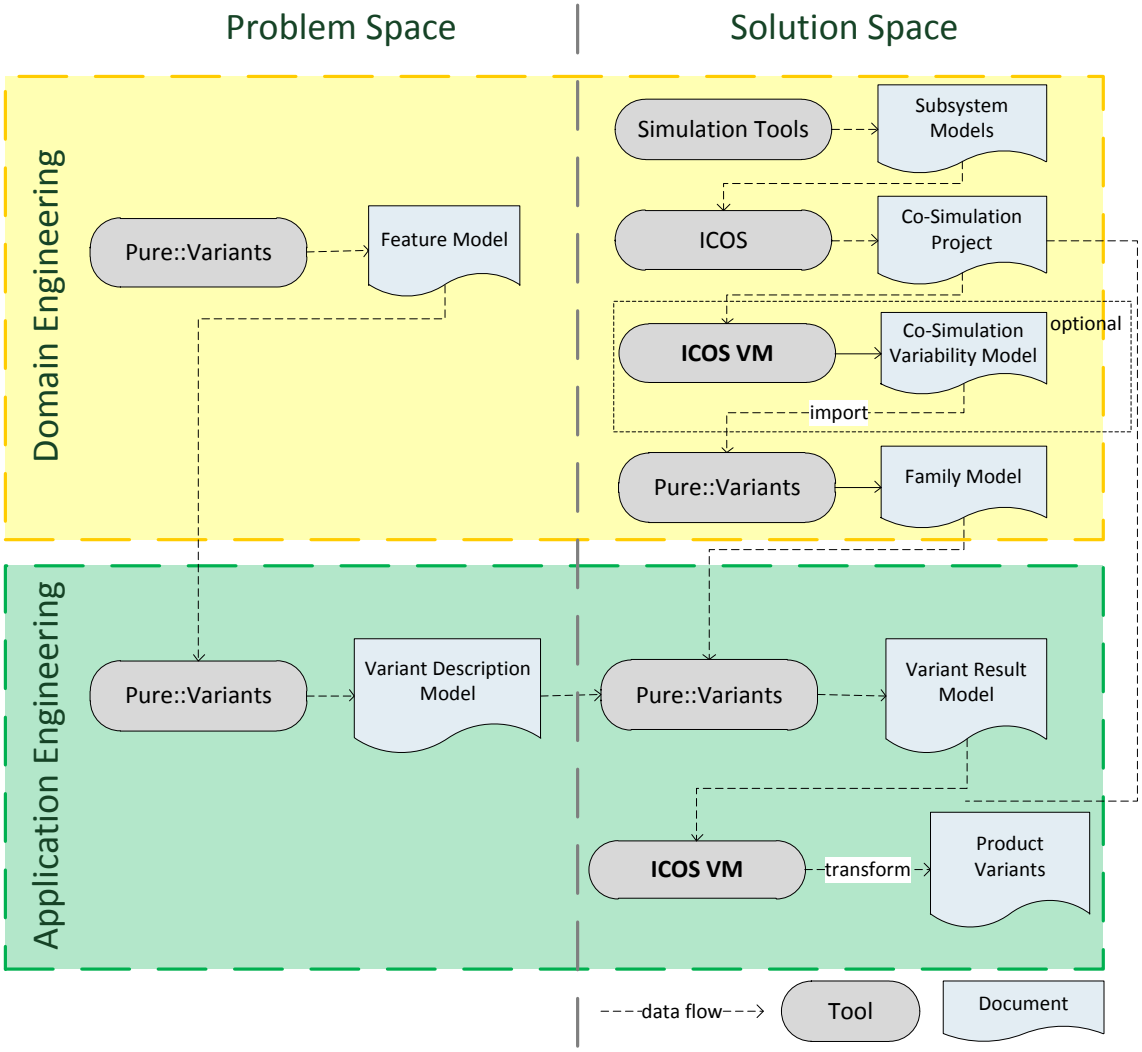


Figure 4.13: Basic concept of the ICOS VM tool chain integrated in pure::variants

# Chapter 5

# Evaluation

In order to evaluate the concept and implementation, it has been applied in a case study. A co-simulation scenario consisting of a partial model of a hybrid electric vehicle (HEV), based on Zehetner et al. [ZLWB12], was created. Several variation points were identified in the co-simulation environment, which were described in an ICOS co-simulation variability model. Based on the co-simulation variability model, an application model was generated by the ICOS VM tool. At the same time, an application model was created by hand, as only several product variants had to be simulated in order to interpret results.

## 5.1 Co-Simulation Environment

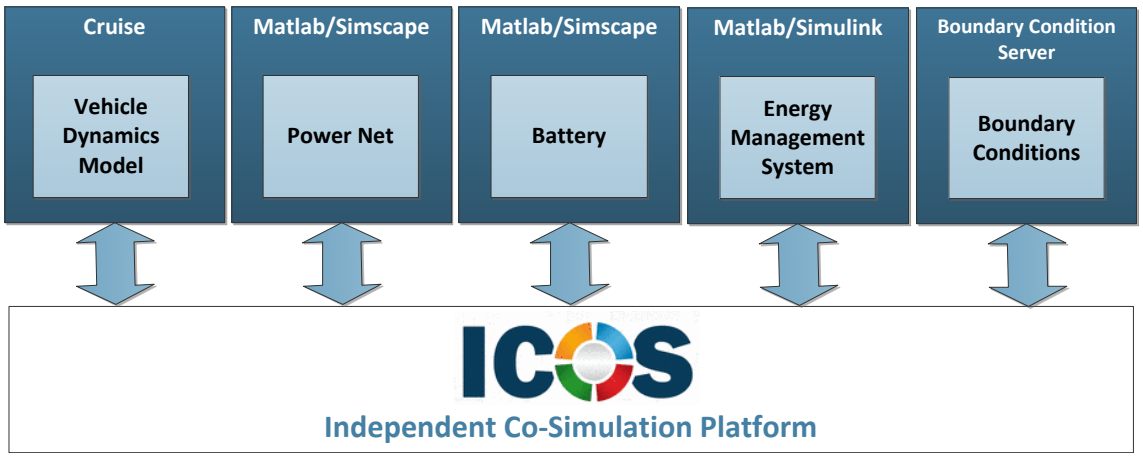


Figure 5.1: Co-simulation environment of the case study (partial hybrid electric vehicle model)

