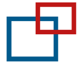
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

DISTANCE AND INDEPENDENT
STUDIES CENTER



Herausragende Masterarbeiten

am Distance and Independent Studies Center

Studiengang:	Software Engineering for Embedded Systems, M.Eng.
Masterarbeitstitel:	Behavioural Customisation of State Machine Models
Autor*in:	Josef Hofer

Declaration

I assure you that I have written the Master's thesis independently and have only used the sources and resources indicated and that I have identified the passages taken literally or in terms of content from the sources used.

Place, date Signature

Abstract

For the development of the Extremely Large Telescope (ELT), the European Southern Observatory (ESO) uses state machines to model life cycles and basic behaviour of control software components. To provide certain degrees of freedom, the component life cycles need to be customisable but in order to remain compatible, they must also conform to specific standard behaviour.

Clearly, these two goals are competing. High customisation causes difficulties in maintenance and may also lead to incompatible solutions. The introduction of strict compatibility requirements on the other hand may increase maintainability but it also makes the system less flexible. To avoid spending a significant portion of the Assembly, Integration and Verification (AIV) phase in integration hell, it is of high importance to find the right balance between customisability and compatibility early enough.

To address this problem, this thesis examines different variability realisation mechanisms with respect to their applicability for the behavioural customisation of state machine models. Based on this information, a novel approach is presented that combines a set of variability realisation mechanisms and thereby enables open and stepwise customisation, systematic reuse and separation of concerns. Concretely, the method enhances a framework approach with model manipulation capabilities and mixin composition while also supporting conditional compilation and conditional execution. Moreover, the thesis demonstrates that compatibility can be ensured by combining constructive and analytical methods, namely feature orientation and conformance testing. Finally, feasibility and soundness of the elaborated solution concept are demonstrated using a proof of concept implementation that has already been applied to a real-world project in scope of the ELT program.

KEYWORDS: Software Product Line Engineering (SPLE), Behavioural Customisation, State Machines, Statecharts, Component Life Cycle, Extension, Refinement, Variability, Compatibility.

Acknowledgements

Throughout this thesis project, I have received a great deal of support and guidance from many people.

First, I would like to thank Dr.-Ing. Martin Becker and Andreas Schäfer from the Fraunhofer Institute for Experimental Software Engineering (IESE). Their expertise and support was invaluable and this work could not have been realised without their involvement. Their comments, ideas and suggestions pushed the quality of my work to a higher level.

Next, I would like to acknowledge my colleagues and co-workers from the European Souther Observatory (ESO) for their support and valuable input. In particular, I would like to highlight my group lead Bogdan Jeram who gave me the opportunity to work out a solution for an interesting, real-world problem. Knowing that the work you do has an impact and that the outcomes will be used in the ”World’s Biggest Eye On The Sky” is a great motivation. Moreover, I would like to thank Calle Rosenquist, Gianluca Chiozzi and Luigi Andolfato who provided helpful advice but also critical feedback.

I could not have completed this thesis without the support of my friends and fellow software engineers David Raneburger and Stefan Klikovits who provided stimulating discussions, important feedback and also mental support.

Last but not least, I would like to thank my family for their continuous support and encouragement throughout this very intense and sometimes also exhausting endeavour. I am very grateful and I know that I could not have completed this thesis without their support and patience.

Table of Contents

Table of Contents . . . . **I**

List of Abbreviations . . . . **IV**

List of Tables . . . . **V**

List of Figures . . . . **VI**

List of Listings . . . . **VIII**

1. Introduction **1**

1.1. Motivation 1
1.2. Problem Statement 2
1.3. Research Goals 2
1.4. Research Questions 3
1.5. Research Scope and Limitations 3
1.6. Research Approach 4
1.7. Thesis Structure 4

2. Foundation **5**

2.1. Software Product Lines 5
2.1.1. Core Concepts 6
2.1.2. Product Line Approaches 9
2.2. Customisation Mechanisms 11
2.2.1. High-level Techniques 11
2.2.2. Characterisation of Mechanisms 11
2.2.3. List of relevant Mechanisms 13
2.3. Behavioural Modeling with State Machines 21
2.3.1. Finite State Machines 21
2.3.2. State Diagrams 21
2.3.3. Harel Statecharts 23
2.3.4. UML State Machines 23
2.3.5. SCXML State Machines 25
2.4. Behavioural Consistency and Compatibility 28
2.4.1. Types of Behavioural Consistency 29
2.4.2. Ensuring Compatibility 30

3. Concept **32**
3.1. Concretisation of the Problem Statement 32
3.1.1. ELT Control Software Development 32
3.1.2. Component-based Architecture 36
3.1.3. Behavioural Customisation 39
3.2. Detailed Analysis 43
3.2.1. Customisation Strategies 43
3.2.2. Reuse Strategies 44
3.2.3. Customisation Scenarios 44
3.2.4. Customisation Operations 48
3.2.5. Compatibility 53
3.3. Requirements and Constraints 56
3.4. Examination of Mechanisms 57
3.4.1. Framework Approach 57
3.4.2. Conditional Execution 61
3.4.3. Model Annotation 64
3.4.4. Model Superimposition 66
3.4.5. Model Inheritance 69
3.4.6. Frame Technology 72
3.4.7. Delta Orientation 77
3.5. Solution Concept 82
3.5.1. Design Decisions 82
3.5.2. High Level Design 86
3.5.3. Customisation Workflow 89

4. Realisation **91**
4.1. Use Case for Realisation 91
4.2. Common Infrastructure 92
4.2.1. Component Runner 92
4.2.2. Support Libraries 93
4.3. Life Cycles and Extensions 94
4.3.1. Defining a Basic Life Cycle 94
4.3.2. Extending a Basic Life Cycle 99
4.3.3. Defining a Life Cycle Extension 103
4.3.4. Defining a Parametrised Life Cycle 104
4.3.5. Defining Life Cycle Aliases 107
4.4. Creating Custom Applications 108
4.4.1. Life Cycle Selection 108

4.4.2. Custom Business Logic 109
4.4.3. Component Invocation 109
4.5. Conformance Testing 110
4.5.1. Reusable Test Assets 110
4.5.2. Test Selection and Invocation 114

5. Conclusion 115
5.1. Results 115
5.2. Discussion 116
5.3. Future Work 117

References 118

Appendices 122
A. Statechart Inheritance in Rhapsody 122
B. State Machine Model Data Structure 126
C. State Machine Model Definitions 127

List of Abbreviations

AIV	Assembly, Integration and Verification.
ALMA	Atacama Large Millimeter/Sub-Millimeter Array.
AO	Adaptive Optics.
API	Application Programming Interface.
BSM	Behavioural State Machine.
CSP	Communicating Sequential Processes.
ELT	Extremely Large Telescope.
ESO	European Southern Observatory.
FOSD	Feature-Oriented Software Development.
FSM	Finite-State Machine.
GUI	Graphical User Interface.
ICD	Interface Control Document.
LSP	Liskov Substitution Principle.
MDA	Model Driven Architecture.
MDE	Model Driven Engineering.
OMG	Object Management Group.
PSM	Protocol State Machine.
RTC	Real Time Computer.
SCXML	State Chart XML.
SOA	Service Oriented Architecture.
SPLE	Software Product Line Engineering.
SQA	Software Quality Assurance.
SysML	Systems Modeling Language.
UML	Unified Modeling Language.
VLT	Very Large Telescope.
W3C	World Wide Web Consortium.
XML	Extensible Markup Language.

List of Tables

Table 1: SCXML libraries and tools. 27
Table 2: Supported customisation operations in scxml4cpp. 68
Table 3: Supported customisation operations in Rhapsody. 71
Table 4: Model import and export API. 78
Table 5: Model manipulation API. 78
Table 6: Requirements coverage matrix. 83
Table 7: Customisation heuristics. 90

List of Figures

Figure 1: Artist's impression of final telescope design. [ESO20b] 1
Figure 2: Basic idea of SPLE. [Bec17] 5
Figure 3: Asset types. [Bec17] 6
Figure 4: Feature model example. [LKL02] 8
Figure 5: Product line approaches. [Bec17] 9
Figure 6: SPLE techniques. [Bec17] 11
Figure 7: Interacting software components. [ESO21b] 17
Figure 8: Collaboration-based design. [Ape+13] 18
Figure 9: Sensor node realisation with frame technology. [Pat11] 19
Figure 10: General shape of a delta module. [Sch+10] 20
Figure 11: Moore machine with effects on states. 22
Figure 12: Mealy machine with effects on transitions. 22
Figure 13: Harel statechart with hierarchy, orthogonal regions and broadcasts. . . . 23
Figure 14: Abstract syntax of behavioural state machines. [OMG15] 24
Figure 15: Layer stack with dependencies. 33
Figure 16: Software development workflow. 34
Figure 17: Interacting software components. 36
Figure 18: State machine in its environment. 37
Figure 19: Standard interface structure. 38
Figure 20: Standard interface behaviour. 38
Figure 21: Component with actions and activities. 39
Figure 22: Alternative component life cycle. 40
Figure 23: RTC component life cycle. 41
Figure 24: Provision of custom behaviour. 45
Figure 25: Provision of custom input stage. 45
Figure 26: Predefined ELT component life cycles. 46
Figure 27: Predefined RTC component life cycles. 47
Figure 28: Stepwise component customisation. 47
Figure 29: Basic state machine model. 48
Figure 30: Extension via addition of transition. 48
Figure 31: Extension via addition of sub-diagrams. 49
Figure 32: Extension via addition of parallel regions. 49
Figure 33: Transition label refinement. 50
Figure 34: Transition refinement via addition of transient state. 50
Figure 35: Refinement of simple state. 51

Figure 36: Refinement of simple state into composite state. 51
Figure 37: Refinement of simple state into parallel state. 52
Figure 38: Refinement of composite state into parallel state. 52
Figure 39: Framework approach. 57
Figure 40: Conditional execution mechanism. 61
Figure 41: Model annotation with UML stereotypes. [ESO20a] 64
Figure 42: Model superimposition. [Ape+09] 66
Figure 43: Model inheritance. 70
Figure 44: Component structure. 86
Figure 45: Component life cycle. 87
Figure 46: Life cycle extension. 87
Figure 47: Example customisation. 88
Figure 48: Life cycle expression. 89
Figure 49: Customisation workflow. 89
Figure 50: Use case for realisation. 91
Figure 51: Model inheritance: base model. 122
Figure 52: Model inheritance: first specialisation. 123
Figure 53: Model inheritance: second specialisation. 124
Figure 54: Model inheritance: third specialisation. 125

List of Listings

Listing 1. Conditional execution using feature toggles. 13
Listing 2. Conditional compilation using preprocessing. 14
Listing 3. Conditional compilation using templates. 14
Listing 4. Example SCXML state machine model. 26
Listing 5. Framework code. 58
Listing 6. Application specific code. 59
Listing 7. SCXML model with guard condition. 62
Listing 8. Framework code with guard method. 62
Listing 9. Model superimposition with scxml4cpp. 67
Listing 10. Variant module for feature selection. 73
Listing 11. Variant module for feature ”Enabling”. 73
Listing 12. Core module with basic state machine model. 74
Listing 13. Definition of the basic model. 78
Listing 14. Addition of a specific feature. 79
Listing 15. Core and delta module defintion and selection. 80
Listing 16. Generic component runner method. 92
Listing 17. Basic life cycle: structure. 94
Listing 18. Basic life cycle: events. 94
Listing 19. Basic life cycle: commands. 95
Listing 20. Basic life cycle: model builder. 96
Listing 21. Basic life cycle: business logic interface. 97
Listing 22. Basic life cycle: output stage. 97
Listing 23. Extended life cycle: structure. 99
Listing 24. Extended life cycle: events. 99
Listing 25. Extended life cycle: commands. 99
Listing 26. Extended life cycle: model builder. 100
Listing 27. Extended life cycle: business logic interface. 101
Listing 28. Extended life cycle: output stage. 102
Listing 29. Life cycle extension: structure. 103
Listing 30. Life cycle extension: model builder. 104
Listing 31. Parametrised extension: structure. 104
Listing 32. Parametrised extension: events. 105
Listing 33. Parametrised extension: model builder. 106
Listing 34. Life cycle alias definition. 107
Listing 35. Life cycle selection. 108
Listing 36. Custom business logic. 109

Listing 37. Component invocation. 109
Listing 38. Partial mock for business logic. 110
Listing 39. Mock for state changed callback. 111
Listing 40. Template fixture class. 111
Listing 41. Test suite definition. 112
Listing 42. Typed test case. 112
Listing 43. Test case registration. 113
Listing 44. Conformance test invocation. 114
Listing 45. State machine model data structure. 126
Listing 46. Basic life cycle without omissions. 127

1. Introduction

1.1. Motivation

The European Southern Observatory (ESO) is developing software platforms, toolkits and frameworks that are used by the organisation and by various international partners to implement custom control software for specific sub-systems and scientific instruments of the upcoming Extremely Large Telescope (ELT).

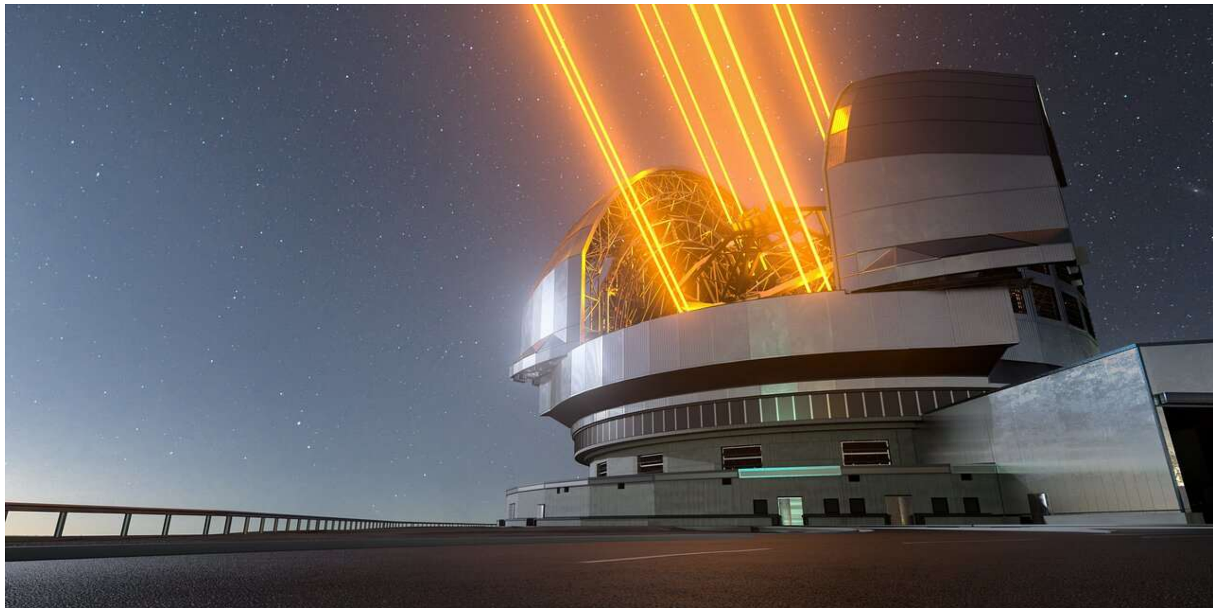


Figure 1: Artist's impression of final telescope design. [ESO20b]

ESO decided to constrain the solution space by providing reference architectures, software platforms and standardised interfaces with well-defined structure and behaviour to avoid spending a significant portion of the Assembly, Integration and Verification (AIV) phase in "Integration Hell" [Cun09] and to ensure compatibility and maintainability for decades to come. At the same time, the system must also be kept open for extension and future evolution so that developers have enough freedom to implement their custom solutions.

ESO makes extensive use of state machines to define life cycles and behaviour of software components. Application developers create custom solutions by customising pre-defined state machine models according to their needs but to ensure interoperability and integrability they also have to maintain a certain degree of compatibility.

The purpose of this thesis is to address the issues of behavioural variability and compatibility and to come up with methods and tools that enable telescope and instrument software developers to create software components that are both customisable and compatible.

1.2. Problem Statement

Building a software system that is easy to integrate and at the same time open for extension, customisation and future evolution is a non-trivial task. Aspects of this problem have already been studied in the field of Software Product Line Engineering (SPLE) that focuses on creating coherent product families consisting of individual products with common and variable features. Additionally, software and systems modeling tools, such as *MagicDraw*, provide support for variant creation and management.

However, a thorough investigation of currently available methods and tools revealed that there is no holistic way of managing variability on both the structural and the behavioural level. Current methods and tools work reasonably well for structural variability, but their support for behavioural variability is limited [TSG08].

Moreover, these methods and tools only work well if detailed information about variation points and variants is available upfront. This is not the case for the ELT program, which does not apply an established product line engineering approach due to its unique nature, large scale and distributed development. In particular, variation points and variants that need to be implemented by external application engineering teams are often unknown to framework developers.

This thesis focuses on the behavioural customisation of state machine models, taking systematic reuse as well as preservation and validation of compatibility into account. The thesis contributes to an approach that supports the customisation of software component life cycles in context of the ELT control system.

1.3. Research Goals

The main goal of this thesis is the development of a solution concept that enables behavioural customisation of state machine models while supporting stepwise refinement, systematic reuse as well as preservation and validation of compatibility. The concept shall be based on thorough research of currently available variability and customisation mechanisms. The main goal can be broken down into the following high-level goals:

- **G1**: Obtain practical and theoretical understanding on customisation mechanisms, behavioural modeling with state machines, behavioural consistency and relevant technologies and tools.
- **G2**: Elicit and characterise customisation scenarios and constraints by studying concrete use cases and by reviewing related literature.

- **G3**: Evaluate available variability mechanisms regarding their applicability using the information from G2.
- **G4**: Elaborate a solution concept that covers the concrete use cases by selecting or combining different customisation mechanisms.
- **G5**: Demonstrate the soundness of the solution concept by implementing a proof of concept that is used to solve a real-world problem.

1.4. Research Questions

The following research questions can be derived from the problem statement and from the research goals. The questions shall be discussed and answered in the scope of this thesis.

- **Q1**: Which variability mechanisms can be used to customise component life cycles and state machine models in an open and stepwise manner while enabling systematic reuse?
- **Q2**: On which level or representation can these mechanisms be used (graphical model, textual representation, graph representation, source code)?
- **Q3**: How can existing methods, mechanisms or tools be used to provide sufficient flexibility?
- **Q4**: How can behavioural consistency and compatibility be ensured?

1.5. Research Scope and Limitations

The goal of this thesis is not to develop a production ready solution but to provide a proof of concept that can be used as a basis to create a production ready solution that can be integrated in concrete ELT software frameworks. Therefore, it is important that the elaborated concepts, methods and tools are not only valuable from an academic point of view, they should also be usable in practice.

Even though this thesis talks about state machine models the focus of the thesis is not on graphical modeling with the Unified Modeling Language (UML) or the Systems Modeling Language (SysML). The thesis does also not focus on code generation using some modeling tool. The primary focus of this thesis is on behavioural customisation of state machine models in a product line or feature-oriented context where different software component life cycles that are characterised by different state machine models need to be supported.

1.6. Research Approach

The research approach used in this thesis project is a combination of the Constructive Research Methodology from [Luk03] and the Quality Improvement Paradigm from [WHH06].

The approach consists of the following steps that can also be repeated iteratively:

1. **Find a Problem.** Find an interesting and practically relevant problem that also has the potential for theoretical contribution.
2. **Understand.** Establish a baseline and obtain a practical and theoretical understanding of both the concrete problem and the topic area. This includes the elicitation of the state of practice via e.g. interviews or observations and a literature review to get an overall picture of the relevant body of knowledge.
3. **Set goals.** Define quantifiable goals that can later be used to validate the solution.
4. **Innovate.** Use the obtained knowledge to construct an innovative solution concept and implement a proof of concept solution.
5. **Analyse.** Analyse the solution and demonstrate that it works. Examine the scope of applicability and generalise. Reason how the solution can be applied to other problem domains or organisations. Show theoretical contributions and novelty.

1.7. Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 introduces the body of knowledge that represents the foundation of this thesis. This includes relevant engineering approaches, concepts, techniques and tools.

Chapter 3 makes use of the information from Chapter 2 and elaborates a solution concept. This is done by summarising the state of practice at ESO, analysing the problem in detail and exploring various solution approaches. Finally, a concrete solution concept is presented.

Chapter 4 demonstrates soundness and feasibility of the elaborated solution concept by applying the developed methods and techniques to a concrete use case from the ELT project. It also explains important and interesting implementation aspects.

Chapter 5 summarises the results and provides an outlook on future activities.

2. Foundation

2.1. Software Product Lines

A *product line* is defined as "a set of products in a product portfolio of a manufacturer that share substantial similarities and that are, ideally, created from a set of reusable parts" [Ape+13, p. 4].

SPLE is a software development methodology that applies product line thinking to software system development. It allows creating custom software products with reduced cost, improved quality and shorter time to market. This is achieved by developing a portfolio of reusable *assets* that can be combined into different products. Such assets are reusable artefacts of various types such as architecture, design, specification, source code, documentation, tests, etc.

A *software product line* is defined as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of assets in a prescribed way [CN01].

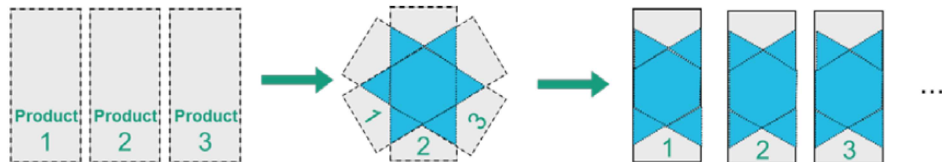


Figure 2: Basic idea of SPLE. [Bec17]

Figure 2 shows the basic idea of SPLE. Assets that are common to multiple products are identified and reused for the whole product portfolio. In order to instantiate a new custom product only product specific parts must be developed and combined with the reusable core assets.

Clearly this approach comes at a price, it requires a significant upfront investment that is larger than the cost of developing a single, custom product. This is because the reusable parts need to be identified, designed, implemented and managed. But the approach pays off in the long run, the positive effects of reuse will become especially apparent if the number of similar but custom products in the family is large enough and growing with time [Bec17].

2.1.1. Core Concepts

This section introduces important core concepts of SPLE that are relevant in the scope of this thesis.

Domain and Application Engineering

In SPLE the engineering process is divided into two main activities: domain engineering and application engineering. While *domain engineering* focuses on aspects that are common to the whole product family and provides reusable domain assets, *application engineering* creates concrete products by reusing existing domain assets and by combining them with application specific assets.

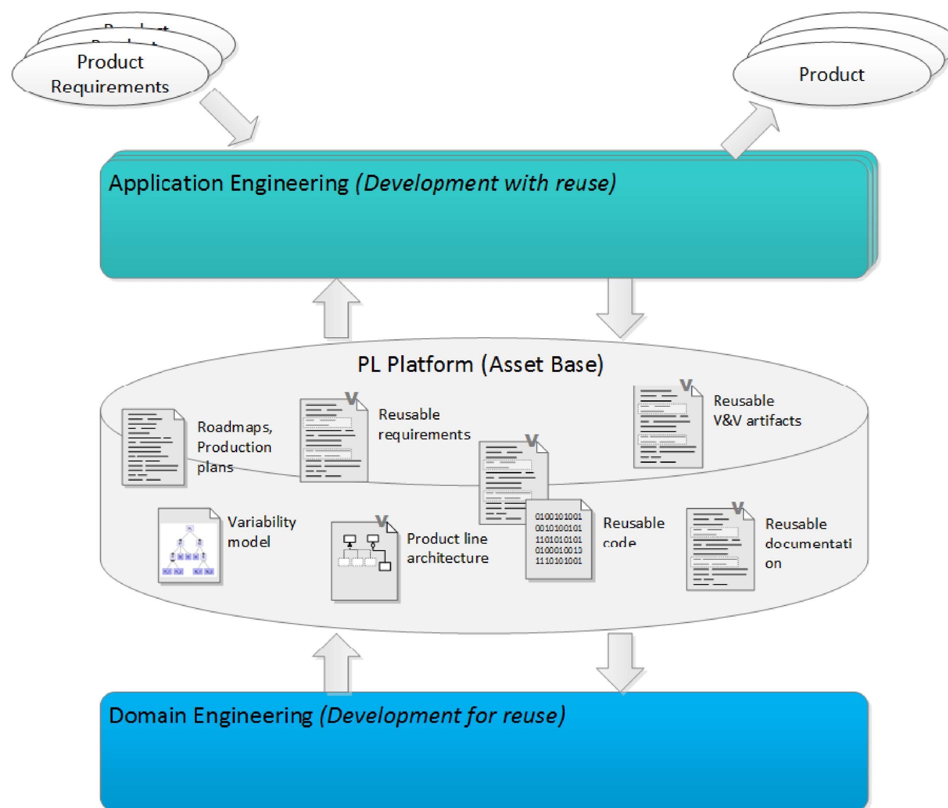


Figure 3: Asset types. [Bec17]

Note that domain and application engineering can be performed by the same team or by separate teams of *domain engineers* and *application engineers*, depending on the organisation structure and on the size of the project.

Product Line Platform

The *product line platform* contains reusable *domain assets*. The platform is populated and managed by domain engineering and it enables application engineering teams to efficiently and cheaply instantiate new products by selecting and combining provided domain assets.

In SPLE not only software artefacts can be reused, other assets can have a high reuse potential too and are thus good candidates for reuse. Figure 3 shows examples for different domain assets such as requirements, designs, conventions, user manuals, functional or variability models and supporting tools.

Domain Architecture

According to ISO/IEC 26550:2015, the *domain architecture* is defined as the "core architecture that captures the high-level design of a software and systems product line including the architectural structure and texture (e.g. common rules and constraints) that constrains all member products within a software and systems product line" [ISO15]. It is also referred to as *reference architecture* or *product line architecture*.

Variability

In a software product line, characteristics that differ among different members or products are called *variability* [ISO15]. While *external variability* can be perceived end-users, *internal variability* remains hidden since it mainly concerns implementation details.

Other important concepts in the scope of variability are *variation points* and *variants*: While variation points show which parts of the product line vary, variants are possible alternatives that can be used to realise variation points. Usually at least one variant is associated to each variation point. [ISO15]

Feature Orientation

Feature-Oriented Software Development (FOSD) is a software development paradigm that focuses on the construction and customisation of large-scale software systems where the concept of a feature plays a central role. The paradigm is also closely related to SPLE.

The reference model for product line engineering and management defines a *feature* as an "abstract functional characteristic of a system of interest that end-users and other stakeholders can understand" [ISO15].

Kwanwoo et al. describe features as "prominent and distinctive concepts or characteristics that are visible to various stakeholders" [LKL02].

In FOSD, software systems are being decomposed in terms of provided features. The goal of this decomposition is the construction of variants of the software that are tailored to

specific needs of the user. By selecting specific features from a set of available features, it is possible to generate or instantiate different software variants that share common features but differ in specific features.

Feature Models

Kwanwoo et al. describe *feature modeling* as an "activity of identifying externally visible characteristics of products in a domain and organising them into a model called a feature model".

A *feature diagram* is a compact, graphical representation of a feature model that captures structural and conceptual relationships between individual features. Typically, features are arranged hierarchically. [LKL02]

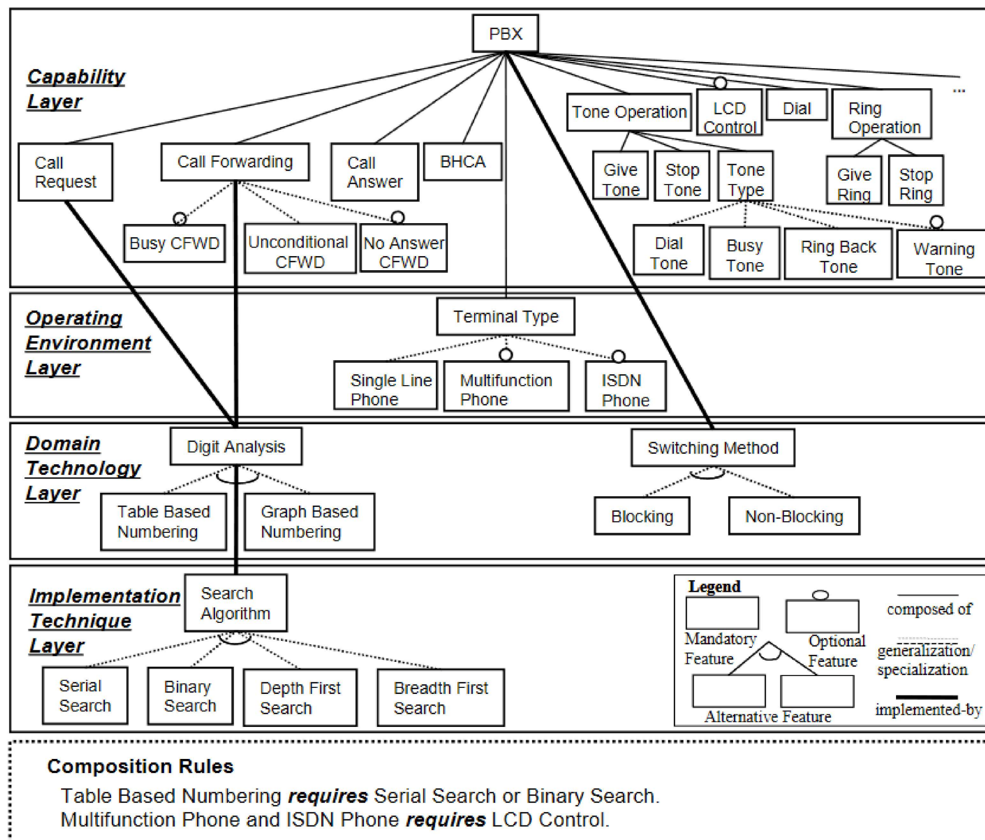


Figure 4: Feature model example. [LKL02]

Figure 4 shows a *feature model* that classifies product features in terms of capabilities, domain technologies, implementation techniques and operating environments. The model defines common features that are *mandatory*, as well as *optional* and *alternative* features.

In this context, Prehofer introduces different types of *feature interactions*. While contradictory or incompatible features cannot be used together, certain features may have to

be adapted in the presence of others. Additionally, features may complement each other, e.g. if two complementary features are combined, additional functionality may be required that is not present if either feature is added separately. [Pre04]

2.1.2. Product Line Approaches

Software product lines can be based on different reuse approaches. Figure 5 provides an overview of practical reuse approaches for the realisation of software product lines.

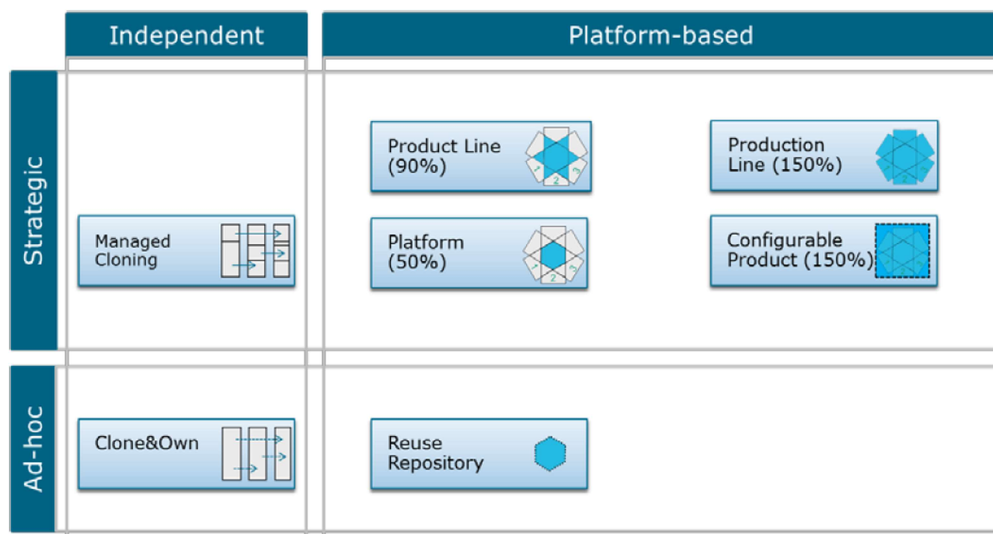


Figure 5: Product line approaches. [Bec17]

Clone-and-Own is an ad-hoc reuse approach where assets are copied and modified. Even though this approach brings the general benefits of reuse, it does not scale well. The initial advantages are lost with time due to an ever-increasing cost of maintenance. For this reason Clone-and-Own should only be used for small projects or for initial prototyping.

