DISCOVERY OF PATTERNS
IN SIMULINK SYSTEMS

by

FRANCISCO DE LA PARRA

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
July 2017

Copyright © Francisco de la Parra, 2017
Abstract

Model-Driven Engineering tools and techniques have become mainstream aids in the development of embedded cyber-physical systems. The large-scale use of visual models is a powerful paradigm for designing reliable and maintainable systems assembled with complex hardware, software and mechanical elements. Modular design approaches can readily capture the requirements for embedded systems into executable and evolutionary models.

We propose a framework to discover, classify and visualize model-patterns in large repositories of Simulink models. We design, and implement in a software toolchain, a set of scalable algorithms to fully traverse and contextually parse the source code of these repositories in order to compute topology hash functions for the size, name connectivity and type connectivity of each model subsystem, as well as properties of Stateflow charts. We use these extracted properties to cluster the subsystems into a set of addressable-by-property classes. We develop an interactive model visualization and querying facility, embodied by the MoSART user interface, to identify further relations, similarities and abstractions in the elements of the cluster classes. This tool provides a configurable environment for inferring structural model-patterns and elaborating on semantic model-patterns with the assistance of skilled domain knowledge. We propose a query system, based on regular expressions that match values of previously computed subsystems properties, as an effective method to iteratively refine Simulink model-patterns abstractions into accumulated modeling knowledge.
Our implementation works cooperatively with a 3-phase iterative model-pattern discovery approach that we conceptualize as 1) model clustering: repository traversal and subsystem-topology evaluation for clustering subsystems, 2) primary model-pattern inference: visualization, query-based search and clustering for identifying basic semantic patterns, 3) model-pattern refinement: continuous refinement of basic patterns into specialized groups for application in targeted use cases. We effectively support the iterative nature of this approach because our repository traversal, property evaluation and clustering algorithms have low time-complexity, that is, between \( \Omega(n) \) and \( O(n \log n) \) (\( n = \) total lines of source code in a repository), and our interactive tool provides a complete-coverage view of a model repository.
Acknowledgements

This thesis is dedicated to my wife, Joanne

I thank you for your unwavering support and encouragement. At every step of this journey, you discovered a way to teach me how to find the energy to continue and enjoy the challenges in life.
Statement of Originality

I, Francisco de la Parra, hereby certify that the content of this thesis and the scope of the research it refers to are the result of my own original work. All references to the work of others, published or unpublished, are fully acknowledged in accordance with standard referencing practices.
# Table of Contents

Abstract i

Acknowledgements iii

Statement of Originality iv

Table of Contents v

List of Tables viii

List of Figures ix

List of Algorithms x

List of Acronyms xi

Chapter 1: Introduction 1
  1.1 Thesis Statement ................................................. 2
  1.2 Scope of the Thesis ............................................... 3
    1.2.1 Model-Pattern Discovery Approach .......................... 4
    1.2.2 Detection of Syntactic Patterns ............................ 7
    1.2.3 Inference of Semantic Model-Patterns ...................... 10
    1.2.4 Sets of Model-Patterns ........................................ 11
    1.2.5 Model-Patterns Use Cases ..................................... 12
  1.3 Thesis Contributions ............................................ 13
  1.4 Thesis Organization .............................................. 15

Chapter 2: Background .................................................. 17
  2.1 Embedded Systems ................................................ 18
    2.1.1 Overview .................................................... 18
    2.1.2 Modeling Embedded Systems ................................. 20
  2.2 Modeling Patterns and their Reuse ............................... 21
    2.2.1 Model-Patterns of Embedded Systems ....................... 22
    2.2.2 Reuse of Simulink Patterns .................................. 24
  2.3 Pattern Discovery Approaches ................................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4 Pattern Representation and Reasoning</td>
<td>28</td>
</tr>
<tr>
<td>2.5 Refinement of Model-Patterns</td>
<td>31</td>
</tr>
<tr>
<td><strong>Chapter 3: Model-Pattern Discovery Framework</strong></td>
<td>34</td>
</tr>
<tr>
<td>3.1 Assumptions</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Architecture of the Framework</td>
<td>36</td>
</tr>
<tr>
<td>3.3 Automated Model-Pattern Detection</td>
<td>38</td>
</tr>
<tr>
<td>3.3.1 Primary Clustering (PCLU) Module</td>
<td>40</td>
</tr>
<tr>
<td>3.3.2 Subsystem Topology Hashing (STH) Module</td>
<td>43</td>
</tr>
<tr>
<td>3.3.3 Subsystem Name Usage Analysis (SNUA) Module</td>
<td>45</td>
</tr>
<tr>
<td>3.3.4 Clustered-Subsystem Directories</td>
<td>47</td>
</tr>
<tr>
<td>3.4 Model-Pattern Exploration Facility (MPEF)</td>
<td>48</td>
</tr>
<tr>
<td>3.4.1 Architecture of MoSART</td>
<td>49</td>
</tr>
<tr>
<td>3.4.2 The Browse View</td>
<td>50</td>
</tr>
<tr>
<td>3.5 Ancillary Tools</td>
<td>53</td>
</tr>
<tr>
<td>3.6 Framework Evaluation</td>
<td>55</td>
</tr>
<tr>
<td>3.6.1 Adaptability to Simulink Repositories</td>
<td>55</td>
</tr>
<tr>
<td>3.6.2 Support for Empirical Analysis</td>
<td>57</td>
</tr>
<tr>
<td><strong>Chapter 4: Model-Pattern Discovery and Topology Assessment Algorithms</strong></td>
<td>59</td>
</tr>
<tr>
<td>4.1 Detection of Model Subsystem Classes</td>
<td>60</td>
</tr>
<tr>
<td>4.1.1 Repository Traversal and Subsystems Clustering</td>
<td>61</td>
</tr>
<tr>
<td>4.2 Statechart-Sequence Classes</td>
<td>66</td>
</tr>
<tr>
<td>4.3 Subsystem-Connectivity Hash Functions</td>
<td>70</td>
</tr>
<tr>
<td>4.3.1 Name-Connectivity Hash Code (NCHC) Function</td>
<td>71</td>
</tr>
<tr>
<td>4.3.2 Type-Connectivity Hash Code (TCHC) Function</td>
<td>73</td>
</tr>
<tr>
<td>4.4 Subsystem Name Usage Analysis</td>
<td>75</td>
</tr>
<tr>
<td><strong>Chapter 5: Model-Pattern Inference Processes</strong></td>
<td>78</td>
</tr>
<tr>
<td>5.1 Model-Pattern Discovery Workflow</td>
<td>79</td>
</tr>
<tr>
<td>5.2 Interactive Elicitation of Model-Patterns</td>
<td>81</td>
</tr>
<tr>
<td>5.3 Class Querying System</td>
<td>87</td>
</tr>
<tr>
<td>5.4 Entailment and Abstraction of Model-Patterns</td>
<td>89</td>
</tr>
<tr>
<td><strong>Chapter 6: Empirical Analysis of an Industrial Repository</strong></td>
<td>94</td>
</tr>
<tr>
<td>6.1 Model-Pattern Elicitation</td>
<td>95</td>
</tr>
<tr>
<td>6.1.1 Subsystem Size and Class Consolidation (CC)</td>
<td>95</td>
</tr>
<tr>
<td>6.1.2 Subsystem Frequency and Sub-Classing (SC) or Class Consolidation (CC).</td>
<td>100</td>
</tr>
<tr>
<td>6.1.3 Subsystem Name and Class Consolidation (CC).</td>
<td>105</td>
</tr>
<tr>
<td>6.2 Semantic Entailment of Model-Patterns</td>
<td>108</td>
</tr>
<tr>
<td>6.2.1 Optional Signals and Events Sources</td>
<td>110</td>
</tr>
<tr>
<td>6.2.2 Synchronous Events Generators</td>
<td>111</td>
</tr>
<tr>
<td>6.2.3 Optional Execution Enablers</td>
<td>112</td>
</tr>
<tr>
<td>6.2.4 Special Waveform Generators</td>
<td>114</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Regular Expressions on Subsystem Property Strings</td>
<td>89</td>
</tr>
<tr>
<td>6.1</td>
<td>Industrial Repository Case Study - Subsystem Size Statistics</td>
<td>96</td>
</tr>
<tr>
<td>6.2</td>
<td>Distribution of Sizes for Subsystem: <em>Operating System Tasks (For Simulation)</em></td>
<td>102</td>
</tr>
<tr>
<td>6.3</td>
<td>Types of Small <em>If Action Subsystems</em></td>
<td>103</td>
</tr>
<tr>
<td>6.4</td>
<td>Types of Signal Generators</td>
<td>103</td>
</tr>
<tr>
<td>6.5</td>
<td>Types of Event Counters</td>
<td>105</td>
</tr>
<tr>
<td>6.6</td>
<td>3B1L Subsystems - Types of Reaction</td>
<td>123</td>
</tr>
<tr>
<td>6.7</td>
<td>Virtual Powertrain Subsystems</td>
<td>124</td>
</tr>
<tr>
<td>6.8</td>
<td>Sub-classes of the 3B1L Class</td>
<td>125</td>
</tr>
<tr>
<td>6.9</td>
<td>Profiles of Simulink Repositories in Study</td>
<td>127</td>
</tr>
<tr>
<td>6.10</td>
<td>Industrial Repository Case Study. spSearch and SIMONE Clustering Results</td>
<td>132</td>
</tr>
<tr>
<td>A.1</td>
<td>Internal Representation of Simulink Models</td>
<td>157</td>
</tr>
<tr>
<td>A.2</td>
<td>Stateflow Tags - Detected in Test Repositories</td>
<td>157</td>
</tr>
<tr>
<td>A.3</td>
<td>Simulink Tags - Detected in Test Repositories</td>
<td>158</td>
</tr>
<tr>
<td>B.1</td>
<td>Content Description of File <em>srid_pattern_Simulink_00.txt</em></td>
<td>160</td>
</tr>
<tr>
<td>B.2</td>
<td>Content Description of File <em>srid_pattern_Simulink_01.txt</em></td>
<td>161</td>
</tr>
<tr>
<td>B.3</td>
<td>Content Description of File <em>srid_pattern_Simulink_00.csv</em></td>
<td>161</td>
</tr>
<tr>
<td>B.4</td>
<td>Class Queries Storage Format Description</td>
<td>162</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Model-Patterns Inference Framework and Approach ........................................ 5
2.1 Combustion Engine Tachometer and Revolutions Limiter Control .......................... 19
2.2 Models and Implementation Architectures for Embedded Systems ....................... 20
3.1 Model-Pattern Discovery Framework - Implementation Architecture .................... 37
3.2 Model Subsystem Clustering Workflow \textit{srid} ................................................. 40
3.3 Subsystem Connectivity Types ........................................................................... 44
3.4 Architecture of MoSART ................................................................................. 50
3.5 \textit{Browse} View of MoSART - Manage, Query and Display Subsystem Classes ....... 52
3.6 Display and Navigation of Selected Subsystems Triggered from the \textit{Browse} View ... 53
4.1 Architecture and Traversal of a Repository of Simulink Models ......................... 61
4.6 \textit{26C56S} Pattern - Engine State Chart ........................................................... 68
4.7 \textit{26C56S} Pattern - Process Events Chart ......................................................... 69
4.8 \textit{26C56S} Pattern - Event Scheduling Chart ...................................................... 69
4.9 \textit{26C56S} Pattern - EngineSpeed Chart ............................................................. 69
4.10 \textit{26C56S} Pattern - Basic Charts ...................................................................... 70
4.13 Recurring Subsystem Due Only to Multiple Versions in Repository .................... 76
4.14 \textit{Event Scheduler} Instantiated for Subsystem Sizes 3B1L and 4B2L ................ 77
4.15 \textit{Event Scheduler} Instantiated for Subsystem Size 45B41L .............................. 77
5.1 Simulink Pattern Discovery with MoSART’s \textit{Browse} View ................................. 79
5.2 MoSART’s Client/Server Model Display Implementation Architecture .............. 83
5.3 Interactive Model-Pattern Inference Process with MoSART ............................... 92
6.1 Enabled Signals and Events .............................................................................. 111
6.2 Synchronous Events ....................................................................................... 112
6.3 Optional Subsystem Execution ....................................................................... 113
6.4 Special Waveform Generation ....................................................................... 114
6.5 Task Scheduling ............................................................................................ 117
6.6 Event Scheduling - Macro Event Generation .................................................... 118
6.7 Signals for Testing ......................................................................................... 119
6.8 Physical Variable Sensor Ring ....................................................................... 120
List of Algorithms

4.2 Detection of Size-Based Classes of Subsystems ................. 62
4.3 Repository Traversal and Subsystem Clustering ................. 63
4.4 Processing of Subsystems at Specified Depth ................. 64
4.5 Detection of Stateflow Chart Size Patterns ................. 67
4.11 Computation of a Name Connectivity Hash Code .......... 72
4.12 Computation of a Type Connectivity Hash Code .......... 74
# List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>AVS</td>
<td>Advanced Vehicle Simulator</td>
</tr>
<tr>
<td>BD</td>
<td>Block-Dataflow</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>CC</td>
<td>Class Consolidation</td>
</tr>
<tr>
<td>CICo</td>
<td>Class Converter</td>
</tr>
<tr>
<td>MPEF</td>
<td>Model Pattern Exploration Facility</td>
</tr>
<tr>
<td>CHSM</td>
<td>Cluster and Hash Subsystem Models</td>
</tr>
<tr>
<td>CSP</td>
<td>Chart Sequence Patterns</td>
</tr>
<tr>
<td>DCD</td>
<td>Display-Classes Directory</td>
</tr>
<tr>
<td>DSML</td>
<td>Domain Specific Modeling Language</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HiL</td>
<td>Hardware-in-the-Loop</td>
</tr>
<tr>
<td>HEVPS</td>
<td>Hybrid Electric Vehicle Propulsion System</td>
</tr>
<tr>
<td>MiL</td>
<td>Model-in-the-Loop</td>
</tr>
<tr>
<td>MoSART</td>
<td>Model Search, Analysis and Reuse Tool</td>
</tr>
<tr>
<td>PCD</td>
<td>Process-Classes Directory</td>
</tr>
<tr>
<td>PClu</td>
<td>Primary Clustering</td>
</tr>
<tr>
<td>PEF</td>
<td>Pattern Exploration Facility</td>
</tr>
<tr>
<td>PM</td>
<td>Preprocess Models</td>
</tr>
<tr>
<td>RE</td>
<td>Regular Expression</td>
</tr>
<tr>
<td>RSM</td>
<td>Reduce and Split Models</td>
</tr>
<tr>
<td>RSiR</td>
<td>Reduced Simulink Repository</td>
</tr>
<tr>
<td>RStR</td>
<td>Reduced Stateflow Repository</td>
</tr>
<tr>
<td>rpm</td>
<td>Revolutions per Minute</td>
</tr>
<tr>
<td>SC</td>
<td>Sub-Classing</td>
</tr>
<tr>
<td>SiL</td>
<td>Software-in-the-Loop</td>
</tr>
<tr>
<td>SCSP</td>
<td>Search for Chart Sequence Patterns</td>
</tr>
<tr>
<td>SLOC</td>
<td>Source Lines of Code</td>
</tr>
<tr>
<td>SNUA</td>
<td>Subsystem Name Usage Analysis</td>
</tr>
<tr>
<td>SO</td>
<td>Statechart Object</td>
</tr>
<tr>
<td>STH</td>
<td>Subsystem Topology Hashing</td>
</tr>
<tr>
<td>STR-R2015a</td>
<td>Simulink Toolbox Release R2015a Model Set</td>
</tr>
<tr>
<td>TS</td>
<td>Task Scheduler</td>
</tr>
<tr>
<td>srId</td>
<td>Simulink Repository Identification</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Software, either replacing or enhancing the functionality of mechanical and electrical components, has made it possible to achieve the sophisticated levels of automation and features found in current automobiles [45], aircraft [46] and system-on-a-chip controllers [68], among many others. Building reliable systems using this paradigm dramatically increases the complexity, size and diversity of the software required. System designers and software developers have to address complex requirements to control and synchronize the real-time behaviour of large sets of embedded components, which contribute to the overall functionality of a major device [1], commonly referred to as a cyber-physical system [6].

Nowadays, practitioners routinely develop visual models to efficiently manage this level of complexity. Using these models, they capture and validate requirements, debug and test systems [14], create virtual environments that simulate the dynamic behaviour of targeted subsystems [47], design control schemes [80], or generate code for hardware controllers (i.e., ECUs), etc. The Simulink environment [56] has been extensively used in industry to develop models for these types of applications, due to its stability, versatility and extensive library support. Simulink models can speed up the capture and expression of complex requirements with fairly practical levels of formality. Their wide spectrum of application has resulted in the proliferation of large repositories of Simulink models that exist today.
1.1 Thesis Statement

Over time, these Simulink repositories have accumulated modeling knowledge that is likely to be reapplied in situations such as the design of new systems, the upgrade of existing ones, or the testing at large of any systems in the given domain. Developing systems for a product line environment [70], where multiple variants of a given product co-exist, would greatly benefit from reusing model-patterns inferred from these repositories. In this dissertation, we propose a framework to discover and manage model-patterns existing in these large repositories of Simulink models.