Figure 5.1 shows the co-simulation environment for this case study. The system is composed of several subsystems, which are subsequently described in detail. The co-simulation environment aims to study the power net of a hybrid electric vehicle.

Therefore, the co-simulation was developed and simulated in ICOS.

**Battery**

Currently two different models of the battery exist:

- a simple model, with a low degree of detail, developed in Simulink[1] and
- a more detailed, physical model, developed in Simscape[2].

For the co-simulation scenarios of this case study, the Simscape model was used. However, as the models have compatible interfaces, both models can be substituted easily.

Among others, the battery model provides an input parameter to set the initial state of charge (SOC). Other properties of the battery, such as the capacity, are set in an initialisation script file.

**Power Net**

The on-board power net was modelled using Simscape. It consists of several electrical components of a vehicle. Among other components, an electric double-layer capacitor (EDLC, also known as supercap or supercapacitor) is part of this model. EDLCs are electrical storage devices, which have relatively high power densities, long lifetime, as well as a great cycle number (number of times it can be recharged) [SG00]. In hybrid electric vehicles, EDLCs are often used as energy storage in combination with a conventional chemical battery.

For the co-simulation scenario presented below, two different models of the on-board power net exist: one including an EDLC and one without an EDLC.

**Energy Management System (EMS)**

This model simulates a control sequence of electrical loads of the vehicle. In other words, it provides a sequence of signals, which state whether an electrical load is turned on or off at a specific point in time. This information is used by the power net model. Even though the model is a simple Simulink model, it has been proven to be accurate enough in a number of co-simulation scenarios.

**Vehicle Dynamics Model**

A model of the vehicle's dynamics was developed using AVL Cruise[3], a powertrain and vehicle dynamics simulation software. It simulates the powertrain in a standardised drive cycle, known as UDC (urban driving cycle). A drive cycle is a set of data points, which represent a vehicle's speed at specific points in time. The urban driving cycle (UDC) was defined to represent city driving conditions. One of the output parameters the vehicle dynamics model provides is the engine's revolutions per minute. This parameter is used by the power net.

## 5.2 Defining Variability

By introducing variability to the co-simulation, several variants of the system (vehicle) can be modelled. Furthermore, several scenarios of the vehicle's environment can be simulated.

[1] http://www.mathworks.de/products/simulink/
[2] http://www.mathworks.de/products/simscape/
[3] https://www.avl.com/cruise1

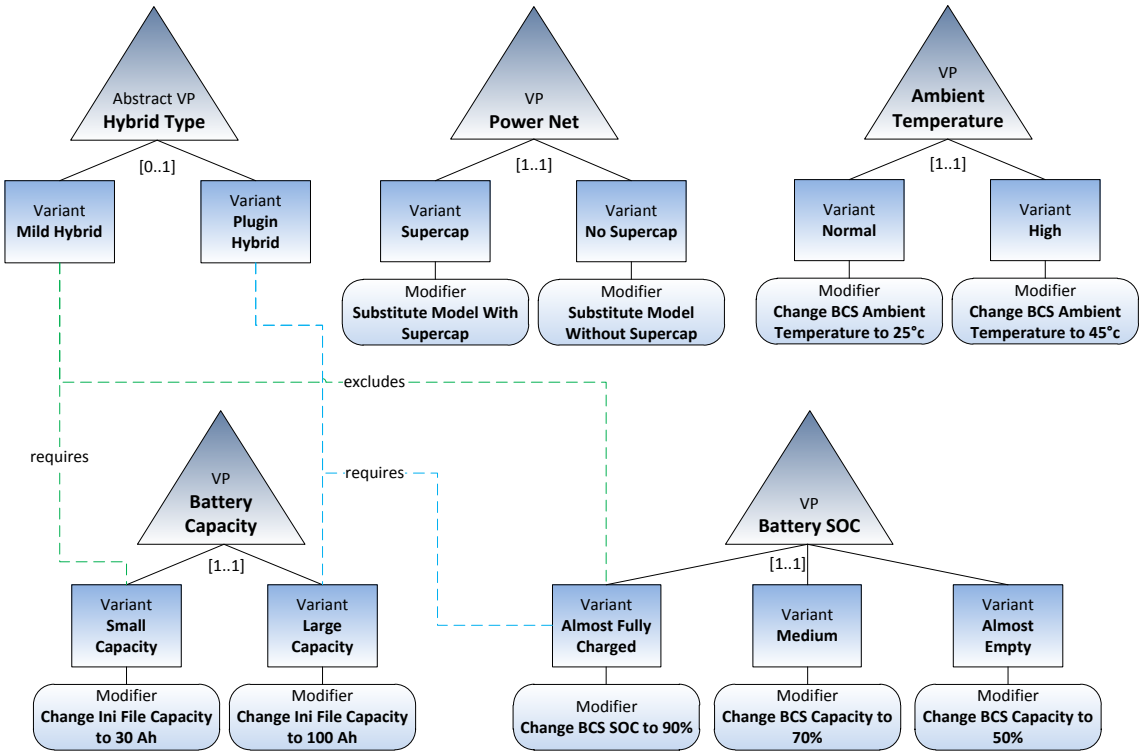


Figure 5.2: Co-simulation variability model of the case study

Therefore, an ICOS co-simulation variability model (depicted in Figure 5.2) containing the following *variation points* was developed:

**Power Net**
: As stated before, two alternative models of the power net exist. One of the models includes a supercapacitor (EDLC), while the other does not. Using this variation point allows the investigation of the impact of a supercapacitor on the overall system.