Managed Cloning is a light-weight approach that attempts to remedy the problems of Clone-and-Own by introducing minor management activities. Maintaining explicit traces between cloned assets helps to follow, compare and align clones during maintenance.

Reuse Repository. This light-weight approach introduces a reuse repository containing a collection of frequently used assets. Usually the repository starts as a hodgepodge of useful stuff that is put together without much deliberate planning. Since the assets in the repository are typically not designed with all the variability considerations in mind, this approach causes maintainability issues with increasing scale.

Platform. In a platform approach, the content of the product line platform is deliberately planned. It only contains domain assets that are common to all members of the product

family. Since the product line platform does not contain any variability, all custom assets must be developed and added by application engineering teams.

Product Line. A platform approach can be evolved into a product line approach by adding assets to the product line platform that are only common to some members of the product family. In this case the product line platform covers more reusable assets but it now also contains variability that needs to be addressed by application engineering teams.

Production Line. Here the product line platform contains all common and all product specific assets from the entire product family. There is no need for application engineering to develop custom assets any more, new products are instantiated by selecting and combining existing assets.

Configurable Product. There is only a single product that contains all the variability of the entire product family. The product can be configured to a specific flavour at run-time by providing configuration information, e.g. in the form of configuration files. When this approach is used, the configurable product is the product line platform and application engineering just needs to provide valid configuration information to instantiate new product flavours.

The decision, which approach fits best for a specific product family, does not only depend on technical aspects. According to *Conway's Law*, organisational constraints and the selected development process have a major impact as well. This means that certain organisational structures may doom certain product line approaches to fail or at least to be totally inefficient. [SPM19]

2.2. Customisation Mechanisms

This section introduces important customisation mechanisms and techniques that are relevant in the scope of this thesis.

2.2.1. High-level Techniques

According to Becker, there are four basic techniques that can be used to realise variation on a high level of abstraction:

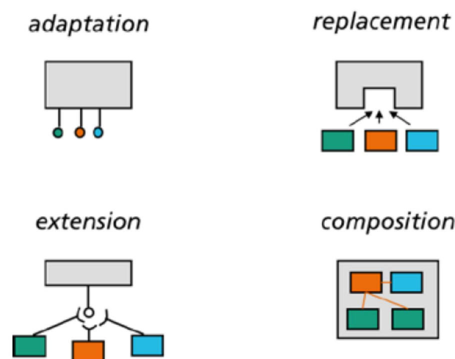


Figure 6: SPLE techniques. [Bec17]

Adaption. Architectural elements offer interfaces to adjust their behaviour. Examples are configuration files, parameters and patches.

Replacement. Several implementations that adhere to a specification are available, a specific implementation is chosen to be used for a concrete application.

Extension. New assets can be added using generic interfaces that allow the addition of elements. The added elements then may use services provided by the infrastructure. A well known example for extension are plug-in mechanisms.

Composition. The composition of a system can be changed by adding or removing assets and changing their interconnections. Examples for this approach are service-oriented or component-based systems that can be deployed in different service configurations. [Bec17]

2.2.2. Characterisation of Mechanisms

Zhang et al. introduced a characterisation of different variability mechanisms in various aspects:

- **Binding Time.** The time in the software life cycle when variability is bound to a certain product variant. This can happen at *construction time* or at *run-time*.

While mechanisms with *early binding* resolve variability early, mechanism with *late binding* provide more flexibility and thus support dynamic adaption.

- **Granularity.** The granularity of variants differs depending on the applied customisation mechanism. While some mechanisms are text-based and thus support *any* granularity of code-based assets other mechanisms require variants of a certain form or size (e.g. method, class, file). While mechanisms that support any granularity are more flexible, mechanisms with *limited* granularity are usually easier to maintain because they are more disciplined and well-structured.
- **Explicit Variation Points.** While some customisation mechanisms have *explicit* variation points that are visible in the code (e.g. via if statements) other mechanisms have *implicit* variation points that cannot be as easily spotted. For large-scale software systems, mechanisms with implicit variation points can cause challenges in development and maintenance.
- **Variant Isolation.** Depending on the applied customisation mechanism, variants for each variation point can either live together in a shared source file or they can be kept in separate files or modules. If variants share the same file or module, the code is more integrated, but also more complex. Variant isolation on the other hand leads to less complex but more fragmented code.
- **Open Variation.** In mechanisms that support *closed variation*, no new variants can be added to a variation point after compilation. Mechanisms that support *open variation* allow the provision of additional variants even after compilation e.g. using plug-in mechanisms. This gives external developers more freedom to customise the product in ways that have not been foreseen by domain engineering.
- **Non-Code Artefacts.** Since software systems include many types of reusable assets besides code (data files, models, etc.), it may be required to make them available as well. But not all customisation mechanism support such non-code artefacts.
- **Defaults.** Variability mechanisms that support default variants can reduce the number of variants significantly and thus simplify the variation logic. This is especially convenient for optional features. [ZDB16]

Trujillo et al. categorise SPLE approaches into two major groups depending on how variability in artefacts is expressed:

- **Compositional Approaches**, also known as **Positive Variability**, encapsulate variable parts in modular units which are assembled into a system according to a feature selection.

- **Subtractive Approaches**, also known as **Negative Variability**. Here software artefacts contain both common and variable parts. A concrete system is instantiated by removing variable parts belonging to unselected features. [Tru+10]

Note that the categorisation used by Trujillo et al. is closely related to the concept of *variant isolation* that was mentioned before.

2.2.3. List of relevant Mechanisms

This section provides a non-exhaustive list of existing customisation and variability realisation mechanisms.

2.2.3.1. Conditional Execution

Conditional execution is a simple and frequently used technique to realise variability. Developers make use of conditional statements (such as `if/else` and `switch/case`) to change the control flow of a program during run time. Conditional statements are typically controlled by configuration parameters that are set via command line options or using some sort of configuration file [Ape+13].

Fowler refers to this mechanism where the system behaviour can be modified without changing code as *feature toggles* or *feature flags* [Fow17].

```
void runAlgorithm() {
  if( featureIsEnabled("use-special-algorithm") ) {
    runSpecialAlgorithm();
  }else {
    runDefaultAlgorithm();
  }
}
```

Listing 1: Conditional execution using feature toggles.

Listing 1 shows an example for conditional execution that makes use of configuration parameters from a feature selection. Note that variation points are explicitly visible in code when using this mechanism.

2.2.3.2. Preprocessing

Preprocessors are tools that manipulate the source code before compilation by replacing or removing marked code fragments. In product line development, preprocessors are used

for *conditional compilation*, where marked or annotated code fragments in the source code are conditionally removed before compilation.

A well-known and popular preprocessor is the C preprocessor, it is used in the C and C++ languages to include header files and to realise include guards in headers. Since the preprocessor also provides statements for conditional compilation (such as `#if` or `#ifdef`) it can also be used as a variability realisation mechanism.

```
void runAlgorithm() {
  #ifdef VP_USE_SPECIAL_ALGORITHM
    runSpecialAlgorithm();
  #else
    runDefaultAlgorithm();
  #endif
}
```

Listing 2: Conditional compilation using preprocessing.

Listing 2 shows an example for preprocessing where a special algorithm is selected if variation point `VP_USE_SPECIAL_ALGORITHM` is defined. Before compilation the preprocessor removes the variant that is not selected so that the compiler only sees the selected variant. Also note that variation points are explicitly visible in code when this mechanism is used.

Template preprocessors provide powerful functionality that goes beyond the functionality of the C preprocessor. C++ templates for instance are a *Turing-complete* language, they are applied later during compilation and are aware of types.

```
template<typename T>
void runAlgorithm(T data) {
  if constexpr (has_contiguous_memory_v<T>) {
    runOptimisedAlgorithm(data);
  }else {
    runDefaultAlgorithm(data);
  }
}
```

Listing 3: Conditional compilation using templates.

Listing 3 shows a trivial example for conditional compilation using C++ templates. Depending on the provided data type `T` a special computation algorithm is selected that is optimised to operate on contiguous data.

2.2.3.3. Design Patterns

Design patterns provide general solutions to common, reoccurring design problems. Since implementing variability is a reoccurring problem, several design patterns provide guidance on how to realise variability.

The following list shortly introduces design patterns that are important for concepts and solutions presented later in this thesis. The presented list of patterns can be regarded as generally useful for the realisation of software product lines. The listed patterns are also prominently described in [Ape+13].

- **Observer Pattern.** This pattern is also known as *publish/subscribe* pattern. It provides a common solution for distributed event handling where a subject notifies registered observers of changes to its state. In SPLE the pattern can be used to realise optional features that can be implemented as observers.
- **Template Method Pattern.** This pattern is based on inheritance. The skeleton of an algorithm is defined in an abstract class that defines hooks. Application engineers customise the skeleton by providing custom implementations for the hooks in a subclass. In SPLE the pattern is used to realise alternative features by using different subclasses. Optional features can also be realised if the skeleton provides default implementations for certain hook methods.
- **Strategy Pattern.** The pattern is similar to the template method pattern but it uses delegation instead of inheritance to provide custom behaviour. A context implements a incomplete algorithm and delegates missing functionality to a strategy object. This object must implement a *strategy interface* that is defined by the context. To apply custom strategies clients only need to register strategy objects to the context. In SPLE the pattern is used to realise alternative features by providing different strategies. Optional features can only be realised by providing empty strategies. The mechanism provides better decoupling than the template method pattern because it defines an explicit interface.
- **Mixin Composition.** This pattern is used to extend classes with additional functionality in a stepwise manner. Additional functionality is implemented in so called mixin classes which are abstract subclasses that can be applied to different superclasses. In SPLE the approach is used to overcome the limitations of inheritance. Features are mapped to mixins so that they can be applied to a superclass that defines basic functionality. The application of multiple mixins to a superclass allows feature composition in a stepwise manner while separating concerns and facilitating code reuse.

The patterns listed above only represent a small fraction of existing design patterns. A longer and more comprehensive list of available design patterns including detailed descriptions can be found in [Gam+95].

2.2.3.4. Frameworks

Frameworks are sets of classes that represent an abstract design for a particular kind of application. These incomplete sets of collaborating classes can be customised by developers according to specific needs. [JF88]

Frameworks provide explicit *extension points* which can be used by application developers to provide custom *extensions* or *plug-ins*. Similar to the template-method and the strategy design patterns frameworks define the main control flow and delegate custom behaviour to available plug-ins. [Ape+13]

Johnson and Foote distinguish between *white-box frameworks* and *black-box frameworks*:

- In **white-box frameworks**, customisation is performed through subclassing. Developers have to identify extension points or template methods and then provide custom overrides for them. The possibility of overriding existing behaviour provides a lot of flexibility but it requires detailed knowledge of the internals and it also weakens modularity.
- In **black-box frameworks**, extension points are more explicit. The framework provides dedicated interfaces that can be used to register observers and strategies. Developers can add extensions without detailed understanding of internal implementations because they only need to understand the interfaces representing the extension points. While this approach simplifies understandability and increases modularity, it also limits flexibility because extension is restricted to explicit interfaces and directly accessing internals of the framework is not possible. [JF88]

2.2.3.5. Components and Services

According to Trapp and Kuhn, software components are units of composition with contractually specified interfaces and explicit context dependencies. They can be deployed independently and they are subject to composition by third parties. [TK14]

Flexible, elastic and scalable systems can be assembled by composition and deployment of different software components. Since components have contractually specified interfaces, it is possible to simply replace a component by a plug-compatible substitute.

Basic component interfaces and life cycles are typically defined in a *component model* that defines common communication, computation and composition principles that all components in a component based system must follow.

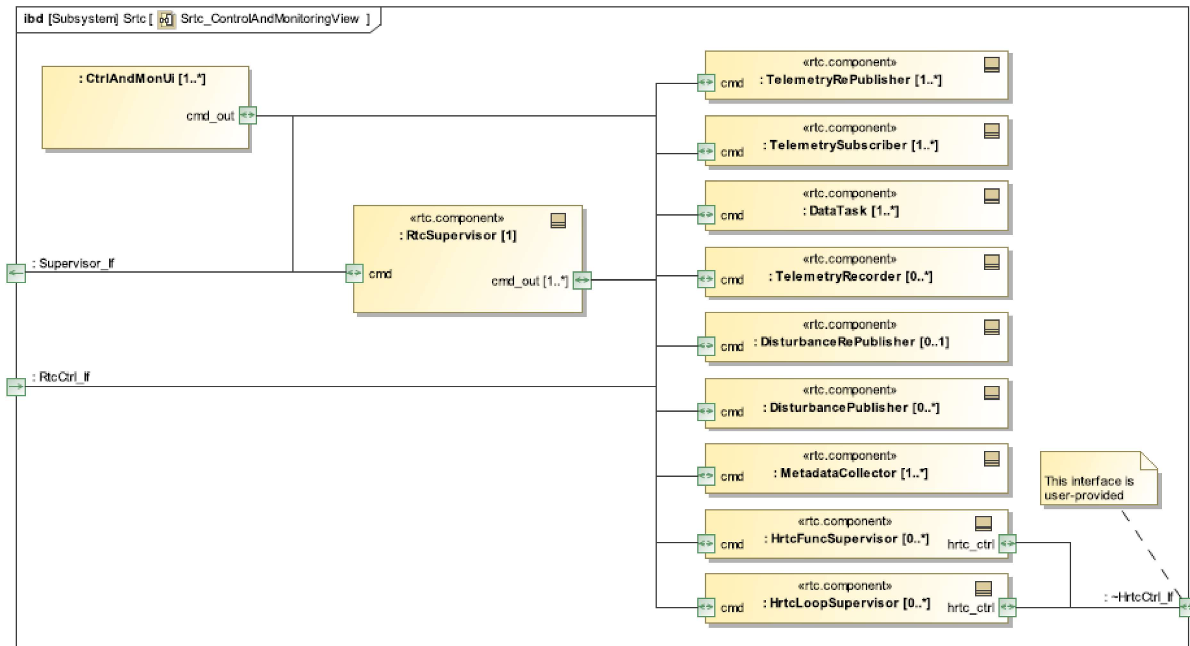


Figure 7: Interacting software components. [ESO21b]

Figure 7 shows a concrete example of interacting software components from the ELT. Components with the stereotype `<<rtc.component>>` follow the same component model which makes them compatible so that they can be used together in an Adaptive Optics (AO) Real Time Computer (RTC).

The *component infrastructure* encompasses conceptual and technical support to build, run and manage software systems that are assembled from individual components. Examples are mechanisms and tools for *service discovery*, *persistence*, *code generation*, a *development environment*, as well as *rules* and *conceptual support* to create new components from existing assets.

In an SPLE context, components and services can be used to assemble different system variants from a set of available building blocks according to a user-provided system configuration or feature selection. The configuration defines which and how many components a certain system variant requires. A deployment system takes care that components are deployed according to the selected system configuration.

2.2.3.6. Feature Modules

In FOSD, *collaboration-based designs* and *feature modules* allow the stepwise development and customisation of programs by adding new elements or modifying existing elements. While a base program provides basic functionality, different combinations of feature modules can be applied to base program to customise it according to specific needs. Typically, there is a one to one mapping between features and feature modules, so that one feature maps to one feature module.

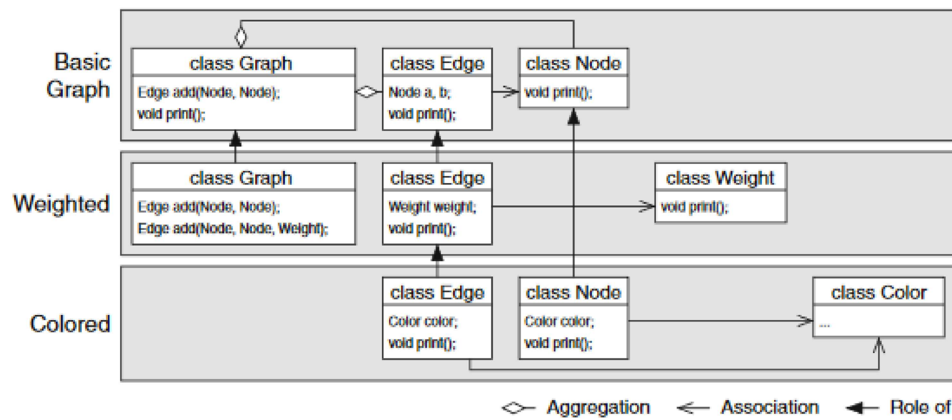


Figure 8: Collaboration-based design. [Ape+13]

Figure 8 shows a collaboration-based design consisting of a base module `BasicGraph` and two applied feature modules `Weighted` and `Colored`. Each feature module consists of a set of collaborating classes (e.g. `Graph`, `Edge`, `Node`) where each class plays a specific role within the feature module. The same classes can also be defined in multiple feature modules so that a class can play different roles depending on the containing feature module.

To realise feature modules, specific language extensions such as the *Jak* language from [Ape+13] are available. Muthig and Patzke describe another way to realise collaboration-based designs, they can be implemented using language features such as *parametrised inheritance* and *nested classes*, where the enclosing class represents the collaboration and the nested classes represent individual roles. The resulting collaborations or *mixin layers* can then be stacked via *mixin composition* to assemble the desired functionality. [MP02]

To instantiate a custom program, users simply provide a feature expression that defines which feature modules need to be applied to the base module. The combination of the base modules with the selected feature modules will be performed by a tool or if supported by some mechanism that is part of the programming language.

2.2.3.7. Frame Technology

When using *frame technology*, custom software is created from reusable and adaptable building blocks that are called *frames*. Frames are considered core assets and a hierarchy of frames is used to compose custom software.

According to Basset, frame technology has been developed to fight the liabilities of cloning. A frame can be described as a fixed theme plus the means to accommodate unforeseen variations on that theme. Frames formalise the notion of *same-as-except* to facilitate reuse and they can be applied to any language. [Bas87]

A frame contains both program code to be reused and also frame commands. Modules are split into *common modules* and *variant modules*. While *common modules* contain code that is enriched by frame technology annotations to make variation points explicit, *variant modules* contain the code that is inserted at the variation points. A *frame processor* ingests a hierarchy of modules and creates custom software products by inserting code from variant modules into the annotated spaces of the common modules.

Figure 9: Sensor node realisation with frame technology. [Pat11]

Figure 9 shows a concrete example for frame technology. The basic program is hosted in common module `main`. Variant modules such as `xpos_sensor`, `sound_sensor` or `drop_detection`

provide additional code segments for specific variation points. Variation points are annoted with `vp` and frame commands such as `insert_after` or `insert_before` specify how the variant texts are woven into the basic program. A frame processing tool can create custom software by combining the common module with a set of selected variant modules.

Patzke describes frame technology as a relatively unknown but very powerful customisation mechanism that supports unpredicted changes through *open variation*. The mechanism allows for *construction time binding*, *explicit variation points* and *variant isolation*. In general, the mechanism combines the advantages of conditional compilation and module replacement without sharing their disadvantages. [Pat11]

2.2.3.8. Delta Orientation

Delta-oriented programming is a rather novel approach that promises more flexibility for the implementation of software product lines. In delta-oriented programming, a program consists of a *core module* and a set of *delta modules*. The core module corresponds to the code of a product with a valid feature configuration and it is the starting point for the generation of other products by delta module application. A delta module specifies modifications to the core module so that other product variants can be implemented.

```
delta <name> [after <delta names>] when <application condition> {
    removes <class or interface name>
    adds class <name> <standard Java class>
    adds interface <name> <standard Java interface>
    modifies interface <name> { <remove, add, rename method header clauses> }
    modifies class <name> { <remove, add, rename field clauses> <remove, add, rename method clauses> }
}
```

Figure 10: General shape of a delta module. [Sch+10]

Figure 10 shows the general shape of a delta module. According to Schaefer et al., each delta module has a `name` and an `application condition` that specifies for which feature configurations the specified changes need to be applied. A delta module can add, modify and also remove elements on both the class level and on the class structure level.

To generate a product for a selected feature configuration, a sequence of delta modules is applied to the core module in an incremental manner. To ensure that delta modules are applied in the right order, an ordering can be defined by providing `delta names` to be applied first in the `after` clause.

Depending on the specific implementation, a delta module can both add new functionality to the core or remove functionality from it. In addition, delta modules also provide ways and means to cope with feature interaction and feature dependencies. This makes the approach more powerful than feature modules. [Sch+10]

2.3. Behavioural Modeling with State Machines

2.3.1. Finite State Machines

A Finite-State Machine (FSM) is a mathematical model of computation that performs a predetermined sequence of actions or outputs depending on a sequence of events or inputs.

Poore and Prowell describe finite state machines as mathematical objects or abstract control mechanisms that recognize inputs and produce outputs. Mathematically, a finite state machine can be represented as a five-tuple:

$$M = \langle S, I, O, T, U \rangle$$

with the members:

S - the finite set of states including the initial state

I - the finite input alphabet

O - the finite output alphabet

T - the state transition function

U - the output function [PP11]

When receiving input events from `I` a state machine can transition from a state in `S` to another according to its state transition function `T`. It will produce output events from `O` according to its output function `U`.

In addition to the algebraic form, finite state machines can also be represented using tables, graphical notations or textual representations.

2.3.2. State Diagrams

State diagrams are graphical representations of FSMs that are used to model the behaviour of systems that are represented in an abstract way using a finite set of states and a series of events that can occur.

In software development, state machine diagrams are used in various ways, e.g. for specification of behaviour, verification, validation, simulation, code generation and also for run-time interpretation. Another important application area for state diagrams is the specification of *object protocols* or *object life cycles*. According to Heckel and Küster these life cycles define the order of operations called upon an object during its lifetime [HK01].

State diagrams are *directed graphs* with vertices and edges where vertices correspond to

states from `S` and edges between states correspond to transitions from `T`.

There are two classical flavours of state machines that make use of this graph representation: *Moore* machines and *Mealy* machines.

2.3.2.1. Moore Machines

The outputs or actions of a *Moore Machine* only depend on its current state. This means that the output function `U` can be seen as a simple mapping from states to output actions.

Figure 11: Moore machine with effects on states.

Figure 11 shows a simple Moore machine with states `On` and `Off`. In addition, state `On` also contains an output action `Output`.

2.3.2.2. Mealy Machines

The outputs or actions of a *Mealy Machine* depend on its current state and also its current input. This means that output actions are associated with transitions, so the output function `U` can be seen as a simple mapping from transitions to output actions.

Figure 12: Mealy machine with effects on transitions.

Figure 12 shows a simple Mealy machine with states `On` and `Off`. There is also an output action `Output` associated to the transition between the states.

Both Moore and Mealy machines have their respective advantages and disadvantages. While Moore machines seem simpler at the first glance, Mealy machines of equivalent functionality tend to be more compact and have fewer states.

Based on classical Mealy and Moore machines, more sophisticated and powerful state machine representations that differ in syntax and semantics have been developed over time. The following sections introduce the most important state machine representations that are used in industry today and that are relevant for this thesis.

2.3.3. Harel Statecharts

Modeling non-trivial, reactive systems with traditional Moore or Mealy state machine notations usually leads to an unwanted phenomenon called "state and transition explosion", where the large number of states and transitions greatly reduces readability and understandability of the state diagram. To overcome this problem, David Harel came up with a graphical state machine notation that extends traditional finite automata by adding hierarchy, concurrency and broadcast communication [Har87].

Harel Statecharts allow the description of state machines in a more compact and expressive way while also supporting composition and modularity. They combine the ideas of Moore and Mealy automata by allowing action labels on both states and transitions. They also differentiate between actions and activities: Actions are considered instantaneous and ideally take zero time, activities on the other hand are durable and take some time to complete, they can only be associated with states. In addition, Harel statecharts also support guard conditions that can prevent transitions from being taken if the guard condition evaluates to false.

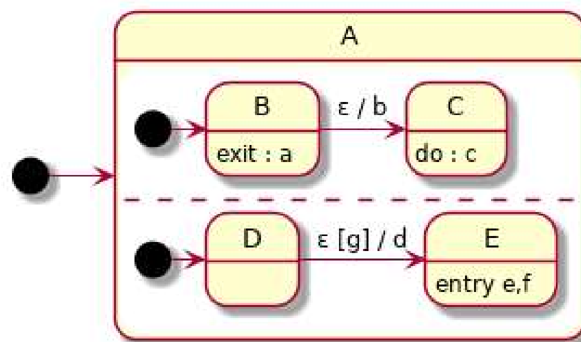


Figure 13: Harel statechart with hierarchy, orthogonal regions and broadcasts.

Figure 13 shows a simple Harel statechart. Composite states that host other states and transitions are used to realise hierarchy, orthogonal regions are used as a means to model concurrency and broadcast communication is used to propagate events to all orthogonal regions.

2.3.4. UML State Machines

With the Unified Modeling Language (UML) the Object Management Group (OMG) introduced UML state machines. They are an object-oriented variant of Harel statecharts with very similar characteristics but with some semantic differences [OMG15]. UML statecharts are supported by major software and systems modeling tools and they are the

de facto standard for modeling discrete event-driven behaviours using a graphical state machine formalism.

The UML standard distinguishes between two types of state machines that are used for different purpose: *Behavioural State Machines* and *Protocol State Machines*.

2.3.4.1. Behavioural State Machines

According to the UML specification, Behavioural State Machines (BSMs) are used to express the behaviour of entities like classes, components, subsystems or entire systems. [OMG15]

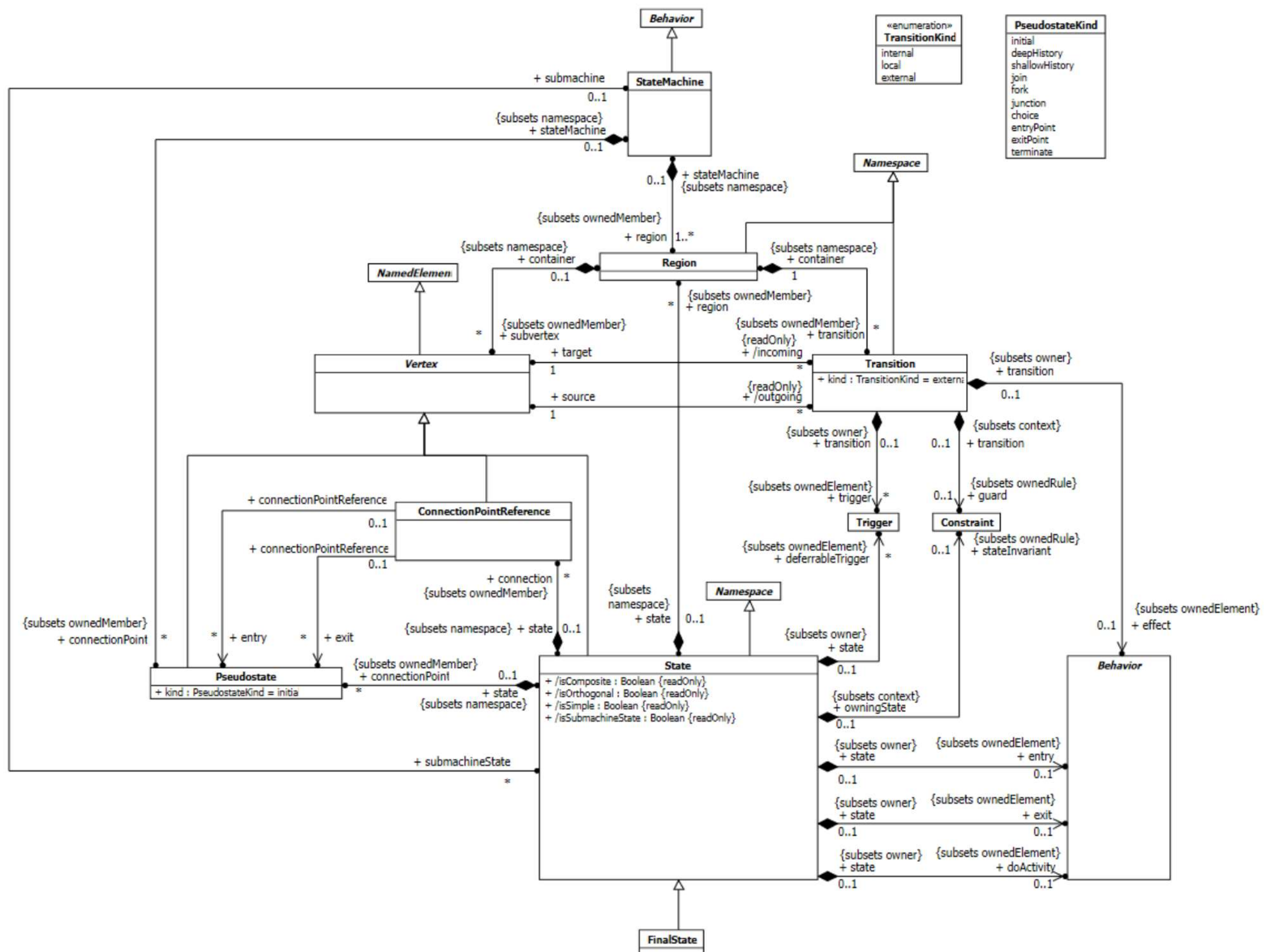


Figure 14: Abstract syntax of behavioural state machines. [OMG15]

Figure 14 provides an overview about the elements of a UML state machine and their relations. A `StateMachine` hosts at least one `Region` that in turn contains elements of type `Transition` and `Vertex`. A `State` is a special type of vertex with associated behaviour. `Behaviour` is optional and can be associated in the form of actions and activiteis (`entry`,

`exit` or `doActivity`). States can have different types: `isSimple`, `isOrthogonal`, `isComposite` and `isSubmachineState`. While simple states are usually found at the leafs of the state hierarchy, orthogonal states contain parallel regions and sub-states. A sub-machine state represents a whole state machine compressed into a single state. A `Transition` is a directed relation between a source and a target vertex with optional `Trigger`, `Guard` and `Effect` behaviour. While external transitions exit the source state, local transitions do not exit the containing composite state. State internal transitions do not leave their state either, they only execute associated behaviour when they are activated. Other vertices worth mentioning are different `PseudoStates` such as `initial`, `history`, `fork`, `join`, `choice`, `junction`, `entryPoint`, `exitPoint`, `terminate` and a special `FinalState` that is used to indicate that an enclosing region has completed.

2.3.4.2. Protocol State Machines

According to the UML specification, Protocol State Machines (PSMs) are used to express usage protocols, legal sequences of event occurrences and the order of invocations from an external perspective, therefore they are mainly associated with interfaces and ports. Since PSMs only provide an external black-box view, their states do not necessarily have to correspond to the states of the BSMs that define the inner workings of the respective class or component.

When defining PSMs, the same elements as for BSMs can be used, but there are some important differences:

- PSMs do not have associated effect behaviour, therefore states and transitions must not define any actions or activities.
- Protocol transitions allow the specification of pre-condition, trigger and post-condition. [OMG15]

2.3.5. SCXML State Machines

State Chart XML (SCXML) is an event-based state machine notation for control abstraction defined by the World Wide Web Consortium (W3C). The notation is based on Harel statecharts, and it makes use of the Extensible Markup Language (XML) to persist state machine models in a textual format.

According to the W3C specification, the SCXML notation was created because both Harel and UML statecharts were only defined as a graphical specification language, but they were lacking an XML representation [W3C15].

In addition to the textual representation, the SCXML standard also defines precise execution semantics and algorithms for runtime interpretation. This makes SCXML an attractive, human- and machine-readable interchange format that can be generated from graphical statecharts and then ingested by state machine interpreters and code generators.