1.1 Thesis Statement

Exploratory work of public domain and industrial repositories of Simulink models for the purpose of identifying modeling patterns, as well as previous industry experience integrating automation components embedded in railway systems and manufacturing production lines, has influenced us to elaborate on the hypothesis for this thesis as:

Large repositories of Simulink models contain a great deal of valuable modeling knowledge that we can extract as model-patterns. A framework that implements 1) low-time-complexity algorithms to traverse the source code of the models in these repositories to evaluate pattern-like subsystem properties, 2) a subsystem-clustering scheme of classes that allows users to execute further model-pattern inference operations, and 3) an interactive user interface that supports queries and operations on the computed cluster classes, can effectively produce groups of Simulink model-patterns assembled for application in targeted use cases. Such a framework supports an iterative and progressive approach to discovering semantic patterns, consisting of three major phases: model clustering, primary model-pattern inference, and model-pattern refinement.

This hypothesis acknowledges and addresses the main challenges arising when attempting to extract actionable model-patterns from the large, heterogeneous and content-rich repositories of Simulink models that we target in this work. We identify these challenges as:
1.2 Scope of the Thesis

- **Search Scalability.** The search for similar entities, with potential to become model-pattern components, should scale to very large repositories of Simulink models. It should scan and evaluate pattern-like properties of entire repositories within time intervals that make it feasible to implement an iterative approach to infer Simulink model-patterns. That is, the model-pattern discovery process can efficiently retrace, from potential semantic patterns in the later stages, all the way back to the content of repositories, and quickly reprocess any repository reconfigurations geared to reinforce the refinement of targeted patterns.

- **Entity Reduction.** The clustering, reduction, aggregation, comparison and mapping of the granular entities in the repository should have a high level of repository coverage and be conducive to assembling model-patterns at an abstraction level that allows their refinement and traceability.

- **Pattern Evaluation.** Model-patterns of varying complexity and abstraction exist in large Simulink repositories. Automated pattern extraction systems can produce any type of similar syntactical structures that we can assimilate to patterns. However, in a software engineering context, only practitioners with good domain knowledge can refine and abstract them to targeted model-patterns, assign design meaning to them, and decide about their applicability. Therefore, there should be an interactive environment that presents a complete summary view of the repository, with drill-down capabilities, which practitioners can use to expedite these processes and record pattern-refinement decisions.

1.2 Scope of the Thesis

To address the challenges laid out in section 1.1, this dissertation proposes a model-pattern discovery and management framework to extract, infer, classify and manipulate for application in targeted use cases, modeling patterns originated in large industrial-strength repositories of Simulink models [56].

The scope of the work includes:

- The design of algorithms to 1) efficiently traverse a repository to identify, parse and evaluate
1.2 Scope of the Thesis

properties of all the subsystems in each Simulink model, 2) compute for each of these subsystems representational values of its size and its encapsulated graph-like topology of blocks and dataflows, and 3) cluster the subsystems by size into a set of classes for further model-pattern identification. To maintain accuracy and facilitate comprehension, we provide formal and semi-formal descriptions of the different components of these algorithms.

- The implementation of these algorithms into a software tool, which we have named spSearch (Simulink Pattern Search), to automate the clustering and evaluation of subsystems, in order to search for patterns on multiple repositories and iteratively refine our searches.

- The development of the interactive user interface MoSART (Model Search Analysis and Reuse Tool) for querying, visualizing, inferring, refining and managing Simulink model-patterns. In this tool we implement a query system, based on matching regular expressions on subsystem property values, to support operations on subsystem cluster classes. This feature is an aid to practitioners for inferring semantic model-patterns based on their domain expertise. In addition, this tool provides configuration, persistence and recall of related groups of subsystems identified as model-patterns.

- A case study involving the analysis of a domain-specific repository of Simulink models, provided by an industrial partner, to identify groups of model-patterns with potential for application in specific use cases. We propose an iterative approach, composed of three phases: 1) model clustering, 2) primary model-pattern inference, and 3) model-pattern refinement, to obtain these groups of patterns with the tools of our framework. We compare the results of this case study with the ones obtained from sets of Simulink models available in the public domain.

1.2.1 Model-Pattern Discovery Approach

We believe that finding Simulink design patterns in large repositories should be a progressive and iterative process, which gradually reduces the size and complexity of the search space, and delegates to trained practitioners the ultimate decisions about the patterns’ value and scope. Building upon
1.2 Scope of the Thesis

this concept, we propose a model-pattern discovery framework whose tools efficiently parse and analyze the source code of the Simulink models, and also provide effective model visualizations. Using the tools of our framework as the keystone, we propose a three-phase iterative approach to discover and refine Simulink design patterns (refer to figure 1.1). We conceptualize the phases of this approach, and describe our framework’s actions in each one, as follows:

- **Model-Pattern Clustering.** The spSearch tool 1) consolidates and annotates the source code of the Simulink models in a repository, 2) separates structural and behavioural subsystems, 3) traverses the entire annotated repository and parses each subsystem to compute subsystem properties, 4) clusters the subsystems by size to produce a set of property-addressable classes, and 6) produces a series of reports on subsystem-name usage and behavioural subsystems (i.e., Stateflow charts). The classes and reports produced in this phase become inputs to the next one.

- **Primary Model-Pattern Inference.** The interactive user interface MoSART takes as input the set of classes produced in the previous phase and integrates it into a Simulink-model query and visualization environment. Practitioners optionally display selected classes, query and visualize...
1.2 Scope of the Thesis

their subsystems, and manipulate (i.e., configure, combine, save and recall) their elements to assemble and/or refine specific Simulink model-patterns, that is, primary patterns. They may optionally use the reports produced in the previous phase.

- **Model-Pattern Refinement.** Again, practitioners use the capabilities of MoSART to obtain more abstract and complex patterns, which may involve using the name-usage and behavioural-subsystem analysis reports produced in the model clustering phase. The goal of this phase is to obtain model-patterns that have a system design meaning within the context of either a domain, system, project or use case. That is, the objective is to obtain semantic Simulink model-patterns.

The tools of the framework are geared to implement highly efficient model-pattern search algorithms and practitioner-friendly user interfaces. They support a three-phase pattern discovery approach, as shown in figure 1.1, which produces partial results and data structures designed to make it effortless to trace, review and refine the patterns in an incremental manner. We can rerun the workflow spSearch multiple times during the model-pattern clustering and primary inference phases to, for example, extract more abstract pattern trends from different versions of the same repository. We can interactively query and manipulate the generated cluster classes with the MoSART user interface, during the model-pattern primary inference and refinement phases to, for example, refine and contextualize a set of model-patterns. The main thrust is to preserve the design knowledge and business decisions accumulated in those repositories of Simulink models, which in most cases are mission-critical and not easily changeable, to extract the best structures and practices as model-patterns for application in targeted use cases.

We believe that many of the abstractions in our pattern-search algorithms, for example, the bottom-up repository traversal, the size-based subsystem clustering approach and the subsystem topology hash functions, could be applied to discover patterns in repositories developed with languages similar to Simulink. A good example would be a repository of UML-RT [60] models. In this specific case, we could associate capsules to subsystems, connectors to links and state machines to Stateflow charts to traverse and cluster the modeling entities in a way similar to our work.
1.2 Scope of the Thesis

In summary, we concentrate on providing tools and methods that efficiently highlight and augment the modeling knowledge embedded in repositories of Simulink models, so that we can help practitioners to:

- Infer sets of Simulink model-patterns, with application to specific projects or use cases, whose elements are theoretically sound in definition and pragmatically useful in abstraction and scope.

- Infer partial and more elaborate Simulink model-pattern results within short elapsed times which would allow increased levels of iterative analysis and refinement.

1.2.2 Detection of Syntactic Patterns

Visual modeling frameworks such as Matlab Simulink [56] and Rhapsody [43] store textual versions of the models in a proprietary and a standard model interchange format (XMI) [64], respectively. Software tools can parse these textual versions for automated syntactic analysis and interpretation of many of the structural and behavioural configuration parameters embedded in the visual designs they represent. However, these textual model representations usually extend to several million lines of source code for the large and complex model repositories typical of industrial domains (i.e., avionics, automotive, etc.). Therefore, effective and scalable approaches for extracting syntactical model-patterns from them have to use, if possible, algorithms that execute with time-complexities equal or lower than \( O(n\log(n)) \), where \( n \) is the total amount of source lines of code (SLOC) required to define the models in a repository. Furthermore, they should be compatible with processes that use multiple, short and progressive model-pattern refinement iterations. We believe that these qualities create the conditions that allow practitioners to assign design meaning (i.e., semantic model-patterns) to syntactical patterns obtained through the automated or semi-automated processing of models.

The core entities in building Matlab Simulink models are blocks, representing units of functionality with input and output ports, and links, representing typed dataflows between the input/output ports of different blocks. Blocks can encapsulate, into a single unit denominated subsystem, any
1.2 Scope of the Thesis

number of blocks and links connected in any allowed configuration, allowing designers to model the concepts of functionality units, composition of functionality and hierarchical top-down or bottom-up modeling to any level of component nesting. This feature allows us to conceptualize Simulink models as directed rooted-trees whose nodes are subsystems and whose edges represent instances of a containment relation. Furthermore, we can view a repository of Simulink models as a forest of directed and rooted trees. Finally, we can conceptualize Stateflow charts as subsystems containing statechart and/or flowchart diagrams that represent modal behaviours of the Simulink subsystems in which they are embedded.

These conceptualizations are fundamental in our work to develop low-complexity model-pattern search algorithms that analyze the source code definitions of Simulink models. The Simulink environment uses a markup language (refer to appendix A, section A.1), consisting of meta-symbols and tags, to stores the visual representation and configuration parameters of a model. These syntactic structures are a key enabler for building software tools that analyze large repositories of Simulink models with grammar-based or context-sensitive parsing approaches. Our framework (described in section 1.2.1), with its spSearch tool, views a repository of Simulink models as a forest of directed trees and contextually parses subsystems of equal nesting depth, in each model, in a bottom-up fashion. (That is, it moves from the deepest nesting level of any subsystem in a repository to the root-subsystem of each model.) Finally, it clusters all the Simulink subsystems in a repository into a set of classes optimized for the inference, refinement and visualization of model-patterns. Its central problem reduction and pattern-search strategies are as follows:

- Instrument the textual definitions of models with indexing fields that would allow the identification of: 1) the type and location of a text line within a model, 2) the nesting level, within a model, of the entity for which the text line provides some definition, and 3) the model file where a text line belongs.

- Consolidate all textual model definitions in a repository into a single unit for further analysis and identify all formatting tags identifying any type of allowable modeling entities.
1.2 Scope of the Thesis

- Reduce the model text line definitions in the consolidated area to those that only define structure in models, namely, those that delimit models, subsystems, blocks and links. This step filters out any other text lines defining properties, comments, formulas, tables, etc.

- Traverse and parse the text lines of the filtered consolidated area in a bottom-up fashion, using the entity nesting level field previously defined, to identify each subsystem, in every model of the repository, and its size defined as the total number of blocks and links it encapsulates. Label each subsystem with a size-based identifier.

- Cluster the subsystems by size into basic pattern classes, where the combination of the number of blocks and number of links values serve as a unique identifier for each class. Produce a consolidated set of subsystem-size-based classes.

- Refine the syntactical model-patterns reduction, extension, sub-classing, class consolidation, intersection of derived classes, and other operations on the subsystem-size-based cluster classes. Workflow-oriented (e.g., spSearch) and interactive (e.g., MoSART) tools, implementing efficient entity-clustering algorithms, versatile pattern refinement queries and flexible subsystem visualizations, along with domain expertise, combine to efficiently and successfully complete these tasks.

- Repeat the previous step to identify subsystem patterns by frequency of use, type of component it encapsulates, potential for composition with other subsystems, role of the subsystem within the repository, and any other relevant indicators. The objective is to isolate groups of model-patterns with a desired syntactical structure.

- Repeat the previous step to identify a potential set of model-patterns applicable in identified use cases.

In summary, in our framework, the process of detecting syntactic Simulink model-patterns produces as output a set of base and derived classes of Simulink-subsystems. Base classes cluster
1.2 Scope of the Thesis

subsystems by size. Derived classes organize them by any practitioner-originated rules, imple-
mented as queries on the base classes or manual reorganizations of any subsets of classes.

1.2.3 Inference of Semantic Model-Patterns

Industrial repositories of Simulink models are usually domain-specific. Examining these reposito-
ries, to identify similarities of syntactical properties (i.e., size, topology, names, etc.) between mod-
eling entities, will initially produce model-patterns that are relevant to specific use cases, projects
or development teams. We will likely need to modify and conceptualize these model-patterns, at
higher abstraction levels, to increase their potential for portability to other projects or domains.

We propose in this work a software toolchain, along with a pattern-discovery approach powered
by this framework, to address these issues and enable the conceptualization of reusable modeling
ideas (i.e., semantic model-patterns) in a home domain, with potential applicability to other do-
 mains. Both framework and approach work together to enable domain experts and engineers to
refine purely syntactic model-patterns detected from the models’ source code into semantic model-
patterns that represent valuable modeling encapsulations. Both work in conjunction to help practi-
tioners decide about the abstraction-level, granularity, scope and design-meaning assigned to those
patterns. Their mission is to support, as far as possible, the identification of classes of semantic
model-patterns associated with one or more of the categories recognized in the practice of embed-
ded systems design, namely, open- and closed-loop control, symmetric or asymmetric redundancy,
fail-safe or self-healing safety, component testing and hardware-in-the-loop simulation. We ac-
knowledge the fact that non-functional and project-oriented factors will be considered in the final
decisions about what constitutes valuable semantic model-patterns. Ultimately, if practitioners are
to justify the use of model-patterns to build, analyze or test models, they need to do it with less
effort, reduced costs, enhanced reliability and extended functionality.

We believe that exhaustive automation of these ideas would require the construction of a highly-
specialized knowledge base with a sophisticated inference engine, which, due to its complexity,
1.2 Scope of the Thesis

would be extremely hard to validate [41]. Our framework and approach to discovering model-patterns represent a pragmatic alternative to obviate these problems by 1) facilitating the quick extraction of the engineering knowledge hidden in repositories of Simulink models, 2) giving experts a flexible environment to make decisions about patterns, and 3) allowing them to assign the most appropriate design meanings to those patterns in agreement with their intended use.

1.2.4 Sets of Model-Patterns

We envision that using our framework and approach, a systems development organization would direct its efforts to the identification of specific sets of model-patterns, deemed applicable in the modeling endeavours of a product-line environment [70] or group of projects. In such a situation, domain experts would likely need to apply refinement and abstraction rules that produce groups of model-patterns that are not only good candidates for reuse, but also meet specific standards and requirements that guarantee their applicability [81]. Ranking these sets of patterns by attributes such as portability, abstraction level, design granularity and compliance with reuse standards would be a most effective way to manage their acceptance. Perhaps some of the model-patterns in these sets will require additional formal or semi-formal assertions, that is, a model-pattern description language that allows one to specify quantitative or descriptive ranges for these attributes. This feature would be most helpful to practitioners when attempting to assemble, propose and formalize emergent Simulink model-patterns.

We believe that our pattern-discovery framework and approach effectively address and support most of the syntax-related concerns in assembling these sets of Simulink model-patterns. Our tools spSearch and MoSART provide extensive support for the implementation of “what-if” scenarios using sets of model-patterns. We can change the size and content of the repositories of Simulink models in a controlled manner, and have our tools act upon them with new sets of model-patterns produced in minimal time. We can interactively reorganize these model sets with MoSART and analyze common properties. Our framework can help experts build a great number of inferences so that they can label these sets with the modeling (i.e., design) semantics deemed most valuable.
1.2 Scope of the Thesis

1.2.5 Model-Patterns Use Cases

Ideally, in our context, the identification of Simulink model-patterns from large repositories should have a largely positive impact on increasing the productivity and reducing the effort to develop, maintain, test and simulate embedded systems modeled in the Simulink language. Depending on the size and complexity of the repositories, the modeling practices in use, and the requirements of a development organization, we can envision the following model-pattern reuse and evaluation environments to address these needs:

1. An interactive tool that allows modeling experts to i) refine syntactic model-patterns, ii) assign design meaning to those model-patterns, and iii) manually reconfigure and externally reapply those model-patterns in specific use cases.

2. A tool that extends the functionality described in item 1, and also identifies sets of patterns and their mappings to configurable Simulink modeling scenarios.

3. An extensible framework that implements all the functionality described in item 2, and also implements i) programmable definition rules for generic sets of Simulink model-patterns, and ii) programmable usage rules that establish customizable mappings from Simulink model-pattern to use cases.

Regardless of the framework’s sophistication or the model-pattern reuse cases under consideration (e.g., impact analysis, systems development, testing, simulation, “what-if” analysis), practitioners would likely expect these tools to 1) scale up the computation of similarities to entire repositories of Simulink models and very large numbers of modeling entities (e.g., subsystems), 2) establish quick-access pointers to groups of those entities to expedite the evaluation of properties, similarities, relations and frequency of occurrence in model-pattern components, and 3) provide the necessary views about the model-patterns so they can be efficiently modified, saved, evaluated and reused.