**Battery Capacity**
: Several vehicle variants might contain batteries with different capacities. The battery's capacity is set in an initialisation file. The initialisation file can be substituted using a custom modifier, as described in Section 4.4.3.

**Battery - Initial State of Charge**
: The initial state of charge of the battery is provided by a constant BCS parameter. Making this parameter variable, allows the investigation of the impact of different initial states of charge to the system.

**Ambient Temperature**
: This variation point was introduced to investigate the vehicle, given varying ambient temperatures. Among other things, the ambient temperature has an impact on the generator, which is part of the power net model.

Section 3.3 introduces four groups of variation points (model-related, parameter connections, environment and coupling). In regard to these groups, the first variation point (power net) is a model-related variation point, as it represents the variability of the subsystem ‘‘power net’’. The second variation point (battery capacity) can be seen as model-related too. Even though the subsystem itself is not substituted, two battery models with different capacities, in fact, represent two different variants of the subsystem ’’battery’’. The third variation point (battery SOC) describes variability of the co-simulation's environment. In other words, it allows the simulation of the system under changing boundary conditions. The last variation point (ambient temperature) describes variability of the system's environment.

The co-simulation and co-simulation variability model enable the evaluation of two user-visible vehicle variants:

1. *Conventional mild hybrid electric vehicle*: This kind of hybrid electric vehicle cannot be powered solely by its electric motor.

2. *Plug-in hybrid electric vehicle* (PHEV): This kind of vehicle contains an energy storage that allows recharging by connecting a plug to an external power source.

These variants have a huge impact on the co-simulation scenario. For instance, the capacity of the battery should be much larger for a plug-in hybrid electric vehicle. One way of modelling these variants, is by introducing an abstract variation point to the co-simulation variability model. This can be seen in Figure 5.2, where an abstract variation point ‘‘Hybrid Type’’ is present.

Another way is to describe the hybrid types in a higher-level model, like a feature model. The co-simulation variability model then needs to refer to the feature model. This use-case scenario was also modelled in pure::variants, using a feature model to describe the hybrid type. This is described in another work [Toe12], which is based on ICOS VM.

## 5.3 Application Model

An application model was automatically generated, consisting of all possible combinations of variants. Some of the product variants only showed minor differences from others. Therefore, four configurations were chosen and used for simulation:

- Mild hybrid with supercapacitor, a battery capacity of 30 Ah (Ampere hours), an initial state of charge of 70%, and a medium ambient temperature of $25°c$

- Mild hybrid without supercapacitor, a battery capacity of 30 Ah, an initial state of charge of 70%, and a medium ambient temperature of $25°c$

- Plug-in hybrid with supercapacitor, a battery capacity of 100 Ah, an initial state of charge of 90%, and a medium ambient temperature of $25°c$

- Plug-in hybrid without supercapacitor, a battery capacity of 100 Ah, an initial state of charge of 90%, and a medium ambient temperature of $25°c$

## 5.4 Lessons Learned

The case study showed that ICOS VM can be used to explicitly define variability in an ICOS co-simulation. It was shown that a co-simulation environment can be adapted to the system's variability (e.g. the battery's capacity varied, according to the type of hybrid electric vehicle). Additionally, the co-simulation's environment was varied (e.g. the ambient temperature), to allow the investigation of the system's behaviour in various environments.

During the development of this case study, several benefits of the ICOS VM tool were observed:

- **Single point of change:** when changes had to be applied to all product variants, this could be done once in the core co-simulation project. The next time, the product variants were generated and simulated, these changes took effect.

  Without ICOS VM, many similar co-simulation projects exist. Common changes have to be applied to all co-simulation projects manually. Obviously, this is more time consuming and error prone.

- **Automatic generation of application model:** After the co-simulation variability model was developed, an application model was automatically generated. Using application model generation constraints (fix and omit, see Section 4.4.4), it is easily possible to generate a special subset of all product variants (e.g. all product variants including a supercapacitor).

- **Optimisation:** In order to optimise a simulation parameter (e.g. battery capacity), the parameter value can be changed manually. Subsequently the simulation is started and the simulation results are evaluated. This manual process has to be repeated for a range of values, until a desired result is reached. Using ICOS VM, the parameter can be made variable and the simulation can automatically be executed for a range of parameter values. After the execution, the results can be evaluated and an appropriate parameter value can be determined.

Contrarily, some possible drawbacks of using ICOS VM were identified as well. These issues should be considered before ICOS VM is used within a project:

- **Return of investment:** creating a co-simulation variability model takes a considerable amount of time. For co-simulation projects with a large number of variation points and resulting product variants, this initial investment will be returned. However, for small co-simulation projects, a small number of variation points or product variants, this might not be the case.