2.3.5.1. Syntax

The syntax of the SCXML state machine notation is described in a publicly accessible W3C document that can be found in [W3C15].

```
<?xml version="1.0"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initial="Off">

  <state id="Off">
    <transition event="turn.on" target="on"/>
  </state>

  <state id="On">
    <initial>
        <transition target="NotOperational"/>
    </initial>

    <transition event="turn.off" target="Off"/>

    <state id="NotOperational">
      <transition event="Enable" target="Operational"/>
    </state>

    <state id="Operational">
      <transition event="Disable" target="NotOperational"/>
    </state>

  </state>

</scxml>
```

Listing 4: Example SCXML state machine model.

Listing 4 shows a simple example of a concrete SCXML state machine model. The whole state machine definition is enclosed in a `scxml` element. States *Off* and *On* with sub-states *Operational* and *NotOperational* are defined using the `state` tag. Transitions are either defined explicitly with tag `transition` or implicitly using attribute `initial`.

2.3.5.2. Differences to UML Notation

Even though both UML and SCXML statecharts are derived from Harel statecharts their abstract syntax is slightly different. While both notations share the notion of *states*, *transitions*, *events* and *guards* there are certain differences. Some major differences worth mentioning are:

- SCXML does not directly support actions, but they can be implemented using the language concept *executable content* which provides facilities for user extension.
- The notation does also not directly support activities, they can be realised using the concept of *external services*, which are executed with the *invoke* keyword.
- Regions are implemented using *compound states* and *parallel states*. A state containing other states is considered a compound state. A parallel state is used to realise concurrent execution of multiple states that reassemble orthogonal regions.

Due to these differences, it is not trivial to map a UML state machine to an equivalent SCXML state machine. Performing the mapping basically means applying a model transformation operation that requires detailed knowledge of both standards. Andolfato et al. provide a detailed description of this UML to SCXML mapping in [And+11].

2.3.5.3. Tool Support

Many tools and libraries support the SCXML interchange format because of its high portability. Table 1 shows a short, non-exhaustive list of interesting and relevant tools.

Tool	Description
COMODO	model to text transformation toolchain developed at ESO
scxml4cpp	SCXML run-time interpreter developed and used at ESO
scxmlcc	efficient SCXML to C++ compiler
Qt SCXML Editor	visual editor for SCXML statecharts
Qt SCXML Engine	run-time interpreter for SCXML statecharts
Qt SCXML Compiler	SCXML compiler that produces C++ code
state-machine-cat	tool that creates statecharts from textual models

Table 1: SCXML libraries and tools.

2.4. Behavioural Consistency and Compatibility

The ISO 25010 standard defines *compatibility* as the "degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions while sharing the same hardware or software environment".

The standard explains this characteristic in terms of two sub-characteristics:

- **Co-existence.** The degree to which a component can perform its function while sharing a common environment and resources with other components without impacting other components in a detrimental way.
- **Interoperability.** The degree to which two components can exchange information and use the exchanged information. [ISO11]

Prehofer provides another definition of compatibility that focuses on extendible feature-based systems: "Compatibility means that a new, refined system can be used in any place where the old system was and behaves in the same way" [Pre13].

Even though this second definition seems unrelated to the ISO definition, they are strongly related. In a customisable system, interoperability can be impacted in a detrimental way if components are replaced with incompatible substitutes. For this reason, the second definition focuses more on the concept of *substitutability*. It is therefore more appropriate in the scope of software product lines and customisable systems.

Another concept that is related to compatibility and substitutability is the notion of *behavioural subtyping* as described by Liskov in [LW94]. The concept states that objects of a type T may be replaced with objects of type S without altering desirable properties of a program, if S is a subtype of T. This concept was later coined into the Liskov Substitution Principle (LSP).

Another related concept from the UML is *protocol conformance*. The concept states that when a general Protocol State Machine (PSM) is refined into a more specific PSM every rule and constraint specified for the general PSM also applies to the specific PSM. The concept of protocol conformance can also be used to refine a PSM into a Behavioural State Machine (BSM) [OMG15].

2.4.1. Types of Behavioural Consistency

Stumptner and Schrefl further analyse behavioural compatibility in the scope of object life cycles and state machines. They introduced different types of behavioural consistency that can be used to compare the behaviour of customised objects and to specify which compatibility levels are required for a certain application. Note that the presented concepts are based on previous work about invocable and observable behaviour from Ebert and Engels in [EE94].

2.4.1.1. Observation Consistency

Observation consistency ensures that each instance of a subtype is observable according to the structure and the behaviour given at the level of the supertype [SS04]. In other words, a sequence of visited states (also called *life cycle occurrence*) can be observed at the level of the superclass if all elements that have been added to the subclass are ignored.

Observation consistency is an important property when modeling workflows where e.g. the current state of a process should always be visible to an operator at a high level of abstraction.

2.4.1.2. Invocation Consistency

This notion of behavioural consistency is based on the substitution principle. Stumptner and Schrefl define two types of *invocation consistency*, where one is stricter and subsumes the other:

- **Weak invocation consistency** ensures that instances of a subtype can be used the same way as instances of the supertype. In other words, any *activation sequence* (sequence of events) that is accepted by the supertype is also accepted by the subtype. E.g. a television set with video text can be used in the same way as a device without this feature.
- **Strong invocation consistency** is stricter and guarantees that instances of a subtype can be used the same way as instances of the supertype, even if operations added at the subtype level have been executed. [SS04]

2.4.2. Ensuring Compatibility

When designing and implementing software systems, behavioural compatibility can be ensured using different constructive and/or analytical approaches.

2.4.2.1. Constructive Approaches

Several papers describe techniques for behaviour-consistent inheritance and they also define rules for specialising statecharts so that compatibility is preserved.

Van der Aalst and Basten introduce four notions of inheritance that can be applied to UML state diagrams and used to construct compatible subclasses: *protocol inheritance*, which is based on *invocation consistency*, *projection inheritance*, which is based on *observation consistency*, *protocol/projection inheritance*, which is a combination of both and *life cycle inheritance*, which is a more liberal form. [AB02]

Simons et al. present a specification method that defines under which conditions a software component is behaviourally compatible. They distinguish between *interface realisation* where abstract interfaces are realised by concrete components and *specialisation* where components can be replaced by more specific components. In addition, they also define rules for statechart refinement. [Sim+02]

The superseded UML 1.5 standard provides useful heuristics how state machines can be refined. Three sets of inheritance rules are presented *subtyping*, *strict inheritance* and *general refinement*, where only subtyping aims at the conservation of compatibility [OMG03]. Note that these three rules are not longer described in the currently released UML 2.5 standard [OMG15].

Hansen et al. propose a set of rules to limit the possible statechart refinements so that certain assumptions will hold. They also introduce the notions of *abstract*, *standard*, *locked* and *virtual* states and transitions to be able to control further extension and refinement of statecharts. [HSL15]

Using such rules and guidelines, it is possible to create editors and tools that prevent developers from creating specialised statecharts that are not compatible with the basic behaviour.

2.4.2.2. Analytical Approaches

Analytical approaches can either be based on *verification* or *validation*.

Model Checking

Model checking is a *verification-based* approach in which compatibility of state machine models can be ensured by checking certain rules or criteria.

Stumptner and Schrefl present a set of five rules to check the behavioural consistency of state machine models. Depending on the selection of rules, it can be verified whether a specialisation of a state machine is observation consistent, weak invocation consistent or strong invocation consistent. [SS04]

Heckel and Kuester present techniques for analysing behavioural consistency. Using meta-model rules, UML models are transformed into a semantic domain where consistency constraints can be specified and validated using the language and tools of that domain. Communicating Sequential Processes (CSP) was used as the semantic domain. It provides a mathematical model for concurrency, is based on a simple programming notation and it is also supported by tools. [HK01]

Conformance Testing

A rather simple, *validation-based* approach to ensure behavioural compatibility and to detect incompatibilities of protocol implementations is *conformance testing*, which is sometimes also referred to as *compliance testing*. Conformance tests guide classes or components through their life cycles. The tests apply different activation sequences and assert that certain states are entered or that specific behaviour is being executed.

According to Lee and Yannakakis, conformance testing of state machines aims to determine whether an *implementation machine* correctly implements the protocol that is defined by a *specification machine* by applying a test sequence to the implementation machine and observing the outputs. [LY96]

Trenkaev et al. further distinguish between *conformance testing* and *interoperability testing*. While conformance testing is used to test if a single protocol entity conforms to its protocol specification, interoperability testing is used to check if multiple protocol entities can operate as a system. Here the objective is to see whether the implementations that already pass the conformance test can correctly interact with each other. [TKS03]

Seifert describes a comprehensive approach for conformance testing of reactive systems. Based on a formal specification using UML statecharts, test cases are generated for a previously selected input sequence. The generated tests are then used to check the functional conformance of a system under test. [Sei08]

3. Concept

This chapter describes the solution concept and all aspects that were investigated to derive it. The investigation includes the following activities:

First, the problem statement from Chapter 1 is revisited and further refined using information derived from concrete use cases, internal documents and from relevant literature. Then, the elicited information is analysed in detail to derive customisation operations and criteria for the evaluation and selection of appropriate customisation mechanisms that have been introduced in Chapter 2. Finally, the solution concept is formalised.

3.1. Concretisation of the Problem Statement

This section revisits the problem statement from Section 1.2, it provides additional context information that is later used to derive concrete requirements and constraints.

3.1.1. ELT Control Software Development

Process and Organisation

Intensive cooperation with external partners is necessary to be able to realise such a large program as the ELT. While ESO is focusing on high level design, standardisation and coordination of the entire program, the organisation also develops part of the control software in-house, such as the telescope control software as well as various domain specific platforms and frameworks. Design and development of the six scientific instruments is being carried out by international consortia, each consisting of several universities and companies. A detailed description of the entire ELT program can be found in the E-ELT Construction Proposal [ESO11].

Architecture Documentation

The high-level software design and architecture documentation for the telescope and for major subsystems and frameworks developed in-house is done using SysML and the visual modeling tool *MagicDraw*. Typically, the design documents follow a common structure and use similar viewpoints as described in the IEEE:1016-2009 standard (see [IEE09]).

However, for the design of individual instruments and for the detailed design of telescope sub-systems, architects are not obliged to use SysML and MagicDraw. Therefore, many design documents contain diagrams created with various other tools such as *Visio*, *Gliffy* or *PlantUML*. For this reason it cannot be assumed that there is a holistic, integrated model of the whole telescope available that can act as a single source of truth.

Product Line Platform

ESO makes use of SPLE techniques that support systematic reuse and stepwise customisation to achieve maintainability and to speed up software development. Currently several infrastructure libraries, toolkits and frameworks are in development that will enable internal and external development teams to build custom control software with a common reference architecture, as well as uniform structure and texture.

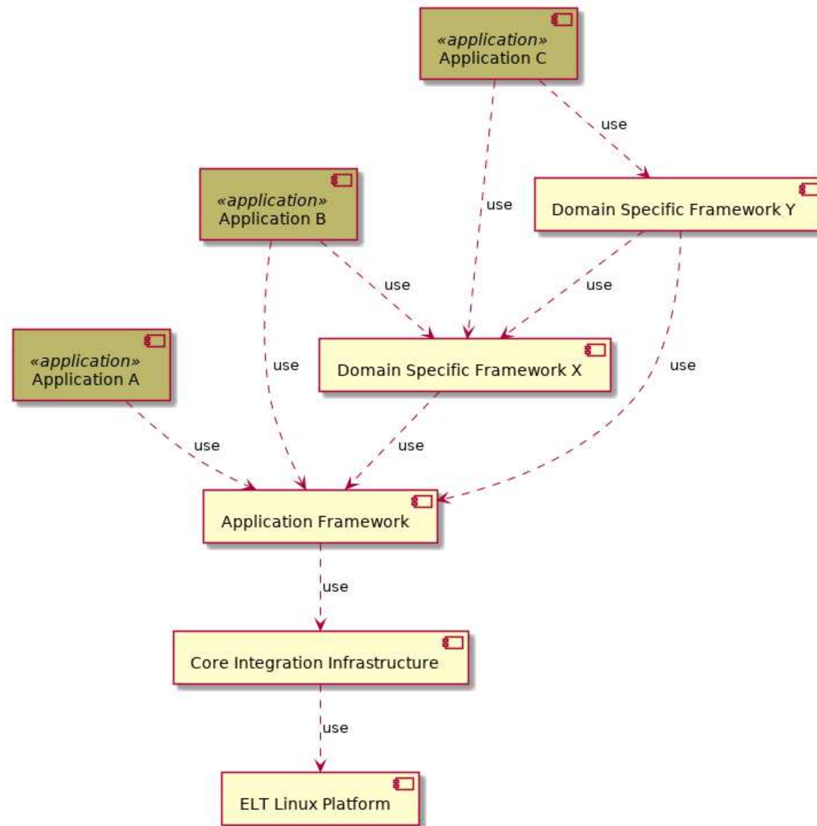


Figure 15: Layer stack with dependencies.

Figure 15 shows a dependency graph of platforms, frameworks and toolkits that are organised in different layers. While the *Core Integration Infrastructure* provides generic base functionality, the *Application Framework* (see [ESO21a]) defines a component model and a common design for ELT software components. Higher level *toolkits* and *frameworks* such as the *RTC Toolkit* (see [ESO21b]) provide more and more concrete and domain specific assets that can be reused and further customised by application engineering teams.

This separation into different layers allows creating a modular and extendible system. Domain engineering teams at ESO provide reusable assets for a certain layer without detailed knowledge of concrete end-applications. Distributed application engineering teams across the globe make use of these domain assets to create final control software applications that conform to the ESO reference architecture and design.

To facilitate reuse and to improve maintainability, it is a goal to keep domain and application assets separate. In addition, layers are split into separate packages, modules and files in order to improve coupling and cohesion. Finally, individual variants are kept separate so that they can be optimally reused and maintained.

Application Development Workflow

The primary application development workflow used in the ELT is based on classical software engineering. Project teams collect requirements, create a design and implement applications and libraries using the available stack of platforms, technologies and tools.

In addition, the application framework also provides a Model Driven Architecture (MDA) based toolchain called COMODO. It was initially developed for the Very Large Telescope (VLT) and Atacama Large Millimeter/Sub-Millimeter Array (ALMA) and it allows developers to generate state machine based application skeletons from a UML or SysML model created with a custom modeling profile [And+11].

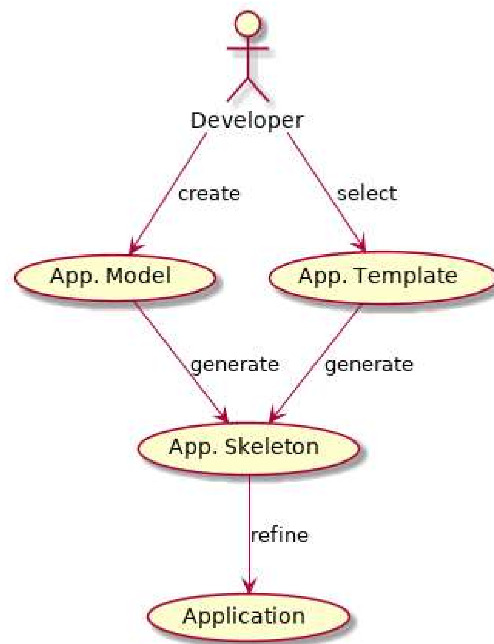


Figure 16: Software development workflow.

Figure 49 shows the two available application development workflows. The edges of the graph are used to depict activities and the vertices correspond to work products.

- When using the Model Driven Engineering (MDE) workflow, developers create component specifications consisting of interface definitions and state machine models with a graphical modeling tool. Then the model is exported into an intermediate format and ingested by COMODO which performs a model-to-text transformation. The generated application skeletons can then be used for further manual refinement.

- When using the traditional software engineering workflow, developers create applications including interface definitions and the executable state machine models manually. To speed up manual application development, the application frameworks also provides a *cookiecutter* tool that generates minimal applications from predefined templates. Developers select a specific template, run the tool and then customise the generated application skeleton according to their needs.

Even though the MDE workflow was adapted to support the ELT technology stack, its use is neither mandated nor actively recommended. A major limiting factor is that COMODO mainly focuses on the refinement-based MDE workflow without providing support for other important SPLE concerns like open customisation, systematic reuse and variability management. Since the *MagicDraw* models are kept in a different version control system than the application source code and since the generated application skeleton are frequently manually edited and committed to the source code repository, it is not clear which repository should be regarded as the single source of truth. For these reasons, several development teams decided to stick to the traditional software engineering workflow instead of generating code from a MagicDraw model.

But also the traditional workflow comes with certain problems: Creating many similar applications in a *Clone-and-Own* manner produces a significant amount of duplicated boilerplate code that needs to be maintained in the long run. Some project teams try to solve this problem by encapsulating common code as much as possible in frameworks and libraries so that the template applications can be very short and trivial programs that only contain a few lines of code.

Software Quality Assurance

The Software Quality Assurance (SQA) process used for ELT control software development is mainly based on validation. Multiple levels of manual and automated tests are run to ensure that the software was implemented correctly. While *unit tests* check correct implementation on the class level, *component tests*, *integration tests* and *system tests* ensure interoperability and integrability of the entire system.

Additionally, static code analysis tools are used and formal and semi-formal reviews are carried out to catch defects and to ensure that the software is developed according to the requirements and standards.

3.1.2. Component-based Architecture

The ELT control and supervision system is based on a Service Oriented Architecture (SOA). It is a highly distributed system consisting of a hierarchical web of interacting software components that together realise the system function. Individual software components follow a more or less sophisticated component model that defines the basic communication, computation and composition principles.

3.1.2.1. Component Interaction

Figure 17 shows a simplified example of interacting software components that are organised in a hierarchy.

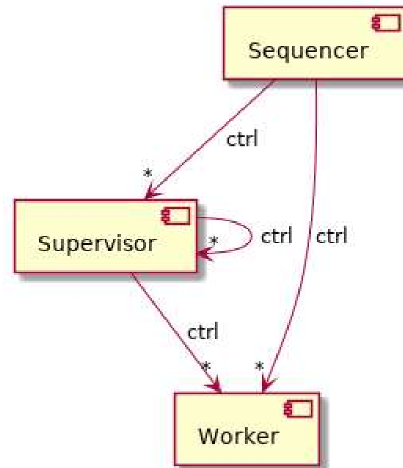


Figure 17: Interacting software components.

While `Worker` components perform computations or act as data gateways between different systems, `Supervisor` components are used to coordinate, monitor and configure other components. In this context, monitoring refers to the estimation of the overall state of the system and coordination refers to bringing a set of subordinate components to a certain state in their life cycle. Since supervisors are only aware of basic functionality common to all components, they can only coordinate other components on a coarse level. Fine granular component coordination is performed by a dedicated `Sequencer` application that steers components individually and that is aware of specific component functionality.

3.1.2.2. Component Structure

The component model, as it is defined in the application framework, foresees that ELT software component applications are based on executable SCXML state machine models

that are interpreted by the *scxml4cpp* state machine engine at runtime. The engine was developed at ESO and it is based on the state machine execution semantics defined in the W3C SCXML standard.

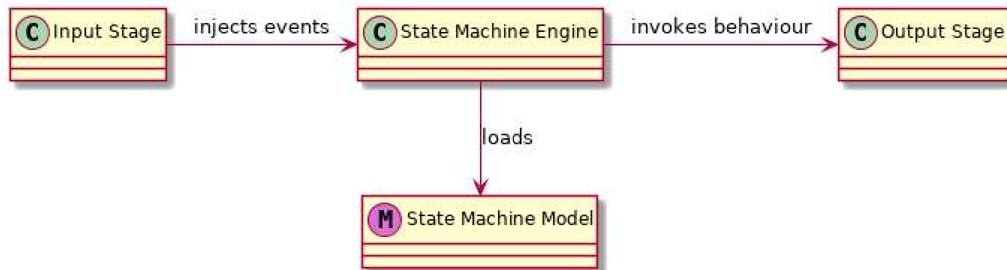


Figure 18: State machine in its environment.

Figure 18 shows the high level component structure as it is defined by the application framework.

- The `Input Stage` receives commands and translates them to input events that are injected into the state machine engine. The set of input events is closely related to the *Input Alphabet* of the state machine.
- The `State Machine Engine` acts as an intermediate unit that translates input events to output events according to a `State Machine Model` that is loaded during component start-up. The engine reacts to external and internal stimuli according to the state transition function and it invokes action and activity methods according to the output function.
- The `Output Stage` provides custom implementations for actions and activities defined in the state machine model. The set of actions and activities is closely related to the *Output Alphabet* of the state machine and the invoked methods are used to send command responses and to implement the application specific *business logic*.

3.1.2.3. Component Life Cycle

Similar to *object life cycles* that have been introduced in Section 2.3, software components in the ELT use the concept of *component life cycles* to define the allowed command sequences and also the observable sequences of visited states.

The basic life cycle, that all software components must follow, is defined in a standardised coordination and monitoring interface `stdif` that specifies the basic set of commands and the associated *interaction protocol*. The standard interface is the common denominator that enables supervisors to coordinate and monitor a set of seemingly different components.

An Interface Control Document (ICD) specifies the structural and behavioural aspects of the standard interface in a contractual way. Structural aspects include the set of accepted commands and events used for coordination, as well as topic definitions used for state observation. Behavioural aspects are defined using a protocol state machine that specifies the sequence of allowed command invocations and observable states. Note that the ICD also defines how invalid command sequences are handled, commands that are not accepted in the current state are rejected and a specific response message is sent.

Figure 19: Standard interface structure.

Figure 19 shows the structure of the `stdif`. The interface defines a set of basic commands that can be sent to the component via the network. In addition, the interface is also used to publish *state changed events* that are used to notify supervisor or Graphical User Interface (GUI) applications.

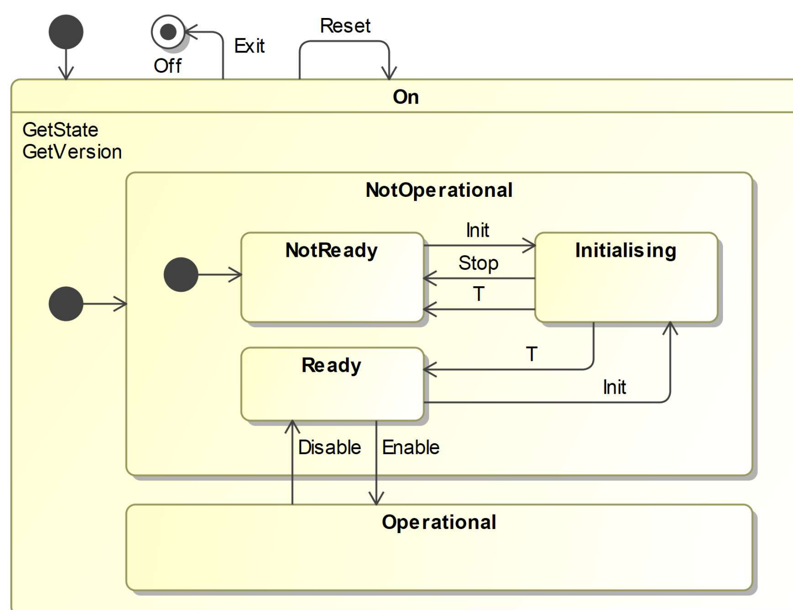


Figure 20: Standard interface behaviour.

Figure 20 shows the protocol state machine that is associated to the standard interface. State machine events correspond to the commands and states correspond to the state changed topic. In addition, different internal events are depicted with `T`.

3.1.3. Behavioural Customisation

Depending on the degree of customisation, software components can be customised in different ways. Note that also combinations of different customisations are possible.

3.1.3.1. Addition of Actions and Activities

The most simple way to customise the component behaviour is to refine the *protocol state machine* into a *behavioural state machine*. This is done by adding actions and activities to the state machine diagram. In a subsequent step, developers map these actions and activities to hard-coded methods constitute the *business logic* of the component. These methods are invoked by the state machine engine at runtime.

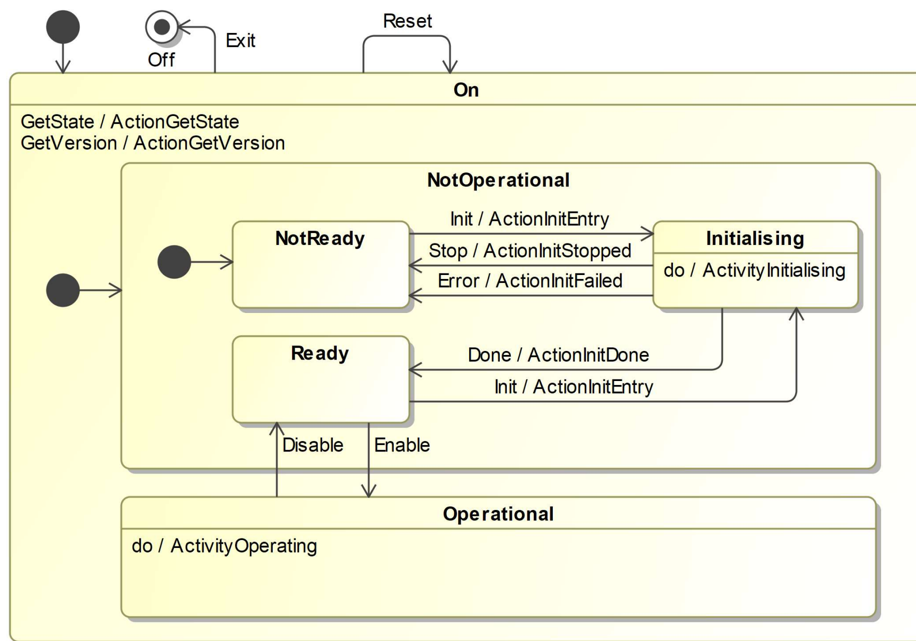


Figure 21: Component with actions and activities.

Figure 21 shows a possible refinement of the basic component life cycle. The behavioural state machine refines the protocol state machine from Figure 20 by adding actions and activities to various states and transitions. Note that the generic internal events, previously marked with `T`, have been replaced with concrete internal events `Done` and `Error`.

3.1.3.2. Selection of Variants

To provide more flexibility to application developers, the product line platform provides different variants of the standard life cycle that introduce additional states and transitions.

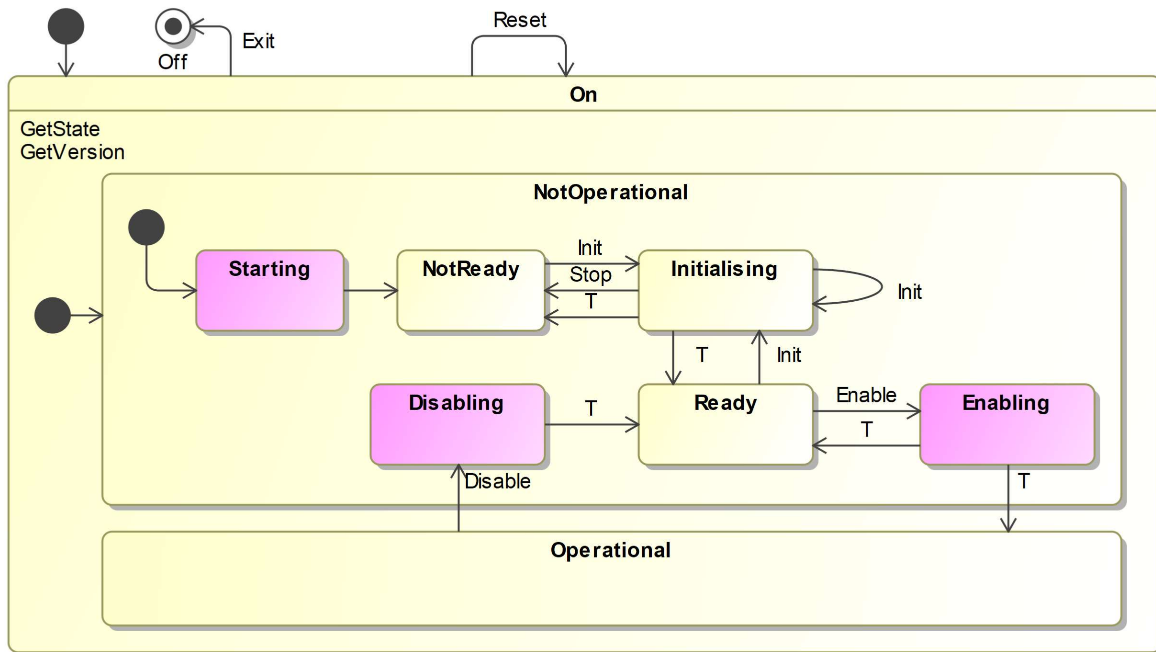


Figure 22: Alternative component life cycle.

Figure 22 shows an alternative component life cycle that introduces new transient states `Starting`, `Enabling` and `Disabling`.

Note that the alternative life cycle needs to be *invocation consistent* with the basic life cycle. This means that all basic activation sequences must also be valid for the alternative life cycles. E.g. activation sequence `Init`, `Enable`, `Disable` must still be valid for the alternative life cycle and it must still bring the component to state `Operational` and back to state `NotOperational`.

The alternative life cycles also need to be *observation consistent* with the basic life cycle, even though the sequence of observable states now also includes additional transient states and transitions. As an example, a concrete sequence of states is shown below. If newly introduced states are removed from this *life cycle occurrence*, the resulting sequence of visited states (depicted as underlined) should still be valid for the basic life cycle: `Starting`, `NotReady`, `Initialising`, `Ready`, `Enabling`, `Operational`, `Disabling`, `Ready`.

3.1.3.3. Stepwise Customisation

Domain specific frameworks such as the ELT RTC Toolkit introduce their own component model on top of the ELT component model. These high level frameworks define custom component life cycles by extending and refining the standard interface. With this approach special *RTC Components* with domain specific capabilities can be introduced that still comply to the basic component model.

In subsequent customisation steps, application engineers further customise RTC Components by configuration, extension or refinement to create final applications that can be deployed in the control system.

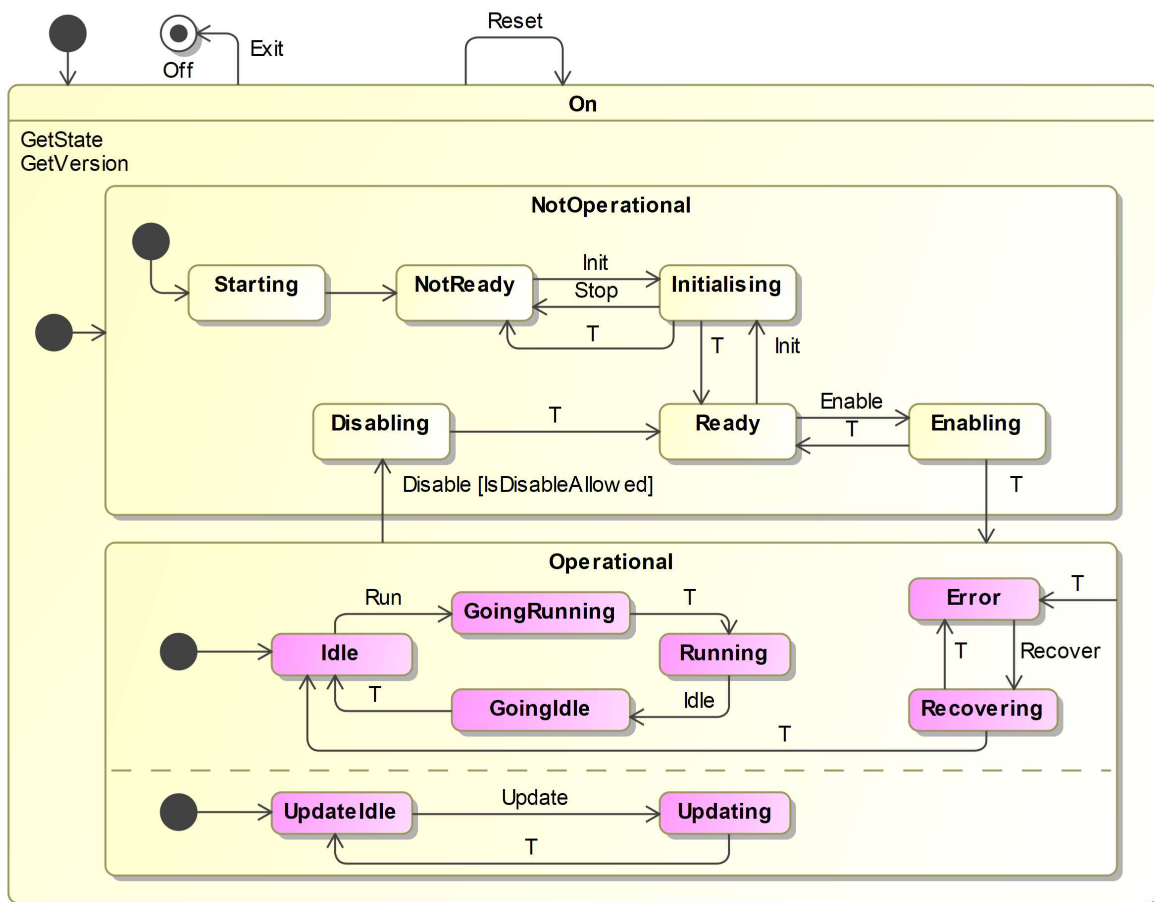


Figure 23: RTC component life cycle.