The MoSART user interface, along with the phases of the model-pattern discovery approach we
1.3 Thesis Contributions

propose (refer to figure 1.1), address these requirements. They implement a flexible and expandable prototyping environment which we use to confirm our hypothesis (refer to section 1.1) about the effectiveness of low-complexity model-pattern search algorithms, flexible subsystem-clustering schemes and visual-analysis user interfaces, in the iterative discovery and subsequent reuse of Simulink model-patterns. Without recurring to model-pattern description and reuse languages, we provide a versatile and pragmatic environment to check our hypothesis and reason extensively about model-patterns and reuse case scenarios. With spSearch, MoSART and our model-pattern discovery approach, we can generate multiple reuse workflows. A generic example of a workflow would consist of the steps to 1) identify a set of model-patterns to address a single type of reuse case, 2) evaluate the set’s application scope, and 3) iteratively extend the set to support additional reuse scenarios.

1.3 Thesis Contributions

The main thrust of this thesis is to devise efficient algorithms and approaches to extract semantic model-patterns for embedded systems from repositories of Simulink models. These patterns represent recurrent micro- or macro-units of structure and functionality in the modelling of one or more systems. Their identification can help practitioners recover valuable design knowledge, highlight best modeling practices, and identify design pitfalls to avoid. To meet these goals, an iterative approach offers many advantages, as it advocates increased levels of repository-coverage and accuracy in detecting and refining these patterns. Finally, a framework consisting of tools for quick, focused and repeated analysis of Simulink repositories is necessary to match this approach. In response to these challenges, the main contributions of this thesis are:

- A Simulink model-pattern discovery framework consisting of a software toolchain that allows domain experts and modelers to 1) quickly parse and iteratively analyze very large repositories of Simulink models to identify sets of ready-to-refine syntactical model-patterns, and 2) query, visualize, refine, preserve and gather information about those model-patterns to identify them
1.3 Thesis Contributions

with useful design meanings, that is, semantic patterns. The spSearch tool clusters by size all
the model-subsystems of a Simulink repository, computes topological properties for each subsystem,
reports on statechart patterns, produces subsystem clustering and name-usage reports, and
generates formatted subsystem clustering data for further processing by other tools. MoSART
reads the output from spSearch, provides a graphical user interface for the interactive querying
and analysis of cluster classes, allows users to perform cluster sub-classing and class consolidation
operations, enables the selective visualization subsystem diagrams, and provides persistence
for queries and operations on classes. Finally, a set of ancillary utilities allows the automation of
support tasks such as profiling and reorganization of Simulink repositories, computation of model
meta-data and automated operations on subsystem cluster classes.

• A semi-automated and iterative approach to discover patterns in industrial-scale and very large
repositories of Simulink models, consisting of three phases: I) model-pattern clustering, II) primary
model-pattern inference and III) model-pattern refinement. Phase I is fully automated and
the tool spSearch, in our framework, implements it. The MoSART tool supports the model-
pattern queries and visualizations executed in phases II and III. These phases require the inter-
tervention of modeling and domain experts to manipulate and evaluate the primary-model-patterns.
This approach is an efficient and flexible alternative for inferring semantic model-patterns, which
does not require to build ontologies or a specialized knowledge base.

• A set of low-time-complexity algorithms (i.e., between $\Omega(n)$ and $O(n \lg n)$, $n =$ total num-
ber of SLOCs in a repository) to traverse the source code definitions of entire repositories of
Simulink models, evaluate modeling entity properties, cluster subsystems into a set of classes
and produce the information necessary to identify syntactical model-patterns. Hash functions
generate distinctive digital signatures for the topology of functional blocks and dataflows of each
model-subsystem. The short running times of these algorithms enable rapid iterations of pattern
discovery and refinement processes. Experts can implement efficient and flexible semantic
model-pattern extraction approaches on very large repositories.
1.4 Thesis Organization

- Empirical sets of semantic model-patterns, extracted from an industrial case study, consisting of Simulink models for embedded components in automotive systems, which validate the application of the framework and approach proposed.

1.4 Thesis Organization

This introduction has given an overall view of the nature and scope of the work completed. The rest of this thesis is organized as follows:

- Chapter 2 introduces some fundamentals of modeling and developing hybrid systems, often referred to as cyber-physical systems, embedded as modules with dedicated functionality in complex multiple-component systems. It also describes the problem of finding modeling patterns in large repositories of Simulink models for these systems, and outlines ways of reusing those patterns in planned use cases.

- Chapter 3 introduces the architecture of the pattern discovery and refinement framework we propose in this dissertation. Three core assumptions delimit the modeling environment, language, and pattern search focus for this framework. The modules of the spSearch workflow automate the initial pattern detection and subsystem clustering tasks. The MoSART client/server graphical user interface allows the interactive elicitation and refinement of Simulink model-patterns. Ancillary utilities support the work of these main tools with additional pattern-search automation, meta-data maintenance and cluster class operations.

- Chapter 4 describes the algorithms to traverse the source code of repositories of Simulink models and selectively parse model-subsystems, compute their core topological properties, cluster them by size into addressable classes, and report on subsystem-name usage. It also introduces an algorithm to detect sequences of Stateflow charts in Simulink models. It provides details of the hash functions to compute the name- and type-connectivity of blocks and dataflows in each subsystem. It presents an analysis of the steps in subsystem name-usage analysis scenarios.
1.4 Thesis Organization

- Chapter 5 describes the processes used to infer semantic model-patterns by operations on the subsystem-cluster classes produced by spSearch, using the class querying and management facilities of the MoSART user interface. The overall pattern discovery workflow with this toolchain consists of two main stages: automated model-pattern discovery and interactive semantic model-pattern elicitation. Cluster-class consolidation and sub-classing operations on MoSART, and the application of expert domain knowledge, support the semantic inferences of the latter stage. The class-querying facility of MoSART expedites these operations on classes. Model-pattern entailment and abstraction processes create the final proposals for semantic patterns.

- Chapter 6 describes an empirical analysis of an industrial-strength repository of Simulink models with the pattern discovery toolchain proposed in this thesis. Initial elicitation of model-patterns using cluster-class operations on MoSART, as well as subsystem size, frequency and name-usage analysis, produced several different types of subsystem model-patterns in this repository. Subsequent semantic entailment of these groups produced a reduced group of more abstract patterns. Overall reasoning about the discovered model-patterns leads to conclusions about existing plant-control architectural model-patterns in this repository. Finally, comparisons with public domain repositories highlight the qualities of this repository.

- Chapter 7 summarizes the work on Simulink model-pattern discovery algorithms, tools, and procedures in the framework introduced in this dissertation. It discusses future work and provides final conclusions about the solutions and results achieved.
Chapter 2

Background

By design, modern automobiles, aircraft, healthcare equipment, and many other complex artifacts are built as ensembles of hybrid assemblies. They host large numbers of embedded systems having complex arrangements of sensing and processing hardware, communication devices, software and mechanical components. These subsystems are reactive in nature and implement specialized functionality, thus their correct operation is often critical and the overall safety of the systems in which they are embedded strongly depends on them. They also interact with each other, using signals and protocols, according to specific rules that guarantee the correct implementation of specified system requirements. Currently, development teams in industry often use visual models to capture complex requirements and manage the diversity of entities involved in the design and development of embedded systems. These models are written in cross-domain languages such as Matlab Simulink [56], or domain-specific modeling languages (DSML) such as the UML-RT [60] profile of UML [63]. Their most common use in the embedded systems domain is to support the simulation, testing, validation and implementation of large and heterogeneous systems of components. Existing software tools can parse them and automatically generate module-specific C or C++ code from model definitions. Consequently, models have become first-class software entities in an organization’s development life cycle [82], resulting in the creation of large and complex model repositories which accumulate extensive and valuable system design knowledge.
2.1 Embedded Systems

Tools to find patterns in repositories of models can be a very effective approach to help practitioners reduce costs and time-to-completion in the modeling of new systems, and manage the complexities involved in upgrading models for existing systems. Product lines [70] and complex technology environments [82] can greatly benefit from reusing model-patterns. The extraction of patterns from these large and heterogeneous repositories requires highly-scalable search algorithms, with low time- and space-complexity. Furthermore, any pattern management framework for these types of repositories should at least support summary and drill-down views of classes and specific sets of modeling entities (e.g., subsystems). Finally, it is critical to identify any existing functional (e.g., communication, synchronization, delegation) [73] and non-functional (e.g., safety, liveness, modularity) [6] specifications that must be preserved in the components of the inferred patterns. The following sections provide an overview of embedded systems, their associated models and the approaches to managing them for reuse.

2.1 Embedded Systems

2.1.1 Overview

Embedded systems are “a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function” [31]. They are reactive building parts of a larger system, to which they contribute with critical and dedicated functionality, subject to logical, physical, execution and timing constraints [42]. They are also known as cyber-physical systems because they usually comprise modules and software which monitor and react to physical variables of their environment [6].

Precise specification and modeling of their functionality and interfaces enables aggregating them into sound, safe and cost-effective modular systems. Complete specifications for these systems must describe: 1) the physical and logical properties of their environment and the interactions with it [65], 2) their roles and limitations as components executing some kind of control software [76], and 3) the timing, concurrency and precedence conditions that regulate the interactions between their
2.1 Embedded Systems

internal components and with their environment [73]. Figure 2.1a shows an example of a simplified assembly with mechanical, electrical and processing components, embedded in an automobile to monitor and limit the maximum revolutions per minute (r.p.m.) of its combustion engine. Figure 2.1(b) shows a Simulink model, containing a Stateflow chart and some function blocks to simulate the behaviour of the control system in figure 2.1(a).

![Figure 2.1: Combustion Engine Tachometer and Revolutions Limiter Control](image)

Overall, embedded systems must behave in a predictable and reproducible manner, carrying out well-defined tasks subject to strict scheduling schemes and completion deadlines. They mostly operate as real-time systems performing scheduled tasks having soft- or hard-completion deadlines. When tolerating soft-deadline tasks, they implicitly accept some degree of performance degradation in the functionality assigned to them. In contrast, critical embedded systems, also known as vital systems, mostly execute hard-deadline tasks where a missed deadline represents a failure state and usually, an alternative recovery procedure must be provided in such situations [65].

Often, their designs include redundant components and multiple connectivity between elements to guarantee highly-reliable operation, eliminating single points of failure and providing fail-safe
2.1 Embedded Systems

operation under abnormal conditions. These are essential requirements for embedded systems in
domains such as automotive and aircraft, where operational safety is strictly assessed and regulated
[1]. Customarily, designers rely on formal or semi-formal languages and techniques to rigorously
specify trustworthy embedded systems [61]. Modeling efforts geared to design and validate control
schemes [50], or develop software for these systems [45], must ensure that the implementations
derived from these models meet rigorous safety, fairness, liveness and interfacing properties [6].

2.1.2 Modeling Embedded Systems

Modeling embedded systems to, for example, generate software from a set of models that would
operate these systems reliably and efficiently, requires multiple internal and external design views.
A productive set of models would normally cover: 1) all necessary internal functional and non-
functional system requirements [1], 2) the relevant properties of the interactions of these systems
with their physical environment [65], 3) the capabilities of the computational devices hosting the
software [68], and often, 4) compliance and interfacing aspects with a general hardware/software
architecture hosting multiple embedded systems [8].

(a) Plant/Control Models  
(b) AUTOSAR Layered Implementation Architecture [8]

Figure 2.2: Models and Implementation Architectures for Embedded Systems
2.2 Modeling Patterns and their Reuse

Figure 2.2(a) shows a diagram of a common approach to separating modeling concerns in the design of cyber-physical systems [1, 7, 13] into: 1) plant model(s), which capture the physical properties and operational requirements of the system, 2) control model(s), which describe the algorithms and equations that control the system’s static and dynamic behaviour, 3) sensor and actuator models, which interface plant and update/sampling models, and 4) update and sampling models, which act upon the plant and sample its behaviour for continuous control. Figure 2.2(b) shows the layered software/hardware implementation architecture for embedded systems proposed by the AUTOSAR standard [8], where software modeling occurs on the application layer.

Overall, designers produce sets of models, each one them representing a partial view of an embedded system, geared to reduce the complexity and extend the coverage of the analysis. The underlying goals are: 1) the rapid implementation of useful system simulation and testing prototypes, 2) the generation of software that is reliable and easy to configure, test, implement and maintain, and 3) the maintenance of central artifacts (i.e., the models) that capture and consolidate the different levels of requirements for heterogeneous sets of components [69]. In this model-driven development approach, visual languages based on functional blocks, typed data flows between blocks and state machines, support a wide range of modeling requirements and scale up very well when used to model large and complex systems [68, 80]. In this dissertation, we parse repositories of models for embedded systems developed with the visual language Simulink [56] and its Stateflow extension [57], with the purpose of extracting modeling patterns. As a case study, we use repositories from the automotive domain for our analysis.

2.2 Modeling Patterns and their Reuse

In the Simulink’s modeling paradigm, designers use atomic or compositional functional blocks, typed data flows between blocks, and state charts as essential entities to build structural and behavioural models of embedded systems [53]. In this context, we define a Simulink model-pattern as 

*possibly nested arrangement of state-driven or stateless functional blocks, connected by a scheme
of typed data flows, which is recurrent due to its applicability and perceived value in designing and maintaining multiple aspects of embedded systems in a given application domain. According to this definition, we can use Simulink models to represent the operation of systems [46], the architecture of components, redundancy for fail-safe schemes, timing and synchronization of protocols, simulation schemes with [58] or without hardware-in-the-loop [47], testing schemes [14], hardware emulation [47], code generation, and many other systems engineering concerns.

2.2.1 Model-Patterns of Embedded Systems

Generally, the initial modeling phases of an embedded system focus on clearly identifying core functionality requirements and system properties. Aspects such as its real-time behaviour, response to its environment, structural reliability and operational safety, set important boundaries on the modeling choices: modeling language, top-down or bottom-up analysis, selection of modeling views, etc. [45, 80] Ideally, the models built under these constraints are a faithful embodiment of the system, their visual representations are highly descriptive, and the system properties they capture can be rapidly verified using semi-formal or formal techniques [61]. Appropriate partition, modularization, organization (e.g. hierarchical, networked) and relational connectivity of the modeling entities play a defining role in endowing a model with these desirable properties [65]. Figure 2.1(b) shows a Simulink/Stateflow model assembled with these goals in mind.

Large industrial repositories of Simulink/Stateflow models, built over years of engineering practice, tend to accumulate hidden or partially-known design schemes associated with types of embedded systems in a specific domain [47, 58]. Focused discovery and proper management for reuse of these modeling patterns, encoded as typed labeled multi-graphs of functional blocks and data flows, can greatly reduce future modeling efforts. In this sense, similar to their software design counterparts, modeling patterns that: 1) reflect and effectively document successful practices in a given domain [16], 2) can be clearly communicated through classifications and diagrams at appropriate abstraction levels [28, 30], 3) have resulted from collective knowledge of proven experiences in a domain (e.g., embedded systems), where they have potential for supporting concrete solutions
2.2 Modeling Patterns and their Reuse

to known problems [34], and 4) reflect the way designers and developers successfully approach systems development challenges in their organizations [74], tend to have significant value for development teams.

The block-and-data flow representation of Simulink models makes a strong case for identifying special types of graphs when attempting to discover patterns in model repositories developed in this language. At a very abstract level, one can view these repositories as sets of directed labeled multi-graphs with typed nodes (i.e. blocks) and edges (i.e. data flows) [20, 21], where each node can, again, be a directed labeled typed multi-graph. Initially, this nested graph view has the potential to reduce the problem of discovering Simulink model-patterns for embedded systems to a multiple graph matching problem that, in theory, could be largely formalized and automated. However, finding all similar or related sub-graphs within a set of graphs, namely, solving the sub-graph isomorphism problem, is a computationally difficult problem (i.e., NP-complete) [32]. Approximated graph-matching and certain specific incarnations of the problem can be solved in polynomial time for relatively large sizes (e.g., 1000 nodes) [33]. However, graph-only analysis techniques to discover modeling patterns in Simulink models, such as determining sub-graph clone pairs in flattened models [22], normalizing graphs to determine semantic clones [3], or computing maximal clique-graphs to determine the variants of a model subsystem [72], are applicable to relatively small exemplars.