- **Frequent changes of the core co-simulation project:** The co-simulation variability model describes the variability of a co-simulation project. Thereby models, parameters, or any other part of the co-simulation project need to be referenced. Therefore, changes in the co-simulation project have to be reflected in the co-simulation variability model. If the co-simulation project is subject to frequent changes, applying those changes to the co-simulation variability model takes a considerable amount of additional time.

# Chapter 6

# Conclusion and Future Work

This thesis aims to introduce variability management in co-simulation environments. At first variability management in general and its role in software product line engineering were studied. Terminology and basic concepts from these areas were presented. Moreover, their role in automotive engineering was investigated. Subsequently, co-simulation and its affiliation to the automotive industry were investigated.

Based on these investigations, the *variability of co-simulation environments* was described. Thereby different scopes of variability in the co-simulation of a particular system were defined:

- Variability of the overall system
- Variability of subsystems that are part of a co-simulation
- Variability of the co-simulation environment

Furthermore, the common variation points of a co-simulation environment were studied and distinguished into four groups:

- Model-related variation points
- Linking variation points
- Environment variation points
- Coupling variation points

Using this classification, several requirements to variability modelling and management in a co-simulation environment were identified. Among others, it was stated that each of the aforementioned types of variation points has to be taken into account when introducing variability management to co-simulation.

On the basis of the identified requirements a prototypical implementation of a variability management tool for the ICOS co-simulation platform was implemented. So far, the ICOS VM tool allows the introduction of variability into a co-simulation project by

- substituting models,
- choosing between alternative models,
- changing boundary condition values,

- changing parameter linking, and
- the use of custom modifiers, which easily allow a developer to make any part of the co-simulation project variable, using XML/XPATH.

The ICOS VM tool was then evaluated in a case study, using the simulation of a partial hybrid electric vehicle model. Using this model, it was shown how variability management in co-simulation can be used to (1) reflect user visible vehicle variants in co-simulation (plugin vs. non-plugin hybrid electric vehicle) and (2) vary the system's environment.

One of the benefits that come with the use of ICOS VM is the explicit definition of variability. This clearly improves communication, and traceability of variability. Furthermore, it enables the integration of the co-simulation variability model into a higher level model, such as a feature model. Another benefit is that changes to all product variants can be done at one single place and are automatically reflected in all product variants. This does not only save time and effort, when faced with a large number of product variants. It further prevents errors that might arise from applying changes to several products.

## 6.1 Future Work

### 6.1.1 Graphical User Interface

As stated before, there is no graphical user interface to create a domain or application model. The implementation of a GUI would have been out of scope for the prototypical implementation of the ICOS VM tool.

The integration of ICOS VM in pure::variants, described in [Toe12], provides a graphical user interface to be able to create and manage a family model. The content of the family model is in some way comparable to the co-simulation variability model.

However, a dedicated graphical user interface is desirable to support users with the creation of domain and application models. The GUI could be implemented as a standalone tool or as an IDE (Integrated Development Environment) plugin, e.g. for the Visual Studio or Eclipse platform.

### 6.1.2 Result Evaluation and Optimisation

When a co-simulation was performed successfully, the user evaluates the results of the simulation. Based on this evaluation the user might change parameters and restart the co-simulation.

The process of evaluation can be automated or partially automated. This can be done using approaches such as simulation-based optimisation, as described in [Den07].

### 6.1.3 Hierarchical Co-Simulation Projects

According to the release plans of the ICOS developer team, the support for hierarchical co-simulation projects is anticipated. Thus, an ICOS co-simulation project can be included in another co-simulation project, just like any subsystem model. This has various reasons, e.g. the support for different coupling strategies for different sub-projects and to decompose large systems.

As long as only the parent co-simulation project is variable, there is no need for changes in the ICOS VM tool. This is due to the fact that the child co-simulation project can be treated just like any other model in the co-simulation.

However, if child co-simulation projects (co-simulations that are referenced as models by other co-simulations) are subject to variability, the ICOS VM tool is affected. Variability in the child co-simulation projects has to be taken into account in the parent project. This is due to the fact that for each variant of the child co-simulation a single co-simulation project file exists. The reference to this file needs to be updated in the parent co-simulation.

### 6.1.4 Model Databases

A model database stores subsystem models along with metadata about the models. The latest version of ICOS supports retrieving subsystem models directly from an existing model database. However, the support for model databases is not yet part of the ICOS VM tool. Referring to subsystem model from a central database in the variability model can be seen as a future extension of ICOS VM.

### 6.1.5 Concurrency

Currently the ICOS VM tool executes sequentially. Thereby most of the execution time is taken to generate the application model as well as to generate the product variants. Both of these processes can be parallelised. For product variant generation, for instance, worker threads can generate one product variant at a time.

### 6.1.6 Non-Constant Boundary Condition Parameters

So far all boundary condition modifiers have only been available for constant boundary condition parameters. However often ICOS co-simulations contain boundary condition values that change over simulation time. Therefore supporting the variability of such boundary conditions is desirable.

A constant BCS parameter value provides a single value as output. Thus, it is possible to provide a range of values (e.g. 1 to 5, step size 1) to make the single value variable. Parameter values that change over simulation time are specified by multiple values (e.g. 1 at time step 0, 42 at time step 20,...). Therefore providing a range of values for each of these time steps has to be implemented.

# Appendix A

# Testing

This section describes how the ICOS VM tool implementation was tested. The purpose of the tests is to execute the program to find as many errors as possible [MBTS04]. How this is done and what measures and strategies were used will be presented.