Figure 23 shows the protocol state machine of a *Runnable RTC Component* that extends the basic component life cycle with new commands `Run`, `Idle`, `Recover` and `Update`. The custom life cycle also introduces new states and transitions in state `Operational` and it refines the transition to state `Disabling` by adding a guard condition that only allows exiting state `Operational` from certain sub-states.

Note that the customisation needs to be *invocation consistent*, i.e. the basic activation sequence `Init`, `Enable`, `Disable` must also be valid for RTC components and it must still bring the component to state `Operational` and then back to state `NotOperational`.

Moreover, the customisation also needs to be *observation consistent*. Even though the sequence of observable states now also includes new sub-states in state `Operational`, the resulting life cycle occurrences still need to be observable through the standard ELT component state machine.

3.2. Detailed Analysis

In this section, the information from Section 3.1 is analysed in detail and mapped to the basic body of knowledge from Chapter 2. Important findings that are relevant for the solution concept are highlighted and discussed.

3.2.1. Customisation Strategies

By allowing a set of basic component life cycles and by letting developers further customise them, ESO is already making use of different customisation strategies.

- **Closed Variation.** This strategy is based on *negative variability* and *feature selection*. The basic component life cycle provides built-in variation points with a set of pre-defined options. By selecting specific options, a variety of alternative life cycles can be instantiated. This gives application developers flexibility to configure the component life cycle according to their specific needs.
- **Open Variation.** This strategy is based on *positive variability* and *feature composition*. While the basic life cycle and the standard interface are defined in the application framework, higher level frameworks and final applications may customise this life cycle by widening the standard interface and by extending or refining the protocol state machine that defines the component behaviour. Such changes to the component life cycle are expected and permitted as long as they do not break compatibility.

Closed variation is mainly used to support a set of well-known, alternative life cycles within a specific layer or framework. When selecting such pre-defined options, developers do not need to worry about compatibility issues because domain engineering already ensured by construction that all selectable life cycles are indeed allowed and compatible with the basic behaviour.

Open variation on the other hand is used for stepwise customisation when crossing the boundary between different layers or frameworks. Typically, a new layer that is added on top of the layer stack may both configure built-in variability of the lower layer and it may also add custom behaviour. When extending or refining the basic component life cycle in a higher layer, developers must consider compatibility issues, because their modifications can lead to incompatibility.

Currently, these strategies are being followed in an ad-hoc way. The solution concept should formalise how both strategies can be applied systematically, and it should also explain how the strategies can be combined.

3.2.2. Reuse Strategies

Since ESO already makes use of customisation strategies that require both positive and negative variability, it is appropriate to follow a *product line approach*. In this case, the product line platform provides both common and product specific assets. Another reasonable way to go is to start with a *platform approach* and gradually evolve it into a product line by adding more common and shared assets.

The stack of platforms and frameworks facilitates stepwise customisation and systematic reuse and it can be regarded as a platform of platforms. While lower layers provide common or basic functionality, higher layers focus on domain specific aspects and provide more concrete functionality.

The component-based and service-oriented system architecture also facilitates reuse by providing a set of re-usable software components and services that can be deployed in different system configurations.

Reuse of Executable Models

While external component interfaces are specified using protocol state machines, concrete component implementations make use of behavioural state machine models that refine protocol state machines by adding custom actions and activities.

These behavioural state machine models are also used at run-time. At component startup SCXML models are ingested by the `scxml4cpp` state machine engine. The engine reacts to external and internal events according to the state machine model and it executes user-provided action and activity methods that are implemented directly in code.

Since ELT software components are required to conform to the default life cycle and many components also share the same or similar custom life cycle, it is important to treat state machine models as domain assets and to systematically reuse them in order to avoid duplication and problems during development, integration and maintenance.

3.2.3. Customisation Scenarios

Four concrete scenarios have been identified and analysed to better understand how software components are customised. Note that the scenarios are not mutually exclusive, typically multiple scenarios are combined to create a real-world component. Since the component life cycle does not only depend on the state machine model, this section also takes customisation of the input and output stage into account.

3.2.3.1. Scenario 1: Provision of Custom Behaviour

Experience from legacy systems shows that a significant number of software components can be implemented using a carefully designed default component life cycle and pre-defined hooks for actions and activities that can be implemented by application engineers.

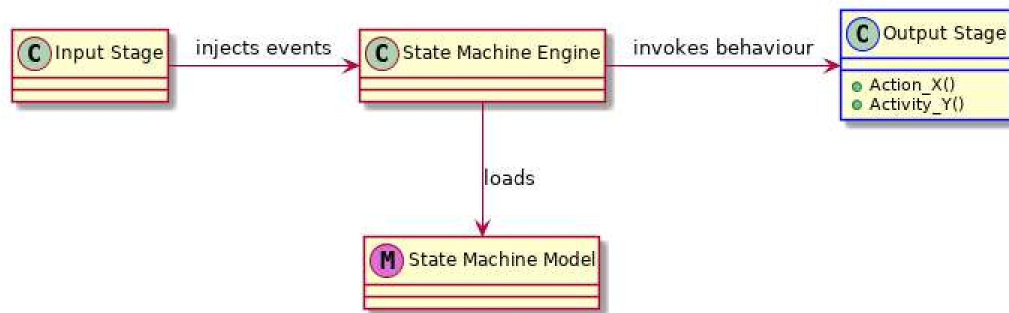


Figure 24: Provision of custom behaviour.

To create a software component with specific behaviour, application developers provide a custom output stage that implements a set of action and activity methods for a specific component life cycle. The state machine model and the input stage remain in control of domain engineering and thus do not need to be changed by application engineering.

This scenario can also be used to unit-test complicated computations that are implemented in the output stage in isolation. In this case, the state machine engine and the input stage are replaced by a mock object that drives the output stage according to a pre-defined sequence of actions and activities.

3.2.3.2. Scenario 2: Provision of Custom Stimuli

Application engineers customise the behaviour of software components by mapping additional external stimuli (e.g. commands or messages) to existing state machine events.

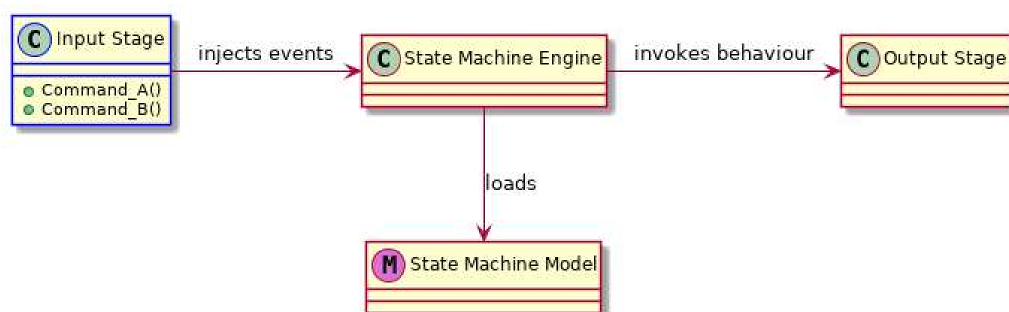


Figure 25: Provision of custom input stage.

The input stage is customised so that it can receive commands and subscribe to topics from additional sources. The received messages are then mapped to existing state machine events. The state machine model does not need to be changed, since the modification is confined to the input stage.

This scenario applies to a small sub-set of components that need to automatically react to additional external stimuli. A concrete example are Data Task components in the RTC Toolkit. They subscribe to state change topics from other components and post a `Run` event to the state machine once other components reach a certain stage in their life cycle.

The scenario is also applicable to enable component-level testing without requiring an external supervisor or sequencer application. In this case, the input stage is replaced with a mock object that drives the state machine engine using pre-defined sequences of state machine events.

3.2.3.3. Scenario 3: Selection of Predefined Life Cycles

As described in Section 3.1.3.2, the component model foresees a set of alternative component life cycles. While domain engineering defines and provides different basic life cycles, application developers are free to select the life cycle that best fits their needs.

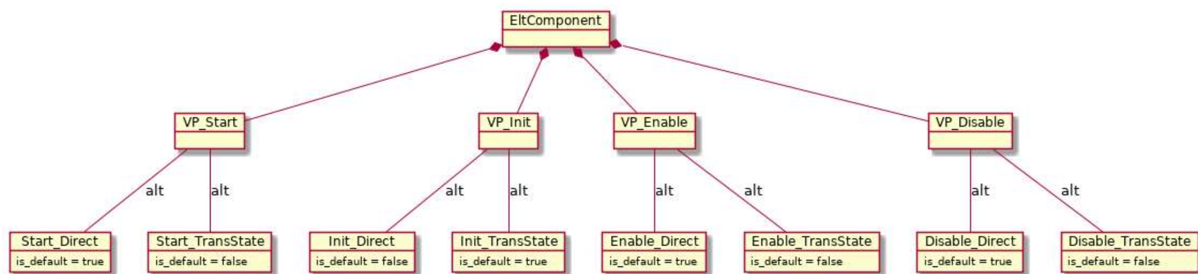


Figure 26: Predefined ELT component life cycles.

Figure 26 shows a feature model for life cycle `EltComponent` with variation points `VP_Start`, `VP_Init`, `VP_Enable` and `VP_Disable`. Concrete variants are available for direct transitions and transitions via additional transient states. Note that also defaults are specified.

Figure 27 shows a feature model of the `RtcComponent` life cycle[1]. While all components are `Updatable` optional capabilities such as `Runnable`, `Loopaware`, `Suspendible`, `Optimisable` and `Measurable` can be selected on demand. While some of these features can be combined, others exclude each other.

[1]Note that this feature model is incomplete, it only contains features that have been identified by domain engineering. Application engineering may introduce additional features to cover specific needs.

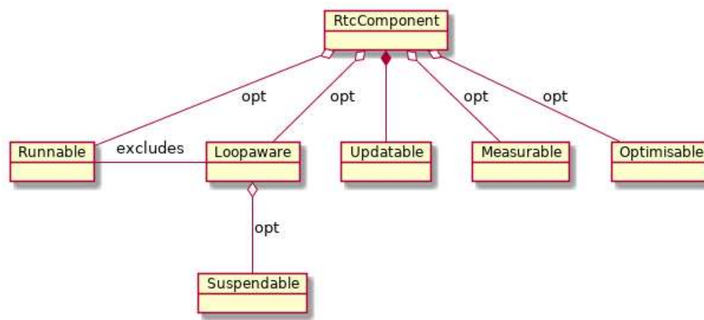


Figure 27: Predefined RTC component life cycles.

The selection of a different component life cycle leads to a changed state machine model. In case the selection causes changes in the input or output alphabet, provision of a custom input or output stage will be required as well (see Sections 3.2.3.1 and 3.2.3.2).

3.2.3.4. Scenario 4: Open Life Cycle Customisation

Developers customise the component life cycle in an open and stepwise manner via extension and refinement of the state machine model. This type of customisation can be performed by both domain and application engineering.

Incomplete software components or component life cycles that are defined in a lower layer of the software stack are refined in more concrete and domain specific higher layers.

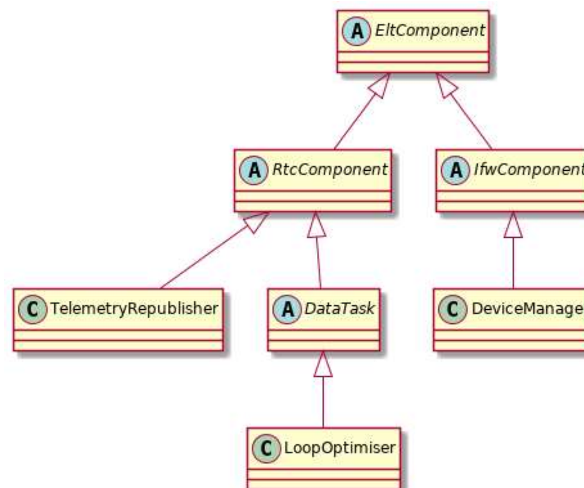


Figure 28: Stepwise component customisation.

Figure 28 shows a hierarchy of software components that are based on stepwise customisation. The `EltComponent` defines the basic component life cycle. It can be refined into more specific components using higher level frameworks that define their own life cycle extensions. Examples are `RtcComponent` and `IfwComponent`. Components can be further

refined multiple times until concrete end applications can be created. Examples for such end applications are the `TelemetryRepublisher`, the `LoopOptimiser` and the `DeviceManager`.

Stepwise component customisation includes life cycle customisation which in turn includes the addition of new commands, events, guards, states, transitions, actions and activities. It may also include modification or even deletion of certain state machine elements.

In case the modification of the state machine model also causes changes in the input or output alphabet, provision of a specific input or output stage is required as well (see Sections 3.2.3.1 and 3.2.3.2). Typically, stepwise customisation also includes the selection of pre-defined life cycles (see Section 3.2.3.3). E.g. an `RtcComponent` is based on a specific variant of the `EltComponent` where transient states `Starting`, `Initialising`, `Enabling` and `Disabling` are used.

3.2.4. Customisation Operations

The concept of open life cycle customisation was introduced in Section 3.2.3.4. To better understand in which ways the component life cycle can be modified, concrete customisation operations were identified and analysed. Concretely, these operations were derived from the ELT RTC Toolkit project. Same or similar operations can be found in related literature, see [ESO20a], [EE94], [Pre13], [SS04], [AB02], [Sim+02], [FM00] and [HSL15].

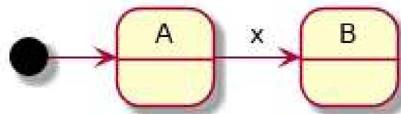


Figure 29: Basic state machine model.

Figure 29 introduces a simple state machine that acts as a base on which customisation operations are applied. Added or modified elements will be highlighted in blue colour.

3.2.4.1. Operation 1: Addition of Transitions

New transitions are added to the model, but the existing behaviour remains unchanged. The added transitions may also introduce new events, guards and actions.

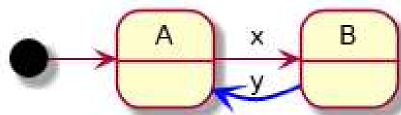


Figure 30: Extension via addition of transition.

This operation can be performed to add optional or alternative behaviour while leaving the original behaviour unchanged. Transitions can also be introduced to create shortcuts between states by bypassing intermediate states.

3.2.4.2. Operation 2: Addition of Sub-Diagrams

A new *sub-diagram* is added that consists of one or more states with associated transitions. The sub-diagram is connected to already existing states using new transitions.

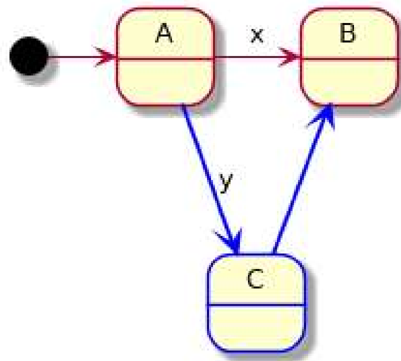


Figure 31: Extension via addition of sub-diagrams.

This operation can be used to add optional or alternative behaviour that is realised as a sequence of states. If no shortcuts are introduced, the original behaviour remains unchanged.

3.2.4.3. Operation 3: Addition of Parallel Regions

This operation is use to add concurrent behaviour that does not interact strongly with existing behaviour. A new orthogonal region is added to an existing parallel state. The new region may contain a sub-diagram consisting of various states and transitions.

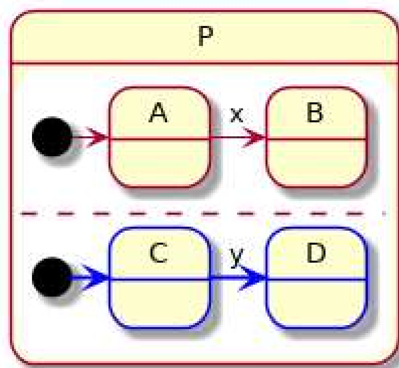


Figure 32: Extension via addition of parallel regions.

3.2.4.4. Operation 4: Refinement of Transitions

Existing transitions are refined by adding elements such as events, guards or actions.

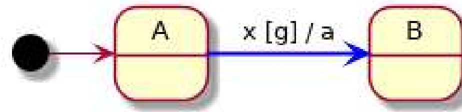


Figure 33: Transition label refinement.

The addition of new events and guards can be used to implement conditional execution and feature flags. Depending on the evaluation of the event and guard condition, a transition can be blocked or an alternative transition can be selected instead.

This operation also includes the refinement or modification of guard conditions. While a base state machine may introduce transitions with guards, a specialisation of this state machine may refine these guards to either relax or strengthen the condition.

When protocol state machines are refined into behavioural state machines, actions are added to existing transitions to provide user-specific behaviour in action methods.

3.2.4.5. Operation 5: Refinement of Transitions into Sub-Diagrams

A transition can be refined into a sub-diagram consisting of one or multiple new transient states connected by new transitions.

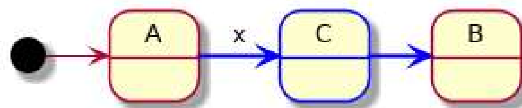


Figure 34: Transition refinement via addition of transient state.

This refinement can be used to adjust the behaviour of an existing sequence. Transitions that can take longer or that can fail can be refined into one or more transient states.

The sub-diagram is introduced between the source and the target state of the original transition and it typically starts with a transition with the same event and/or guard. If the original transition had an action with associated behaviour, it can either remain there or it can also be deleted and implemented via other states or transitions of the newly created sub-diagram.

3.2.4.6. Operation 6: Refinement of Simple States

Additional behavioural elements such as entry-actions, exit-actions and do-activities can be added to an existing simple state. Note that also the addition of internal transitions to a state is possible using this operation.

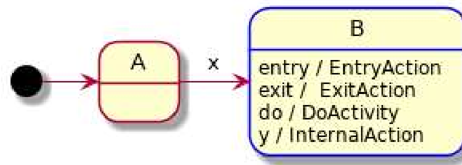


Figure 35: Refinement of simple state.

When protocol state machines are refined into behavioural state machines, actions and activities can be added to existing protocol states to provide user-specific behaviour in corresponding action and activity methods.

3.2.4.7. Operation 7: Refinement of Simple States into Composite States

A simple state can be refined into a composite state by adding new sub-states that are connected by new transitions.

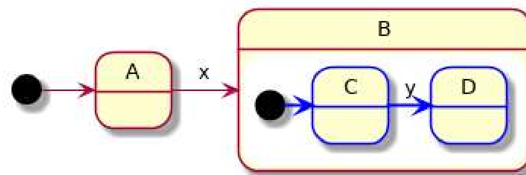


Figure 36: Refinement of simple state into composite state.

This operation may also include:

- deletion of actions and activities from the refined state, because they are now covered by the new sub-states
- modification of the target state of incoming transitions to newly created sub-states
- modification of the source state of outgoing transitions to newly created sub-states

3.2.4.8. Operation 8: Refinement of Simple States into Parallel States

A simple state can be refined into a parallel state by adding new orthogonal regions that contain sub-diagrams.

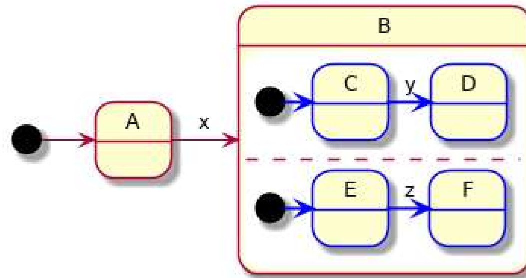


Figure 37: Refinement of simple state into parallel state.

This operation may also include:

- deletion of actions and activities from the refined state, because they are now covered by the new sub-states
- modification of the target state of incoming transitions to newly created sub-states
- modification of the source state of outgoing transitions to newly created sub-states

3.2.4.9. Operation 9: Refinement of Composite States into Parallel States

To be able to execute new behaviour in parallel to an existing sub-diagram, the containing composite state can be refined into a parallel state.

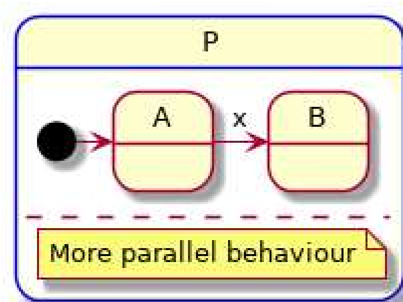


Figure 38: Refinement of composite state into parallel state.

States and transitions of the existing sub-diagram are moved into a newly created orthogonal region of the parallel state via *re-parenting*. In a subsequent step, new orthogonal behaviour can be added by introducing additional parallel regions, as described in Operation 3.

3.2.5. Compatibility

In this section, the information from Section 3.1 is analysed with regard to compatibility and behavioural consistency.

3.2.5.1. Required Level of Behavioural Consistency

Both *invocation consistency* and *observation consistency* need to be preserved when customising software components (see Section 3.1.3).

Invocation Consistency

A component can be steered by both a supervisor and a sequencer. Additionally, it can also react to events emitted by other components. Coarse granular coordination is performed by the supervisor on a set of different components (e.g. initialise all, enable all, disable all). This is only possible if component life cycles are *invocation consistent*. Fine granular steering of individual components using more specific commands is performed by the sequencer (e.g. close a certain loop, trigger a certain optimisation step, etc.). Since the sequencer is aware of specific component life cycles, invocation consistency is not important in this case.
When control is handed over from the supervisor to the sequencer, it can be assumed that all components have been steered to a certain minimum state (e.g. state *Operational*). When control is handed back to the supervisor, it can also be assumed that the sequencer will gracefully bring the components to a state where the supervisor can take over. This means that it is sufficient to assume *weak invocation consistency*.

Observation Consistency

Observation consistency is important for monitoring. Supervisors need to be able to observe the state of subordinate components, even if the life cycle of these components has been customised. This is only possible if observation consistency is preserved.

3.2.5.2. Compatibility According to Scenarios

In **Scenario 1**, the component life cycle can be considered immutable and in control of domain engineering. Therefore, compatibility is guaranteed by construction because the state machine model, the input stage and the output stage cannot be changed by application engineering.

In **Scenario 2**, the extension of the input stage does also not change the state machine model. Therefore, the component life cycle is still compatible. However, from an external

perspective, compatibility may be affected if the mapping from commands to state machine events changes. For this reason, it is recommended to perform conformance testing on the component level to ensure that important command sequences are still accepted.

In **Scenario 3**, optional and alternative features of the component life cycle are defined by domain engineering. Platform developers ensure by construction that all selectable component life cycles are correct and compatible with the standard behaviour. In addition, they also put mechanisms in place that prevent application engineers from selecting or combining invalid feature configurations.

In **Scenario 4**, developers are free to modify the component life cycle in an open and stepwise manner. This means, the state machine model can be modified in ways that break compatibility. To prevent this, constructive methods that preserve invocation and observation consistency should be used (see Section 2.4.2.1). Additionally, conformance testing should be performed. Reusable conformance test assets can be provided by domain engineering as part of the product line platform.

3.2.5.3. Behaviour Consistent Life Cycle Customisation

It is important to make developers aware that the usage of certain customisation operations can break behavioural consistency and compatibility. Important effects that different customisation operations have on behavioural consistency are explained below:

Operations 1 and **2** only add new transitions or states without modifying existing state machine elements. Invocation consistency can be preserved by only adding new transitions that do not bypass existing transitions, e.g. by defining new transitions with a trigger. Observation consistency cannot be preserved easily because newly added transitions can also introduce shortcuts between existing states that break observation consistency.

In **Operation 3**, new orthogonal regions are added so that new behaviour can be executed in parallel. Here invocation consistency and also observation consistency can be preserved since existing elements are not modified and new functionality is only added orthogonally. However, developers must be careful not to add transitions into new parallel regions that prematurely exit the containing parallel state because this can break both invocation consistency and observation consistency.

Operations 4 and **5** refine transitions. Here, invocation consistency cannot be preserved if transitions are refined with events or guards. Observation consistency however is preserved because the sequence of visited states can still be mapped to the basic behaviour by omitting newly introduced intermediate states.

In **Operation 6**, a state is refined by adding actions, activities or internal transitions. This operation preserves both invocation and also observation consistency.

Operations 7 and **8** refine a state into a composite or a parallel state. Here, invocation consistency can be preserved if exiting the refined state is allowed from each newly introduced sub-state. In this case, a basic activation sequence will still be accepted and lead to the same target state. Observation consistency can also be preserved as long as no transitions are introduced in sub-states that exit the refined state and thus introduce a life cycle occurrence that can not be mapped to the basic behaviour.

In **Operation 9**, existing state machine elements are moved into an orthogonal region via re-parenting so that new behaviour can be added in parallel. Since existing functionality is not modified, invocation consistency can be preserved. Observation consistency is preserved since the life cycle occurrences of the modified model can still be mapped to the basic model.

It becomes apparent that the preservation of both invocation and observation consistency is a non-trivial task. Conformance testing can be used to ensure that important activation sequences are invocation consistent and that the sequence of visited states can be mapped to the basic behaviour. Another constructive approach to make open customisation ”safer” in regard to compatibility is to disallow certain customisation operations or aspects of certain operations. E.g. when refining states according to Operations 7 and 8 observation consistency can be ensured by disallowing the introduction of new transitions that leave the refined state. Of course introducing such limitations also limits flexibility which is a trade off that needs to be considered.

3.3. Requirements and Constraints

After describing and analysing the problem in more detail, concrete requirements and constraints for the solution concept can be derived and formulated.

- **R1**: The solution concept shall support all customisation scenarios from 3.2.3 and all customisation operations from 3.2.4. The concept shall also enable developers to perform all types of customisation described in 3.1.3.
- **R2**: The solution concept shall support the systematic reuse of code as well as state machine models (which are likely to be *non-code artefacts*) in a platform or product line setup (see Section 3.2.2).
- **R3**: The solution concept shall support *open variation* to facilitate stepwise customisation and distributed development by multiple, loosely coupled teams spread all around the globe. This is important not to work against Conway's law.
- **R4**: To be able to cleanly separate domain assets from application assets, the solution concept shall support *variant isolation*.
- **R5**: The notion of features shall be supported to simplify component life cycle customisation. Feature orientation simplifies understanding and selection of variants, it also introduces the possibility of providing a feature model.
- **R6**: Since ELT control software projects are not mandated to use a software modeling tool and a MDA based workflow, the solution concept shall primarily support the classic software engineering based workflow. It is considered "nice to have" if the solution can also be applied to the MDA based workflow.
- **R7**: The elaborated solution shall work well in concert with concepts, technologies and tools used for ELT software development. Concretely, this means the solution shall support the SCXML format and it shall work well in a service-oriented and component-based system.
- **R8**: To ensure understandability and long-term maintainability, the solution shall be based on mature technology. The introduction of experimental third party libraries and tools shall be avoided.
- **R9**: The solution concept shall facilitate preservation and/or validation of *invocation consistency* and *observation consistency* so that compatibility can be ensured.

3.4. Examination of Mechanisms

In this section, different customisation mechanisms are examined to find out which mechanisms are most suitable to cover the scenarios from Section 3.2.3 and the operations from Section 3.2.4. Concrete prototypes were implemented in modern C++ to gain experience with less known mechanisms.

3.4.1. Framework Approach

Basic Idea

Domain engineering provides an abstract design or skeleton with pre-defined extension points. Application engineers customise the behaviour of the software component by writing application specific extensions.

Description

This approach is based on software frameworks that were introduced in Section 2.2.3.4. Variability is realised using C++ function templates and the *Strategy* design pattern.

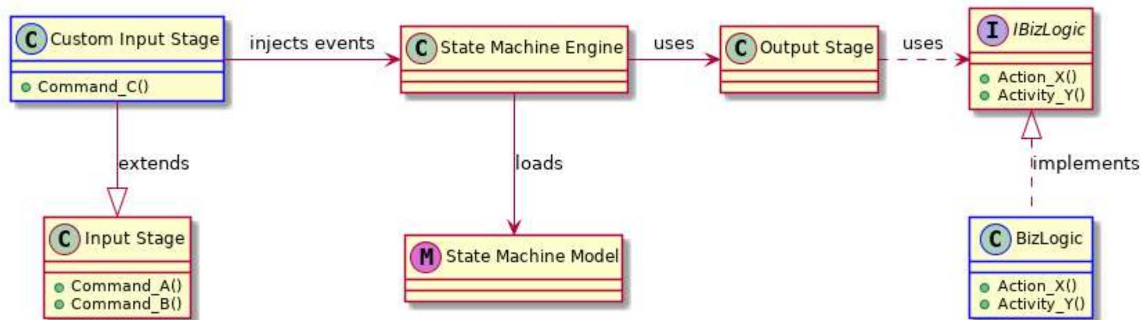


Figure 39: Framework approach.

Figure 39 shows the overall structure of a software component including `Input Stage`, `State Machine Engine`, the `State Machine Model` and the `Output Stage`. These parts can be regarded as a skeleton that defines the high-level component behaviour. To allow for customisation of the output stage, the skeleton defines abstract methods or hooks with empty bodies. The hooks are grouped into a business logic interface `IBizLogic` that defines concrete action and activity methods. Application engineering provides custom behaviour by implementing class `BizLogic` that implements interface `IBizLogic`. To make the skeleton aware of the custom business logic, the class needs to be registered using some form of dependency injection.

Similarly, the skeleton also allows the provision of a custom input stage, this enables application developers to map additional external messages to state machine events.

Prototyping

A simple framework was created for prototyping. It contains common infrastructure code and also life cycle specific code. While the common infrastructure can be used for all life cycles, each life cycle is represented by a specific set of collaborating classes. A generic component runner is provided as a function template, it allows to start a component with a user-provided business logic and it facilitates the provision of a custom input stage.

```
// common infrastructure (same for all life cycles)
class Services {};
class Engine {};

// generic component runner method
template<class IS, class OS, class BL>
int RunComponent(string const& model) {
    Services services;
    Engine engine;

    BL logic(services);
    IS input_stage(engine);
    OS output_stage(engine, logic);

    engine.LoadModel(model);
    return engine.Work();
}

// implementation for specific life cycle: RunnableRtcComponent
namespace RunnableRtcComponent {

    // user-facing business logic interface with actions and activities
    struct IBizLogic {
        virtual void ActivityStarting() {}
        virtual void ActivityInitialising() {}
        virtual void ActivityEnabling() {}
        virtual void ActivityDisabling() {}
        virtual void ActivityRunning() {}
    };

    // state machine model in textual representation
    const string MODEL = "the scxml model as a string";

    // default input stage, receives messages and posts events to sm
    class InputStage { };
```

```
    // implements req/rep sending and delegates to business logic
    class OutputStage { };

    // component runner method
    template<class BL, class IS=InputStage>
    int RunComponent() {
        return ::RunComponent<IS, OutputStage, BL>(MODEL);
    }
}

// implementation for specific life cycle: LoopawareRtcComponent
namespace LoopawareRtcComponent {
    // same as above with different details ...
}

// and so on, for other provided life cycles
```

Listing 5: Framework code.