Clone detection techniques applied to the source code of the Simulink models in a repository have produced similar uncertain scalability results. They have to deal with large scalability versus recall rates trade-offs [4]. That is, in order to have reduced subsystem clustering times, they only cluster subsystems whose size is within a certain range. This situation represents a very partial coverage of a repository, from a model-pattern search standpoint, with high numbers of false negatives. Also, these techniques produce clusters of quasi-identical subsystems, which are difficult to associate with each other, in order to infer model-patterns, and are susceptible to interpretations that represent false positive patterns. Furthermore, often the identified clone classes lead to unclear design meaning assignments to clustered graph-like structures [29].
2.2 Modeling Patterns and their Reuse

As a scalable alternative to the limitations imposed by graph-based analysis and source code clone detection techniques, in this dissertation we propose an iterative approach to finding modeling patterns in a repository, which selectively parses and traverses the source code definitions of the Simulink models. Our main objective is to synthesize semantic model-patterns from elements with rich meanings and properties, such as Simulink’s functional blocks and statecharts. The approach uses a software toolchain to cluster model subsystems into uniquely identifiable classes, using a specific metric for subsystem size, to provide a compact view of a model repository useful for fast interactive querying and iterative pattern refinement. Additionally, an interactive software tool provides the following capabilities to manage the size-based classes of model subsystems identified in a repository: 1) select specific classes for further analysis, 2) browse selected model subsystems, 3) query classes to identify sub-classes, overlapped classes and related classes, and 4) provide persistence for class-based identified patterns to quickly rebuild them and visualize them.

2.2.2 Reuse of Simulink Patterns

The approach and tools we propose to discover and manage model-patterns act upon the source code definitions of the Simulink models, however, users can visualize these patterns as classes of similar or related block diagrams. Simulink provides the subsystem and chart blocks to encapsulate structural and behavioural system modeling ideas, respectively [56, 57]. These blocks can be nested to any depth, and therefore, we can abstract a complete view of a Simulink model as a hierarchical typed-graph whose nodes can also be typed-graphs. Overall, these features of the language support the top-down or bottom-up modeling of systems.

We can view a syntactic Simulink modeling pattern as a set of related typed-graphs, having one or more common structural properties, within a set of hierarchical typed-graphs contained in one or more model repositories. The reuse of these patterns in identified use cases requires an understanding of the architectural and systemic abstractions (e.g. plant control, redundancy module, test interface, task scheduler) that can be attributed to these graph-like structures in a given application domain. Natural language and semi-formal descriptions of these semantic mappings expedite the
2.3 Pattern Discovery Approaches

work of identifying patterns and clearly describe them for reuse. They represent a good precision vs usability trade-off in the modeling practice when application of formal methods is not mandatory. In this dissertation, 1) we label these fully-qualified structures as semantic modeling patterns, 2) our application domain is embedded systems, 3) the framework for reusing them is based on a partially-automated approach and software toolchain, which requires the active participation of analysts and modelers in the more abstract iterative pattern-inferencing steps (i.e., semantic interactions) [67], and 4) the use cases for reusing our model-patterns can involve their reinsertion in new models (i.e., new, derived or upgraded systems) or their role as templates in maintaining existing ones (i.e., impact analysis, testing, simulation).

In general, clear reuse strategies and design semantics for model-patterns are determinant factors in identifying good candidates for reuse in a specific domain. They help establish many of their desirable structural and functional properties, such as: 1) optimal granularity: full model, model component, functionality block, or code segment, and 2) optimal functionality scope: cross-domain sub-model component, core domain entity model-pattern, generic function block, etc., and 3) adaptability and compatibility with new modeling requirements [71, 81].

Finally, an environment that consolidates pattern detection and management would allow the exploration of emergent patterns and the expansion, refinement and validation of proven ones. Such a framework should be flexible enough to facilitate the top-down (i.e., from pattern rules) and bottom-up (i.e., from sets of exemplars) recognition of model-patterns [23]. This work concentrates on these goals.

2.3 Pattern Discovery Approaches

Simulink models have an external graph-like representation for modelers, and an internal one, for model persistence, which consists of structured and tag-delimited text stored in a file. Structural, behavioural and display properties of models have corresponding definitions in both representations. Researchers have used the one-to-one nature of this mapping to attempt to detect similarities in
2.3 Pattern Discovery Approaches

Simulink models using graph-matching and graph-transformation techniques [3, 72] on the external representation of models, and string clone detection and pattern-matching on the internal ones [4]. The central premise is that the analysis and algorithms to answer a pattern research question on either representation should turn out equivalent results. Graph-based approaches tend to have higher time and space complexities and are less scalable than text-based approaches. The quality of the results, characterized by their precision, recall, granularity, scope, manageability, applicability, etc., depends more on the specifics of the searching, filtering and clustering algorithms applied than on the graph vs text duality [15, 23]. Both approaches attempt to discover cloned and similar substructures within the models, with a specific granularity, that could represent modeling patterns or identify common useful properties in a model set. The following are representative examples of both approaches.

Ryssel et al [72] represent in their work the subsystems of a Simulink model as undirected graphs, and then use a heuristic algorithm to produce a set of best-pair graph matches. These pairs of graphs form a first-level of subsystem clustering. Thereafter, the algorithm iterates over first-level graph-pair clusters to obtain best matches between them and merge them, and continues to do so with the successive cluster levels. The algorithm stops with a single cluster including all the subsystems in a model. However, the optimal answer to their research question, about finding representative subsystems and their most likely variants, is found in an intermediate iteration of the proposed algorithm. They acknowledge that their approach might run into a large number of graph comparisons and that the use of heuristics makes the processing more feasible, however, the degree of optimality provided by the solution cannot be guaranteed.

Al-Batran et al [3] use labeled directed graphs to define semantic model clones as subsystems of Simulink models having slightly different graph structure and identical meaning (i.e. encapsulated purpose or functionality). Semantics-preserving graph transformation rules act upon fragments of each subsystem and convert them to unique equivalent normal forms. Thereafter, a graph- and heuristic-based clone detection algorithm, introduced by Deissenboeck et al [22], processes the models with the normalized subsystems. Finally, a manual review of the clone classes generated by
2.3 Pattern Discovery Approaches

the previous process eliminates irrelevant clone clusters and produces a short list of semantic clones. Overall, the approach uses graph transformations for simple mathematical, logical and structural operations represented by basic blocks in the Simulink language. The results show experimentation with a single Simulink model and the scalability of the approach is not discussed, however, it is mostly determined by the algorithm in [22].

Alalfi et al [4] adapt a clone detection tool, which parses structured text, to detect Simulink model clones. A specific grammar for parsing Simulink text definitions with a source transformation language and customized sub-model extraction, layout attribute filtering, syntactic substructure sorting and identifier renaming steps, enhance the entity clustering and clone detection on Simulink models of the original tool. The resulting prototype detects exact, structurally-identical and graded syntactic-similarity model clones. The emphasis of the approach is on finding clone classes whose elements have a high degree of similarity (i.e., 70% or more) and the same granularity (i.e., models, subsystems or blocks). Experimentation with medium-size models and a comparative evaluation with the work in [22] highlight the approach’s scalability in the detection of near-miss Simulink clones.

In the field of mining software repositories, Hassan [40] discusses the challenges and benefits encountered in extracting useful information from various sources of textual data, including program code and logs of the activities involved in software development and maintenance. In previous work, Hassan uses special extractor modules to mine repositories of C-programs, recover their evolutionary history and identify change patterns [39]. This empirical study resulted in a proposal of a set of techniques to assist developers and managers to better understand legacy software, estimate the impact of using new software maintenance tools and allocate testing resources. With similar challenges and objectives, we address in this thesis the problem of extracting valuable modeling patterns from the source code of large and highly-structured repositories of Simulink models for embedded systems, which practitioners can reuse in multiple applications. We have developed a tool to traverse these repositories and cluster subsystems (i.e., spSearch) to identify syntactic
2.4 Pattern Representation and Reasoning

Simulink model-patterns, and an interactive user interface to query and visualize these syntactic patterns (i.e., MoSART) for refinements focused on endowing them with useful design or application semantics. We envision a variety of use cases for these patterns, for example, 1) pattern-insertion in new models, 2) support for refactoring existing models, 3) support for building testing models and 4) configuration of simulation environments (e.g., software-in-the-loop, hardware-in-the-loop and model-in-the-loop).

To accomplish our objectives, we introduce an iterative, interactive and multiphase approach and toolchain for the identification of modeling patterns in large repositories of Simulink models by clustering sub-models with a common property that creates a partition of the available model set. Initial text-based parsing and processing of the models clusters their subsystems by size, determined by the number of block and data flows between blocks in a subsystem, into a set of uniquely-identified classes. An interactive subsystem query and visualization software tool facilitates the exploration of selected groups of models and sub-models, using a graph-oriented approach. The iterative use of this latter facility allows analysts to infer more abstract modeling patterns (i.e. semantic patterns) by focusing on specific structures, relationships, and operations (e.g. sub-classing, class consolidation and intersection, etc.) on the sub-model classes.

2.4 Pattern Representation and Reasoning

Based on intrinsic good design goals, the overall implementation architecture of a computerized embedded system can be characterized by the synergistic configuration and operation of four types of elements: 1) multiple hardware devices cooperating with each other in an arbitrated manner, 2) a set of software components executing monitoring, control and supervision algorithms, 3) physical connections between the hardware devices to channel internal signals and interactions with their environment, and 4) logical connections between software modules to support data exchanges and implement inter-module dependencies [65]. Ideally, the embedded system assembled in this configuration meets or exceeds the original specification, and is fit to run under sound testing and
2.4 Pattern Representation and Reasoning

verification techniques. In practice, multiple equivalent configurations could meet the specification, and additional factors, such as form and cost, will determine the implementation of choice.

Practitioners designing embedded systems quantize their continuous physical properties when a symbolic analysis of discrete values offers advantages, use mathematical models when precise reasoning is required in all temporal and spatial situations, and abstract their elements into related and/or compositional modules to reduce the complexity of the analysis on internal/external interactions [7, 27, 73]. Similar to the concept of configurations in a system’s implementation, since multiple equivalent model choices could satisfy established modeling requirements, one can view these arrangements of entities as modeling patterns which specify and represent an embedded system with structures that allow one to reason about them.

This cognitive approach finds broad application in the modeling of complex hybrid systems. For example, Barouni and Moulin [9] use the configurations view to reason about dynamic spatio-temporal patterns in the domain of power utilities management. In their case study, they call a configuration of inter-related elements in a geographically-distributed data acquisition system (i.e., computers, networks, sensors, actuators and programs) a “situation”. Events and processes make these situations evolve in time, in somehow predictable ways, creating an identifiable set of spatio-temporal patterns. Special “agents” analyze instances of these patterns, within specific contexts, and help take corrective actions about the operation of the system.

The design of embedded systems relies on the models and principles governing cyber-physical systems [6] and systemic abstractions such as the plant-control view shown in figure 2.2a. System developers need to express their essential structure and behaviours in modeling views with sufficient and practical degrees of formality, so that they can effectively validate design objectives and integrity constraints [61]. Industrial practice has widely used domain-specific modeling languages (DSML) to create visual models of embedded systems that support these goals, that is, the models are self-documenting, executable, evolutionary and can directly lead to system implementations through code generation. Simulink [56] is the most prominent of these DSMLs in industry, due to its maturity and large library support for various application domains, however, environments
that support UML profiles for real-time systems (e.g., UML-RT in Papyrus-RT [24]), or the AU-
TOSAR modeling standard (e.g., AUTOSAR DSML in Rhapsody [43]) are gaining acceptance. A
great deal of academic research has been directed to studying customizable environments, such as
the Model Integrated Computing (MIC) of Vanderbilt University, in order to define highly specific
DSMLs using meta-modeling techniques [48]. These studies attempt to answer questions about the
effectiveness of DSMLs in handling issues of scalability, system complexity and modeling scope
[45, 61].

In theory, each DSML can support a range of syntactically and visually different model repres-
entations for a common highly-abstract systemic model. In practice, modeling requirements and
objectives, as well as language constraints (e.g., required level of detail, available language’s prim-
itives, etc.) determine the possible range and scope of these variations, that is, the multiplicity of
model representations for a common system design. Clustering these instances into representation
classes for models allows one to: 1) reason about common structural and behavioural properties,
2) infer useful semantic interpretations by denoting arrangements of syntactic elements based on
domain practice, and 3) identify potential patterns of common design principles and objectives.

The approach and toolchain introduced in this dissertation parse and analyze the textual defini-
tions of Simulink models contained in large repositories and proceed to: 1) identify representation
classes of modeling entities expressed in the Simulink language, 2) gather these classes into an envi-
nronment which allows query-based inferences on class groups, and 3) infer more abstract Simulink
modeling patterns and embedded system modeling patterns in general, querying these classes and
applying domain knowledge. The supporting tools to refine classes into modeling patterns are 1)
regular-expression-based queries applied on strings representing properties of model subsystems to
create specific subsets of them and 2) visualizations of classes of subsystems that lead to inferring
common graph-like structures that belong to more general or abstract model-patterns. Users can
interact with the MoSART interface multiple times, and refine the subsystem cluster classes pro-
duced by the spSearch workflow, by setting up controlled subsystem visualizations and reasoning
2.5 Refinement of Model-Patterns

Models of embedded systems combine abstractions of physical variables, discrete functionality modes, subsystem interfaces and software entities to represent their design as sets of hybrid components behaving and interacting with each other according to protocols, algorithms and mathematical models [13]. Mature modeling languages, such as Simulink, provide a comprehensive set of primitives and operations to capture these abstractions in models through 1) direct mappings between language primitives (e.g., functional blocks) to domain elements, 2) specialization of primitives to conform to specific domain entities, and 3) composition of primitives to represent aggregated structures and behaviours [53]. These features help designers build models that contain multiple views of a system, are easier to conceptualize and reduce the overall complexity of the system development efforts. Ideally, the resulting models capture all necessary functional and non-functional requirements of a system [58, 65], represent a modular and maintainable view of the system which is verifiable and potentially reusable [81], and meet stringent standards set for their intended roles in the system development life-cycle (e.g., simulation, testing, or code generation) [14, 47]. This is the essence of the model-driven development approach to building industrial-strength embedded systems.

In this context, the analysis of large repositories of Simulink models can be approached with increasing specialization in the model-pattern-inference process. There could be a wide range of
2.5 Refinement of Model-Patterns

catalysts for focusing on determined types of modeling patterns such as 1) a product line that uses a common set of large abstractions, 2) a model-based test facility that checks compliance with design standards, or 3) a model-in-the-loop simulation prototype that exercises specific control models. Regardless of the size and composition of the repositories of Simulink models, as well as the requirements of the modeling projects involved, a pattern-identification initiative that addresses these objectives will likely strive to infer one or more of the following sets of model-patterns:

1. **Activity-Oriented Model-Patterns.** Designers usually turn to well-understood patterns of abstraction and design approaches to specify and build embedded systems [42]. Simulink models can capture requirements for these designs, document them and also serve as executable specifications of a system [46]. The resulting models contain typical elements that can be associated with activities (e.g., requirements analysis, simulation, testing) in the development life-cycle of a system in a given domain. A pattern recognition technique can initially cluster these modeling elements into classes, using, for example, relevant indicators of size, topology, behaviour and relationships between entities. Experts can further interpret and segregate these classes into semantic model-patterns that represent pieces of higher-level modeling activities in the domain [23, 41].

2. **Variations of a Model-Pattern.** The nature and relevance of a specific activity-oriented model-pattern can justify processing further pattern searches related to it. A modeling project could benefit from describing the pattern, and a set of modeling elements associated to it, in a more formal manner. Experts can determine that specific variations of the pattern apply to useful “what-if” scenarios. Furthermore, they can estimate that these variations have regularities which apply to a significant number of modeling concerns in the domain, so that describing the model-pattern with a grammar would be beneficial for studying known and emergent modeling scenarios using it [12].

3. **Target Model-Pattern.** Models of embedded systems are notorious for the heterogeneity of their component entities. Attempting to describe a set of Simulink model-patterns with a grammar or
other formalism might be impractical, perhaps unfeasible. However, practitioners can identify a draft version of a model-pattern and determine that, after adding and/or deleting specific elements from it, the pattern is applicable to desired modeling scenarios. As a result, they can re-focus their analysis of a repository of Simulink models on capturing the scope of these changes when reusing the pattern in a given set of modeling use cases.

4. **Model-Pattern Representation Language.** The analysis of the elements in a given set of Simulink model-pattern classes may uncover conceptual modeling patterns [35] that are potentially applicable in different use cases and show a sizable set of common abstractions. Under these circumstances, a model-pattern language (either textual or visual) could facilitate the identification, manipulation and reuse of the elements in these classes, similar to how ontological pattern languages describe entities in conceptual modeling [36]. Experts would use the primitives of such a language to describe conceptual modeling patterns (i.e., design patterns) with specific templates of Simulink entities. As a result, they would be able to control the granularity and abstraction level of the model-patterns, as well as the rules for their application. Researchers would be able to describe known and emergent model-patterns in a given domain, which can be expressed with templates of elements in the Simulink language.