## A.1 Details

Basically, the developed tests can be split into two groups:

- *Unit tests* for the JAVA implementation of the ICOS VM tool.
- *Integration tests* in the form of test scripts, which test the integration of the JAVA implementation within the ICOS VM tool chain.

While this section mainly deals with unit test, integration tests are explained in Section A.3.

### A.1.1 Fixtures

Test *fixtures* are data that is being used among several test cases [BG00]. In other words, fixtures are objects that are created during the setup phase of testing and are shared during execution and verification by multiple test cases.

Fixtures can be separated into shared and fresh fixtures [Mes09a, Mes09b]. Shared fixtures are objects that are created once (during setup phase). One and the same instance is used for the execution of several test cases. The object is destroyed in the tear down phase of testing.

Fresh fixtures, on the other hand are recreated for every test case. But the way they are built and the data they hold is the same among all test cases.

Fixtures were used to reduce the complexity of our test cases. For instance: in order to load and check a co-simulation variability model, a simulation project must be present. In the co-simulation variability model test cases, a simulation project was used as fixture.

In most of the cases it was possible to use shared fixtures. However, for test cases in which fixture objects are modified, fresh fixtures are required. For instance: when modifiers are tested to modify a simulation project, the simulation project can be seen as a fresh

fixture. As the modifiers changed properties of the simulation project it obviously had to be instantiated for each test case.

### A.1.2 Code coverage

Code coverage tools measure how thoroughly tests exercise programs. In other words they give an answer to the question about the percentage of code that was executed during testing. Code coverage has to be used with care. A code coverage of 100% means that all parts of the code have been executed at least once. However, this does not prove the program to be correct at all [Mar99, MBTS04].

It was not tried to verify our program by using code coverage. On the contrary, code coverage analysis was used to identify parts of the source code that were not covered well or at all by test cases.

In order to perform code coverage analysis, two JAVA tools were used: EclEmma[1] and Cobertura [2]. Both tools reported similar results for the code coverage for almost all components of our tool. With the help of those tools two subcomponents were identified that were not tested at all. Finally a code coverage of 80.4% for the entire program was achieved. Big parts of the remaining uncovered code are the POJOs (Plain Old Java Object) that were generated by JAXB.

## A.2 Test Scope

ICOS VM unit tests were developed with different scopes. The most detailed test cases were developed for a single class. As a next step, test cases for functionality spanning several classes in a single component were written. Moreover, component tests were written that aimed to test the functionality of a whole component. Finally several test cases were developed that aimed to test communication between components and tried to test the whole program's behaviour.

The remainder of this section describes the tests of several program components.

### A.2.1 User Interface

The main user interface of the ICOS VM tool is a command line application, represented by the class *ConsoleInterface*. The class parses the command line arguments and initiates the actions intended by the user.

The test cases of this component cover the parsing of correct and incorrect arguments and other command line specific behaviour, as well as the communication and correct initialisation of other components.

### A.2.2 Simulation Project

After the interfaces of the simulation project were defined, the SimulationProjectHandler, the most important part of this component, was developed using test-driven (test first)

[1] http://www.eclemma.org/
[2] http://cobertura.sourceforge.net/

development.

The tests cover loading and saving correct, as well as dealing with corrupted simulation project files. Further, the correct behaviour of all modifications (model deletion, parameter renaming...) was tested. This was done by calling the appropriate modifying method and then verifying the changes in the underlying XML file.

### A.2.3 Co-Simulation Variability Model Component

The co-simulation variability model as a whole was tested (1) using sample XML variability model files, and (2) by building variability models in code. Generally the resulting test cases can be separated into three categories:

- **Modifiers:** All modifiers implement behaviour to check and apply a certain modification. The check and apply behaviour of each modifier was tested. This was done using correct and corrupted input.
- **Custom Modifiers:** In addition to the general test for all modifiers, several dedicated tests for custom modifications were performed. In particular there was thorough testing of the loading and parsing custom modification description files as well as their boolean expressions.
- **Constraints:** Co-simulation variability model constraints were tested by observing the behaviour of the model in the presence of correct, corrupted and conflicting constraints.

### A.2.4 Application Model Component

A great part of the application model tests addressed two of the components main functionalities:

- **Application Model Generation:** As stated above, the application model generation is implemented using two distinct strategies. These strategies were used to test one another. Therefore, a range of test data was created, which is used to generate application models using both strategies. Afterwards their results are compared.
- **Product variant generation:** In order to reduce the complexity of testing product variant generation, different methods to evaluate results were used. For some simple co-simulation variability models, all of the resulting product variants were checked in detail. Hence, it was checked if all modifications were carried out successfully for each variant and so on. On the other hand, for complex co-simulation variability models only the number of generated variants and some particular modifications were checked.

## A.3 Integration Testing

The purpose of the integration test scripts is to test the integration of the JAVA console application in the ICOS tool chain. Thus, the test scripts do not only test the JAVA console application but the wrapper script. This includes the execution of the output co-simulations. Therefore in order to run the integration test scripts, ICOS must be

installed on the host that runs the integration tests. Moreover the remote hosts that run the simulation tools required for the tested projects must be accessible from the test host.

So far all test cases only include Matlab/Simulink models. Thus, only a host running Simulink and the ICOS remote server must be available.