Listing 5 shows a condensed snippet of framework code that defines different collaborating classes for life cycle `RunnableRtcComponent`. The life cycle is fully defined by the `MODEL`, the `InputStage`, the `OutputStage` and interface `IBizLogic`. To support other life cycles, the same structure can be repeated in different namespaces. A more detailed listing can be found in Appendix 5.3.

Listing 6 shows how to create a custom application using the framework. First, developers select the desired life cycle (e.g. `RunnableRtcComponent`). Then, they implement the custom business logic class `BizLogic`. Finally, the component runner method `RunComponent` is invoked in `main` with class `BizLogic` as a template argument.

```
// selection of life cycle
namespace LifeCycle = RunnableRtcComponent;

// custom business logic class
class BizLogic : public LifeCycle::IBizLogic {
    public:
        BizLogic(Services services) {
            // retrieve handles to services
        }
        void ActivityInitialising() override {
            // application specific implementation
        }
        void ActivityEnabling() override {
            // application specific implementation
```

```
        }
        void ActivityDisabling() override {
            // application specific implementation
        }
        void ActivityRunning() override {
            // application specific implementation
        }
};

// main entry point
int main() {
    return LifeCycle::RunComponent<BizLogic>();
}
```

Listing 6: Application specific code.

Discussion

With this framework approach, customisation Scenarios 1 and 2 can be fully covered. In both cases, the state machine model is considered immutable and application engineers customise the component behaviour by providing custom implementations for the `InputStage` or for the `BizLogic` class.

To support the provision and selection of predefined component life cycles as described in customisation Scenario 3, the framework can be enhanced so that it provides different life cycles. Application developers can then select the desired life cycle and implement the business logic interface accordingly. However, if many feature combinations need to be supported, a potentially huge number of life cycles have to be defined in the framework (to cover all required feature combinations). Here the approach reaches its limits in terms of scalability and maintainability.

Even though application developers can introduce their own life cycles (that can even reuse parts of the framework code) the mechanism cannot be used to extend or modify the state machine model. For each provided life cycle a full state machine model needs to be defined which causes duplicated information and maintenance problems. Therefore, Scenario 4 cannot be covered.

Additional advantages (+) and limitations (-) have been identified:

+ The mechanism cleanly separates user-provided code from framework code, this enables domain engineering to maintain and evolve the framework separately.
+ The approach enables domain engineers to provide default implementations for methods of the Business Logic. In this case, application developers are not required

to implement empty methods just to make the compiler happy.

- The granularity of customisable elements is limited to entire life cycles and methods in `IBizLogic` and in the custom `InputStage`.

Conclusion

The framework approach can be used to cover customisation Scenarios 1, 2 and partially 3. Since it is not possible to reuse and customise the state machine model with this approach, it needs to be combined with another mechanism that fully covers Scenario 4.

In addition, the problems of duplicated information and combinatoric explosion of provided life cycles need to be addressed to make this approach viable and scalable.

3.4.2. Conditional Execution

Basic Idea

Variability is resolved during run-time using mechanisms for conditional execution that are provided by the state machine language.

Description

Both UML and also SCXML statecharts provide built-in mechanisms for conditional execution. Examples for such mechanisms are transitions with guards and junction/choice pseudo states. By using such mechanisms, a 150 % model can be created that covers multiple variants. A similar approach was also applied by Rohlf in [Roh20].

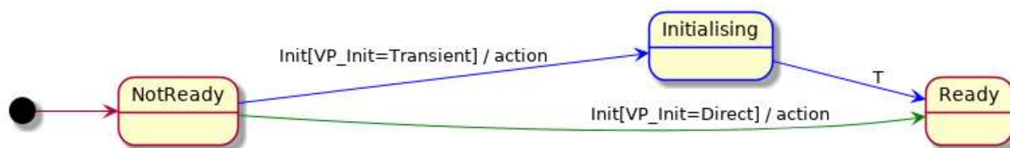


Figure 40: Conditional execution mechanism.

Figure 40 shows a 150 % model of a statechart that covers component initialisation with and without transient state `Initialising`. The desired variant is selected at run-time using guard `VP_Init`.

Prototyping

To support conditional execution, the framework approach from Section 3.4.1 was enhanced so that the SCXML model now also makes use of guards.

Since the *scxml4cpp* engine does not support conditional expressions that are directly encoded in the model, it is necessary to implement guards using hard-coded *guard methods*

that return either `true` or `false`. Whenever the engine needs to evaluate a guard, it invokes the associated guard method. If the method returns `true`, the transition can be taken, otherwise it is blocked.

```
<state id="Update" initial="Idle">
  <state id="Idle">
    <transition event="Update" cond="GuardUpdatingAllowed" target="Busy"/>
  </state>
  <state id="Busy">
    <invoke id="ActivityUpdating"/>
    <transition event="Done" target="Idle"/>
  </state>
</state>
```

Listing 7: SCXML model with guard condition.

Listing 7 shows parts of an SCXML model that contains a guard. The transition from state `Idle` to `Busy` can only be taken if event `Update` arrives and the associated guard method `GuardUpdatingAllowed` evaluates to `true`.

Guard methods can be implemented in the framework but they can also be provided by application engineering. In the latter case, hooks for the guard methods are added to the business logic interface. This enables application engineering to provide custom guard expressions together with the business logic.

```
// implementation for specific life cycle: UpdatableRtcComponent
namespace UpdatableRtcComponent {

    // user-facing business logic interface with guard and activity
    struct IBizLogic {
        virtual bool GuardUpdatingAllowed() {return true;}
        virtual void ActivityUpdating() {}
        // ...
    };
    // ...
}
```

Listing 8: Framework code with guard method.

Listing 8 shows a snippet of framework code that defines a business logic interface with `Update` functionality. `ActivityUpdating` is only invoked if the user-provided guard condition `GuardUpdatingAllowed` evaluates to true. Note that the code already defines default implementations for both the guard and the activity.

Discussion

Conditional execution of state machine models is based on *model annotation* and *negative variability*. The approach can be used to cover Scenario 3 if variation points and variants are known up-front and if binding needs to happen late during run-time.

In addition, the following advantages (+) and limitations (-) have been identified:

+ Variation points can be made explicitly visible in the model using special naming conventions, e.g. guard conditions starting with prefix `VP_`.
+ The statechart is well-formed and can be used for simulation. This is important for conformance testing and validation.
- The approach cannot be used for open and stepwise customisation, because it is based on closed variability.
- Conditional refinement of states is not possible with this approach. It is not possible to draw a well-formed 150 % model where state `Initialising` has different internal refinements. This would require duplication of the state with different names, which is not ideal in terms of consistency and also compatibility.
- The scalability of the approach is limited, understanding and maintaining a 150 % model for non-trivial state machines with dozens of states, transitions, guards, actions and activities is very difficult. Developers can work around this by creating different views or sub-machines for individual features, but also this is hard to understand and maintain as the model grows.
- At run-time only a part of the 150 % model will be traversed, but the execution engine still has to load and interpret the entire model. This causes overhead.
- The SCXML standard and also the *scxml4cpp* engine only support a limited set of model elements used for conditional execution. Concretely, there is no support for *junction/choice pseudo-states* and *conditional expressions*. For this reason the mechanism can only be applied in a limited fashion where effectively only transitions with guards can be used to realise conditional execution.

Conclusion

Since conditional execution is based on *model annotation* and *negative variability*, it can be used to cover customisation Scenario 3.

The mechanism is not suitable as a primary customisation mechanism, but when combined with a framework approach, the mechanism can be used to cover specific cases where variability is well-known and where binding needs to happen late during run-time.

3.4.3. Model Annotation

Basic Idea

Domain engineering provides a 150 % model that makes use of UML stereotypes to annotate individual elements with information regarding variability. A custom pre-processing tool resolves variability according to a user-provided feature selection and transforms the 150 % model into a concrete 100 % model.

Description

Both UML and SysML support the annotation of model elements with stereotypes. This mechanism can be used to make variation points explicitly visible and to mark statechart elements as optional or alternative. During variability resolution, a concrete state machine model is generated by a pre-processing tool that can be implemented as a plug-in of the modeling tool. The plug-in ingests the 150 % model and transforms it into a 100 % model with resolved variability according to a user-provided feature selection.

Prototyping

Prototyping was performed in *MagicDraw*. The outcomes were used to document mandatory, alternative and optional features of the basic ELT component life cycle in [ESO20a].

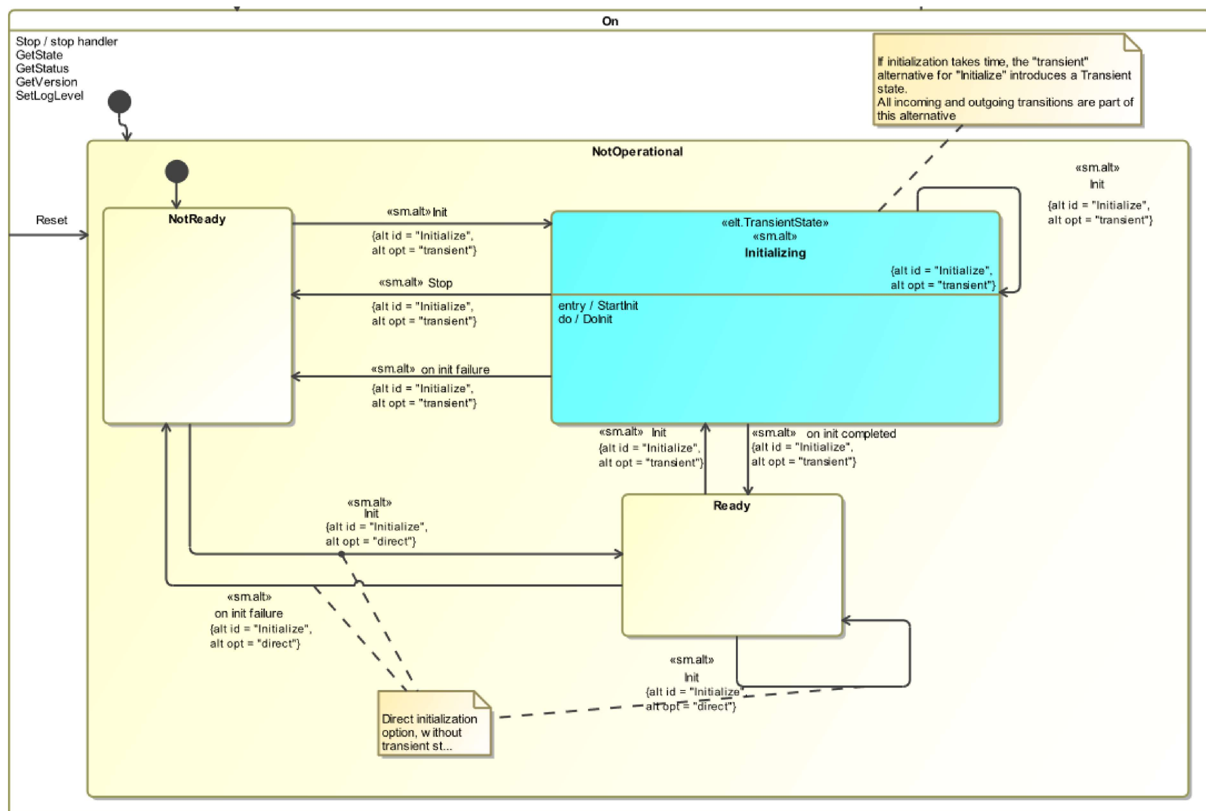


Figure 41: Model annotation with UML stereotypes. [ESO20a]

Figure 41 shows parts of the standard ELT component life cycle. The statechart makes use of stereotypes to document alternative options for component initialisation. Model elements that belong to variants are annotated with stereotype `<<sm.alt>>`. The stereotypes also contain additional information: Attribute `alt.id` identifies the variation point and attribute `alt.opt` specifies the variant.

Discussion

Annotation of state machine models with UML stereotypes is based on negative variability. The approach can be used to cover Scenario 3 and it works well if variation points and variants are known up-front. A similar approach was applied by Trujillo et al. in [Tru+10].

The following advantages (+) and limitations (-) have been identified:

+ Variation points are explicitly visible in the 150 % model, since they are encoded using stereotypes.
+ The resolved 100 % model only contains selected functionality, so it is easier to read and understand and it does not cause overhead during interpretation.
- The mechanism is based on closed variability, therefore it cannot be used for open and stepwise customisation.
- The approach requires usage of a modeling tool, the definition of a custom profile for model annotation and the development of a custom transformation tool that transforms the 150 % model into a 100 % model.
- The 150 % model cannot be directly used for simulation or interpretation, it first needs to be transformed into a concrete 100 % model.
- A 150 % model containing multiple alternative variants may not be well-formed because it can include conflicting states and transitions. While some modeling tools do not even allow the addition of conflicting elements, other tools may report errors during model validation.
- Conditional refinement of states can only be implemented with additional effort and workarounds. A well-formed state machine model requires states to have unique identifiers. To achieve this, state names need to be suffixed, e.g. `Initialising.Refined` and `Initialising.Unrefined`. When creating the 100 % model, the transformation tool needs to strip away this information to produce correct state names.
- Scalability of this approach is limited. Understanding and maintaining a 150 % model for non-trivial state machines with dozens of states, transitions, guards, actions and activities is very difficult. Annotation with stereotypes makes the model even more difficult to read and to maintain, especially if many elements have stereo-

types. Developers can work around this by creating different views for individual features, but also this is hard to understand and maintain as the model grows.

Conclusion

Model annotation with stereotypes can be considered if a project is fully model driven and if a complete variability model exists upfront. For the ELT control system, however, this is not the case. Therefore, the approach must be disregarded. Other limitations discourage the use of this approach further.

3.4.4. Model Superimposition

Basic Idea

Pre-defined state machine model fragments are combined into a single, custom model according to a user-provided feature selection.

Description

Model superimposition is a rather simple, composition-based customisation technique that assembles a custom state machine model from individual model fragments. Apel et al. define superimposition as "the process of composing software artifacts by merging their corresponding substructures on the basis of nominal and structural similarity" [Ape+09].

First, domain engineering defines major features that are mapped to model fragments. The fragments must be well-formed state machine models that have certain structural similarities. Next, application engineering selects a set of features and uses a custom superimposition tool to merge the corresponding model fragments into a single, customised state machine model. The tool combines model fragments according to a user-provided feature expression. Model elements are matched by name and type and an output model is created that represents the union of all ingested input models.

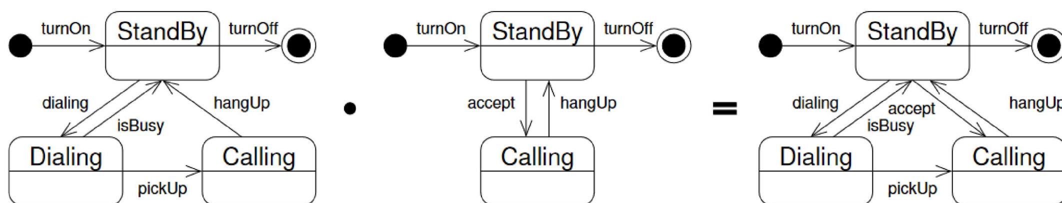


Figure 42: Model superimposition. [Ape+09]

Figure 42 shows an example where two state machine model fragments are merged. The resulting model can be regarded as the union of the provided fragments. The fragment

in the middle effectively adds a new transition with event `accept` between states `StandBy` and `Calling`.

Prototyping

For the evaluation of the mechanism two different tools were used:

- *FeatureHouse* from [Ape]
- *scxml4cpp* from [ESO20c]

Even though both tools are capable of superimposing state machine models, preference was given to the ESO tool that is based on SCXML models. Since *scxml4cpp* is developed in house, it can easily be modified and improved if necessary. Development of *FeatureHouse* seems to have stagnated in 2011. Additionally, the tool makes use of the *XMI ArgoUML* format that is not supported by the tools used at ESO.

```
void Superimpose(Engine& engine, string base, list<string> fragments) {
    engine.Load(base);
    for(auto fragment : fragments)
        engine.Append(fragment);
}
```

Listing 9: Model superimposition with scxml4cpp.

Listing 9 shows how *scxml4cpp* can be used to superimpose a list of model fragments in SCXML format. First, a base model is loaded using `engine.Load()`. Then, the list of model fragments is appended with `engine.Append()`. The resulting in-memory model represents the union of loaded and appended model fragments.

During the prototyping activities, several limitations and also bugs in the *scxml4cpp* engine and also in the model merger were detected and reported. Most of the reported issues have already been fixed by the application framework development team. Other limitations are inherent to the superimposition mechanism and thus cannot be fixed.

Discussion

Model superimposition is a composition-based customisation technique that makes use of positive variability. While the selection and combination of pre-defined model fragments corresponds to customisation Scenario 3, the provision of reusable model fragments corresponds to Scenario 4.

Table 2 shows which customisation operations are supported by model superimposition with the *scxml4cpp* model merger.

Operation	Supported	Comment
1	yes	-
2	yes	-
3	yes	-
4	partially	no extension of guards
5	no	change of transition target not possible
6	yes	-
7	partially	no deletion of actions
8	no	change of state type not possible
9	no	re-parenting not possible

Table 2: Supported customisation operations in scxml4cpp.

Since the mechanism only supports addition of new elements, not all customisation operations can be realised. The introduction of new of behaviour via adding transitions, sub-diagrams or parallel regions is fully supported. The refinement of transitions is only partially supported: It is possible to add new guard conditions but it is not possible to further constrain or relax existing guards. It is also not possible to refine transitions into sub-diagrams or transient states. This would require the modification of transition target states, which is not possible using superimposition. Simple states can be refined by adding new actions, activities or internal transitions and it is also possible to refine them into composite states with certain limitations. The refinement of a simple state into a parallel state however is not possible because the hierarchical structure is too different in the underlying SCXML notation. Similarly, composite states cannot be refined into parallel states because re-parenting cannot be done without allowing modification.

The following advantages (+) and limitations (-) have been identified:

+ Model superimposition is a rather simple operation that can be easily implemented in a custom tool. It is already supported by the *scxml4cpp* model merger.
+ The mechanism is based on positive variability and thus allows extension and refinement of the state machine model via the addition of new state machine elements.
+ The approach can be combined with the MDE workflow. Individual model fragments are well-formed state machines that can be kept in the modeling tool, a custom modeling tool plug-in can perform the merging of the fragments.
- Model superimposition is limited to the addition of elements. It does not support replacement and/or deletion, which is required for certain customisation operations.
- Individual model fragments are required to have a certain structural similarity. This similarity introduces redundant information, which is not ideal in terms of reuse.

- Since only addition of elements is possible, there is no way to specify defaults when dealing with alternative features. This means, the user-provided feature expression must contain the full selection of features including defaults.
- Features that depend on other features must be factored out into dedicated model fragments that can be combined with the respective features. Since not all selectable feature combinations are meaningful, additional constraints are necessary to define which combinations are valid.
- Feature interactions require features to behave differently in the presence of related other features. This is not supported by superimposition.

Conclusion

Model superimposition is a rather simple, composition-based customisation technique that partially covers customisation Scenarios 3 and 4. The mechanism works well to add large granular features orthogonally to existing core functionality. When the granularity of the features becomes smaller or when features interact with each other, the approach reaches its limits. Another major limitation of model superimposition is that it cannot deal with the modification and deletion of elements. This makes the mechanism unsuitable if certain customisation operations are required.

3.4.5. Model Inheritance

Basic Idea

A general state machine model is specialised in a stepwise manner using inheritance semantics. While a base model contains common functionality, different specialisations extend and refine the base model and add more specific capabilities.

Description

According to the UML 2.5 specification, state machines are generalisable elements. A general state machine can be specialised so that a specialised state machine inherits all elements of the general state machine. The specialised state machine can also include additional elements like regions, states or transitions. Inherited regions, states or transitions can be redefined. Simple states can become composite states, which in turn can be redefined by adding new regions, states and transitions. States may also be redefined by adding entry/exit/doActivity behaviours if the general state does not have any. When redefining transitions, source state and trigger must be preserved but action and target state can be replaced. [OMG15]

Prototyping

Prototyping with *MagicDraw 19* showed that the tool does not support inheritance semantics of state machine models. Even though developers can add generalisation relations between statecharts, the elements of the general state machine are not available in the specialisation. Therefore, the tool cannot be used to realise customisation via inheritance.

Additional prototyping was carried out with *Rhapsody 9.0*. Support for statechart inheritance is explicitly mentioned in the supporting documentation [IBMb]. According to a white paper, state machine extension and state redefinition are implemented via class inheritance, where the inherited statechart can redefine inherited elements [IBMa].

An example state machine model was created that acts as a base for customisation using inheritance semantics. This base model was then specialised in a stepwise manner. Different customisation operations were applied to find out if they are supported by *Rhapsody*. The full sequence of performed customisation steps can be found in Appendix A.

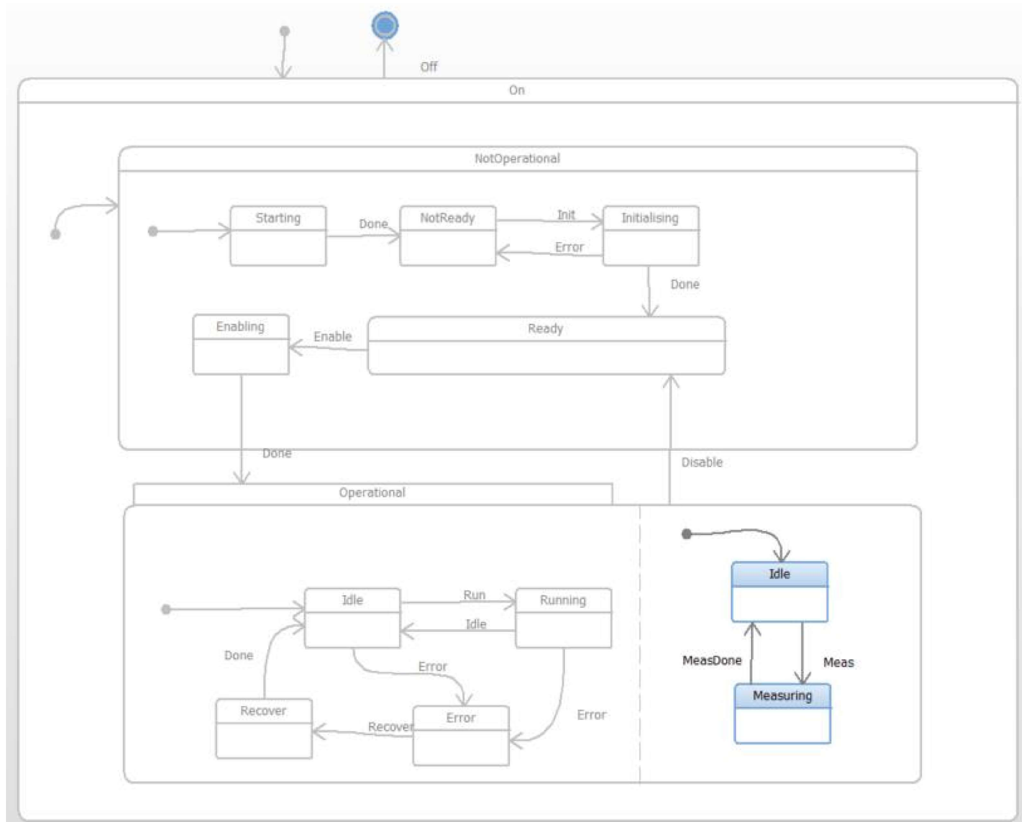


Figure 43: Model inheritance.

Figure 43 shows the state model after applying a sequence of customisation operations. The last operation applies parallel extension (see Operations 9 and 3) to state `Operational` and adds a new parallel region with states `Idle` and `Measuring`.

More prototyping was performed in *Rhapsody* to find out if mixin-based inheritance is supported. The usage of mixins for individual specialisation steps would enable application engineers to compose custom statecharts by selecting and combining mixins as needed. Unfortunately, statechart inheritance is implemented via copying elements from the base class into the specialised class. This becomes apparent when deleting the inheritance relation. Base class elements are not removed from the specialised class, instead they simply remain in the derived class as if they would directly belong to it. This means, specialisations cannot exist without a base class, therefore it is not possible to realise mixin-based inheritance in Rhapsody. For this reason, Scenario 3 cannot be covered with this mechanism.

Discussion

State machine inheritance is a composition-based customisation mechanism that allows extension and refinement of state machine models via the application of different specialisation steps. This corresponds to customisation Scenario 4.

Similar to class inheritance from object-oriented programing, state machine inheritance semantics allow the addition of new and the modification of existing model elements.

Operation	Supported	Comment
1	yes	-
2	yes	-
3	yes	-
4	partially	change of transition trigger not possible
5	yes	-
6	partially	doActivities not supported
7	partially	change of transition source not possible
8	partially	change of transition source not possible
9	yes	-

Table 3: Supported customisation operations in Rhapsody.

Table 2 provides an overview of supported customisation operations in Rhapsody. While some operations are fully supported others are only partially covered by the mechanism.

The following advantages (+) and limitations (-) have been identified:

+ Inheritance semantics as defined in the UML standard and implemented in Rhapsody support all customisation operations at least partially and thus enable stepwise customisation of state machine models on the level of the graphical modeling tool.

+ Performing the specialisation in a graphical modeling tool greatly simplifies customisation because inherited and newly added elements can be clearly distinguished.

- Statechart inheritance semantics are not well-defined and they are also not known to many developers. While the UML specification from [OMG15] provides some rules for some specialisation operations, the SCXML specification from [W3C15] does not even attempt to define any inheritance semantics.
- Only a limited number of modeling tools such as *Rhapsody* support the specialisation of state machine models using inheritance and even these tools still have major limitations. The modeling tool used at ESO does not support it at all.
- *Rhapsody* does not support mixin-based inheritance, therefore it is not possible to realise mixin-based feature composition.
- Even though all customisation operations are covered, there are limitations: E.g. both UML 2.5 and Rhapsody inheritance semantics disallow changing the source state and the trigger of a transition. For this reason customisation Operations 4, 7 and 8 can only be covered partially.

Conclusion

State machine inheritance is a customisation technique that can be used to cover Scenario 4. Even though the mechanism is in theory very powerful, the evaluated graphical modeling tools do not provide sufficient support for inheritance semantics. Especially, the missing support for mixin-based inheritance is a show-stopper. Moreover, the SCXML standard does not support inheritance semantics either. For these reasons the usage of this mechanism must be disregarded.

3.4.6. Frame Technology

Basic Idea

A frame processing tool is used to assemble a custom SCXML model from individual frames that contain common and variable parts.

Description

Frame technology is an annotation-based and tool-driven customisation technique that makes use of pre-processing. While *common modules* contain basic functionality and frame technology annotations to mark variation points, *variant modules* contain the variable parts of the state machine model and frame processor commands that specify how the variable parts are to be inserted. To create a custom state machine model, application engineers select common and variable modules and use a frame processor to resolve variability and to create the desired output model. Since frame technology supports *open variability*, application developers can also provide custom variant modules.

Prototyping

For prototyping the open source *FrameProcessor* from Patzke was used [Pat]. The tool is written in Python and it supports any kind of text based assets ranging from plain text over source code to markup languages like the XML.

Core modules contain common code and variation points that are marked with tags using keyword `VP` followed by the unique name of the variation point. In addition, statement `OUTFILE` specifies the name of the generated output file with resolved variability.

Variant modules specify with the `ADAPT` statement which core or other variant module they modify. The fact that also other variant modules can be specified allows the definition of an arbitrarily deep hierarchy of variant modules starting from a top level variant module that defines the set of selected features. The concrete modification operation consists of an operator e.g. `REPLACE`, `EXTEND` or `CLONE` followed by an operand that contains the information that is added with the operator. The operand text can also specify variation points as well as clone points.

```
ADAPT feature_Runnable
ADAPT feature_GetStatus
ADAPT feature_Enabling
ADAPT feature_Updatable
ADAPT feature_Optimisable
```

Listing 10: Variant module for feature selection.

Listing 10 shows the top level variant module that applies 4 different adaptations that can be mapped to features `Runnable`, `GetStatus`, `Enabling`, `Updatable` and `Optimisable`.

```
ADAPT core_module
REPLACE enable.s
    <state id="Enabling">
        <transition event="Done" target="Operational"/>
        <invoke id="ActivityEnabling"/>
    </state>
REPLACE enable.t
    <transition event="Enable" target="Enabling"/>
```

Listing 11: Variant module for feature "Enabling".

Listing 11 shows the implementation of the variant module belonging to optional feature `Enabling`. The variant module refines a transition into a sub-diagram with new transient state `Enabling` according to Operation 5. The operation is implemented in two steps:

First, the new transient state with an outgoing transition to state `Operational` is added using variation point `enable.s`. Then, the original transition is replaced with a transition to the new transient state using variation point `enable.t`.

```
OUTFILE fsm.xml
    <?xml version="1.0" encoding="us-ascii"?>
    <scxml xmlns="http://www.w3.org/2005/07/scxml"
        xmlns:customActionDomain="http://my.custom-actions.domain/CUSTOM" >
        version="1.0" initial="On">
        <state id="On" initial="NotOperational">
            <state id="NotOperational" initial="NotReady">
                <state id="NotReady">
                    <transition event="Init" target="Ready"/>
                </state>
                <state id="Ready">
<VP enable.t>
                    <transition event="Enable" target="Operational">
                        <customActionDomain:ActionEnabling name="
                            ActionEnabling"/>
                    </transition>
</VP>
                </state>
<VP enable.s> </VP>
            </state>

<VP operational>
            <state id="Operational">
                <transition event="Disable" target="Ready">
                    <customActionDomain:ActionDisabling name="
                        ActionDisabling"/>
             </transition>
             <invoke id="ActivityRunning"/>
            </state>
</VP>
            <transition event="Reset" target="NotOperational"/>
<VP status> </VP>
        </state>
        <state id="Off"/>
    </scxml>
```

Listing 12: Core module with basic state machine model.

Listing 12 shows the core module that defines the basic SCXML state machine model. The core module also contains frame processor annotations for variation points `enable.t`,

`enable.s`, `operational`, `status` and default implementations for variation points `enable.t` and `operational`. In addition, the output file is specified as `fsm.xml`.

Discussion

Frame processing works well if variability is well understood upfront, therefore it can be used to cover foreseen variability as described in Scenario 3. In addition, the approach also supports the selection of features via the top level variant module.

Frame command `REPLACE` can be used to implement optional and alternative features, command `EXTEND` can be used to support features that allow the co-existence of multiple variants at the same time. The `CLONE` command and clone point annotations can be used to implement repetitive features that are very similar and only differ in small details, this greatly reduces duplication of information.

Since the frame processor tool can digest all sorts of text-based assets, it can also be used to modify source code. Therefore, customisation Scenarios 1 and 2 can be covered as well. Concretely, this means that in the output stage the template method or strategy pattern can be implemented using command `REPLACE` and additional message handling can be added to the input stage with `EXTEND`.

Since the mechanism supports open variation, application engineers are free to create additional variant modules for existing variation points. This can be an advantage in terms of flexibility but it can be a disadvantage in terms of compatibility, since custom variant modules could break behavioural consistency.

Unfortunately, the supported notion of open variation is not flexible enough to properly support customisation Scenario 4 where also unforeseen or unexpected variability needs to be covered. Extension can only happen if variation points are foreseen in common modules. Therefore, domain engineers must anticipate changes and add variation points upfront. In practice this is difficult to achieve: Domain engineering does not always have enough information regarding upcoming features, so it is not always possible to add variation points at the right place in advance. Moreover, variation points need to be named (usually according to specific capabilities or features) but when it comes to unforeseen variation these variation point names lose their meaning and degrade to unique identifiers.

A concrete example where this lack of flexibility becomes apparent is *parallel extension*, as described in customisation Operation 9. If domain engineers can foresee that parallel regions will be added to a specific state at a later stage, they can introduce a parallel state and a variation point upfront. Then application engineers can use command `EXTEND` to add more parallel regions. But if this customisation is not foreseen, application engineers are not able to introduce orthogonal regions for two reasons: First, the state cannot be

changed into a parallel state and second, there is no variation point to invoke the `EXTEND` command on.