The joint work of the spSearch and MoSART tools, in the pattern-discovery framework proposed in this thesis, allows practitioners and researchers to infer and elaborate on Simulink model-patterns in the four categories described above. They automate most of the tasks related to identifying syntactical Simulink patterns (i.e., templates of Simulink elements in a repository) and greatly facilitate the work of experts in conceptualizing high-level modeling and design abstractions expressed in the Simulink language (i.e., semantic model-patterns).
Chapter 3

Model-Pattern Discovery Framework

This chapter presents the environment and all the core building blocks of a framework to discover, formalize and manage for reuse, a set of semantic modeling patterns from one or more repositories of Simulink models for embedded systems. A modeling-pattern discovery methodology implements a set of techniques - supported by pattern search and evaluation algorithms embedded in a tool chain - that are applicable, in particular, to a wide range of domain-oriented repositories of Simulink models, and in general to repositories of block-and-dataflow models. Primarily, this work deals with the processing and model management issues that arise when attempting to discover modeling patterns for the design of embedded systems in industrial-scale repositories, built by large organizations, supporting complex development projects. The framework is also applicable in answering experimental model-pattern questions on public domain repositories, which contain models of small prototype systems, heterogeneous model examples, and incomplete-component designs. Its overall objective is to help practitioners synthesize, visualize and analyze the content of large repositories of Simulink models as classes of syntactical patterns, of determined sizes and topology, which facilitate elaborating on pattern refinements, composition and encapsulation of modular modeling entities, establishing relations between models and subsystems, building “what-if” scenarios of modeling reuse, evaluating the impact of modeling changes, building of test cases, and many other use cases.
3.1 Assumptions

The exact nature of the application domain, the software development process and the organization’s software delivery goals define the characteristics of the models and the modeling patterns that one can expect to find in these usually large repositories built by practitioners. Overall, the following core assumptions, which this work uses as a baseline, effectively help optimize the task of finding modeling patterns for reuse in these repositories:

- The modeling pattern reuse occurs in a product line software development environment [70] or similar, where certain very high-level abstractions (i.e., modeling-language-independent “conceptual design templates”) exist of the embedded artifacts to be produced (i.e., “products”). This consideration ensures that model repositories meet minimum consistency and validity requirements that make the pattern-finding effort relevant. It also serves as a reference to appropriately qualify the relevance and applicability of the patterns when this compliance level is not achieved. Finally, it helps to provide a level of correlation between the quality of the model instances in a repository, and existing software practices.

- The modeling language in use, Simulink in our case, allows the modeling of all universal, compositional, behavioural, temporal and recurrent abstractions required in the design of embedded systems within the given application domain. This assumption helps to envision reasonable frontiers in the pattern-finding effort based on systemic conceptual ideas that are common in the modeling practice of cyber-physical systems. Furthermore, it encourages searching in repositories of Simulink models for syntactical modeling entities that could be part of or contribute to conceptual modeling entities in a given domain, since they must be a representation of one or more of the latter entities.

- The modeling pattern discovery focuses on finding pattern instances in a single application domain or in the intersection of highly-overlapping domains that would be of interest to a software development team. The effort concentrates on discovering design knowledge embedded in the
3.2 Architecture of the Framework

model repositories that entails broad modeling semantic categories such as control, redundancy and safety. This work analyzes models whose implicit end goal is to generate software systems, within a layered software/hardware architecture such as AUTOSAR (refer to figure 2.2b), that safely operate and control embedded systems. Finding domain-oriented modeling patterns takes precedence over identifying generic patterns that would require extensive adaptation and validation for application in a specific domain.

3.2 Architecture of the Framework

The implementation architecture of the framework to discover model-patterns in repositories of Simulink models, which we propose in this thesis, consists of the ensemble of modules, processes and dataflows depicted in Figure 3.1. This framework supports the iterative analysis of Simulink-subsystem classes in a repository, and the identification of targeted model-patterns according to the approach and deliverables proposed in section 1.2.1. As indicated in figure 1.1, it automates the traversal of the repositories, the generation of subsystem-cluster classes and the computation of subsystems’ topological properties. The spSearch software tool, which is a multiple-step workflow implementing the repository traversal and property computation algorithms described in chapter 4, executes all these tasks. The MoSART tool allows practitioners to query, reorganize into derived classes and visualize the elements of the subsystem-cluster classes generated by the spSearch tool. Experts can use this interactive user interface to infer and refine model-patterns into higher-level abstractions (i.e., semantic model-patterns) that are more useful in their modeling practice.

In conceptual terms, the framework processes $M = m_1 + ... + m_n$ textual representations

$\{T_{1,1}, ..., T_{1,m_1}, ..., T_{n,1}, ..., T_{n,m_n}\}$ (refer to Appendix A) of $M$ Simulink models contained in $n$ repositories $\{R_1, ..., R_n\}$, where each $T_{i,j}$ contains $S_{i,j}$ lines of source code and the total for all repositories is $S = \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq m_i} S_{i,j}$. The framework produces as output $\{P_1, ..., P_s\}$ persistent data structures referencing sets of model subsystems (refer to Appendix B) clustered in a set $C =$
\{C_1, ..., C_p\} \text{ of classes, such that } \bigcup_{1 \leq i \leq n} R_i \subseteq \bigcup_{1 \leq j \leq p} C_j. \text{ These } C_j \text{ classes identify modeling patterns, of varying size, complexity and abstraction-level, based on subsystem properties such as similarity, functional recurrence, structural utility, readiness for composition, and many others which result from the automated processing by } \text{spSearch} \text{ and the interactive model-pattern assessments performed by practitioners on MoSART.}

Sections 3.3 and 3.4 describe the components in this architecture, which collaborate to implement the workflow from \textit{models in repositories} to \textit{model-pattern abstractions} encoded for visualization and further processing.
3.3 Automated Model-Pattern Detection

As indicated in section 3.2 and depicted in figure 3.1, the spSearch tool performs the automated stages of the Simulink model-pattern discovery approach proposed in section 1.2.1. This tool implements a workflow of eleven main steps, whose input is a repository $R_{srid}$ of Simulink models stored under a location named $srid$, and whose output is a directory of subsystem classes with additional topological attributes computed for each subsystem. The following is a description of these functional steps:

1. Consolidate and instrument the models in repository $R_{srid}$ into single-storage version $R'_{srid}$.

2. Update the Master Catalogs of Simulink and Stateflow tags and block types, respectively. This step parses $R'_{srid}$ and detects all the possible Simulink tags (e.g., Model, System, Port etc.), Simulink block types (e.g., Subsystem, Mux and Inport) and Stateflow tags (e.g., machine, chart and state) used in the models it contains. Next, it updates the master lists above.

3. Isolate the Simulink model definitions from the Stateflow charts and filter out all the display settings, property configuration, tables and comment source code. This step separates block-and-dataflow definitions from statechart definitions in all the models of $R'_{srid}$. This allows the separate processing of structural aspects in Simulink models.

4. Isolate the Stateflow charts from the Simulink model definitions and filter out all the display settings, property configuration, tables and comment source code. This step allows the separate processing of behavioural aspects in Simulink models.

5. Cluster the subsystems of the models in $R'_{srid}$ into size-based classes. This step also computes topology properties for all the subsystems in $R'_{srid}$ based on the configurations of dataflows, block names and block types that they encapsulate.

6. Parse all the Stateflow charts in $R'_{srid}$ to identify their sequence in each Simulink model and the number of states they contain. These numbers assign a profile to each chart in $R'_{srid}$. 

38
3.3 Automated Model-Pattern Detection

7. Compute Stateflow chart sequence patterns, which consist of sets of charts appearing in the same or closely-equal orders in different models and subsystems, with the numbers of states each one contains forming equal numeric sequences or slightly different permutations of them. These patterns help identify similar behaviours modeled in different areas of a Simulink repository.

8. This step links Simulink block-dataflow patterns to Stateflow chart-size profiles.

9. Determine specific chart-sequence patterns. This step concentrates on further describing a specific Stateflow pattern detected in the industrial repository studied in chapter 6.

10. Export the cluster classes of subsystems to the PEF for visualization and further pattern elaboration with the MoSART interactive tool.

11. Execute the Subsystem Name Usage Analysis (refer to section 3.3.3) to report on the assignments of names to subsystems 1) within the same size-based cluster class for identification of identical or closely similar intended design meanings in multiple occurrences of a subsystem, and 2) across cluster classes for subsystems of different sizes having the same name and possibly a high-level similar design meaning.

spSearch has a command line interface whose syntax is as follows:

```
python spSpearch.py srid <StepsTuple>
```

where the <StepsTuple> modifier is optional, with the following syntax and meaning:

- If <StepsTuple> = empty then spSearch executes all its steps.
- If <StepsTuple> = GT<StartStep> then spSearch executes <StartStep> and all the subsequent steps in the workflow.
- If <StepsTuple> = GT<StartStep> (S₁, ..., Sₖ) then spSearch executes <StartStep> and the steps Sⱼ (j = 1, ..., k) after it.
3.3 Automated Model-Pattern Detection

Sections 3.3.1, 3.3.2, 3.3.3 and 3.3.4 describe the different modules and data structures in the architecture of spSearch shown in figure 3.1, respectively.

3.3.1 Primary Clustering (PClu) Module.

The pattern discovery workflow starts at this module, which traverses in a bottom-up fashion (i.e., from more deeply-nested definitions to the root of each model, refer to section 4.1 and figure 4.1b) the source code definitions of the Simulink models in one or more repositories, searching for model subsystems. When it finds the beginning of a subsystem definition (refer to table A.1 in appendix A), it proceeds to parse the subsystem’s content to compute its size and some topological properties, and then deletes its entire content before resuming the search for the next subsystem. Next, it encodes the size of each subsystem in a hash code with the format $mBnL$ ($m =$ number of blocks, $n =$ number of dataflows). Finally, it clusters the subsystems by size into a set of classes $\{C_1, ..., C_p\}$, where the elements of each class $C_i$ represent subsystems with the same size hash code.

The PClu module also parses all the Stateflow charts embedded in the Simulink models of a repository to detect statechart sequence patterns, using the approach and algorithm described in detail in section 4.2.

![Diagram of Model Subsystem Clustering Workflow]

Figure 3.2: Model Subsystem Clustering Workflow srid
3.3 Automated Model-Pattern Detection

As depicted in figure 3.1, the PClu module sequentially process multiple repositories $\mathcal{R}_{srid}$ of Simulink models, where $srid$ is a unique repository identifier. This module labels all the processing that performs on a single repository with this identifier (e.g., data structures, cluster classes, reports) to facilitate a seamless merge or clear separation of results in the analysis of multiple repositories. Figure 3.2 shows the architecture of processing modules and data structures involved in an execution of this clustering workflow. The following are descriptions of these components:

- **Preprocess Models (PM) Module.** This module searches for Simulink models in a storage area identified by $srid$ and consolidates them into a single persistent data structure of preprocessed models. It annotates each source code line of a model (or library of model components) with the following characterization fields: a type of container model (i.e., single model or library), a model-id which uniquely associates the current line with a model, a location-index of the current line within a model, and a nesting-level number which indicates the depth of containment within a model of the entity that the current line defines (refer to Appendix A). It also generates a set of catalogs unique to repository $\mathcal{R}_{srid}$, which: 1) provide a quick reference index to its models, 2) capture the set of Simulink-model and Stateflow-chart structural delimiters (i.e., syntactical tags) used in its models and libraries, and 3) capture a list of the block types used in its models and libraries.

- **Reference Catalogs.** The *Models Catalog* provides unique numeric identifiers for the models in repository $\mathcal{R}_{srid}$, which make the processing of their source code faster and independent from their address in secondary storage. Simulink and Stateflow *Tag Catalogs* lists all the possible syntactic tags used anywhere in $\mathcal{R}_{srid}$. The *Block Type Catalog* list all the possible block types used anywhere in $\mathcal{R}_{srid}$.

- **Annotated Models Repository.** It contains all the source lines of code, of all models and libraries in $\mathcal{R}_{srid}$, annotated with fields inserted by the PM module.

- **Reduce and Split Models (RSM) Module.** This module filters out all the source lines of code
3.3 Automated Model-Pattern Detection

of models and libraries that define configuration options, data structure options, display properties, data tables, embedded programming code, and other declarative options. It keeps only the structural definitions of data structures, of subsystems within Simulink models, of blocks within subsystems, and of data flow lines between the blocks of subsystems, as well as the names and types of subsystems and blocks. It also keeps the structural definitions of the entities in existing state machines in Stateflow charts embedded in Simulink models: machines, states, transitions, junctions, events and data declarations. Finally, it stores these reduced-definition Simulink models and Stateflow state machines into two separate repositories: Reduced Simulink Repository (RSiR) and Reduced Stateflow Repository (RStR).

- **Reduced Models Repositories.** RSiR and RStR contain the structural-only model source line of code definitions, as filtered by the RSM module, for all the Simulink subsystems and Stateflow charts, respectively, of all models in repository \( R_{srid} \).

- **Cluster and Hash Subsystem Models (CHSM) Module.** This module traverses and parses the source lines of code of a Reduced Simulink Repository, in a bottom-up fashion (refer to section 4.1.1), to determine the size of each subsystem \( s \) in each model \( m \) of repository \( R_{srid} \) as a pair of values \( (B, L) \) \( (B = \text{number of blocks in } s, \ L = \text{number of data flows in } s) \). It also determines the name \( N_c \) and type \( T_c \) connectivity of each subsystem, as defined in section 3.3.2). Thereafter, it computes \( N_c \) and \( T_c \) hash codes for the name- and type-connectivity, respectively, of each subsystem \( S_{i,j} \), using the hash functions described in section 4.3. Finally, it clusters all subsystems in \( R_{srid} \) by size.

- **Model-Subsystem Classes.** Each class represents a clustering of Simulink model-subsystems, in a repository \( R_{srid} \), having the same number of blocks and dataflows, that is, the same size. This clustering produces classes whose subsystems may have different topologies, that is, they may have different name- and type-connectivity hash codes. However, the complete set of classes is a full partition of all the Simulink model-subsystems in \( R_{srid} \), and it is a closed set for operations such cluster-class merge and sub-classing.
3.3 Automated Model-Pattern Detection

- **Search for Statechart Sequence Patterns (SCSP) Module.** This module parses all the Stateflow chart definitions in the *Reduced Stateflow Repository* generated by the RSM module to identify clusters (i.e., patterns) of *sequences of Stateflow charts* having the following common properties: 1) all charts in a sequence belong to the same Simulink model or library, 2) the total number of charts in a sequence is a fixed number $C$, and 3) the sum of all the states in all the charts of a sequence is a fixed number $S$. *Chart Sequence Patterns (CSP)* embody the idea of having a set $C = \{c_1, ..., c_n\}$ of $N$ Stateflow charts, each $c_i$ having $s_i$ ($i = 1, ..., n$) states respectively, which appears in multiple models or libraries of a repository and that very likely represents a core behavioural pattern embedded in the design of multiple model subsystems, perhaps with small or equivalent modifications. The existence of these patterns greatly facilitates the identification of more specific and complex behavioural patterns by finding relations (i.e., dependencies, inclusions, exclusions, etc.) between the charts they contain and other charts in a repository. They establish a productive baseline for the inclusion of topological and logical elements (i.e., state transitions, guards and constraints) in the analysis of Statecharts, that is, of model subsystems, for the identification of more abstract and/or complete behaviours in design patterns. Section 4.2 provides details about the general definition of these patterns and the algorithm to discover them, as well as an exemplar found in the empirical analysis of an industrial repository, as detailed in chapter ch:EmpiricalAnalysis.

3.3.2 Subsystem Topology Hashing (STH) Module.

This module computes hash codes $N_{i,j}$ and $T_{i,j}$, which are strings of hexadecimal digits, from the name-connectivity $N_{i,j}$ and type-connectivity $T_{i,j}$, respectively, that module PClu computes for each subsystem $S_{i,j}$ in a repository $R_{sruid}$ of Simulink models as follows:

1. A list of the association pairs between the block names, that is, its *name connectivity*

   $N_{i,j} = \{(name_{u_{i,j}}, name_{v_{i,j}})| k_{i,j} \}$, $u_{i,j} \neq v_{i,j}$, $\exists$ dataflow $u_{i,j}, v_{i,j}$, $blocks u_{i,j}, v_{i,j} \in S_{i,j}$,
   $k_{i,j} = 1, ..., n_{i,j}$). (Figure 3.3b - List of Name-Connectivity Pairs for ConnectivityExample)
3.3 Automated Model-Pattern Detection

2. A list of the association pairs between the types of each block, that is, its type connectivity

\[ T_{i,j} = \{(type_{u_{i,j}}, type_{v_{i,j}})_{k_{i,j}} \mid u_{i,j} \neq v_{i,j}, \exists \text{ dataflow}_{u_{i,j}, v_{i,j}}, \text{blocks } u_{i,j}, v_{i,j} \in S_{i,j}, \]

\[ k_{i,j} = 1, \ldots, n_{i,j} \}. \]

(Figure 3.3b - List of Type-Connectivity Pairs for ConnectivityExample)

---

Figure 3.3: Subsystem Connectivity Types

Hash codes \( N_{i,j} \) and \( T_{i,j} \) capture a “digital signature” of the topology of functional blocks and dataflows encapsulated by each of the subsystems \( S_{i,j} \). The code associated with a subsystem’s name-connectivity is very sensitive to changes in block names, which makes it a good indicator for identifying cloned subsystems. The code associated with a subsystem’s type-connectivity, although sensitive to block-name differences, is more dependent on the types of functionality encapsulated by the respective subsystem. Both codes are sensitive to the configuration of dataflows existing in a given subsystem. These properties greatly increase the resolution of the reasoning processes to logically regroup subsystems, for further model-pattern inferences, with the querying and model manipulation capabilities supported by the MoSART user interface. Using these codes for detecting subsystem similarity or differencing relies on the following:

- Let subsystems \( S_1, S_2 \in S \) contain unequal block names and data flow schemes between blocks, that is \( N_1 \neq N_2 \), then they will be assigned hash codes \( N_1 \) and \( N_2 \) respectively, such that
3.3 Automated Model-Pattern Detection

\(N_1 \neq N_2\). The cases where \(S_1\) and \(S_2\): 1) contain the same block names but different data flow interconnections, and 2) contain different block names but identical data flow interconnections, will also result in \(N_1 \neq N_2\).