# Appendix B

# Examples

## B.1 Example Co-Simulation Variability Model File

```
<?xml version="1.0" encoding="utf-8"?>
<DomainModel>

  <VariationPoint ID="variationPoint1" Name="Heat Model" >
      <Variant ID="variant_1_A" Name="Heat Model Variant 1">
        <ExchangeModel ModelName="Model1"
            FilePath="..\models\Simple1_Part1_var2.mdl" />
        <ChangeParameterNameInSimTool
            CurrentParameterNameInIcos="Part1_InputB"
            NewName="Part1_Input_var2" />
      </Variant>
      <Variant ID="variant_1_B" Name="Heat Variant 2">
        <ExchangeModel ModelName="Model1"
            FilePath="..\models\Simple1_Part1_var3.mdl" />
      </Variant>
  </VariationPoint>

  <VariationPoint ID="variationPoint2" Name="Heat Model Engine"
     >

    <ModelAlternative ID="alternative1"/>

    <Variant ID="variant_2_A" Name="Abstract Engine 1">
      <ChooseModel ModelAlternative="alternative1"
          ModelName="Model2" />
    </Variant>
    <Variant ID="variant_2_B" Name="Detailed Engine 2">
      <ChooseModel ModelAlternative="alternative1"
          ModelName="Model2_var2" />
      <ChangeOutputParameterMapping
```

```
        CurrentParameterNameInIcos="Part2_OutputA"
        NewName="Part2_OutputA_var2" />
    </Variant>
    <Variant ID="variant_2_C" Name="No Engine">
      <!-- model works without the engine model -->
      <ChooseModel ModelAlternative="alternative1" None="true"/>
    </Variant>

  </VariationPoint>

  <VariationPoint ID="variationPoint3" Name="Boundary
      Conditions for model1" >

    <Variant ID="variant_3_A" Name="BCS Values 1">
      <BCSValue ParameterName="BCS_Output1">
        <Range StartValue="5" EndValue="10" Step="0.5" />
      </BCSValue>
    </Variant>
    <Variant ID="variant_3_B" Name="BCS Values 2">
      <BCSValue ParameterName="BCS_Output1">
        <Entry Value="4" />
        <Entry Value="6" />
        <Entry Value="11" />
      </BCSValue>
    </Variant>

  </VariationPoint>

  <!-- change time step using custom modifications -->
  <VariationPoint ID="variationPoint4" Name="Change time step" >

    <Variant ID="variant_4_A" Name="AdapeTime Step" >
      <Modification Type="changeTimeStep">
        <Argument Name="minTimeStep" Value="1" />
        <Argument Name="maxTimeStep" Value="10" />
      </Modification>
    </Variant>

  </VariationPoint>

  <!-- constraints are checked during generation of app model
      as well as generation of prodcuts -->
  <Constraints>
    <Exclude VariantA="variant_1_A" VariantB="variant_2_C" />
    <Exclude VariantA="variant_2_B" VariantB="variant_3_B" />
```

```
    <Require VariantA="variant_1_B" VariantB="variant_3_A" />

    <!-- The default limit is [1, 1], we set it anyways-->
    <Limit MinVariants="1" MaxVariants="1"
        VariationPoint="variationPoint1" />
    <Limit MinVariants="1" MaxVariants="1"
        VariationPoint="variationPoint2" />
    <Limit MinVariants="0" MaxVariants="1"
        VariationPoint="variationPoint3" />
    <Limit MinVariants="0" MaxVariants="unbounded"
        VariationPoint="variationPoint4" />
  </Constraints>

  <!-- constraints are used for generation of app model, but do
      not effect the product generation-->
  <GeneratingConstraints>
    <!-- only generate variants including variant_1_A -->
    <Fix VariationPoint="variationPoint1" Variant="variant_1_A"
        />

    <!-- don't generate variants including variant_2_C -->
    <Omit VariationPoint="variationPoint2"
        Variant="variant_2_C" />
  </GeneratingConstraints>

</DomainModel>
```

Listing B.1: Example co-simulation variability model file

## B.2 Example Custom Modification Description File

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>

<CustomModifications>

  <CustomModification Name="ChangeModelTimeStep">
    <!-- arguments that have to be provided by the user -->
    <Argument
        Name="timeStep"
        DataType="double" />
    <Argument
        Name="maxTimeStep"
        DataType="double" />
    <Argument
        Name="modelName"
        DataType="string" />
```

```
    <!-- a list of models and parameters that are going to be
        modified -->
    <ModifiedModel Name="{modelName}" />

    <!-- the modifications that are carried out (xpath
        expression) -->
    <Modification
        Selector="//wrapper[@name='{modelName}']/@timeStep"
        Value="{timeStep}"
        CreateIfNotExists="true" />

    <Modification
        Selector="//wrapper[@name='{modelName}']/@maxTimeStep"
        Value="{maxTimeStep}"
        CreateIfNotExists="true" />

     <!-- expressions to be checked before the modifications
         are carried out -->
    <Rules>
      <Expression
          Action="Fail"
          Message="The maximum time step ({maxTimeStep}) can not
              be less than the time step ({timeStep}).">
        <And>
          <Leq>
            <ArgumentValue ArgumentName="timeStep" />
            <ArgumentValue ArgumentName="maxTimeStep" />
          </Leq>
          <Gt>
            <ArgumentValue ArgumentName="timeStep" />
            <Value Value="0" DataType="double" />
          </Gt>
        </And>
      </Expression>
    </Rules>
  </CustomModification>

</CustomModifications>
```

Listing B.2: Example custom modification description file

## B.3 Example Application Model File

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationModel>
    <Configuration Name="Simulation Product 1">
        <SelectVariant VariantID="variantA"/>
```

```
        <SelectVariant VariantID="variantC"/>
        <SelectVariant VariantID="variantD"/>
    </Configuration>
    <Configuration Name="Simulation Product 2">
        <SelectVariant VariantID="variantB"/>
        <SelectVariant VariantID="variantD"/>
        <SelectVariant VariantID="variantE"/>
    </Configuration>
</ApplicationModel>
```