Additional advantages (+) and limitations (-) have been identified for this approach:

+ Variation points are explicitly visible.
+ Common and variable features are kept in separate modules.
+ The mechanism supports options, alternatives, extension and cloning.
+ All sorts of text-based assets are supported, therefore the mechanism can also be used to customise the input and output stage.
- It is difficult to prepare and evolve the common and variant modules, especially for large, non-trivial SCXML models this becomes a maintainability issue.
- As the model grows, it becomes more and more difficult to read and understand common and variant modules before they are processed. SCXML files that are split into multiple modules and contain multiple variation and clone points are especially difficult to understand.
- Features that require changes in different places require definition of multiple variation points with unique and meaningful names. This is a usability issue which does not occur when using conditional compilation where the same feature flag can be tested in multiple places.

Conclusion

Frame technology is a powerful customisation technique that works well if variability is well understood upfront. Even though the mechanism can be used to cover all customisation scenarios, it reaches its limits if unexpected changes need to be implemented for which no variation points have been foreseen. Due to the distributed project setup and the lack of a holistic feature model, the approach is unsuitable for ELT control software development. Moreover, the approach constitutes a for ESO unknown technology that requires custom tools and significant learning effort, especially when larger and non-trivial modules need to be handled.

3.4.7. Delta Orientation

Basic Idea

A basic state machine model is modified in a stepwise manner according to a user-provided feature selection. Individual modification steps can add, modify but also remove elements. The approach is based on delta orientation from Section 2.2.3.8.

Description

Domain engineering defines a *basic model* and a set of *delta modules* that correspond to individual features. The features can be regarded as model or graph transformations that map an input model to an output model according to a transformation function.

The basic model acts as the starting point for a sequence of model transformations. The transformation function of a feature is defined as a sequence of model manipulation operations that allow the addition, modification and also the deletion of individual state machine elements. While a simple feature may only define a single manipulation operation that e.g. modifies a transition, features with large granularity may apply a long list of manipulation operations to e.g. add an entire sub-diagram.

To create a custom state machine model, application engineers provide a feature expression that specifies the basic model and the sequence of desired features. A feature composition tool ingests the expression and creates the desired state machine model by applying the provided transformation steps in sequence to the base model.

Note that application engineers may also define their own features and combine them with existing domain assets.

Prototyping

For the evaluation of this customisation technique, a custom state machine model manipulation tool was developed. The tool provides an Application Programming Interface (API) that allows the manipulation of the graph-based, in-memory representation of the state machine model. The definition of the in-memory representation can be found in Appendix B.

Model import and export functionality was added to be able to import the state machine model from textual representations and to export it again to same or different representations after manipulation.

Operation	Description
ImportModel	creates state machine model from textual representation
ExportModel	exports state machine model to textual representation

Table 4: Model import and export API.

An imperative model manipulation API was introduced that can be used to create a state machine model from scratch. The API also allows stepwise modification of an existing model.

Operation	Description
AddState	adds a new state to existing parent state
AddTrans	adds a new transition between existing states
DelState	removes a state recursively
DelTrans	removes a transition
RepState	replaces a state with another
ModStateId	renames a state
ModStateType	modifies state type
ModStateParent	modifies state parent
ModStateDoActivity	modifies do-activity of a state
ModStateEntryAction	modifies entry action of a state
ModStateExitAction	modifies exit action of a state
ModTransTarget	modifies transition target state
ModTransSource	modifies transition source state
ModTransEvent	modifies transition event
ModTransGuard	modifies transition guard
ModTransAction	modifies transition action

Table 5: Model manipulation API.

Table 5 provides an overview of supported model manipulation operations. Note that individual methods may throw exceptions if developers attempt to perform bad or forbidden operations. With this mechanism, it is possible to constrain behavioural customisation in a constructive manner.

Listing 13 shows an example for the definition of a basic state machine model using the model manipulation API. A new state machine is constructed and methods `AddState` and `AddTrans` are used to define basic states and transitions.

```
unique_ptr<StateMachine> CreateBasicModel() {

    auto model = make_unique<StateMachine>();

    //              type        id      parent  activity  entry  exit
    AddState(model, Initial,   "A",    "",     "",       "",    ""  );
    AddState(model, Composite, "B",    "",     "",       "",    ""  );
    AddState(model, Initial,   "B.1",  "B",    "",       "",    ""  );
    AddState(model, Simple,    "B.2",  "B",    "",       "",    ""  );
    AddState(model, Simple,    "B.3",  "B",    "",       "",    ""  );
    AddState(model, Final,     "C",    "",     "",       "",    ""  );

    //              source  target  event  guard  action
    AddTrans(model, "A",    "B",    "",    "",    ""    );
    AddTrans(model, "B",    "C",    "a",   "",    ""    );
    AddTrans(model, "B.1",  "B.2",  "",    "",    ""    );
    AddTrans(model, "B.2",  "B.3",  "b",   "",    ""    );
    AddTrans(model, "B.3",  "B.2",  "c",   "",    ""    );

    return model;
}
```

Listing 13: Definition of the basic model.

In Listing 14 new functionality is added to the basic state machine model introduced before. Existing states are refined and states and transitions are added. In addition, an existing guard condition is modified using method `ModTransGuard`.

```
void ApplyFeatureA(StateMachine& model) {

    //              type        id      parent  activity  entry  exit
    AddState(model, Simple,    "B.4",  "B",    "",       "",    ""  );
    AddState(model, Simple,    "D",    "",     "",       "",    ""  );

    //              source  target  event  guard  action
    AddTrans(model, "B.1",  "B.4",  "d",   "",    ""    );
    AddTrans(model, "B",    "D",    "e",   "",    ""    );

    //                   source  target  event  old guard  new guard
    ModTransGuard(model, "B.2",  "B.3",  "b",   "",        "g1");
}
```

Listing 14: Addition of a specific feature.

The tool is also able to group sequences of model manipulation operations into named features. Application developers then specify a sequence of features to be applied and the tool adapts the model accordingly.

```
DeltaTool tool;

tool.RegisterCore("BasicBehaviour", CreateBasicModel);

tool.RegisterDelta("Feature_A", ApplyFeatureA);
tool.RegisterDelta("Feature_B", ApplyFeatureB);
// ...

auto sm = tool.MakeModel("BasicBehaviour", { "Feature_A" });

cout << ExportModel(*sm, "scxml");
```

Listing 15: Core and delta module defintion and selection.

Listing 15 shows how the tool can be used to create a custom model using the previously defined model manipulation methods. First, the manipulation methods are mapped to concrete features using calls to `RegisterCore` and `RegisterDelta`. Then, the model is created by invoking method `MakeModel` that takes the name of the basic model and a list of features as arguments. Finally, the model is printed in SCXML format using method `ExportModel`.

Discussion

Delta-orientation is closely related to FOSD because it also uses the notions of features, feature selection and feature composition. The approach presented above is based on composition of model manipulation operations and thus supports both positive and negative variability. By applying sequences of additive model manipulation operations the models grows and by applying subtractive operations the model shrinks.

The approach can be used to cover customisation Scenarios 3 and 4 and it supports all customisation operations fully because it can add, modify and delete state machine elements.

Additional advantages (+) and limitations (-) have been identified for this approach:

+ The approach supports stepwise customisation and open variability

+ The approach also supports feature orientation and feature selection.

+ Features can be kept in separate delta modules, this benefits maintainability.

+ Delta modules support different granularities (small to large).

+ Custom model manipulation operations can be composed of basic operations.
+ Delta modules can be applied to different base modules, therefore something like mixin-based composition can be realised.
+ The approach supports feature interactions and feature dependencies, this enables features to behave differently in the presence or absence of related features.
- The mechanism is not well-known and is not supported by modeling tools.
- The possibility to modify and delete elements introduces additional complexity.
- Delta modules cannot be represented as statecharts because they are just sequences of modification operations applied to an unknown core.
- There is no implementation ready to use, development of a custom tool is necessary.

Conclusion

From all customisation mechanism that cover Scenario 4, delta orientation is the most promising candidate because it covers all customisation operations fully and provides most flexibility in terms of customisation.

The mechanism can be used both as a standalone model manipulation tool but also in combination with a software framework. The latter would enable the framework to cover all customisation scenarios fully.

3.5. Solution Concept

In this section the solution concept is presented. First, the most important design decisions are described. Then the high level design and the resulting customisation workflow are presented.

3.5.1. Design Decisions

The most important design decisions are summarised and explained below. They form the basis of the solution concept.

3.5.1.1. Combine Customisation Mechanisms

In Section 3.4 it was shown that there is no single customisation mechanism that can solve the problem at hand completely, therefore several mechanisms need to be combined.

- **Framework Approach.** Since most ELT control software projects already follow a platform or framework approach, it makes sense to adapt the design of the frameworks according to the description in Section 3.4.1 so that customisation scenarios 1, 2 and partially 3 can be covered.
- **Model Manipulation**. In Section 3.4.7 delta orientation was identified as the most appropriate mechanism to cover customisation scenario 4. To facilitate stepwise refinement and reuse of the state machine model, an imperative model manipulation API is added to the framework. With this API it is possible to add, modify and also delete individual state machine elements.
- **Mixin Composition.** To compensate and mitigate the drawbacks of the framework approach regarding feature selection, combinatoric explosion of required variants and amount of duplicated information, the concept of *life cycle extensions* is introduced. Instead of providing a potentially huge number of component life cycles that cover all possible feature combinations developers specify the required life cycle by selecting and combining life cycle extensions that represent specific features or capabilities. Life cycle extensions are implemented as *mixin layers* which can be combined using *parametrised inheritance* as described in Sections 2.2.3.3 and 2.2.3.6.
- **Conditional Compilation.** To realise well-understood, fine-grained variability inside a specific life cycle extension, conditional compilation is used. Fine granular variation points can be added to a life cycle extension so that it can be customised at compile time via parametrisation (see Section 2.2.3.2).

- **Conditional Execution.** To support well-understood variability that needs to be evaluated dynamically at run-time, conditional execution is used as described in Section 3.4.2. The state machine elements used to realise conditional execution are transitions with guards.

Table 6 shows a requirements coverage matrix that describes which requirements from Section 3.3 are covered by which customisation mechanism. Full coverage is depicted with `+`, partial coverage with `o` and requirements that are not covered at all are marked with `-`. Note that only the combination of the selected mechanisms covers all applicable requirements.

Customisation Mechanism	R1	R2	R3	R4	R5	R6	R7	R8
Framework Approach	o	o	+	+	+	+	+	+
Model Manipulation	o	o	+	+	+	+	+	+
Mixin Composition	o	o	+	+	+	+	+	+
Conditional Compilation	o	-	-	-	+	+	+	+
Conditional Execution	o	-	-	-	+	+	+	+
Overall Concept	+	+	+	+	+	+	+	+

Table 6: Requirements coverage matrix.

3.5.1.2. Use Conformance Testing

In addition to the state machine model, also the input and the output stage have a major impact on the component life cycle, therefore they also need to be taken into account when ensuring compatibility.

Conformance testing is used to validate behavioural consistency of customised life cycles in an end-to-end manner. Specific test cases ensure that the application of new life cycle extensions does not break compatibility with the basic behaviour. The tests check both invocation and observation consistency by applying the most important activation sequences and checking resulting replies, guard, action and activity invocations, as well as the sequence of visited states.

To realise this, all framework-provided basic life cycles and life cycle extensions come with associated conformance test suites that ensure that the behaviour of the respective layer is correct and consistent. These test suites can also be used to check if a customised life cycle is still consistent with the framework-provided basic behaviour. In Figure 28 a component hierarchy is shown. In this case, a conformance test suite for `EltComponents` can be used to check if different flavours of `RtcComponents` are still compatible with the basic `EltComponent` life cycle. When new life cycle extensions are introduced by application

engineering, existing conformance test suites from the framework can again be used to validate that the resulting behaviour is still compatible with basic, framework-provided behaviour.

Even though conformance testing can only prove the existence but not the absence of compatiblity issues, it is still considered a ”good enough” approach to cover requirement R9. In addition, testing is common practise at ESO, therefore, the approach can be implemented at relatively low cost. Note that it is still possible to combine conformance testing with formal verification as described in Section 2.4.2.2, but since ESO does not have much experience in this field, conformance testing is considered more appropriate.

3.5.1.3. Manage Configuration Complexity

The possibility to assemble custom component life cycles from framework-provided and also application-specific building blocks provides the required flexibility but it also introduces additional complexity that can cause maintainability and compatibility issues.

This section introduces important tactics that can be used to limit complexity and thus prevent developers from making mistakes.

Provide Comprehensive Documentation

The software framework should come with comprehensive documentation regarding life cycle customisation. The documentation shall be kept up-to-date and shall include the following information:

- a general description of the available customisation mechanisms
- a feature model describing available life cycles and extensions
- a description of the development workflow including the customisation workflow
- advice on when to use which customisation mechanism
- a description of relevant behavioural consistency levels

Moreover, a reference implementation or concrete examples and tutorials should be provided to show how the mechanisms are used in practice.

Check Feature Dependencies

Developers need to be prevented from selecting and combining incompatible life cycle extensions. This can be achieved by introducing a feature model that specifies relations between available basic life cycles and life cycle extensions. Useful relations to control allowed feature combinations are *requires* and *excludes*.

At compile time, the provided life cycle expression can be checked using the information from the feature model and compilation can be aborted with an error message if an invalid expression is detected.

Since the selected approach is based on open variation, the framework-provided feature model is incomplete. When new life cycle extensions are introduced as application assets, the information from the feature model may no longer be sufficient to check if the feature selection is still valid. In this case, additional tactics need to be applied to prevent errors.

Constrain Open Variation

To further control open variation, modification of the state machine model can be constrained so that only certain customisation operations from Section 3.2.4 can be applied to specific state machine elements.

The model manipulation API from Section 3.4.7 can be enhanced so that certain unwanted customisation operations can be detected and rejected. Meaningful error messages can be generated to inform developers why a specific operation was rejected. In addition, a model validation step can be introduced to check that the fully customised model is well-formed and consistent.

It is also possible to add meta-data to individual state machine elements that provides concrete information whether future modification of the element is allowed or not. A concrete example is the introduction of *abstract*, *standard* and *locked* states and transitions as proposed by Hansen et al. in [HSL15]. While *abstract* elements require future refinement, *standard* elements do not require but allow modification at a later stage. *Locked* elements on the other hand are considered final cannot be modified after they have been added.

The built-in model checking and validation facilities can use this meta-data to disallow certain operations or to notify developers that certain elements still need refinement.

Use Introspection

Introspection functionality enables developers and also client applications to retrieve the capabilities of a component at run-time. On request, the component can report its basic life cycle and also the sequence of applied life cycle extensions. This information can be used for debugging, but it can also be used by client applications to dynamically figure out the capabilities and thus the communication protocol of components.

In addition, the possibility to export the resulting state machine model in various formats is useful for manual debugging and for visualisation of the component life cycle in generated documentation.

3.5.2. High Level Design

This section describes the high level design and formulates important concepts more clearly. In addition, concrete examples are shown to make the rather abstract design more tangible.

3.5.2.1. Component Structure

To support the combination of mechanisms, the component model is modified so that a component can be parametrised with a `LifeCycle` and with a `BusinessLogic`. During compilation, the life cycle and the business logic are bound to the overarching component structure. Note that this type of customisation can also be regarded as a twofold application of the strategy pattern where both the selected life cycle and also the provided business logic are concrete strategies.

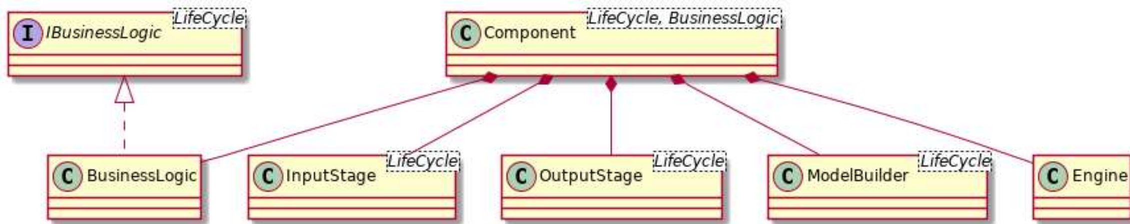


Figure 44: Component structure.

By providing a specific `LifeCycle` class, parts of the component that depend on the life cycle can be adjusted or even replaced. Similarly, a custom `BusinessLogic` class that provides application specific implementations for action and activity methods can be provided. Note that the abstract methods of the business logic class are defined in the life cycle dependent business logic interface.

In addition, the component now also hosts a life cycle dependent `ModelBuilder` class that is responsible for creating and adjusting the state machine model using the imperative model manipulation API. This class can also be used to import a base model in some textual representation and to export the customised model to various formats that can be used for debugging, visualisation and most importantly for subsequent interpretation by the state machine engine.

3.5.2.2. Life Cycle

To separate concerns, the `LifeCycle` class is implemented as a nested class with four inner classes.

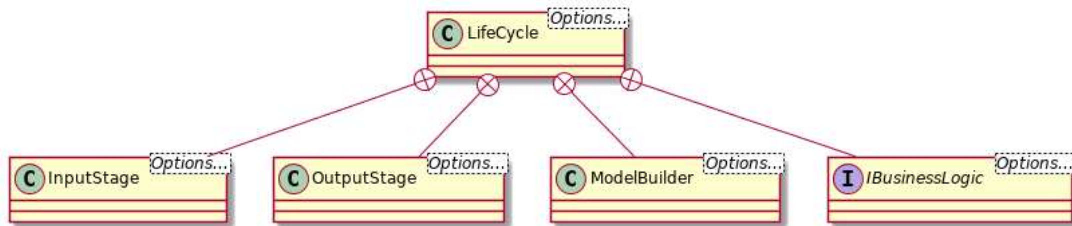


Figure 45: Component life cycle.

The `InputStage` receives commands and translates them to state machine events. The `ModelBuilder` creates or modifies the state machine model used for interpretation. The `OutputStage` delegates actions and activities to the business logic. Interface `IBusinessLogic` defines hook methods that need to be implemented by application developers.

A `LifeCycle` class can also be parametrised to support well-known variability. By providing `Options`, developers can select pre-defined variants that are bound at compile time.

3.5.2.3. Life Cycle Extension

To allow for stepwise customisation and open variation, the component life cycle can be customised using mixin composition. A `LifeCycleExtension` class is a mixin layer that can customise a `LifeCycle` class by extending any of its inner classes. An `ExtendedLifeCycle` can be created by applying multiple life cycle extensions sequentially to a `BasicLifeCycle`.

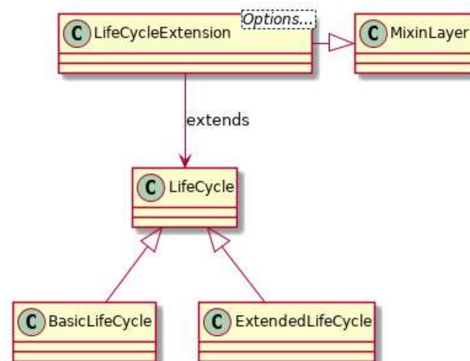


Figure 46: Life cycle extension.

By extending the `InputStage`, new commands can be added and then mapped to new or existing state machine events. When extending the `ModelBuilder` class, the state machine model can be customised by either adding, modifying or removing elements. By

extending the `OutputStage`, replies of new commands can be handled and new actions and activities can be delegated to the user-provided business logic. Finally, extending interface `IBusinessLogic` allows making new actions and activities available to business logic implementers.

A `LifeCycleExtension` can be parametrised to support well-known variability. By providing `Options`, developers can select pre-defined variants that are bound at compile time.

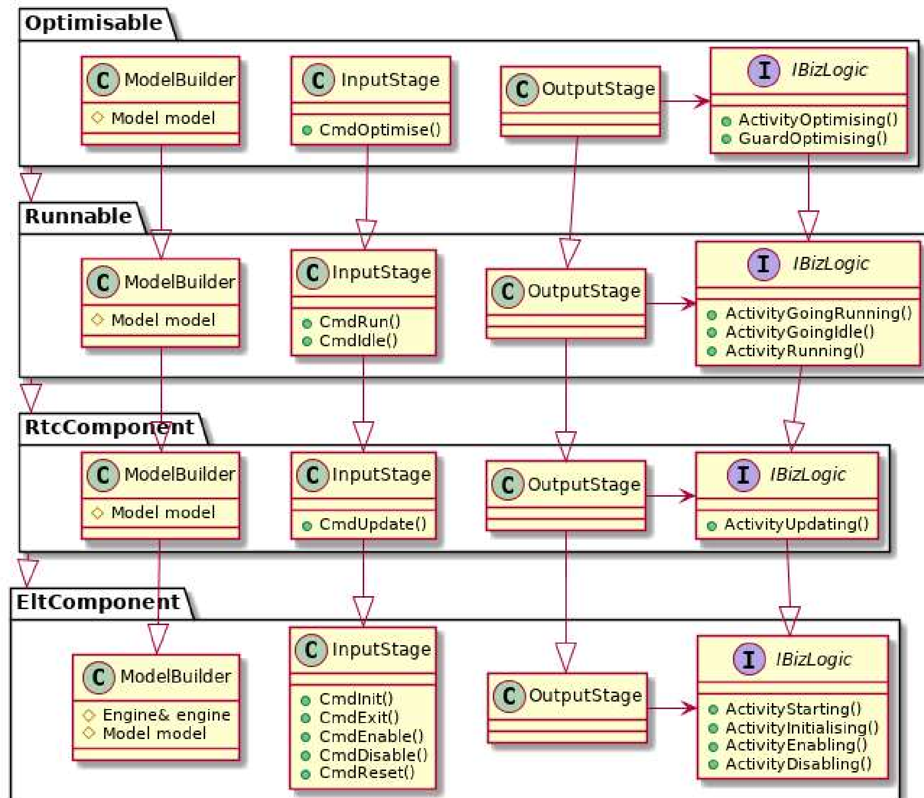


Figure 47: Example customisation.

Figure 47 shows a concrete example of a custom component life cycle that was created by combining basic life cycle `EltComponent` with life cycle extensions `RtcComponent`, `Runnable` and `Optimisable`. Note that a life cycle extension only needs to define a certain inner class if that class needs to be customised.

3.5.2.4. Life Cycle Expression

Developers specify a `LifeCycleExpression` to assemble the desired component life cycle from a set of basic life cycles and life cycle extensions. This expression basically defines which features or capabilities a component shall have.

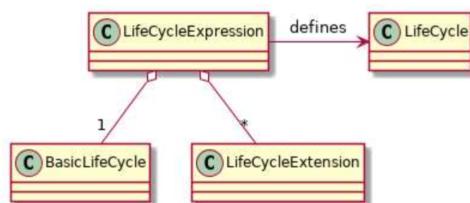


Figure 48: Life cycle expression.

The `LifeCycleExpression` defines which `LifeCycleExtensions` are applied to a `BasicLifeCycle` and it also defines the order in which the extensions are applied. The expression can also define options for fine-grained configuration of the basic life cycle and of individual life cycle extensions.

A life cycle expression that describes the feature selection from Figure 47 could look like:

```
LifeCycle = EltComponent + RtcComponent + Runnable + Optimisable
```

The resulting `LifeCycle` consists of basic life cycle `EltComponent` and life cycle extensions `RtcComponent`, `Runnable`, `Optimisable`. Also note that this notation is only an example to explain the general concept, the precise syntax will be defined later in the realisation.

3.5.3. Customisation Workflow

To create a new component application engineers perform the following steps:

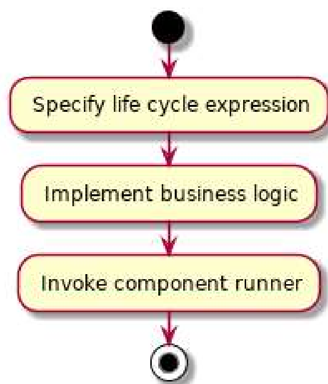


Figure 49: Customisation workflow.

1. **Specify life cycle expression.** First, application developers select the desired component life cycle by providing a life cycle expression that specifies the basic life cycle and the sequence of applied life cycle extensions including configuration options.

2. **Implement business logic.** Next, a custom business logic class that satisfies the business logic interface of the selected life cycle is implemented. Here, developers provide custom implementations for pre-defined guard, action and activity hooks.
3. **Invoke component runner.** Finally, the component is started by invoking the framework-provided component runner method with the life cycle expression and the custom business logic.

In the majority of cases the specification of the component life cycle is expected to be as simple as selecting a basic life cycle or combining and configuring several framework-provided life cycle extensions. However, in some cases things can be more involved: In case the framework does not support the required life cycle extension, developers may need to implement custom life cycle extensions themselves.

Question	Recommended Action
Can a basic life cycle be used?	Select and configure basic life cycle.
Does the framework provide desired extensions?	Select and configure extensions.
Should missing functionality be a domain asset?	Ask domain engineering to provide it.
Otherwise	Create custom life cycle extension.

Table 7: Customisation heuristics.

Table 7 provides important heuristics that developers should use when creating custom components with special life cycles.

Traditional vs. Model-Based Workflow

The customisation workflow presented above can be applied to both traditional and also to model based development:

- When using the traditional software engineering workflow, the life cycle expression, the business logic and the invocation of the component runner method are specified directly in code. To further simplify component creation, it is also possible to provide a supporting tool that generates the application skeleton code for a provided life cycle expression.
- When using the MDA based workflow, the life cycle expression and the name of the business logic class are specified in the modeling tool. A specific COMODO plug-in then parses the model and creates the application skeletons that use the enhanced framework as a platform. Also in this case, developers have to manually implement the business logic class.

4. Realisation

This chapter describes how the solution concept from Section 3.5 was realised. Implementation work was carried out in scope of the ELT RTC Toolkit project using C++ version 17. To keep the focus on relevant key aspects, only code snippets are presented in this chapter. Many implementation details are omitted to keep the code snippets concise and to not overwhelm the reader with implementation details. Additional code listings that also contain omitted parts can be found in the appendix. Moreover, concrete tutorials and additional information can be found in the ELT RTC Toolkit user manual (see [ESO21b]).

4.1. Use Case for Realisation

To demonstrate soundness and feasibility of the solution concept, a concrete use case from the ELT RTC Toolkit project was selected for realisation. The use case is based on the state machine diagrams introduced in Section 3.1 and it covers the customisation scenarios identified in Section 3.2.3.

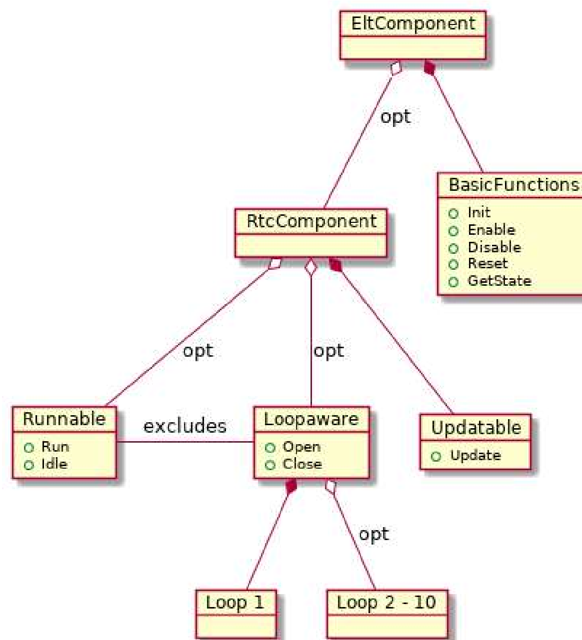


Figure 50: Use case for realisation.

Figure 50 shows a feature model of the selected use case. Core component functionality is covered by basic life cycle `EltComponent`. The life cycle can be refined into an `RtcComponent` which adds AO RTC specific functionality. While each `RtcComponent` is `Updatable`, dedicated life cycle extensions are created for optional features `Runnable` and `Loopaware`. The latter can be parametrised to support 1 to 10 control loops.

4.2. Common Infrastructure

In this section, the realisation of common infrastructure libraries and classes is described. The prototype from Section 3.4.1 was used as a basis for implementation. The code was adapted to support the enhanced component structure introduced in Section 3.5.2.

4.2.1. Component Runner

The generic component runner method `RunComponent()` is introduced to be able to run any user-provided business logic with framework-provided infrastructure and services.

```
template<typename LC, typename BL>
int RunComponent() {
    static_assert(is_base_of_v<LC::IBizLogic, BL>);

    Services services;
    Replier replier;
    Publisher publisher;
    Engine engine;

    BL logic(services);
    typename LC::InputStage input_stage;
    typename LC::OutputStage output_stage(engine, logic);
    typename LC::ModelBuilder model_builder;

    auto in_memory_model = model_builder.MakeModel();
    auto scxml_model = ExportModel(*in_memory_model, "scxml");
    cout << scxml_model << endl;
    engine.LoadModel(scxml_model);

    engine.RegisterStateChangedCB([&publisher](auto state) {
                publisher.Publish(state);
            });

    input_stage.Start(engine, replier);
    return engine.Work();
}
```

Listing 16: Generic component runner method.

Listing 16 shows the component runner method with template arguments `LC` for the life cycle and `BL` for the business logic. A static assertion is used to ensure at compile time that the provided business logic class `BL` implements `IBizLogic`.

First, objects for common support classes, the business logic class `BL` and the life cycle dependent classes `InputStage`, `OutputStage` and `ModelBuilder` are instantiated. The `input_stage` receives commands and posts events to the state machine engine. The `output_stage` registers callbacks for guards, actions and activities to the state machine `engine` and delegates application specific methods to the business logic. The `model_builder` object creates and modifies the state machine model.

Next, `model_builder.MakeModel()` is invoked to assemble the state machine model. After converting the `in_memory_model` to SCXML format, the `scxml_model` is printed to standard output for debug purpose and registered to the state machine engine by calling `engine.LoadModel()`.

Then, a callback for state publication is registered with `engine.RegisterStateChangedCB()` and command reception is started using method `input_stage.Start()`.

Finally, control is handed over to the event loop of the state machine engine using method `engine.Work()`.

4.2.2. Support Libraries

Several support libraries are required to run the component. Since they are not essential for understanding the realisation, they are only briefly introduced here:

- **Services.** A container providing different services to the business logic. Concrete examples for services are configuration, persistence, database access and logging.
- **Replier.** A class used for receiving commands over the network. Commands are realised using the *request-reply* messaging pattern.
- **Publisher.** A class used for publishing the current state over the network. Here the *publish-subscribe* messaging pattern is used to notify remote subscribers.
- **Engine.** A custom wrapper around the *scxml4cpp* state machine engine. The wrapper is used to simplify setup and configuration of the state machine engine.

4.3. Life Cycles and Extensions

In this section, the implementation of basic life cycles and different life cycle extensions is described. A top down approach is used. First, the overall class structure is described. Then, relevant implementation details of the specific classes are shown and discussed.

4.3.1. Defining a Basic Life Cycle

A basic life cycle is defined using a nested class with several inner classes that define specific aspects of the life cycle. This approach was selected over a single god-class approach to be able to separate concerns and improve understandability and maintainability. Note that in C++ classes can be defined with keywords `class` or `struct`.

```
struct EltComponent {
    struct Events { /*...*/ };
    struct InputStage : IInputStage { /*...*/ };
    struct ModelBuilder : IModelBuilder { /*...*/ };
    struct IBizLogic { /*...*/ };
    struct OutputStage { /*...*/ };
};
```

Listing 17: Basic life cycle: structure.

Listing 17 shows the overall structure of basic life cycle `EltComponent`. The following inner classes are defined: `Events` for state machine event definition, `InputStage` for command reception and event injection, `ModelBuilder` for model creation and manipulation, `IBIzLogic` for the definition of abstract methods to be implemented by application engineers and `OutputStage` for command reply sending and invocation of user-provided code.

Event Definition

The *scxml4cpp* state machine engine requires the definition of state machine events. Each event is defined as a data type in inner struct `Events` using a convenience macro. The events are used or referred to by other inner classes, e.g. for defining the state machine model, for event injection or for sending of command reply messages.

```
struct Events {
    DEFINE_EVENT (Init,      CommandEvent);
    DEFINE_EVENT (Enable,    CommandEvent);
    DEFINE_EVENT (Disable,   CommandEvent);
    DEFINE_EVENT (Reset,     CommandEvent);
    DEFINE_EVENT (GetState,  CommandEvent);
```

```
    DEFINE_EVENT (Done,      InternalEvent);
    DEFINE_EVENT (Error,     InternalEvent);
};
```

Listing 18: Basic life cycle: events.