- Let subsystems \(S_1, S_2 \in \mathcal{S}\) contain unequal types and data flow schemes between blocks, that is \(T_1 \neq T_2\), then they will be assigned hash codes \(T_1\) and \(T_2\) respectively, such that \(T_1 \neq T_2\). The cases where \(S_1\) and \(S_2\): 1) contain the same block types but different data flow interconnections, and 2) contain different block types but identical data flow interconnections, will also result in \(T_1 \neq T_2\).

- Let \(S_0 \in \mathcal{S}\) and \(S_{eq} = \{S_i \in \mathcal{S} \mid N_i = N_0, T_i = T_0, i = 1, \ldots, n\}\), then \(C_{S_0} = S_{eq} \cup \{S_0\}\) constitutes a class of topologically equivalent subsystems characterized by \(S_0\).

The cases where \(S_1 \neq S_2\) and, \(N_1 = N_2\) or \(T_1 = T_2\), represent hash code collisions. We would interpret these situations as the occurrence of false negatives in subsystem differencing checks, or false positives in subsystem similarity inferences. However, experimentation with the hashing algorithms for \(\mathcal{N}\) and \(\mathcal{T}\) proposed in this dissertation has shown either non-existent or extremely low incidence of collisions. Furthermore, we can usually diagnose these situations with a correct answer by straightforward contextual analysis of their occurrences. Section 4.3 describes the algorithms of the hash functions that module STH implements to generate these two codes. Both algorithms generate hash codes having a very low incidence of collisions.

3.3.3 Subsystem Name Usage Analysis (SNUA) Module.

In Simulink, models identify blocks (e.g., functional blocks, first-level-nesting subsystems) within each subsystem with unique names, that is, they disallow name duplicity within the same subsystem [56]. However, it is possible to assign the same name to blocks encapsulated by subsystems in different containment branches from the root of a model, or to blocks in different models of a repository \(R_{srid}\). This latter feature opens up the possibility of overloading the semantic usage of
3.3 Automated Model-Pattern Detection

names, as practitioners working on large projects or compartmentalized environments often assign
the same name to subsystems that are topologically and functionally different, or conversely, they
assign different names to subsystems that are identical. Understanding the nature and scope of
these name duplicity situations is very useful because: 1) they can provide an insight into evolutionary patterns in models, 2) they may represent similar modeling approaches for different entities or different modeling approaches for the same entity labeled with a common name, 3) they may represent accidental name clashes for different modeling approaches of different entities that need to be sorted out for modeling consistency, clarity and accuracy purposes.

Module SNUA analyzes these name-assignment situations for all the subsystems $S_{i,j}$ in repository $R_{srid}$ and reports all their occurrences. That is, it detects whether the same subsystem-name has been assigned to subsystems of different sizes in the same model and across the models of one or more repositories. Also, for a given subsystem size, it finds all the occurrences of a given subsystem-name across the models of a repository. Its output consists of a series of reports, organized by cluster class, to assist an analyst in inferring more abstract semantic patterns, involving multiple subsystems, and understanding the impact of name-duplicity scenarios. This module detects the reuse of names across all the cluster classes that module PClu generates. Section 4.4 describes the name-analysis algorithm implemented by module SNUA and provides examples of subsystem name usage scenarios detected in the case study of chapter ch:EmpiricalAnalysis. We implement all this functionality in tool spNames (Simulink Pattern - Names Usage Profile).

spNames takes as input the identifier $mBnL$ ($m =$ number of blocks, $n =$ number of dataflows) of a subsystem-cluster class, generated by spSearch for a repository $R_{srid}$. For each subsystem name $N_{i,j}(n,m)$ corresponding to a subsystem $S_{i,j}(n,m)$ in a class $mBnL$, the tool reports all the subsystems $S_{k,l}(p,q)$ in $R_{srid}$, of any size $pBqL$, which have the same name as $S_{i,j}(n,m)$ (i.e., $\forall (p,q), \forall (k,l)$ such that $N_{i,j}(n,m) = N_{k,l}(p,q)$), including their location in $R_{srid}$, their computed topology hash codes and the size of their source code.

The role of spNames is to complement the core work of spSearch and serve as an analysis/synthesis aid for abstracting semantic patterns with the interactive tool MoSART. Its command
3.3 Automated Model-Pattern Detection

line interface uses the following syntax:

```
python spNames.py srid <PatternCode>
```

where `<PatternCode>` has the following syntax and meaning:

- If `<PatternCode> = mBnL` then `spQuery` analyzes and lists the subsystems in class `mBnL`.
- If `<PatternCode> = all` then `spQuery` performs the single-class analysis for all the classes in \( R_{srid} \).

### 3.3.4 Clustered-Subsystem Directories

The main processing strategies we employ to make our Simulink model-pattern discovery approach scalable, feasibly iterative and with expandable capabilities, are 1) build temporary annotated versions of the original Simulink models’ source code with values that expedite the repository traversals (i.e., nesting, file indexes, source code line indexes), 2) compact all the annotated versions of the Simulink models in a repository \( R_{srid} \) into a single temporary data structure optimized for subsystem pattern identification, and 3) uniquely identify the original models and the location within their source code for each subsystem included in any cluster class generated by the pattern search algorithms. The spSearch workflow tool, upon completion of its automated pattern search steps, creates two directories to link subsystems-in-cluster-classes to source-code-of-subsystems in Simulink models:

- **Process-Classes Directory (PCD)**. Module PClu creates this directory, which is a data structure that captures the following attributes for each subsystem: 1) `size` expressed as number of blocks and dataflows, 2) pointer to an entry in a directory of models’ physical locations, 3) source code start and end indexes, 4) `nesting depth` from the root of the containing model, 5) `nesting path` from the root of the containing model, and 6) name and type topology hash codes. The main role of this directory is to support any further automated processing of the annotated versions of the Simulink models and the cluster-by-size classes of subsystems.
3.4 Model-Pattern Exploration Facility (MPEF)

- **Display-Classes Directory (DCD).** Module ClCo (Class Converter) creates this directory, from the PCD, as a data structure that captures the following attributes for each subsystem: 1) containing *cluster-class identifier* (i.e., code format *mBnL*), 2) *nesting path* from the root of the containing model, 3) physical location of the containing model, 4) name and type topology hash codes, and 5) formatting for display of subsystem attributes on MoSART. This directory supports the interactive operation of the Model-Pattern Exploration Facility (MPEF), implemented in the MoSART tool, to arbitrarily select classes and subsystems, build queries and derived new classes of subsystems.

3.4 Model-Pattern Exploration Facility (MPEF)

The MPEF takes as input a set of subsystem-cluster classes that the spSearch tool (refer to section 3.3) produces as the output of detecting and clustering by size all the model-subsystems in a Simulink repository \( R_{srid} \). The MoSART (Model Search, Analysis, and Reuse Tool) software tool implements all the capabilities of the MPEF centered around a visual interface that allows users to query, analyze and manipulate 1) all the Simulink models, and their subsystems, in a repository \( R_{srid} \), 2) the subsystem-cluster classes computed by module PClu, and 3) any model-patterns obtained directly from the cluster classes or from user-driven inferences using the querying and display capabilities of MoSART’s *Browse* view. Overall, we have developed this tool for the experimental work of this thesis related to verifying and recording the processes we propose to elicit Simulink semantic model-patterns. For this purpose, practitioners can access MoSART’s *Browse* view to select lists of cluster classes and display the computed properties on their elements (i.e., subsystems), on a dynamic grid, for comparisons and model-pattern refinements. Additionally, they can build arbitrary configurations of subsystems on this grid, through queries or manual selections, display their content and configure permanent patterns (i.e., associations) of subsystems. The following sections explain the architecture of MoSART and basic operations of the MPEF.
3.4 Model-Pattern Exploration Facility (MPEF)

3.4.1 Architecture of MoSART

MoSART is a client graphical user interface (GUI) application, written in C++/CLI on the .NET platform, whose expandable architecture consists of a set of tabbed views, each view representing an aspect of Simulink model-pattern analysis which requires a category of related operations, for example, *Pattern Browse*, *Pattern Adaptation*, *Pattern Language* and *Pattern Reuse* (refer to figure 3.5). Creating a new view in MoSART translates into adding, to the set of existing views, a new display container filled with operators (e.g., class querying and reorganization) and pattern manipulation entities (e.g., pattern assembling, refinement, testing, persistence and reuse). All the views must share the set of subsystem-cluster classes that spSearch generates from a repository \( R_{sr{\text{id}}} \) and can interact with each other through intermediate, transient or permanent data structures. However, each view presents to the users its own output. As shown in figure 3.4, all the views operate in a client-server environment where:

1. The Simulink software [56] and the model repositories \( R_{sr{\text{id}}} (\forall sr{\text{id}}) \) represent the server side.

2. The views of MoSART, the subsystem-cluster classes, and MoSART’s facilities to manipulate these classes, constitute the client side.

3. The Matlab-engine software [54] and MoSART’s Matlab-Simulink interface module provide the necessary communication software between the client and server sides.

As depicted in figure 3.4, for the experimental work of this dissertation, we have only developed the *Browse* view and the functionality deemed common to all the views, that is, the interpretation of the classes from spSearch and the communication with the Matlab-Simulink software to display selected models and subsystems. Other views, envisioned as compliant with the general architecture for MoSART, are left to be designed in future research efforts on the approach depicted in figure 1.1, to address topics such as domain specialization and targeted use cases for Simulink model-patterns.
3.4 Model-Pattern Exploration Facility (MPEF)

3.4.2 The Browse View

The main objectives of the Browse view (refer to figure 3.5) are to facilitate knowledge-driven model-pattern inferences and the study of potential ways to formalize and reuse these patterns. This view implements the interactive mechanisms to display and to query the subsystems in any group of classes from the set that the spSearch tool provides as input. It can display the content of any subsystem, from any cluster class, in a repository $R_{sril}$ of Simulink models, and save configured groups of subsystems to display. Users can assemble those groups by manual selection on a dynamic grid that lists references to those subsystems, through executing queries on this grid, or by reloading previously saved groups of subsystems. The Browse view consists of four operational panels, which implement the functionality to manage the subsystem-cluster classes, visualize subsystems, query classes and save queries and arbitrary groups of subsystems:

- A Manage Classes panel that 1) loads into MoSART the set of classes produced by spSearch, 2) allows users to build an arbitrary class subset from the loaded set, 3) implements mechanisms to save and retrieve that class subset, and 4) can trigger user-initiated updates of the dynamic grid with the list of subsystems contained in this subset.
3.4 Model-Pattern Exploration Facility (MPEF)

- A **Dynamic Grid** panel that 1) lists the subsystems in the classes selected on the class management panel, and 2) plays the role of a data set for the querying facility in the Browse view, becomes a display pad for the results of a query, and allows a user to highlight and select arbitrary subsystems for display and inspection. Each row of the grid displays a set computed properties for a subsystem (e.g., Block-Link (BL) class, Name-Connectivity Hash Code, Type-Connectivity Hash Code, Subsystem Path within a model, Model Path on secondary storage).

- A **Display Models** panel that 1) implements a mechanism to trigger the display of user-selected subsystems on the grid (e.g., highlighted grid rows on figure 3.5) as instances of the Matlab-Simulink editor, as shown in figure 3.6, and 2) implements the functionality to trigger the mechanisms that save and retrieve those subsystem selections.

- A **Query Grid** panel that allows users to configure queries for filtering specific rows of the grid according to the logical expression:

\[
(BL\text{-Code} \in \{BL\text{-class}\} \subseteq C_{srid}) \text{ AND }
(RE_{TCHC}(\text{Type-Connectivity-Hash-Code})) \text{ AND }
(RE_{SP}(\text{Subsystem-Path}) \text{ AND/OR } RE_{MP}(\text{Model-Path}))
\]

where 1) **BL-Code** identifies class \(mBnL\) \((m, n = 1, 2, \ldots)\) which is in the selected subset **BL-class** of the set of classes \(C_{srid}\) provided by spSearch for repository \(R_{srid}\), and 2) \(RE_P(\text{Property})\) means “execute regular expression [59] \(RE_P\) on a string representing a value of **Property** and return a TRUE result if a match exists, return FALSE otherwise”. This panel also implements the functionality to trigger the queries and update the grid with the results from their execution. Users can save and retrieve for execution any queries assembled with this panel.

Using the panels of **Browse view**, users can save and retrieve 1) configurations of cluster classes assembled with **Manage Classes**, 2) configurations of subsystems highlighted on the grid with **Display Models**, and 3) class queries created with **Query Grid**.
3.4 Model-Pattern Exploration Facility (MPEF)

Figure 3.5: Browse View of MoSART - Manage, Query and Display Subsystem Classes
3.5 Ancillary Tools

Figure 3.6: Display and Navigation of Selected Subsystems Triggered from the *Browse View*

Figure 3.5 shows a subset of 5 classes selected on the *Manage Classes* panel. A user has highlighted on the grid 4 subsystems from this subset to display their contents (refer to figure 3.6).

3.5 Ancillary Tools

The main tools of our model-pattern discovery framework work directly (i.e., spSearch) or indirectly (i.e., MoSART) on the source code of the Simulink models. They need to process Simulink models whose source code contains valid formats, consistent with the rules of the Simulink mark-up language (refer to appendix A). Additionally, spSearch uses special meta-data structures (e.g., lists
3.5 Ancillary Tools

of keywords and block types) to optimize its operation, which are filled with data extracted from models' source code. Finally, some additional aids such as models’ source code profiling, name usage analysis, class merge and sub-classing operations, and model management, are necessary to effectively work with spSearch and MoSART.

To meet these requirements, we have developed a set of auxiliary tools to ensure the correctness and consistency of the repositories of Simulink models used in the empirical work for this thesis. We also use them to profile repositories, provide utility functions and build data structures that support the operation of spSearch. The following provides a summary of these tools, which we have written in Python language:

- **spProfile** checks the source code definitions of all the Simulink models in a repository for a consistent structure (i.e., valid Model and Stateflow sections), and the existence of specific modeling elements (e.g., Stateflow charts).

- **spNames** reports on subsystem-names usage across a repository of Simulink models. Section 3.3.3 provides a detailed description of this program.

- **selectModels** selects specific versions of subsets of models, in repositories containing multiple versions of the same model and/or spurious models.

- **getLatest** selects the latest version of the models in a repository containing multiple versions of the same model.

- **commDefs** is a library-style module which provides configuration and utility functions for spSearch and the auxiliary tools (e.g., set environment variables, dictionary upload, timing, workflow control, export formats and hash functions).

- **getTags** extracts from all the analyzed repositories the different types of Simulink and Stateflow tags (i.e., metadata keywords) used, and maintains cumulative directories of either type of tags.
3.6 Framework Evaluation

- **getBlockTypes** extracts from all the analyzed repositories values for the different types of Simulink functional block used, and maintains a cumulative directory of block types.

- **getSubNames** finds and reports all the different subsystem names in a clustering class generated by spSearch.

- **spClassMerge** merges two DCD data structures (refer to section 3.3.4) into a single one, using class identifiers of format $mBnL$ as matching keys.

- **spClassSelect** selects a specific subset of subsystem cluster classes from a DCD data structure, using class identifiers of format $mBnL$ as selection keys.

- **spDepth** reports the highest depth of subsystem nesting in each Simulink model of a repository, and selects models where this parameter is under a specified maximum value. For those models, it reports the size of the subsystems and the path to the most deeply nested subsystem.

3.6 Framework Evaluation

The main goal of our pattern discovery framework, toolchain and approach, is that they be powerful aids for practitioners in the elicitation of Simulink model-pattern patterns. For that purpose, it strives to implement algorithms and processes that are repository-agnostic, summarize enormous amounts of information about models into indicators of similarity that are easy to visualize, and efficiently implements fast iterations of pattern refinement tasks. The following sections provide more details about these concepts.