Listing B.3: Example application model file

## B.4 Example ICOS Batch File

```
1 "C:\Temp\Simple1_0001_config_0001.icos"
    "C:\licencefile_path\ValidLicence.lic"      127.0.0.1:1234
    "superSecreteRemotePassword"
2 "C:\Temp\Simple1_0001_config_0002.icos"
    "C:\licencefile_path\ValidLicence.lic"      127.0.0.1:1234
    "superSecreteRemotePassword"
3 "C:\Temp\Simple1_0001_config_0003.icos"
    "C:\licencefile_path\ValidLicence.lic"      127.0.0.1:1234
    "superSecreteRemotePassword"
```

Listing B.4: Example ICOS batch file

# Bibliography


[AHLAO+07] Ahmad Al-Hammouri, Vincenzo Liberatore, Huthaifa Al-Omari, Zakaria Al-Qudah, Michael S. Branicky, and Deepak Agrawal. A co-simulation platform for actuator networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 383--384, New York, NY, USA, 2007. ACM.

[AMO+02] A. Amory, F. Moraes, L. Oliveira, N. Calazans, and F. Hessel. A heterogeneous and distributed co-simulation environment [hardware/software]. In *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, pages 115 -- 120, 2002.

[ASB99] D. Atef, A. Salem, and H. Baraka. An architecture of distributed co-simulation backplane. In *Circuits and Systems, 1999. 42nd Midwest Symposium on*, volume 2, pages 855 --858 vol. 2, 1999.

[BBM05] Kathrin Berg, Judith Bishop, and Dirk Muthig. Tracing software product line variability: from problem to solution space. In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, SAICSIT '05, pages 182--191, Republic of South Africa, 2005.

[BC05] Felix Bachmann and Paul C. Clements. Variability in Software Product Lines. Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon, September 2005.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 13--21, London, UK, 2002.

[BG00] Kent Beck and Erich Gamma. More java gems. chapter Test-infected: programmers love writing tests, pages 357--376. Cambridge University Press, New York, NY, USA, 2000.

[BHR+07] D.A.v. Beek, A. T. Hofkamp, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Co-Simulation of Chi And Simulink Models, 2007.

[BPSP04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333--352, 2004.

[BSW10] M. Benedikt, H. Stippel, and D. Watzenig. An adaptive coupling methodology for fast time-domain distributed heterogeneous co-simulation. In *SAE Technical Paper*, volume 2010-01-0649, 2010.

[BZWB11] Martin Benedikt, Josef Zehetner, Daniel Watzenig, and Jost Bernasch. Modern coupling strategies: is co-simulation controllable? . In *NAFEMS Seminar: The Role of CAE in System Simulation*, Wiesbaden, Germany, 2011.

[Cen11] Virtual Vehicle Competence Center. *ICOS User Manual version 1.5*. Virtual Vehicle Competence Center, Graz, Austria, November 2011.

[Cen12a] Virtual Vehicle Competence Center. ICOS - independent co-simulation product page. http://vif.tugraz.at/en/products/icos/, January 2012.

[Cen12b] Virtual Vehicle Competence Center. *ICOS User Manual version 2.0*. Virtual Vehicle Competence Center, Graz, Austria, January 2012.

[Den07] Geng Deng. *Simulation-Based Optimization*. PhD thesis, University of Wisconsin - Madison, 2007.

[Dro04] S. Dronka. *Die Simulation gekoppelter Mehrkörper- und Hydraulik-Modelle mit Erweiterung für Echtzeitsimulation*. Shaker, 2004.

[FSSD09] M.O. Faruque, M. Sloderbeck, M. Steurer, and V. Dinavahi. Thermo-electric co-simulation on geographically distributed real-time simulators. In *Power Energy Society General Meeting, 2009. PES '09. IEEE*, pages 1 --7, Calgary, AB, Canada, July 2009.

[Fuj99] R.M. Fujimoto. Parallel and distributed simulation. In *Simulation Conference Proceedings, 1999 Winter*, volume 1, pages 122 --131, Phoenix, AZ , USA, December 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[GKL06] Marcus Geimer, Thomas Krüger, and Peter Linsel. Co-Simulation, gekoppelte Simulation oder Simulatorkopplung? Ein Versuch der Begriffsvereinheitlichung. In *O+P Zeitschrift fr Fluidtechnik (50)*, pages 572--576, Wiesbaden, Germany, 2006.

[GNLG11] Francisco Gonzalez, Miguel Naya, Alberto Luaces, and Manuel Gonzlez. On the effect of multirate co-simulation techniques in the efficiency and accuracy of multibody system dynamics. *Multibody System Dynamics*, 25:461--483, 2011.

[JB02] Michel Jaring and Jan Bosch. Representing variability in software product lines: A case study. In Gary Chastek, editor, *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 219--245. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45652-X_2.

[JC99] Steve DeRose James Clark. XML Path Language (XPath). http://www.w3.org/TR/xpath/, 1999. Retrieved on March 8 2012.

[Knu05] D.E. Knuth. *The art of computer programming, Fascicle 3: Generating all combinations and partitions*. Addison-Wesley, 2005.

[Kru02] Charles Krueger. Variation management for software production lines. In *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 107--108. Springer Berlin / Heidelberg, 2002.

[Kru04] Charles Krueger. Towards a taxonomy for software product lines. In *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 323--331. Springer-Verlag Berlin Heidelberg, Germany, 2004.