Listing 18 shows the definition of events `Init`, `Enable`, `Disable`, `Reset`, `GetState`, `Done` and `Error`. Note that an event can be a `CommandEvent` that supports command payloads and reply messages or an `InternalEvent` that is used internally to signal that activities have completed or failed.

Input Stage

The input stage is responsible for command reception and event injection. To achieve this, command handlers are registered to a replier. Note that one command handler can manage multiple commands. On command reception, command events are created and injected to the state machine engine. A future-based interface is used for handling command replies, this was done to be in line with the asynchronous communication pattern used in the ELT control system.

```
struct InputStage : IInputStage {

    void Start(Replier& replier, Engine& engine) override {
        // register and start command handlers that inject events
        StdCmdHandler::Register(replier, engine);
    }

    class StdCmdHandler {
        // ...
        future<string> OnCmdReceived(Cmd cmd) {
            if(cmd.id == "Init")
                return InjectEvent<Events::Init>(engine, cmd.payload)
            // ...
        }
    };
};
```

Listing 19: Basic life cycle: commands.

Listing 19 shows how command handler `StdCmdHandler` can be defined, registered and started in the input stage. Template method `InjectEvent` is used to create event instances and post them to the state machine engine. Note that the exact implementation of the command handler is omitted here, because it highly depends on the used middleware.

Model Builder

The inner class for the model manipulator is used to create and manipulate the state machine model. The class implements interface `IModelBuilder` which requires overriding method `MakeModel`. The method is used to create the model using the mechanisms and libraries introduced in Section 3.4.7.

```
struct ModelBuilder : IModelBuilder {

    unique_ptr<Model> MakeModel() override {
        auto model = make_unique<Model>("sm");

        AddState(*model, Initial,   "Initial");
        AddState(*model, Composite, "On");
        AddState(*model, Final,     "Off");
        AddState(*model, Initial,   "On.Initial",        "On");
        AddState(*model, Composite, "On.NotOperational", "On");
        AddState(*model, Simple,    "On.Operational",    "On");
        // ...

        AddTrans(*model, "Initial", "On");
        AddTrans(*model, "On",      "Off",
                         "events.Exit", "", "ActionExit");
        AddTrans(*model, "On",      "",
                        "events.GetState", "", "ActionGetState");
        AddTrans(*model, "On",      "On",
                         "events.Reset");
        AddTrans(*model, "On.Initial", "On.NotOperational");
        AddTrans(*model, "On.NotOperational.Initial",
                         "On.NotOperational.Starting");
        // ...

        return model;
    }
};
```

Listing 20: Basic life cycle: model builder.

Listing 20 shows how the basic state machine model can be defined using the model manipulation API. The complete definition can be found in Appendix C. Note that it is also possible to import the basic state machine model from an SCXML file using API method `ImportModel()`.

Business Logic Interface

The business logic interface defines virtual methods that need to be implemented by application engineering. Currently, methods for guards, actions and activities are supported. The methods are prefixed with their type for better understanding. The virtual methods can either be defined as pure virtual, or they can be defined with a default implementation. While the former forces application engineers to implement the method, the latter makes the implementation of the method optional.

```
struct IBizLogic {

    virtual void ActivityStarting() { };
    virtual void ActivityInitialising() = 0;
    virtual void ActivityEnabling() = 0;
    virtual void ActivityDisabling() = 0;
};
```

Listing 21: Basic life cycle: business logic interface.

Listing 21 shows the inner class that defines the business logic interface. Note that `ActivityStarting` has an empty default implementation while the other activity methods are pure virtual. This means, they must be implemented by application engineering.

Output Stage

The output stage contains glue code that connects the state machine engine with the virtual methods defined in the business logic interface. In addition, the class is also responsible for sending command replies and handling succeeded and failed activities. As a result of having multiple responsibilities the implementation of the output stage can become quite complex.

```
struct OutputStage {

  OutputStage(Engine& engine, IBizLogic& bl) {

    engine.RegisterRejectHandler(Events::Init, [&](auto ev) {
        static_pointer_cast<Events::Init>(ev)->SetValue("REJECTED");
    });

    engine.RegisterActivity("ActivityInitialising", [&](auto ev) {
        tmp_ev = ev;
        try {
            bl.ActivityInitialising();
            engine.PostEvent(make_shared<Events::Done>());
```

```
        }catch(...) {
            engine.PostEvent(make_shared<Events::Error>());
        }
    });

    engine.RegisterAction("ActionInitDone"), [&](auto ev){
        static_pointer_cast<Events::Init>(tmp_ev)->SetValue("OK");
        tmp_ev.reset();
    }

    engine.RegisterAction("ActionInitFailed"), [&](auto ev){
        static_pointer_cast<Events::Init>(tmp_ev)->SetValue("FAILED");
        tmp_ev.reset();
    }

    // ...
  }

  shared_ptr<Event> tmp_ev;
};
```

Listing 22: Basic life cycle: output stage.

Listing 22 shows the inner class that defines the output stage. Implementation of the class requires the definition and registration of callbacks for guards, actions, activities and rejected events. This is done using modern C++ lambda expressions to keep the code short and compact.

Note that only a simplified implementation of functionality related to event `Init` is shown here to keep the listing short and readable. The omitted functionality is handled in a similar manner.

First, a reject handler for event `Init` is registered with `engine.RegisterRejectHandler()`, the handler returns a reply message with text `"REJECTED"` if the state machine engine does not accept the received event in the current state. Then, `engine.RegisterActivity()` is called to register the callback for `ActivityInitialising`. In the activity callback the event is first retrieved and temporarily stored, then the activity method from user-provided business logic is invoked and event `Done` or `Error` is posted to the state machine engine to signal the outcome of the business logic invocation. Next, the action handlers are registered using `engine.RegisterAction()`. The handlers take care of sending the command reply message using the temporarily stored event.

4.3.2. Extending a Basic Life Cycle

A new basic life cycle can be defined by extending an existing basic life cycle using inheritance. To achieve this, a life cycle extension is created that extends and refines a specific, hard-coded basic life cycle.

```
struct RtcComponent : EltComponent {
    struct Events : EltComponent::Events { /*...*/ };
    struct InputStage : EltComponent::InputStage { /*...*/ };
    struct ModelBuilder : EltComponent::ModelBuilder { /*...*/ };
    struct IBizLogic : EltComponent::IBizLogic { /*...*/ };
    struct OutputStage : EltComponent::OutputStage { /*...*/ };
};
```

Listing 23: Extended life cycle: structure.

Listing 23 shows how this kind of customisation can be realised. Both the outer and the inner classes of the new basic life cycle `RtcComponent` extend the existing basic life cycle `EltComponent`. With this approach, new functionality can be added on top of existing functionality with a high level of reuse.

Event Definition

New events can be added to the life cycle by extending struct `Events` as shown in Listing 24.

```
struct Events : EltComponent::Events {
    DEFINE_EVENT (Update,      CommandEvent);
    DEFINE_EVENT (UpdateDone,  InternalEvent);
    DEFINE_EVENT (UpdateError, InternalEvent);
};
```

Listing 24: Extended life cycle: events.

Input Stage

In the input stage additional command handlers can be registered to be able to receive new commands. On command reception, the commands are mapped to new or existing events which are then injected into the state machine engine.

```
struct InputStage : EltComponent::InputStage {

    void Start(Replier& replier, Engine& engine) override {
        EltComponent::InputStage::Start(replier, engine);
        UpdateCmdHandler::Register(replier, engine);
```

```
    }
    class UpdateCmdHandler {
        // ...
        future<string> OnCmdReceived(Cmd cmd) {
            if(cmd.id == "Update")
                return InjectEvent<Events::Update>(engine, cmd.payload)
            // ...
        }
        // ...
    };
};
```

Listing 25: Extended life cycle: commands.

Listing 25 shows the registration of a new command handler to support command `Update`. Note that in method `Start()` the command handler of the super class is registered first, afterwards the new command handler is added.

Model Builder

In the inner class for the model builder the state machine model can be adjusted using the model manipulation API introduced in Section 3.4.7. Using this mechanism, developers can define their own state machine inheritance semantics without being constrained by the limitations of some third-party, graphical modeling tool.

```
struct ModelBuilder : EltComponent::ModelBuilder {

    unique_ptr<Model> MakeModel() override {
        auto model = EltComponent::ModelBuilder::MakeModel();

        ModStateType(*model, "On.Operational", Parallel);

        AddState(*model, Composite, "On.Operational.RegionUpdate",
                                    "On.Operational");
        AddState(*model, Initial,   "On.Operational.UpdateInitial",
                                    "On.Operational.RegionUpdate");
        AddState(*model, Simple,    "On.Operational.UpdateIdle",
                                    "On.Operational.RegionUpdate");
        AddState(*model, Simple,    "On.Operational.UpdateBusy",
                                    "On.Operational.RegionUpdate",
                                    "ActivityUpdating");

        AddTrans(*model, "On.Operational.UpdateInitial",
                         "On.Operational.UpdateIdle");
```

```
        AddTrans(*model, "On.Operational.UpdateIdle",
                         "On.Operational.UpdateBusy",
                         "events.Update",
                         "GuardUpdatingAllowed");
        AddTrans(*model, "On.Operational.UpdateBusy",
                         "On.Operational.UpdateIdle",
                         "events.UpdateDone",
                         "",
                         "ActionUpdatingDone");
        AddTrans(*model, "On.Operational.UpdateBusy",
                         "On.Operational.UpdateIdle",
                         "events.UpdateError",
                         "",
                         "ActionUpdatingFailed");

        ModTransGuard(*model, "On.Operational",
                              "On.NotOperational.Disabling",
                              "events.Disable",
                              "",
                              "GuardDisablingAllowed");

        return model;
    }
};
```

Listing 26: Extended life cycle: model builder.

Listing 25 shows the manipulation of the basic state machine model in the model builder class. First, the model builder of the basic life cycle is invoked and then the model is adjusted. The type of state `On.Operational` is changed to `Parallel` and a new orthogonal region with sub-states and transitions is added. Finally, the transition with event `Disable` is modified to add new guard condition `GuardDisablingAllowed`.

Business Logic Interface

By extending `IBizLogic` new virtual methods for guards, actions or activities can be added. If needed, the methods can come with default implementations.

```
struct IBizLogic : EltComponent::IBizLogic {

    virtual bool GuardUpdatingAllowed() { return true; };
    virtual void ActivityUpdating() { };
};
```

Listing 27: Extended life cycle: business logic interface.

Output Stage

In the extension of the output stage glue code is added for sending command reply messages and for connecting the state machine engine with the methods defined in the business logic interface extension.

```
struct OutputStage : EltComponent::OutputStage {

 OutputStage(Engine& engine, IBizLogic& bl) {

   engine.RegisterRejectHandler(Events::Update, [&](auto ev) {
       static_pointer_cast<Events::Update>(ev)->SetValue("REJECTED");
   });

   engine.RegisterGuard("GuardUpdatingAllowed"), [&](auto ev){
       for(auto& state : no_update_in_states) {
           if (engine.GetState().find(state) != string::npos) {
               return false;
           }
       }
       return bl.GuardUpdatingAllowed()
   }

   engine.RegisterActivity("ActivityUpdating", [&](auto ev) {
       tmp_ev = ev;
       try {
           bl.ActivityUpdating();
           engine.PostEvent(make_shared<Events::UpdateDone>());
       }catch(...) {
           engine.PostEvent(make_shared<Events::UpdateError>());
       }
   });

   engine.RegisterAction("ActionUpdateDone"), [&](auto ev){
       static_pointer_cast<Events::Update>(tmp_ev)->SetValue("OK");
       tmp_ev.reset();
   }

   engine.RegisterAction("ActionUpdateFailed"), [&](auto ev){
       static_pointer_cast<Events::Update>(tmp_ev)->SetValue("FAILED");
       tmp_ev.reset();
   }

   // ...
 }
```

```
  shared_ptr<Event> tmp_ev;
  list<string> no_update_in_states;
};
```

Listing 28: Extended life cycle: output stage.

Listing 28 shows the registration of callbacks related to event `Update`. Note that the extension also introduces member `no_update_in_states`, which is a black-list of states in which no update is allowed. The black-list is used in `GuardUpdatingAllowed` to allow for stepwise refinement of the guard condition. This had to be done, since evaluation of complex guard expressions is not supported by *scxml4cpp*.

4.3.3. Defining a Life Cycle Extension

Life cycle extensions that do not require a hard-coded basic life cycle are implemented as *mixin layers*. As suggested in Section 3.5.1.1, parametrised inheritance is used and the basic life cycle is provided via template parameter `Super`.

```
template <typename Super>
struct Runnable : Super {

    static_assert(is_base_of_v<RtcComponent, Super>,
        "Error: Runnable requires RtcComponent!");
    static_assert(not is_base_of_template_v<Loopaware, Super>,
        "Error: Runnable excludes Loopaware!");

    struct Events : Super::Events { /*...*/ };
    struct InputStage : Super::InputStage { /*...*/ };
    struct ModelBuilder : Super::ModelBuilder { /*...*/ };
    struct IBizLogic : Super::IBizLogic { /*...*/ };
    struct OutputStage : Super::OutputStage { /*...*/ };
};
```

Listing 29: Life cycle extension: structure.

Listing 29 shows the definition of life cycle extension `Runnable`. The basic life cycle is provided via template argument `Super`.

Feature Dependencies and Interactions

Static assertions and type traits can be used to define constraints for the provided life cycle. With this mechanism, a feature model that supports the notions of *requires* and

excludes can be encoded directly into the life cycle extensions, as suggested in Section 3.5.1.3.

The same mechanism can also be used to support feature interactions:

```
struct ModelBuilder : Super::ModelBuilder {

    unique_ptr<Model> MakeModel() override {
        auto model = Super::ModelBuilder::MakeModel();

        if (is_base_of_v<ExtensionXY, Super>) {
            // add some states and transitions
        }else{
            // add different states and transitions
        }

        return model;
    }
};
```

Listing 30: Life cycle extension: model builder.

Listing 30 shows an example of how the state machine model can be modified in a different way depending on the presence of a specific life cycle `ExtensionXY`.

4.3.4. Defining a Parametrised Life Cycle

Life cycle extensions can also accept additional template parameters so that well-known variability inside a specific life cycle can be covered. As a customisation mechanism, conditional compilation is used as suggested in Section 3.5.1.1.

```
template <typename Super, unsigned NumLoops=1>
struct Loopaware {
    static_assert(NumLoops>=1 and NumLoops<=10,
        "Error: Only 1-10 loops supported!");

    struct Events : Super::Events { /*...*/ };
    struct InputStage : Super::InputStage { /*...*/ };
    struct ModelBuilder : Super::ModelBuilder { /*...*/ };
    struct IBizLogic : Super::IBizLogic { /*...*/ };
    struct OutputStage : Super::OutputStage { /*...*/ };
};
```

Listing 31: Parametrised extension: structure.

Listing 31 shows the definition of parametrised life cycle extension `Loopaware`. The extension accepts the additional *non-type template parameter* `NumLoops` that specifies the number of control loops.

Even though usage of parametrised life cycles is very powerful and flexible, it also leads to non-trivial implementations that require a good understanding of C++ template metaprogramming techniques. This becomes apparent in the implementation of the inner classes.

Event Definition

In a parametrised life cycle the existence of certain events may depend on the provided parameters. A special mechanism is necessary to realise this by manipulating the implementation of structs and classes at compile-time.

```
template<typename T>
struct Empty {};

struct EventsLoop1 {
    DEFINE_EVENT (Close1,     CommandEvent);
    DEFINE_EVENT (Open1,      CommandEvent);
    DEFINE_EVENT (Execute1,   CommandEvent);
    DEFINE_EVENT (Abort1,     CommandEvent);
    DEFINE_EVENT (Done1,      InternalEvent);
};

struct EventsLoop2 {
    // same with index 2
};

// ..

struct Events
    : Super::Events
    , std::conditional_t<NumLoops>=1, EventsLoop1, Empty<EventsLoop1>>
    , std::conditional_t<NumLoops>=2, EventsLoop2, Empty<EventsLoop2>>
    // ...
{};
```

Listing 32: Parametrised extension: events.

Listing 32 shows how event definitions can be conditionally added to the hosting struct `Events` at compile-time. Modern C++ feature `std::conditional_t` is used to add a certain event group (e.g. `EventsLoop1`) to struct `Events` only if a specific condition is met. Note that the same technique can also be used to conditionally widen the business logic interface.

Model Builder

In the model builder, the provided template parameters can be used as variation points. Depending on the existence or the value of supplied parameters, different variants of the state machine model can be created.

```
struct ModelBuilder : Super::ModelBuilder {

    unique_ptr<Model> MakeModel() override {
        auto model = Super::ModelBuilder::MakeModel();

        for(unsigned loop=1; loop<=NumLoops; loop++) {
            AddRegion(*model, to_string(loop));
        }
        return model;
    }

    void AddRegion(Model& m, string idx) {
        string region_id = "On.Operational.Region"+idx;

        AddState(m, Composite, region_id,
                               "On.Operational");
        AddState(m, Initial,   region_id+".Initial",
                               region_id);
        AddState(m, Simple,    region_id+".Open",
                               region_id);
        AddState(m, Composite, region_id+".Closed",
                               region_id);
        AddState(m, Initial,   region_id+".Closed.Initial",
                               region_id+".Closed");
        AddState(m, Simple,    region_id+".Closed.Idle",
                               region_id+".Closed");
        AddState(m, Simple,    region_id+".Closed.Executing",
                               region_id+".Closed",
                               "ActivityExecuting"+idx);

        AddTrans(m, region_id+".Initial",
                    region_id+".Open");
        AddTrans(m, region_id+".Open",
                    region_id+".Closed",
                    "events.Close"+idx,
                    "",
                    "ActionCloseDone"+idx);
        AddTrans(m, region_id+".Closed",
                    region_id+".Open",
```

```
                    "events.Open"+idx,
                    "",
                    "ActionOpenDone"+idx);
        AddTrans(m, region_id+".Closed.Initial",
                    region_id+".Closed.Idle");
        AddTrans(m, region_id+".Closed.Idle",
                    region_id+".Closed.Executing",
                    "events.Execute"+idx,
                    "",
                    "ActionExecutingEntry"+idx);
        AddTrans(m, region_id+".Closed.Executing",
                    region_id+".Closed.Idle",
                    "events.Done"+idx,
                    "",
                    "ActionExecutingDone"+idx);
        AddTrans(m, region_id+".Closed.Executing",
                    region_id+".Closed.Idle",
                    "events.Abort"+idx,
                    "",
                    "ActionExecutingFailed"+idx);
    }
};
```

Listing 33: Parametrised extension: model builder.

Listing 33 shows how a configurable number of parallel regions with sub-states and transitions can be added depending on the supplied template parameter `NumLoops`. By using the model manipulation API, an arbrary number of regions can be populated with the same code just by providing the region index as an argument.

4.3.5. Defining Life Cycle Aliases

It is also possible to define a new basic life as an alias of an existing basic life cycle and a list of life cycle extensions. This allows framework developers to define shorthands for frequently used life cycles.

```
using RunnableRtcComponent = Runnable<RtcComponent>;

template <unsigned NumLoops>
using LoopawareRtcComponent = Loopaware<RtcComponent, NumLoops>;
```

Listing 34: Life cycle alias definition.

Listing 34 shows two examples for defining life cycle aliases. In the first example, a *type alias* is used to define life cycle `RunnableRtcComponent`. In the second example, an *alias template* is used to define parametrisable life cycle `LoopawareRtcComponent`.

4.4. Creating Custom Applications

This section describes how the developed customisation technique can be used to create custom applications. The customisation workflow introduced in Section 3.5.3 is used.

In case application engineers need to define their own custom life cycle extensions they can do this as described in the previous section. Note that life cycle extensions created by application engineering can also be combined with framework-provided extensions.

4.4.1. Life Cycle Selection

A component life cycle is defined using a life cycle expression that specifies the basic life cycle and the list of life cycle extensions to be applied. Since basic life cycles and extensions are defined as structs or classes with template arguments, the life cycle expression defines the order in which the compiler should aggregate the provided parts into the resulting life cycle. A *type alias* is used to represent the resulting life cycle and to avoid repetition of the life cycle expression in the code.

Invalid life cycle expressions can be detected during compilation. In this case the static assertions provided by framework developers fail and meaningful error messages are shown to the application developer.

```
//    life cycle = life cycle expression
using LifeCycle1 = EltComponent;
using LifeCycle2 = RtcComponent;
using LifeCycle3 = Runnable<EltComponent>; // compiler error
using LifeCycle4 = Runnable<RtcComponent>;
using LifeCycle5 = Loopaware<RtcComponent, 3>;
using LifeCycle6 = Loopaware<RtcComponent, 11>; // compiler error
```

Listing 35: Life cycle selection.

Listing 35 shows examples for the definition of different life cycles. Note that the life cycle expressions make use of basic life cycles and extensions introduced in previous sections. Also note that compilation would fail since `LifeCycle3` and `LifeCycle6` violate the constraints defined in life cycle extensions `Runnable` and `Loopaware`.

4.4.2. Custom Business Logic

After defining the component life cycle, the business logic class is implemented. The life cycle dependent business logic interface defines which methods may and which methods must be implemented by the component developer.

```
struct MyBusinessLogic : LifeCycle4::IBizLogic {

    MyBusinessLogic(Services& services) { /* custom implementation */ }
    void ActivityInitialising() override { /* custom implementation */ }
    void ActivityEnabling() override { /* custom implementation */ }
    void ActivityRunning() override { /* custom implementation */ }
    void ActivityUpdating() override { /* custom implementation */ }
};
```

Listing 36: Custom business logic.

Listing 36 shows the outline of a business logic class. Note that not all methods defined in the business logic interface must be overwritten. Framework developers may provide default implementations for certain methods, in this case overriding is only optional.

4.4.3. Component Invocation

After implementation of the custom business logic class, the component can be brought up using the generic component runner method.

```
int main() {
    return RunComponent<LifeCycle4, MyBusinessLogic>();
}
```

Listing 37: Component invocation.

Listing 37 shows how the component runner method is invoked with the selected life cycle and the custom business logic class.

4.5. Conformance Testing

This section describes how conformance testing was realised. The section is split into two major parts. The first part focuses on conformance test definition and the second part explains how conformance tests are selected and invoked.

The goal of the conformance tests is to validate that invocation consistency and observation consistency is preserved even if a life cycle is customised or extended. Invocation consistency is validated by sending specific activation sequences and by checking the returned reply values. Observation consistency is validated by applying specific activation sequences and then observing the sequence of visited states. Additionally, the same tests are also used to check if delegation to business logic methods works as expected.

Conformance tests are implemented using the *GoogleTest* framework from [Goo]. Using a unit testing framework for conformance testing facilitates fast test execution and it also allows to test the component life cycle independently from a specific business logic. GoogleTest facilitates this by its mocking capabilities. Specifically, *type-parametrised tests* are used to check multiple life cycle expressions against a single conformance test asset. This allows to reuse the same test logic for a list of life cycles.

4.5.1. Reusable Test Assets

Reusable conformance test assets are provided as part of the framework in a dedicated *unit test support library* that can be used by both domain and application engineering. This section describes how the test assets for basic life cycle `EltComponent` are implemented.

Mock Class Definition

Mock class `MockBizLogic` is introduced to test that the hook methods defined in the business logic interface are invoked correctly. The class only overrides business logic interface methods that are relevant for life cycle `EltComponent`. Later, in the individual test cases, concrete expectations are defined that validate correct hook method invocation.

```
template <class LC>
class MockBizLogic : public LC::BizLogicIf { public:
    MOCK_METHOD(void, ActivityStarting, (), (override));
    MOCK_METHOD(void, ActivityInitialising, (), (override));
    MOCK_METHOD(void, ActivityEnabling, (), (override));
    MOCK_METHOD(void, ActivityDisabling, (), (override));
};
```

Listing 38: Partial mock for business logic.

Listing 38 shows mock class `MockBizLogic` that overrides hook methods `ActivityStarting`, `ActivityInitialising`, `ActivityEnabling` and `ActivityDisabling`. Note that the class makes use of parametrised inheritance and that it only defines a partial mock by overriding four specific life cycle methods.

Additionally, mock class `MockStateChangedCallback` is defined to validate observation consistency. Using this class, expectations can be defined for the sequence of visited states.

```
class MockStateChangedCallback {
public:
    MOCK_METHOD(void, OnStateChanged, (std::string const&));
};
```

Listing 39: Mock for state changed callback.

Listing 39 shows the definition of mock class `MockStateChangedCallback` with mock method `OnStateChanged`. Note that the callback method reports the active state as a string.

Test Fixture Definition

Next, a *test fixture class* is introduced so that individual tests can share common objects and subroutines. The fixture class also performs setup and teardown work so that the test cases can focus on the test logic.

```
template <class LC>
class TestEltComponentCompliance : public ::testing::Test {
protected:
    TestEltComponentCompliance()
        : bl(), engine(), output_stage(engine, bl), model_builder()
    {
        auto model = model_builder.MakeModel();
        engine.LoadModel(ExportModel(*model, "scxml"));
        engine.RegisterStateChangedCB([&](auto state) {
                cb.OnStateChanged(state);
            });
    }

    virtual ~TestEltComponentCompliance() {
        engine.Stop();
        t.join();
    }

    void StartEngine() {
        t = std::thread([&] { engine.Work(); });
```

```
    }

    MockStateChangedCallback cb;
    MockBizLogic<LC> bl;
    Engine engine;
    typename LC::OutputStage output_stage;
    typename LC::ModelBuilder model_builder;
    std::thread t;
};
```

Listing 40: Template fixture class.

Listing 40 shows the implementation of the test fixture class. By introducing the template parameter `LC`, the same fixture class can be used for different life cycles. In the constructor, the fixture sets up the component in a similar manner as the generic component runner (c.f. Listing 16). Once the state machine is ready to be started, control is handed over to individual test cases. After test execution, the component is shut down gracefully using the tear down code in the destructor.

Test Suite Definition

Next, a type-parametrised test suite is defined using macro `TYPED_TEST_SUITE_P`.

```
TYPED_TEST_SUITE_P(TestEltComponentCompliance);
```

Listing 41: Test suite definition.

Test Case Definition

After the test suite has been defined, individual test cases can be implemented. First, expectations for the sequence of visited states and life cycle methods to be called are defined. Then, the state machine engine is started. Finally, a specific activation sequence is applied and the received replies are checked.

```
TYPED_TEST_P(TestEltComponentCompliance, InitEnableDisableResetExit) {

   // set expectations
   {
      InSequence seq;
      EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Starting")));
      EXPECT_CALL(this->bl, ActivityStarting());
      EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".NotReady")));
      EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Initialising")));
      EXPECT_CALL(this->bl, ActivityInitialising());
```

```
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Ready")));
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Enabling")));
        EXPECT_CALL(this->bl, ActivityEnabling());
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Operational"))).
            Times(AtLeast(1));
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Disabling")));
        EXPECT_CALL(this->bl, ActivityDisabling());
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Ready")));
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".Starting")));
        EXPECT_CALL(this->bl, ActivityStarting());
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr(".NotReady")));
        EXPECT_CALL(this->cb, OnStateChanged(HasSubstr("Off")));
    }

    // start engine
    this->StartEngine();
    WaitUntilInState(this->engine, "On.NotOperational.NotReady");

    // apply test sequence
    ASSERT_EQ("OK", AwaitReply(Inject<Events::Init>(this->engine)));
    ASSERT_EQ("OK", AwaitReply(Inject<Events::Enable>(this->engine)));
    ASSERT_EQ("OK", AwaitReply(Inject<Events::Disable>(this->engine)));
    ASSERT_EQ("OK", AwaitReply(Inject<Events::Reset>(this->engine)));
    ASSERT_EQ("OK", AwaitReply(Inject<Events::Exit>(this->engine)));
}
```

Listing 42: Typed test case.

Listing 42 shows the implementation of a concrete test case for activation sequence: `Init`, `Enable`, `Disable`, `Reset`, `Exit`. The test case is executed for all provided life cycles. Invocation consistency is validated by checking if the sequence of commands is accepted. Observation consistency is checked in a similar fashion. For all provided life cycles the sequence of visited states must match the sequence specified in the test. If mock method `OnStateChanged` reports other states that are not mentioned in the specified sequence the test still succeeds as long as the specified sequence of states is still fully visited.

Test Case Registration

Finally, test cases are registered to the test suite using macro `REGISTER_TYPED_TEST_SUITE_P`.

```
REGISTER_TYPED_TEST_SUITE_P(TestEltComponentCompliance,  // suite name
                            Init, Exit, GetState         // test case
                            InitEnableDisableResetExit); // test case
```

Listing 43: Test case registration.

4.5.2. Test Selection and Invocation

The framework-provided conformance test assets can be used by both domain and application engineering. Domain engineers use the test assets to validate that the behaviour of framework-provided life cycles is correct. Application engineers use the test assets to validate that special life cycle extensions introduced during application engineering do not break compatibility with basic component behaviour.

```
// test suite headers
#include "testEltComponentCompliance.hpp"

// life cycle headers
#include "eltComponent.hpp"
#include "rtcComponent.hpp"
#include "runnable.hpp"
#include "loopaware.hpp"

// list of life cycles to be tested
using TypesToTest = testing::Types<
                        EltComponent,
                        RtcComponent,
                        Runnable<RtcComponent>,
                        Loopaware<RtcComponent>,
                        Loopaware<RtcComponent, 3>>;

// test suite invocation
INSTANTIATE_TYPED_TEST_SUITE_P(EltComponentSuite,
                               TestEltComponentCompliance,
                               TypesToTest);
```

Listing 44: Conformance test invocation.

Listing 44 shows how a conformance test suite can be invoked. First, relevant headers are included for the test suite and the life cycles. Then, the list of life cycles to be checked is specified by defining type alias `TypesToTest`. Finally, the type parametrised test suite is invoked with macro `INSTANTIATE_TYPED_TEST_SUITE_P`. The name of the test suite and the list of types to test are passed as arguments. Note that it is also possible to instantiate multiple test suites for the same set of life cycles.

5. Conclusion

This chapter summarises and discusses the behavioural customisation approach for state machine models developed in this thesis. Additionally, the chapter also presents an outlook on future work.

5.1. Results

In this thesis, a new approach was developed that combines a set of variability realisation mechanisms and thereby enables open and stepwise customisation of component life cycles, which are characterised by state machine models. The developed approach facilitates systematic reuse of common assets and separation of concerns in a product line setup. In addition, this thesis demonstrates that compatibility can be ensured by combining constructive and analytical methods, namely feature orientation and conformance testing. In particular, a proof of concept solution was implemented to demonstrate the feasibility and soundness of the concept formulated in this thesis.

With regard to the research goals from Chapter 1 the following activities were carried out:

- **G1**: Practical and theoretical understanding on software product lines, customisation mechanisms, behavioural modeling with statecharts, behavioural consistency and compatibility as well as relevant technologies and tools was obtained and summarised in this thesis by studying relevant literature (see Chapter 2).
- **G2**: The state of practice at ESO was investigated. Concrete use cases, customisation scenarios, customisation operations and requirements were elicited and documented (see Sections 3.1, 3.2 and 3.3).
- **G3**: Different customisation mechanisms were evaluated and prototypes were created to gain hands-on experience and to test their applicability (see Section 3.4).
- **G4**: A solution concept was elaborated that covers the elicited requirements. This concept combines different customisation mechanisms and applies different techniques for ensuring compatibility (see Section 3.5).
- **G5**: Feasibility and soundness of the solution concept was demonstrated by implementing a proof of concept solution. Additionally, the proof of concept has already been integrated into the ELT RTC Toolkit where it is successfully applied in practice (see Section 4).