3.6.1 Adaptability to Simulink Repositories

In designing the algorithms and tools of our framework, we have assigned a very high priority to abstractions that make them perform well regardless of the size and composition of repositories. We visualize the repositories as forests of trees, the subsystems as nodes of these trees, and
the subsystem-containment relations as their branches. We consider that the maximum number of subsystem-nesting levels $L$ is bound by the expression $c \lg n$ ($c = \text{constant}, c \geq 1; n = \text{total number of SLOCs in the repository}$) since, in practice, $L \ll \lg n$. We traverse these forests, in a bottom-up fashion from the most deeply nested subsystems to the root-subsystem of each Simulink model, with an algorithm whose time-complexity is $O(n \lg n)$. We perform local and context sensitive parsing of each subsystem found in the traversal’s trajectory. Although the targeted modeling language is Simulink, the architecture of these tools is geared to efficiently process block-and-dataflow models having a hierarchical structure of nested components. By the same token, the following characteristics of our framework’s architecture strive to take advantage of our toolchain and make it portable to the analysis of any repositories of Simulink models:

1. Separation of initial pattern detection (i.e., clustering) from more abstract semantic-pattern elicitation.

2. Initial pattern detection 1) consolidates, annotates and filters the Simulink models into a single data structure for faster processing, 2) performs multiple search-space reduction modules that maintain links to the original Simulink models, 3) executes modules that compute subsystem properties on reduced data sets, 4) generates a set of indexed cluster classes that is independent of block types, embedded code, tables, etc., and allows users to perform closed class operations (e.g., class merge, sub-classing).

3. Semantic-pattern elicitation is interactive and supported by a powerful cluster-class querying system that works on strings describing generic subsystem properties such as size, type- and name-connectivity. Extensive cluster-class management facilities allow experts to relate systems, based on similarity, containment, frequency, size, and other block-type-independent properties, to make more inclusive refinements and decisions about model-patterns at any level of abstraction.
3.6 Framework Evaluation

3.6.2 Support for Empirical Analysis

We believe that the major advantage of our toolchain’s architecture lies in the separation of pattern detection and pattern elicitation. It acknowledges the fact that we can gradually increase the scope and automation of the former task, if this action produces increasingly useful data structures and information about Simulink patterns. At the same time, it allows practitioners to exercise expert knowledge on the latter task, necessary to make decisions about patterns and infer more abstract ones, which requires cognition abilities that, at the present time, are very hard to replace with software. In agreement with these considerations, our framework’s architecture has many features that help enhance the empirical analysis of Simulink repositories. Some of the most powerful ones are:

1. Model-Pattern Detection

   (a) Processing by spSearch, of any number of repositories, results in a super-set of classes that is closed under operations such as sub-classing and class merge (e.g., spClassSelect and spClassMerge utility programs in section 3.5). This feature allows analysts to reconfigure the cluster classes into groups of any size, containing classes and subsystems from any of the repositories already processed with spSearch, without having to reprocess them.

   (b) The subsystem-clustering algorithm in spSearch (refer to algorithm 4.2) can compute any number of additional properties on subsystems. So far, it evaluates the size, name- and type-connectivity of subsystems. Although the clustering is always by size, the computation of other properties can help refine the understanding of subsystems and inferred patterns when pursuing more conceptual pattern abstractions and classifications.

2. Model-Pattern Elicitation

   (a) MoSART’s query system supports a large variety of closed operations on cluster classes (i.e., the results are cluster classes, refer to section 5.2) based on subsystem (e.g., class,
3.6 Framework Evaluation

size, name, name- and type-connectivity) and model (e.g. location, name) properties. Prac-
titioners can save and recall, through queries, any association of subsystems that support the
elicitation of model-patterns. They can relate any number of queries, and apply their do-
main expertise, to build powerful inferences about complex model-patterns that may include
concurrency concepts, conditional execution, and others.

(b) MoSART’s dynamic grid allows drill-down operations, on the cluster classes and their sub-
systems, whose results are displayed on the grid itself. These results can be queried in the
same manner as the original content of the grid. Users can clear and refresh the grid at any
time, load it with any desired cluster class content, including manually assembled sets of
classes, and display the content of their subsystems. Using the drill-down, they can reason
about model-patterns using sequences of queries that produce increasingly focused results.
Chapter 4

Model-Pattern Discovery and Topology
Assessment Algorithms

Searching for model-patterns in large repositories requires low-complexity algorithms which can examine the Simulink models on multiple projections. The block-and-dataflow nature of Simulink models steers the analysis in the direction of graphs, graph topology, data flow types, node types, node attributes, etc. Specific properties such as the block name, the size of a composite block measured as the number of encapsulated items, the specific connectivity of encapsulated items in a subsystem captured in hash codes as unique signatures, and the size of a statechart expressed as its state cardinality, provide incremental, yet powerful and tractable indicators of structural and/or behavioural similarity that can drive the clustering of sections of models (e.g., subsystems) into property-defined classes. Furthermore, the hierarchical statechart-based approach used in these models to express component behaviour encourages searching for patterns of sets of statecharts with identical or closely similar structure.

This thesis introduces a series of algorithms designed to address these challenges in identifying Simulink model-patterns. The framework depicted in figures 1.1 and 3.1 uses them to automate a large portion of its operations in the analysis of a given repository $R_{srid}$. The spSearch tool
4.1 Detection of Model Subsystem Classes

implements them to perform the following tasks:

1. Search for model-patterns in repositories, based on the hierarchical structure of subsystems existing in every Simulink model.

2. Assess key topological properties of model-subsystems and their encoding into hash codes, to endow those subsystems with descriptive signatures that facilitate further pattern inferences.

3. Analyze the usage of subsystem names across a repository, to highlight relations that are useful in the inference of more abstract patterns.

4. Analyze statechart patterns existing in Stateflow charts contained in Simulink models, to identify modal behaviours that are common among the models of a repository.

4.1 Detection of Model Subsystem Classes

Subsystems are the mechanism in Simulink to encapsulate system functionality, group common modeling elements, establish hierarchical dependencies, capture optional execution of the subsystems of a system, and in general, create associations of elements that facilitate the Simulink modeling practice through divide-and-conquer, composition, reuse, and many other operations. As such, subsystems reflect most of the architectural and design efforts placed in a modeling project to build, for example, an embedded system. It follows that searching for similar subsystems in a model repository represents a very productive approach to identifying modeling patterns. Clustering subsystems by the number blocks and dataflows they encapsulate, ignoring block types and properties, into cluster classes uniquely identified by this measurement of size, strikes a good initial similarity balance. This direct class system greatly facilitates the identification of model subsystems in iterative class refinements and inductive inferences of Simulink model-patterns. The approach is tractable for very large model repositories, as this type of clustering can be attained with low time and space complexity algorithms. This feature makes the computation of other subsystem properties an approachable task, so that experts can have more indicators to identify and refine model-patterns.
4.1 Detection of Model Subsystem Classes

4.1.1 Repository Traversal and Subsystems Clustering

The architecture of a Simulink model can be visualized as a tree of subsystem nodes connected by a containment relationship. The root node of this tree encapsulates the entire model, which consists of zero or more subsystems and functional blocks interconnected by a scheme of typed data flows. In turn, each subsystem might encapsulate a portion of the model with a configuration of the same nature as the one described for the root node (refer to figure 4.1a). In a tree traversal away from the root, each level of the tree represents an increased depth of subsystem nesting. Extending this view to a repository \( R_{sr} \) of Simulink models, we can refer to it as a forest of models, which we can traverse in a bottom-up fashion, if we examine the more deeply-nested nodes first and move towards the root of each model, or in a top-down fashion, if we examine the root of each model first and move towards the more deeply-nested nodes. We implement this algorithm in the spSearch tool, which traverses a consolidated and annotated version \( R'_{sr} \) of a repository \( R_{sr} \), in a bottom-up fashion, and parses all its model-subsystems.
4.1 Detection of Model Subsystem Classes

classSet clusterSubsystems(\(R_{sr}d\), maxDepth)

// \(R_{sr}d\): Repository of Simulink models
// \(R'_{sr}d\): Temporary repository of Simulink models
// \(C_{sr}d\): Set of size-based classes with connectivity hash codes
// maxDepth: Deepest nesting of a subsystem in \(R_{sr}d\) (int)
// getDepth(subsystem): Nesting of the current subsystem (int)
// \((B, L)\): Size of the current subsystem (int, int)
// \(B\): Number of subsystems and functional blocks in a subsystem
// \(L\): Number of data flows in a subsystem
// CIrepository(\(R_{sr}d\)): Consolidate and Instrument Repository \(R'_{sr}d\)
// NCHC(subsystem): Name connectivity hash code for the current subsystem
// TCHC(subsystem): Type connectivity hash code for the current subsystem
// subsystemList: Storage for the computed properties of each subsystem in \(R_{sr}d\)
// getSubsystemSize(subsystem): Computes the size of the current subsystem
// emptySubsystem(subsystem): Deletes the content of the current subsystem
// getNameConnHC(subsystem): Computes the name connectivity hash code
// getTypeConnHC(subsystem): Computes the type connectivity hash code
// getSizeClasses(subsystemList): Clusters the subsystems in \(R_{sr}d\) by size

1: \(R'_{sr}d = CIrepository(R_{sr}d)\)
2: for depth = maxDepth step -1 until 0 :
3:   for each subsystem in \(R'_{sr}d\)
4:     if getDepth(subsystem) = depth then :
5:       // Parse subsystem to compute the following properties:
6:       \((B, L) = getSubsystemSize(subsystem)\)
7:       NCHC(subsystem) = getNameConnHC(subsystem)
8:       TCHC(subsystem) = getTypeConnHC(subsystem)
9:       // (*) Place holder for computing additional subsystem properties
10:      emptySubsystem(subsystem)
11:      append \((B, L), NCHC(subsystem), TCHC(subsystem)\)
12:      to subsystemList
13:    end if
14: end for
15: end for
16: return \(C_{sr}d = getSizeClasses(subsystemList)\)
end clusterSubsystems

Algorithm 4.2: Detection of Size-Based Classes of Subsystems.

The tool also computes a maxDepth parameter as an integer that represents the maximum level of nesting detected for any block, dataflow or subsystem in \(R'_{sr}d\). The root of each model is
assigned a nesting depth of 0, all other subsystem definitions are nested within another subsystem or the root subsystem at an increasing depth of $2k + 1$ ($k = 0, \ldots, n$), where $\text{maxDepth} = 2n + 1$.

Algorithm 4.2 shows a high-level description of the steps, implemented in module PClu (refer to figure 3.1), to traverse and parse all the models in a repository $R_{srid}$ in a bottom-up fashion, as indicated in figure 4.1b. For each subsystem found, in each model, the algorithm evaluates its size as the number of subsystems, functional blocks and data flows it encapsulates.

\[
\text{classSet clusterSubsystems}(R_{srid}, \text{maxDepth})
\]

\[
\begin{align*}
1: & \quad R'_{srid} = \text{CRepository}(R_{srid}) \\
2: & \quad \text{procStep} = 0 \quad // \text{Initially an even processing step} \\
3: & \quad \text{evenRep} = R'_{srid} \quad // \text{Simulink repository for even processing steps} \\
4: & \quad \text{for depth} = \text{maxDepth} \text{ step -1 until 0 :} \\
5: & \quad \quad \text{if mod(depth, 2) \neq 0} \quad // \text{Odd depth level} \\
6: & \quad \quad \quad \text{procStep} = \text{procStep} + 1 \\
7: & \quad \quad \text{if mod(processingStep, 2) \neq 0} \quad // \text{Odd processing step} \\
8: & \quad \quad \quad \quad \text{evenRep} = \Phi \quad // \text{Empty even repository} \\
9: & \quad \quad \quad \quad (\text{subsysProps, evenRep}) = \text{processCurrentDepth(oddRep, depth)} \\
10: & \quad \quad \quad \text{else} \quad // \text{Even processing step} \\
11: & \quad \quad \quad \quad \text{oddRep} = \Phi \quad // \text{Empty odd repository} \\
12: & \quad \quad \quad \quad // \text{subsysProps captures the following properties for each subsystem:} \\
13: & \quad \quad \quad \quad // (B, L): \text{subsystem size} \\
14: & \quad \quad \quad \quad // \text{NCHC(subsystem) and TCHC(subsystem): connectivity hash codes} \\
15: & \quad \quad \quad \quad (\text{subsysProps, oddRep}) = \text{processCurrentDepth(evenRep, depth)} \\
16: & \quad \quad \text{end if} \\
17: & \quad \text{append} \\
18: & \quad \quad \text{subsysProps} \\
19: & \quad \quad \quad \text{to subsystemList} \\
20: & \quad \text{end if} \\
21: & \quad \text{end if} \\
22: & \quad \text{end for} \\
23: & \quad \text{Cluster subsystems by size and return the set of classes} \\
24: & \quad \text{return } C_{srid} = \text{getSizeClasses(subsystemList)} \\
\end{align*}
\]

Algorithm 4.3: Repository Traversal and Subsystem Clustering.

Using this information, the algorithm clusters all the subsystems by size into a set of classes $C_{srid}$ which is a full partition of $R_{srid}$. That is, each subsystem and each of its encapsulating
4.1 Detection of Model Subsystem Classes

subsystems in $R_{srld}$, up to the root of each Simulink model, belongs to one and only one class in $C_{srld}$. Algorithms 4.3 and 4.4 provide expanded views of the main steps in algorithm 4.2.

currentResults processCurrentDepth(Repository, Depth)
// Parse source code of Repository which defines subsystems at a given Depth
// (scLine $\in$ Repository) = $\bigcup$ Tokens $\bigcup$ IndexFields $\bigcup$ EndOfLine
// Tokens = Metacharacters | Tags | Types | QuotedStrings
1: subsystemFound = False
2: for each scLine $\in$ Repository
3: if not subsystemFound
4: if Tag = System and Depth(scLine) = Depth
5: subsystemFound = True
6: else
7: append scLine to newRepository
8: end if
9: else
10: Parse scLine $\in$ subsystem for Depth(scLine)
11: Parse (scLine) - IndexFields $\in$ subsystem
12: Build Incremental Size(subsystem)
13: Build Incremental Name Connectivity Pairs
14: Build Incremental Type Connectivity Pairs
15: if Depth(scLine) = Depth
16: // Compute a subsystem’s Name Connectivity Hash Code
17: Compute NCHC(subsystem) = NCHC(Name Connectivity Pairs)
18: // Compute a subsystem’s Type Connectivity Hash Code
19: Compute NCHC(subsystem) = TCHC(Type Connectivity Pairs)
20: Append Size(subsystem), NCHC(subsystem), NCHC(subsystem)
21: to subsysProps
22: subsystemFound = False
23: end if
24: end if
25: end for
26: return $C_{srld} = $getSizeClasses(subsystemList)
end processCurrentDepth

Algorithm 4.4: Processing of Subsystems at Specified Depth.

As shown in algorithm 4.3, a temporary repository $R'_{srld}$ is initialized to an annotated version
4.1 Detection of Model Subsystem Classes

of the original repository $R_{srkd}$ under study. Each source line of code in $R_{srkd}'$ is annotated with the nesting depth of the subsystem, block, dataflow, Stateflow chart, etc., it represents. The subsystem clustering algorithm traverses $\text{maxDepth}/2$ times a shrinking repository $R_{srkd}'$. Algorithm 4.2 (i.e., $\text{clusterSubsystems}$) traverses and processes the subsystems in repository $R_{srkd}'$, according to algorithm 4.4 (i.e., $\text{processCurrentDepth}$), only for odd depth values, which corresponds to nesting levels of subsystem definitions. During a traversal pass, $\text{processCurrentDepth}$ scans all the subsystem definitions in $R_{srkd}'$, however, it only parses the content of those whose nesting depth is equal to the current value of the depth variable. Immediately after parsing a subsystem and computing its topological properties, it proceeds to empty its content from $R_{srkd}'$. Algorithm 4.2 processes a repository $R_{srkd}$ in a bottom-up fashion, moving from the most deeply-nested entity definitions to the root of each Simulink model.

From an analysis of the execution of algorithm 4.2, we can establish that for a repository $R_{srkd}$, represented by $n$ SLOCs, the algorithm has a worst-case time-complexity $O(n \times \text{maxDepth})$. Since we consider that $\text{maxDepth} \ll (c \lg n)$ ($c =$ constant, $c \geq 1$; refer to section 3.6.1), the average complexity of this algorithm is $O(n \lg n)$, for a repository $R_{srkd}$ of Simulink models that we can visualize as a forest of multiway trees (figure 4.1b).

Algorithm 4.4 (i.e., $\text{processCurrentDepth}$) computes and encodes the size of each model subsystem in $R_{srkd}'$, so that it can be placed in the correct size-driven cluster class. It also computes the lists of name-connectivity and type-connectivity pairs for each subsystem, and a hash code for each of these two properties according to the hash functions described in sections 4.3.1 and 4.3.2.

As indicated in algorithm 4.2, line 9, we can embed additional functions in this algorithm in order to compute any more properties of a subsystem at this point. A justification for the computation of additional pieces of information about subsystems could be a desire for increased levels of refinement, specialization or abstraction in the identification of Simulink semantic model-patterns. To this point, subsystem size, name connectivity and type connectivity support powerful features in our model-pattern discovery framework, namely, 1) full indexing of the subsystems in a repository for sub-classing, querying and cross-referencing operations that may involve large groups of
subsystems, 2) signatures (i.e., hash codes) that support reasoning iteratively, in an integrated tool environment such as MoSART, about component similarity or dissimilarity, hierarchical dependencies, compositional relationships, pattern inference and pattern refinement.