[Kru12a] Charles W Krueger. Introduction to software product lines. http://www.softwareproductlines.com/introduction/introduction.html, 2012. Retrieved on January 26 2012.

[Kru12b] Charles W Krueger. Introduction to software product lines - binding. http://www.softwareproductlines.com/introduction/binding.html, 2012. Retrieved on January 27 2012.

[Kru12c] Charles W Krueger. Introduction to software product lines - concepts. http://www.softwareproductlines.com/introduction/concepts.html, 2012. Retrieved on January 31 2012.

[Kru12d] Charles W Krueger. Introduction to software product lines - production. http://www.softwareproductlines.com/introduction/production.html, 2012. Retrieved on March 3 2012.

[Lan03] The American Heritage Dictionary Of The English Language. Definition of the word ‘‘model’’. http://www.thefreedictionary.com/model, 2003. Retrieved on January 25 2012.

[Lei09] Andrea Leitner. A software product line for a business process oriented IT landscape. Master's thesis, Graz University of Technology, September 2009.

[LPPFK06] Günter Lang, Heinz Petutschnig, Wolfgang Puntigam, and Josef Hage Filip Kitanoski. Simulation of the warm-up of the combustion engine and the vehicle by coupling of different simulation models . Technical report, Magna - Engineering Center Steyr - Software and Simulation - Energy Management, 2006.

[LSR07] Frank Van Der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action - the best industrial practice in product line engineering*. Springer-Verlag Berlin Heidelberg, Germany, 2007.

[Mar99] Brian Marick. How to misuse code coverage. http://www.exampler.com/testing-com/writings/coverage.pdf, 1999. Retrieved on March 9th 2012.

[MBTS04] G.J. Myers, T. Badgett, T.M. Thomas, and C. Sandler. *The art of software testing*. Business Data Processing: A Wiley Series. John Wiley & Sons, 2004.

[Mes09a] Gerard Meszaros. Fresh fixture. http://xunitpatterns.com/Fresh%20Fixture.html, 2009. Retrieved on March 8th 2012.

[Mes09b] Gerard Meszaros. Shared fixture. http://xunitpatterns.com/Shared%20Fixture.html, 2009. Retrieved on March 8th 2012.

[NSL09] Nicolas Navet and Francoise Simonot-Lion. *Automotive Embedded Systems Handbook*. CRC Press, Taylor Francis Group, Boca Raton, USA, 2009.

[Ort12] Ed Ort. Java Architecture for XML Binding (JAXB). http://www.oracle.com/technetwork/articles/javase/index-140168.html, 2012. Retrieved on February 11 2012.

[OV04] O. Oberschelp and H. Vocking. Multirate simulation of mechatronic systems. In *Proceedings of the IEEE International Conference on Mechatronics. ICM '04.*, pages 404 -- 409, June 2004.

[Pav11] Nicolas Pavlidis. Design and Implementation of a Variant Rich Component Model for Model Driven Development. Master's thesis, Graz University of Technology, October 2011.

[PBL05] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag Berlin Heidelberg, Germany, 2005.

[Pun07] Wolfgang Puntigam. Coupled Simulation: key for a successful energy management. In *Virtual Vehicle Creation*, June 2007.

[Row94] J.A. Rowson. Hardware/software co-simulation. In *Design Automation, 1994. 31st Conference on*, pages 439 -- 440, june 1994.

[RW09] Frank Ruskey and Aaron Williams. The coolest way to generate combinations. generalisations of de bruijn cycles and gray codes/graph asymmetries/hamiltonicity problem for vertex-transitive (cayley) graphs. *Discrete Mathematics*, 309(17):5305 -- 5320, 2009.

[Sch06] Douglas C. Schmidt. Why software reuse has failed and how to make it work for you. http://www.cs.wustl.edu/~schmidt/reuse-lessons.html, 2006. Published in the C++ Report. Retrieved on January 26 2012.

[SG00] Adrian Schneuwly and Roland Gallay. Properties and applications of supercapacitors from the state-of-the-art to future trends. *PCIM 2000*, pages 1--10, 2000.

[Sha98] R.E. Shannon. Introduction to the art and science of simulation. In *Simulation Conference Proceedings, 1998. Winter*, volume 1, pages 7 --14 vol.1, dec 1998.

[SJ04] Klaus Schmid and Isabel John. A customizable approach to full lifecycle variability management. *Science of Computer Programming*, 53(3):259 --284, 2004. Software Variability Management.

[TMKG07] C. Tischer, A. Muller, M. Ketterer, and L. Geyer. Why does it take that long? Establishing Product Lines in the Automotive Domain. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 269 --274, September 2007.

[Toe12] Philipp Toeglhofer. Integration of ICOS Co-Simulation Variability Management in Pure::Variants. Master Project at the Institute for Technical Informatics, Graz University of Technology, March 2012.

[WWLZ09] Jiaxue Wang, Qingnian Wang, Jishun Liu, and Xiaohua Zeng. Forward simulation model precision study for hybrid electric vehicle. In *Mechatronics and Automation, 2009. ICMA 2009. International Conference on*, pages 2457 --2461, August 2009.

[ZLWB12] Josef Zehetner, Wenpu Lu, Daniel Watzenig, and Jost Bernasch. Co-simulation based analysis of a two-voltage electrical system for hybrid electric vehicles. In *Hybrid and Electric Vehicles: 9th symposium, February 13 and 14, 2012*, pages 139--155. ITS Niedersachsen, Germany, 2012.