The research questions introduced in Section 1 could be answered as follows:

- **Q1**: It was demonstrated that component life cycles and state machine models can be customised using a combination of state-of-the-art customisation mechanisms. A flexible solution concept was elaborated and implemented. The concept enhances a *framework approach* with *model manipulation* capabilities and *mixin composition*. Additionally, *conditional compilation* and *conditional execution* were added (see Section 3.5).
- **Q2**: It was shown that customisation mechanisms need to be applied to the textual or even to the graph-based in memory representation of the state machine model to provide sufficient flexibility and to support the traditional software engineering workflow applied at ESO (see Sections 3.4 and 3.5).
- **Q3**: It was demonstrated that a framework approach can be used and enhanced to provide the required flexibility. Additionally, it was shown that the problem cannot be solved by simply using existing methods, tools and techniques separately. It was also shown that graphical modeling tools such as *MagicDraw* and *Rhapsody* have major limitations for modeling composition-based and behavioural variability (see Sections 3.4 and 3.5).
- **Q4**: The concept of behavioural consistency was investigated and conformance testing was identified as the most suitable approach for ensuring compatibility. Additional constructive techniques and tactics were introduced that help developers maintain compatibility and cope with configuration complexity (see Section 3.5).

5.2. Discussion

The presented solution concept was tailored to solve a specific problem in scope of the ELT program where domain engineering teams only have incomplete knowledge about the final system and where they also only have limited control over what application engineering teams will do. To cope with this uncertainty, an approach was chosen that supports open variation but at the same time allows constraining variability by using different constructive and analytical techniques.

The elaborated approach is expected to work well for projects that are facing similar challenges and that are organised in a similar fashion (i.e. a platform or product line setup with highly decoupled domain and application engineering teams).

This approach is not recommended for projects that are organised in a more tightly coupled manner and where more knowledge regarding variability is available upfront. For

such projects, a production line or a configurable product approach that is based on closed variability is more appropriate.

5.3. Future Work

Evolution and further improvement of the developed approach is ongoing work in the context of the ELT RTC Toolkit project. In addition, further ESO project teams expressed interest in using the developed techniques in other ELT software platforms and frameworks. It is planned to evaluate the applicability of the approach for these projects.

Still open issues that need to be addressed in the future are:

- **Model Validation.** It would be beneficial to validate the generated state machine model to be able to detect bad or forbidden model manipulation operations.
- **Conformance Testing.** The definition, selection and invocation of conformance tests should be revisited to make tests more efficient but also simpler to write.
- **Parametrised Life Cycles.** The concept of parametrised life cycles should be revisited. In particular, the trade-off between additional complexity and benefit should be examined.
- **Feature Interactions.** There are still open questions regarding feature interactions, e.g. how can fine-grained feature dependencies across multiple life cycle extensions be realised?
- **MDA Workflow.** Currently, the approach focuses on the traditional software development workflow. Integration into COMODO should be considered to make the approach applicable in MagicDraw and with the MDA workflow.

References

[AB02] W. V. Aalst and T. Basten. “Inheritance of Dynamic Behavior in UML”. In: 2002.

[And+11] L. Andolfato et al. “A Platform Independent Framework for Statecharts Code Generation”. In: (2011).

[Ape] S. Apel. *FeatureHouse: Language-Independent, Automated Software Composition.* URL: `https://www.se.cs.uni-saarland.de/apel/fh/` (visited on 08/09/2021).

[Ape+09] S. Apel et al. “Model Superimposition in Software Product Lines”. In: *Theory and Practice of Model Transformations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 4–19. ISBN: 978-3-642-02408-5.

[Ape+13] S. Apel et al. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-37520-0.

[Bas87] P. Bassett. “Frame-Based Software Engineering”. In: *IEEE Software* 4 (1987), pp. 9–16.

[Bec17] M. Becker. *Software Product Line Engineering.* 2017.

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.

[Cun09] H.G. Cunningham. *Integration Hell.* 2009. URL: `http://wiki.c2.com/?IntegrationHell` (visited on 11/29/2021).

[EE94] J. Ebert and G. Engels. “Observable or Invocable Behaviour - You Have to Choose”. In: 1994.

[ESO11] ESO. *The E-ELT Construction Proposal.* The E-ELT Project Office, 2011.

[ESO20a] ESO. *ELT - Technical Note on Standard Application Interface and State Machine.* 2020. URL: `https://pdm.eso.org/kronodoc/HQ/ESO-321402/3` (visited on 08/14/2021).

[ESO20b] ESO. *ELT4k-12-Night4-cc.* 2020. URL: `https://www.eso.org/public/images/ELT4k-12-Night4-cc/` (visited on 09/22/2021).

[ESO20c] ESO. *scxml4cpp: SCXML intepreter for C++.* 2020. URL: `https://github.com/Open-MBEE/Comodo/tree/support/1.x/Engines/scxml4cpp` (visited on 12/30/2021).

[ESO21a] ESO. *Application Framework Documentation.* 2021. URL: `https://www.eso.org/~eltmgr/ICS/documents/RAD/` (visited on 12/30/2021).

[ESO21b] ESO. *RTC Toolkit Documentation*. 2021. URL: `https://www.eso.org/~eltmgr/RTCTK/documents/latest/` (visited on 12/30/2021).

[FM00] J. M. Fernandes and R. J. Machado. *Object-Oriented Inheritance of Statecharts for Control Applications*. 2000.

[Fow17] M. Fowler. *Feature Flags*. 2017. URL: `https://martinfowler.com/articles/feature-toggles.html` (visited on 08/20/2021).

[Gam+95] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[Goo] Google. *GoogleTest User's Guide*. URL: `http://google.github.io/googletest` (visited on 11/14/2021).

[Har87] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Sci. Comput. Program.* 8 (1987), pp. 231–274.

[HK01] R. Heckel and J. M. Küster. "Behavioral Constraints for Visual Models". In: *Electronic Notes in Theoretical Computer Science* 50 (2001), pp. 257–265.

[HSL15] C. Hansen, E. Syriani, and L. Lucio. "Towards Controlling Refinements of Statecharts". In: *ArXiv* abs/1503.07266 (2015).

[IBMa] IBM. *Differences between Rational Rhapsody 8.0 Statecharts and UML 2.4.1 Behavior State Machine*. URL: `https://www.ibm.com/support/pages/differences-between-rational-rhapsody-80-statecharts-and-uml-241-behavior-state-machine` (visited on 01/17/2022).

[IBMb] IBM. *Statechart Inheritance*. URL: `https://www.ibm.com/docs/en/rhapsody/8.2?topic=statecharts-statechart-inheritance` (visited on 01/17/2022).

[IEE09] IEEE. *IEEE 1016-2009. IEEE Standard for Information Technology–Systems Design–Software Design Descriptions*. Standard. Institute of Electrical and Electronics Engineers, 2009.

[ISO11] ISO/IEC. *ISO/IEC 25010:2011. Software and systems engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Standard. International Organization for Standardization, 2011.

[ISO15] ISO/IEC. *ISO/IEC 26550:2015. Software and systems engineering - Reference model for product line engineering and management*. Standard. International Organization for Standardization, 2015.

[JF88] R. Johnson and B. Foote. “Designing Reusable Classes”. In: *Journal of Object-Oriented Programming* (1988).

[LKL02] K. Lee, K. Kang, and J. Lee. “Concepts and Guidelines of Feature Modeling for Product Line Software Engineering”. In: Apr. 2002, pp. 62–77. ISBN: 978-3-540-43483-2. DOI: 10.1007/3-540-46020-9_5.

[Luk03] K. Lukka. “The Constructive Research Approach”. In: 2003, pp. 83–101.

[LW94] B. Liskov and J. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Trans. Program. Lang. Syst.* (1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383. URL: https://doi.org/10.1145/197320.197383.

[LY96] D. Lee and M. Yannakakis. “Principles and methods of testing finite state machines-a survey”. In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123. DOI: 10.1109/5.533956.

[MP02] D. Muthig and T. Patzke. “Generic Implementation of Product Line Components”. In: vol. 2591. 2002. ISBN: 978-3-540-00737-1. DOI: 10.1007/3-540-36557-5_23.

[OMG03] OMG. *Unified Modeling Language Version 1.5.* 2003. URL: https://www.omg.org/spec/UML/1.5/PDF (visited on 02/20/2021).

[OMG15] OMG. *Unified Modeling Language Version 2.5.* 2015. URL: https://www.omg.org/spec/UML/2.5/PDF (visited on 02/20/2021).

[Pat] T. Patzke. *Frame Processor.* URL: https://sourceforge.net/projects/frameprocessor/ (visited on 07/18/2021).

[Pat11] T. Patzke. “Sustainable Evolution of Product Line Infrastructure Code”. PhD thesis. Kaiserslautern, Germany: TU Kaiserslautern, 2011.

[PP11] J. Poore and S. Prowell. “Textbook: Software Engineering for Embedded Systems”. In: (2011).

[Pre04] C. Prehofer. “Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams”. In: *Softw. Syst. Model.* (2004), pp. 221–234.

[Pre13] C. Prehofer. “Behavioral Refinement and Compatibility of Statechart Extensions”. In: *Electronic Notes in Theoretical Computer Science* 295 (2013), pp. 65–78.

[Roh20] F. Rohlf. “Feasibility Study on Variability Realization Methods in SysML Models”. MA thesis. Kaiserslautern, Germany: TU Kaiserslautern, 2020.

[Sch+10] I. Schaefer et al. “Delta-Oriented Programming of Software Product Lines”. In: 2010, pp. 77–91. ISBN: 978-3-642-15578-9. DOI: 10.1007/978-3-642-15579-6_6.

[Sei08] D. Seifert. “Conformance Testing Based on UML State Machines”. In: vol. 5256. Oct. 2008, pp. 45–65. DOI: 10.1007/978-3-540-88194-0_6.

[Sim+02] A. J. Simons et al. “Plug and Play Safely: Rules for Behavioural Compatibility”. In: 2002.

[SPM19] M. Skelton, M. Pais, and R. Malan. *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution, 2019. ISBN: 9781942788812.

[SS04] M. Stumptner and M. Schrefl. “Behavior Consistent Inheritance with UML Statecharts”. In: (2004).

[TK14] M. Trapp and T. Kuhn. *Model-based Component Engineering*. 2014.

[TKS03] V. Trenkaev, M. Kim, and S. Seol. “Interoperability Testing Based on a Fault Model for a System of Communicating FSMs”. In: vol. 2644. May 2003, pp. 226–242. ISBN: 978-3-540-40123-0. DOI: 10.1007/3-540-44830-6_17.

[Tru+10] S. Trujillo et al. “Coping with Variability in Model-Based Systems Engineering: An Experience in Green Energy”. In: June 2010, pp. 293–304. ISBN: 978-3-642-13594-1. DOI: 10.1007/978-3-642-13595-8_23.

[TSG08] P. Tessier, D. Servat, and S. Gérard. “Variability Management on Behavioral Models”. In: Jan. 2008, pp. 121–130.

[W3C15] W3C. *State Chart XML (SCXML) State Machine Notation for Control Abstraction*. 2015. URL: https://www.w3.org/TR/scxml/ (visited on 02/20/2021).

[WHH06] C. Wohlin, M. Höst, and K. Henningsson. “Empirical Research Methods in Web and Software Engineering”. In: 2006, pp. 409–430. ISBN: 3-540-28196-7. DOI: 10.1007/3-540-28218-1_13.

[ZDB16] B. Zhang, S. Duszynski, and M. Becker. “Variability Mechanisms and Lessons Learned in Practice”. In: *2016 IEEE/ACM 1st International Workshop on Variability and Complexity in Software Design (VACE)*. 2016, pp. 14–20. DOI: 10.1109/VACE.2016.012.

Appendices

A. Statechart Inheritance in Rhapsody

A concrete prototype that makes use of stepwise customisation was implemented in Rhapsody to show how state machine inheritance can be used and to find possible limitations.

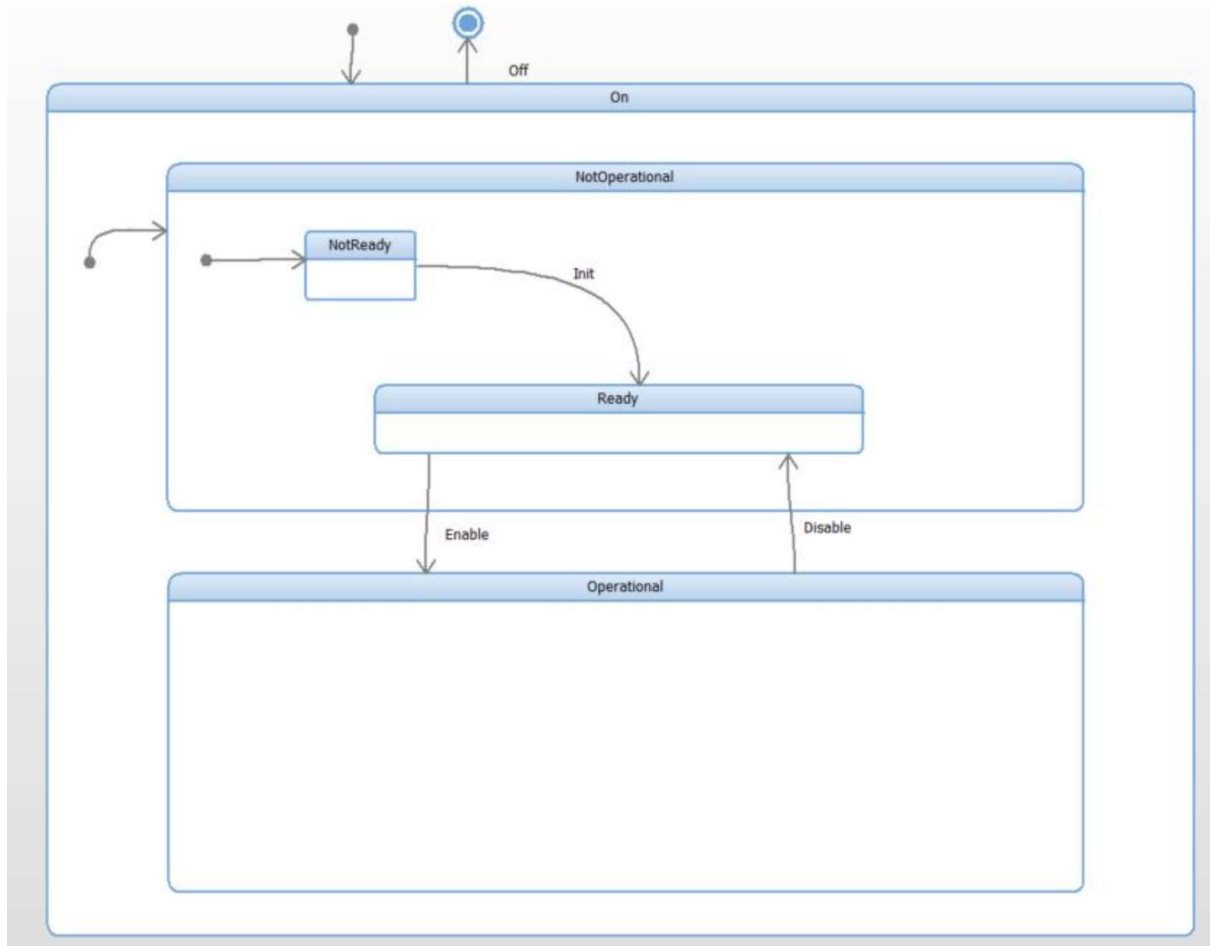


Figure 51: Model inheritance: base model.

Figure 51 shows an example state machine model that acts as a base for stepwise customisation using inheritance semantics. Note that in Rhapsody states of the current specialisation are depicted in blue color whereas inherited states and transitions are greyed out.

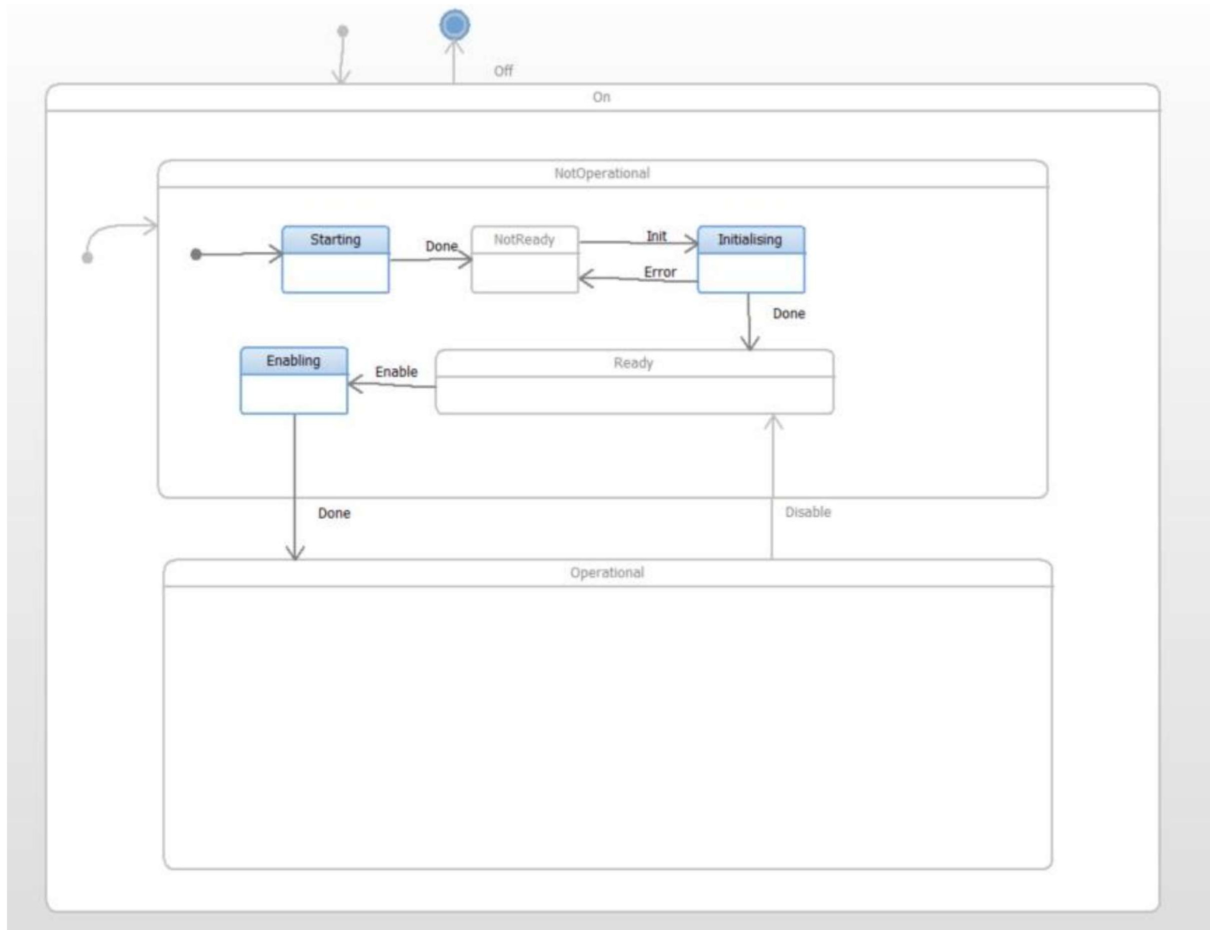


Figure 52: Model inheritance: first specialisation.

Figure 52 shows the first specialisation of the basic model. The specialisation refines three transitions into corresponding transient states using Operation 5. In addition a new transition with event *Error* has been introduced according to Operation 1.

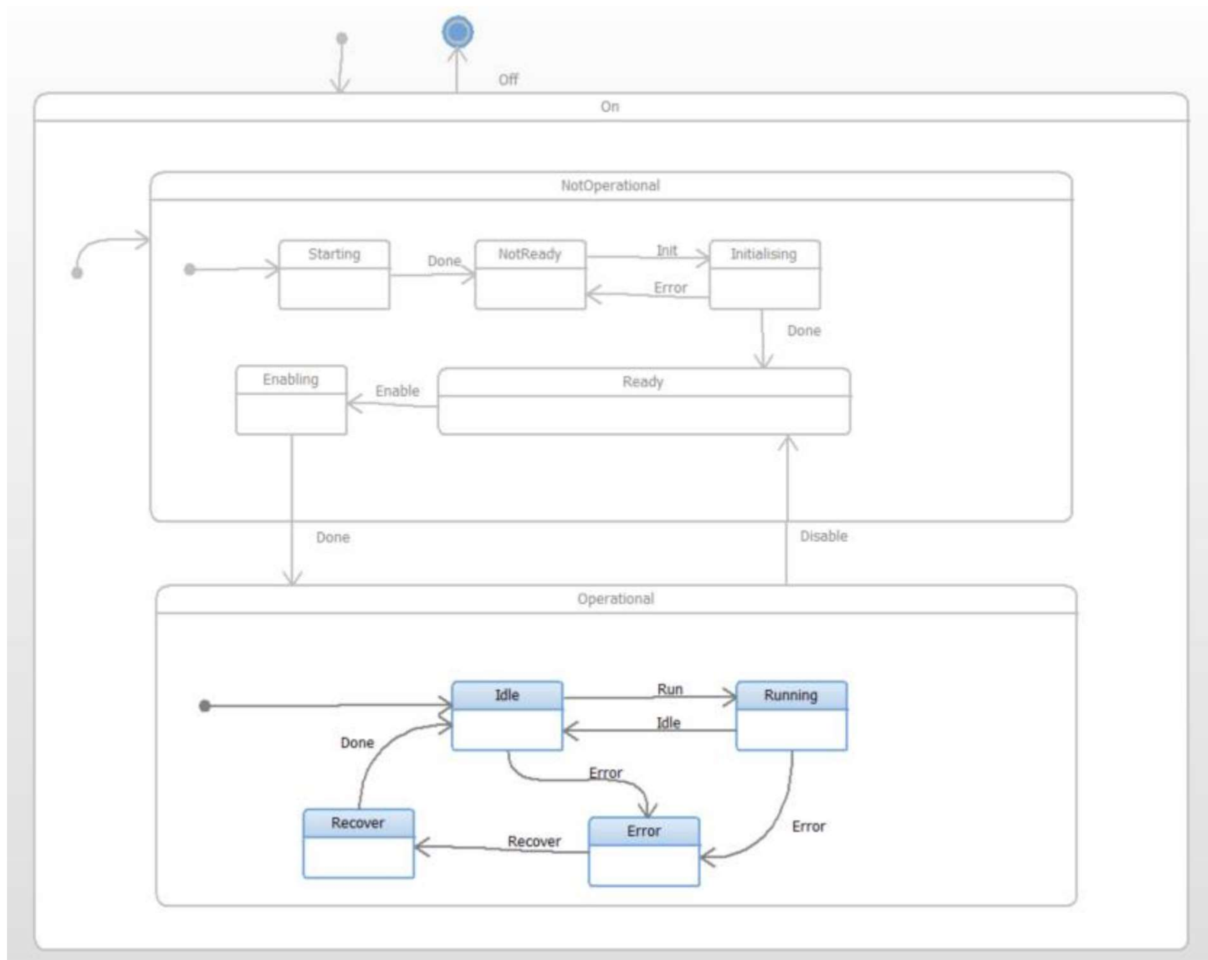


Figure 53: Model inheritance: second specialisation.

The next specialisation of the state machine in Figure 53 shows the refinement of state *Operational* into a composite state according to Operation 7.

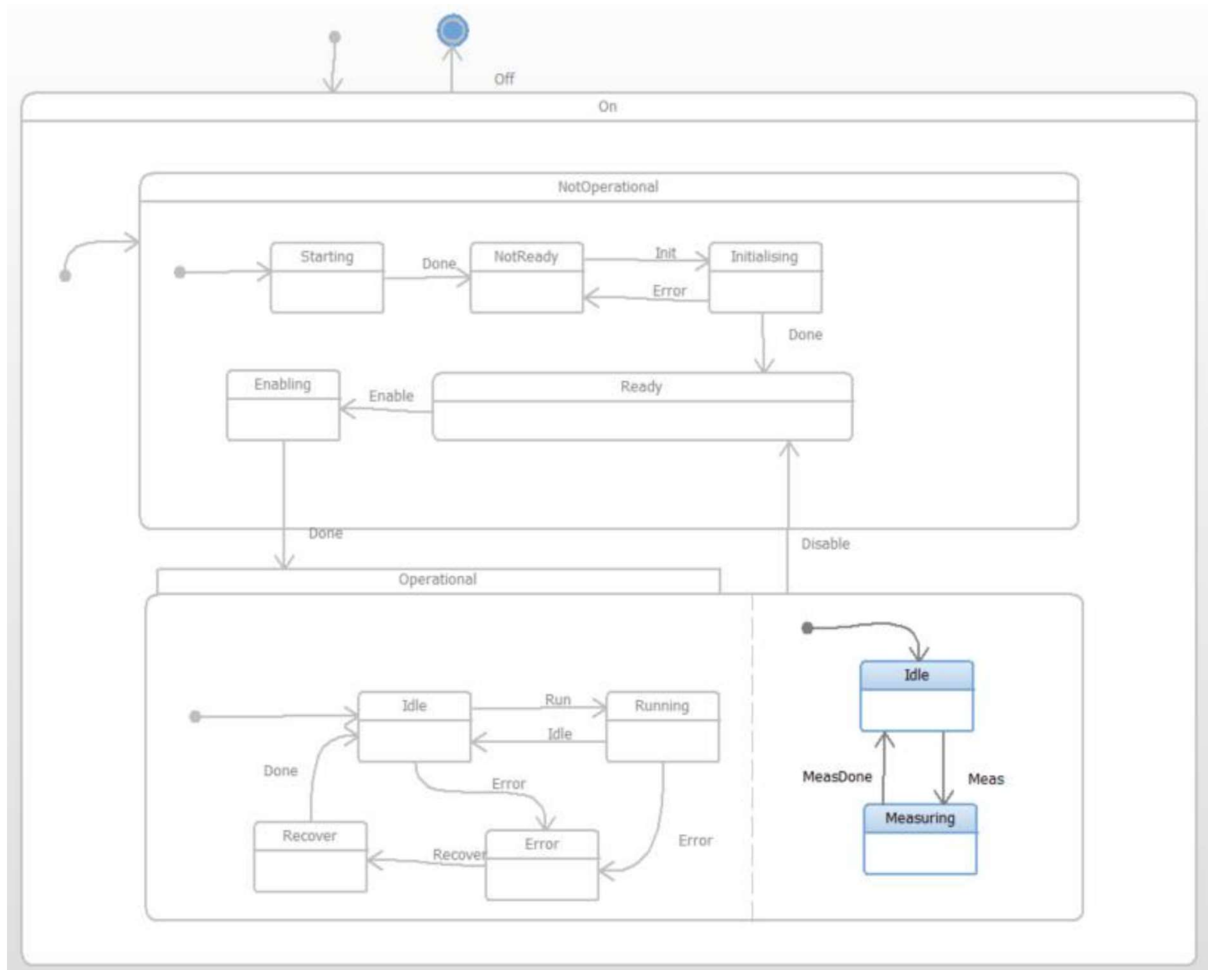


Figure 54: Model inheritance: third specialisation.

The last specialisation step depicted in Figure 54 applies parallel extension (see Operations 9 and 3) to add new behaviour that is running in a dedicated parallel region of state *Operational*.

B. State Machine Model Data Structure

```
enum StateType {
    Machine,
    Simple,
    Parallel,
    Composite,
    Initial,
    Final
};

struct State {
    string id;
    StateType type;
    State* parent;
    list<State*> children;
    string do_activity_id;
    string entry_action_id;
    string exit_action_id;
};

struct Transition {
    State* source;
    State* target;
    string event_id;
    string guard_id;
    string action_id;
};

struct StateMachine : State { // type = Machine;
    list<State*> m_all_states;
    list<Transition*> m_all_transitions;
};
```

Listing 45: State machine model data structure.

Listing 45 shows the data structure that describes the state machine model. A `StateMachine` is a special kind of state that is composed of a list of states and transitions that are organised in a hierarchy. States can have different types and transitions connect a source state with a target state.

C. State Machine Model Definitions

```
struct ModelBuilder : IModelBuilder {

    std::unique_ptr<Model> MakeModel() override {
        auto model = std::make_unique<Model>("sm");

        // add states
        AddState(*model, Initial,   "Initial");
        AddState(*model, Composite, "On");
        AddState(*model, Final,     "Off");
        AddState(*model, Initial,   "On.Initial",
                                    "On");
        AddState(*model, Composite, "On.NotOperational",
                                    "On");
        AddState(*model, Simple,    "On.Operational",
                                    "On");
        AddState(*model, Initial,   "On.NotOperational.Initial",
                                    "On.NotOperational");
        AddState(*model, Simple,    "On.NotOperational.Starting",
                                    "On.NotOperational",
                                    "ActivityStarting",
                                    "ActionStartingEntry");
        AddState(*model, Simple,    "On.NotOperational.NotReady",
                                    "On.NotOperational");
        AddState(*model, Simple,    "On.NotOperational.Initialising",
                                    "On.NotOperational",
                                    "ActivityInitialising",
                                    "ActionInitialisingEntry");
        AddState(*model, Simple,    "On.NotOperational.Ready",
                                    "On.NotOperational");
        AddState(*model, Simple,    "On.NotOperational.Enabling",
                                    "On.NotOperational",
                                    "ActivityEnabling",
                                    "ActionEnablingEntry");
        AddState(*model, Simple,    "On.NotOperational.Disabling",
                                    "On.NotOperational",
                                    "ActivityDisabling",
                                    "ActionDisablingEntry");
        // add transitions
        AddTrans(*model, "Initial",
                         "On");
        AddTrans(*model, "On",
                         "Off",
```

```
                    "events.Exit",
                    "",
                    "ActionExit");
    AddTrans(*model, "On",
                    "",
                    "events.GetState",
                    "",
                    "ActionGetState");
    AddTrans(*model, "On",
                    "On",
                    "events.Reset");
    AddTrans(*model, "On.Initial",
                    "On.NotOperational");
    AddTrans(*model, "On.NotOperational.Initial",
                    "On.NotOperational.Starting");
    AddTrans(*model, "On.NotOperational.Starting",
                    "On.NotOperational.NotReady",
                    "events.Done",
                    "",
                    "ActionStartingDone");
    AddTrans(*model, "On.NotOperational.NotReady",
                    "On.NotOperational.Initialising",
                    "events.Init");
    AddTrans(*model, "On.NotOperational.Initialising",
                    "On.NotOperational.Ready",
                    "events.Done",
                    "",
                    "ActionInitialisingDone");
    AddTrans(*model, "On.NotOperational.Initialising",
                    "On.NotOperational.NotReady",
                    "events.Error",
                    "",
                    "ActionInitialisingFailed");
    AddTrans(*model, "On.NotOperational.Initialising",
                    "On.NotOperational.NotReady",
                    "events.Stop",
                    "",
                    "ActionInitialisingStopped");
    AddTrans(*model, "On.NotOperational.Initialising",
                    "On.NotOperational.Initialising",
                    "events.Init",
                    "",
                    "ActionInitialisingRestarted");
    AddTrans(*model, "On.NotOperational.Ready",
                    "On.NotOperational.Initialising",
```

```
                            "events.Init");
        AddTrans(*model, "On.NotOperational.Ready",
                            "On.NotOperational.Enabling",
                            "events.Enable");
        AddTrans(*model, "On.NotOperational.Enabling",
                            "On.Operational",
                            "events.Done",
                            "",
                            "ActionEnablingDone");
        AddTrans(*model, "On.NotOperational.Enabling",
                            "On.NotOperational.Ready",
                            "events.Error",
                            "",
                            "ActionEnablingFailed");
        AddTrans(*model, "On.Operational",
                            "On.NotOperational.Disabling",
                            "events.Disable");
        AddTrans(*model, "On.NotOperational.Disabling",
                            "On.NotOperational.Ready",
                            "events.Done",
                            "",
                            "ActionDisablingDone");
        AddTrans(*model, "On.NotOperational.Disabling",
                            "On.NotOperational.Ready",
                            "events.Error",
                            "",
                            "ActionDisablingFailed");

        return model;
    }
};
```

Listing 46: Basic life cycle without omissions.