4.2 Statechart-Sequence Classes

Stateflow charts (i.e. statecharts [37]) are central in expressing modes of behaviour of a system modeled in the Simulink language. Their conceptualization in the Simulink language is as functional blocks that can assume one of a set of defined states during their execution within a model. Modelers usually embed them in subsystems to encapsulate modes of operation for an entire Simulink model or for a large abstraction within it. As such, we may expect that models containing similar subsystems in size, topology and modeled functionality, could contain similar-size Stateflow charts up to topology, that is, charts with similar numbers of states and transitions. Even more, these charts will most likely have the same number of states and slightly different transitions, guards, etc.

Statechart-set classes capture the potential existence of these patterns of similar behaviour, based on sequences of Stateflow charts with similar state diagrams, embedded in separate Simulink models.

Based on these observations, let us consider Simulink models $M_a$ and $M_b$, containing sets of Stateflow charts $C_a = \{C_{a_1}, \ldots, C_{a_n}\}$ and $C_b = \{C_{b_1}, \ldots, C_{b_n}\}$, respectively. Let $S_{x_j}$ be the $j$ element of a tuple with $n$ coordinates, which represents the number of states in a chart $j$ of a Simulink model $M_x$. We say that tuples $(S_{a_1}, \ldots, S_{a_n})$ and $(S_{b_1}, \ldots, S_{b_n})$ are instances of the same Statechart-set pattern, that is, they are in the same Statechart-set class, if $S_{a_j} = S_{b_j}$ for $j = 1, \ldots, n$.

We label a Statechart-set class whose elements are tuples $(S_{x_1}, \ldots, S_{x_n})$, where $p = \sum_{1 \leq j \leq n} S_{x_j}$, with the identifier pair $(nC,pS)$.

Algorithm 4.5 (i.e., getChartSequencePatterns) computes the tuples of Stateflow chart sizes for all the models in a repository $R_{srid}$. Three operations on the computed tuples allow the identification of similar patterns of behaviour across the models of a repository, namely 1) clustering
4.2 Statechart-Sequence Classes

them by the number of coordinates (i.e., number of charts) and number of states stored in each co-
ordinate, 2) identifying equivalent permutations of the elements in tuples with the same number of
coordinates, and 3) identifying groups of tuples whose elements are ordered sub-sequences of the
elements in larger tuples.

patternSet getChartSequencePatterns(\mathcal{R}_{srid})
// \mathcal{R}_{srid} : Repository of Simulink models
// S : Temporary storage of Stateflow charts
// \mathcal{P} : Stateflow size patterns for \mathcal{R}_{srid}
// charts(C, S) : Stateflow chart/state size pattern for a model in \mathcal{R}_{srid}
// C : Number of Stateflow charts in a model
// S : Number of states in a model
// states(S_1, ..., S_m) : Distribution of the number of states in a chart set pattern
// S_{chart} : Number of states in chart
// SP : Stateflow (size, sequence) pair patterns for a model in \mathcal{R}_{srid}
// filterStateflow(model) : Isolates Stateflow charts in a model
// getStates(chart) : Parses a chart to compute its number of states
1: \mathcal{P} = \emptyset
2: for each model in \mathcal{R}_{srid} :
3: \quad S = filterStateflow(model)
4: \quad C = 0; S = 0; states() = ()
5: \quad for one StateflowMachine in S :
6: \quad \quad for each chart in StateflowMachine :
7: \quad \quad \quad S_{chart} = getStates(chart)
8: \quad \quad \quad addCoordinateValue S_{chart} to states(...)
9: \quad \quad \quad C = C + 1; S = S + S_{chart}
10: \quad \quad end for
11: \quad end for
12: \quad SP = (charts(C, S), states(S_1, ..., S_m))
13: \quad \mathcal{P} = \mathcal{P} \cup \{SP\}
14: end for
15: return \mathcal{P}
end getChartSequencePatterns

Algorithm 4.5: Detection of Stateflow Chart Size Patterns.

The three operations above can potentially lead to the identification of sets of Stateflow charts
likely implementing more general and abstract behaviours. In the scope of our pattern discovery
4.2 Statechart-Sequence Classes

framework, an analyst can capture, abstract and refine Simulink model-patterns involving sets of Stateflow charts using the iterative approach depicted in figure 1.1, in conjunction with the spSearch and MoSART tools as shown in figure 3.1.

As a proof of concept, using algorithm 4.5, `getChartSequencePatterns`, we have identified a \((26C,56S)\) pattern, whose sequence Stateflow charts and numbers of states can be described by the tuple \((1, 2, 2, 2, 2, 2, 3, 6, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)\). The pattern occurs in 57 different Simulink models of the industrial repository studied in chapter 6. Figures 4.6, 4.7, 4.8, 4.9 and 4.10a show the Stateflow charts involved in the pattern. Figure 4.10b depicts the configuration of states for the most common Stateflow chart type in the pattern, an Event Handler.

Figure 4.6: 26C56S Pattern - Engine State Chart
4.2 Statechart-Sequence Classes

Figure 4.7: 26C56S Pattern - Process Events Chart

Figure 4.8: 26C56S Pattern - Event Scheduling Chart

Figure 4.9: 26C56S Pattern - Signal Zone Determination Chart
4.3 Subsystem-Connectivity Hash Functions

Clustering Simulink-model subsystems of equal size into clearly-labelled classes, evaluating statistical profiles for these cluster classes, and performing an analysis of name assignments to their elements, are pivotal aids in building solid inferences to refine classes of subsystems into semantic model-patterns, or to consider formalizations of these patterns. The computation of additional relevant properties about subsystems translates into the capability to carry out more accurate and sophisticated pattern refinements. Applying this paradigm, we compute two additional topological properties, for each subsystem in a repository $R_{srid}$, that can play a central role in visualizing Simulink model-patterns, namely, its name connectivity $N$ and its type connectivity $T$ (refer to section 3.3.1). We express these properties as lists of name and type ordered-pairs, as indicated for the subsystem exemplar of figure 3.3.

To facilitate the comparative analysis of these properties on MoSART, for arbitrary groups of subsystems, we design and apply a specific hash function for each type of connectivity. These functions generate hash codes that are compact hexadecimal signatures of these connectivity properties for each subsystem, which are also available for querying and clustering using operations on strings. Formally, given a repository $R_{srid}$ of $M_i$ ($i = 1, ..., n$) Simulink models, the set $S_{srid}$ of all subsystems in $R_{srid}$, the name connectivity $N_{ij}$ and type connectivity $T_{ij}$ lists for each $S_{ij} \in S_{srid}$, and $\mathbb{H} = [1-9, A-F]^+$ (the set of strings of hexadecimal digits, whose length is greater than 0), then the
4.3 Subsystem-Connectivity Hash Functions

following definitions apply:

1. \( N_{srid} = \{N_{ij}, \forall i, j\} \) is the name connectivity for repository \( R_{srid} \).

2. \( T_{srid} = \{T_{ij}, \forall i, j\} \) is the type connectivity for repository \( R_{srid} \).

3. \( \text{NCHC} : N_{srid} \rightarrow \mathbb{H} \) is the name connectivity hash code function.

4. \( \text{TCHC} : T_{srid} \rightarrow \mathbb{H} \) is the type connectivity hash code function.

The following sections describe the algorithms to compute the specific hash code in each of these hash functions.

4.3.1 Name-Connectivity Hash Code (NCHC) Function

Function \( \text{NCHC} \) computes a hash code, which is an integer expressed in hexadecimal notation, to identify in a compact format the defined configuration of block names for each subsystem \( S_{i,j} \) in a repository \( R_{srid} \) of Simulink models. Since function \( \text{NCHC} \) applies to a list of connectivity pairs, each one indicating the existence of a dataflow between two blocks whose names are in the pair, the generated hash code also captures a compact view of the topology associated to the diagram of a subsystem. Furthermore, two subsystems with identical sizes, distribution of block names and configuration of dataflows between blocks, get the same name-connectivity hash code, indicating that they have identical topologies up to block names. Ideally, except for very rare collisions produced by the generation algorithm, two subsystems differing in any of these characteristics get assigned different hash codes. This property allows analysts to very accurately discriminate or cluster subsystems for classification and pattern detection purposes. The following are the general steps for computing a name-connectivity hash code for a single subsystem \( S_{i,j} \) in a repository \( R_{srid} \):

- Compute the name-connectivity \( \hat{N}_{ij} \) for \( S_{ij} \), as indicated in section 3.3.1, to obtain list \( L_{ij} \) of \((bname_1, bname_2)\) pairs \((bname_i = \text{block name}, i = 1, 2)\).
4.3 Subsystem-Connectivity Hash Functions

- Apply algorithm 4.11 (i.e., function NCHC) on list \( L_{ij} \) to obtain a name-connectivity hash code for \( S_{ij} \). This function hashes list \( L_{ij} \) according to the following steps:

1. For each pair \((bname_{1k}, bname_{2k})\) in \( L_{ij} \), convert strings \( bname_{1k} \) and \( bname_{2k} \) to integers \( I_{1k} \) and \( I_{2k} \), respectively. Each integer is equal to the sum of the ASCII code values of the characters in the respective name.

2. For each pair \((bname_{1k}, bname_{2k})\) in \( L_{ij} \), compute \( I_{k}(\text{base}10) = v_1 * I_{1k} + v_2 * I_{2k} \), where \( v_1, v_2 \) are repository-specific heuristic integer values (e.g., \( v_1 = 1 \), \( v_2 = 2^3 \)).

3. Compute \( I_{i,j}(\text{base}10) = \sum_k I_{k}(\text{base}10) \).

4. Convert \( I_{i,j}(\text{base}10) \) to \( I_{i,j}(\text{base}16) \) and assign it to return it to \( S_{i,j} \) as its NCHC.

HexString NCHC(namePairsList)

// Compute a name connectivity hash code HexNCHC as a hexadecimal string
// for all blockName1-dataflow-blockName2 pairs in namePairsList
1: DecNCHC = 0
2: for each namePair \( \in \) namePairsList
3:     blockCode1 = 0
4:     for each character \( \in \) blockName1
5:         // Obtain character’s ASCII code and add it to blockCode1
6:         Add ASCII(character) to blockCode1
7:     end for
8:     blockCode2 = 0
9:     for each character \( \in \) blockName2
10:    // Obtain character’s ASCII code and add it to blockCode2
11:    Add ASCII(character) to blockCode2
12: end for
13: // Compute a biased code, to represent a directed dataflow for the pair,
14: // and add it to decNCHC
15: pairCode = blockCode1 + blockCode2 * 2^N \quad // e.g., N = 3
16: Add pairCode to DecNCHC
17: end for
18: // Convert code to a hexadecimal string
19: return HexNCHC = convertToHex(DecNCHC)
end NCHC

Algorithm 4.11: Computation of a Name Connectivity Hash Code
4.3 Subsystem-Connectivity Hash Functions

As an example, if we apply function NCHC on the name-connectivity list generated in figure 3.3b, the returned hash code is the hexadecimal string 22B0. In general, this function generates hash codes that are highly sensitive to the lexical form of names (i.e., type and ordering of characters in names), capture the direction of dataflows by applying different weights to the names in a pair, and advantageously use the uniqueness of names for the blocks in subsystems. These properties make the probability of collisions (i.e., two or more different subsystems getting the same hash code) extremely low. In other words, the probability of getting false positives, when using this hash code to identify similar subsystems, is negligible.

4.3.2 Type-Connectivity Hash Code (TCHC) Function

Function TCHC computes a hash code, which is an integer expressed in hexadecimal notation, to obtain a template of the configuration of block types for each subsystem $S_{i,j}$ of a repository $\mathcal{R}_{srid}$ of Simulink models. It takes as input the a list of connectivity pairs $L_{i,j}$ which has the same structure as the one used by function NCHC, except for the names in each pair being replaced by the types of the blocks involved. Two subsystems $S_{i,j}$ and $S_{p,q}$ with identical sizes and different name-connectivity properties, that is, $N_{ij} \neq N_{pq}$, can get identical type-connectivity hash codes assigned if $T_{ij} = T_{pq}$ (i.e., identical type-connectivity). This situation captures the idea of abstracting the similarity of subsystems to the level of similar typed graphs, where nodes represent functional blocks and edges represent dataflows. This type of reasoning can facilitate the inferring of more abstract classes of subsystems, as the analysis focus on typical behaviours of functional blocks and interfaces (i.e., dataflows) between blocks. The following are the main steps to compute a type-connectivity hash code for a single subsystem $S_{i,j}$ in a repository $\mathcal{R}_{srid}$:

- Compute the type-connectivity $T_{ij}$ for $S_{ij}$, as indicated in section 3.3.1, to obtain list $L_{ij}$ of $(btype_1, btype_2)$ pairs ($btype_i = \text{block type, } i = 1, 2$).

- Apply algorithm 4.12 (i.e., function TCHC) on list $L_{ij}$ to obtain a type-connectivity hash code for $S_{ij}$. This function hashes list $L_{ij}$ according to the following steps:
4.3 Subsystem-Connectivity Hash Functions

1. For each pair \((btype_{1k}, btype_{2k})\) in \(L_{ij}\), replace \(btype_{1k}\) and \(btype_{2k}\) by integers \(J_{1k}\) and \(J_{1k}\), respectively. These integers are indexes to the values of \(btype_{1}\) and \(btype_{2}\), respectively, in a sorted dictionary of all possible block-type values existing in any of the models in repository \(R_{srid}\).

2. For each pair \((btype_{1k}, btype_{2k})\) in \(L_{ij}\), compute \(J_{k(base10)} = w_1 * J_{1k} + w_2 * J_{2k}\), where \(w_1, w_2\) are repository-based heuristic integer values (e.g., \(w_1 = 2^0, w_2 = 2^8\)).

3. Compute \(J_{i,j}(base10) = \sum_k J_{k(base10)}\).

4. Convert \(J_{i,j}(base10)\) to \(J_{i,j}(base16)\) and return it as the type-connectivity hash code for subsystem \(S_{i,j}\).

HexString TCHC(typePairsList)

// Compute a type connectivity hash code HexTCHC as a hexadecimal string
// for blockType1-dataflow-blockType2 pairs in typePairsList
// typesDictionary assigns a unique integer to each block type
1: Import typesDictionary
2: DecTCHC = 0
3: for each typePair ∈ typePairsList
4: Find in typesDictionary codes for blockType1 and blockType2
5: blockCode1 = lookup(typesDictionary, blockType1)
6: blockCode2 = lookup(typesDictionary, blockType2)
7: // Compute a biased code, to represent a directed dataflow for the pair,
8: // and add it to DecTCHC
9: pairCode = blockCode1 + blockCode2 * 2^N // e.g., N = 8
10: Add pairCode to DecTCHC
11: end for
12: // Convert code to a hexadecimal string
13: return HexTCHC = convertToHex(DecTCHC)
end TCHC

Algorithm 4.12: Computation of a Type Connectivity Hash Code

If we apply function \(TCHC\) on the type-connectivity list generated in figure 3.3b, the returned hash code is the hexadecimal string \(F3DB\). Since types can repeat multiple times in the different blocks of a subsystem, function \(TCHC\) produces hash codes with a higher probability of collisions.
4.4 Subsystem Name Usage Analysis

than the name-connectivity counterparts. The use of larger empirical weights (i.e., \( w_1 \geq 2^0 \) and \( w_1 \geq 2^8 \)) produces hash codes that not only better identify the directionality of dataflows, but also have a lower probability of collisions. The probability of getting false negatives decreases when using this hash code, in addition to the name-connectivity hash code, to identify similar subsystems.

4.4 Subsystem Name Usage Analysis

We introduced in section 3.3.3 the overall functionality and typical outputs of module SNUA. In this section, we provide the formal steps and general scope of the algorithm supporting the inner works of this module.

Let \( S_{ij} \) be a subsystem \( j \) of model \( M_i \) in repository \( \mathcal{R}_{sr\text{id}} = \bigcup \forall i M_i \), and \( S = \bigcup \forall ij S_{ij} \) (i.e., the set of all possible model subsystems in \( \mathcal{R}_{sr\text{id}} \)), then a size-based clustering of subsystems, as defined in section 3.3.1, partitions \( S \) into a set of classes \( C = \{ C_k \mid k = 1, ..., n; C_i \cap C_j = \emptyset \text{ if } i \neq j \} \), that is, \( S = \bigcup \forall k C_k \). Let \( N_{k,l} \) be the name of the \( l \) subsystem \( S_{k,l} \) in class \( C_k \), then module SNUA addresses the sub-problems of finding:

1. All \( S_{k,r} \in C_k (\forall r \neq l) \) such that \( N_{k,r} = N_{k,l} \).
2. All \( S_{p,q} \in C_p (\forall p \neq k) \) such that \( N_{p,q} = N_{k,l} \).

Module SNUA also solves the complete subsystem name usage and duplication identification problem by reiterating steps 1) and 2) above, for all \( S_{i,j} \in \mathcal{S} \). Case analysis of the latter results can significantly aid in inferring one or more Simulink modeling patterns, for example:

- Given a subsystem \( S_{k,r} \in C_k \cap S_{M_i} \), where \( S_{M_i} = \{ \text{all subsystems of model } M_i \} \), whose name is \( N_{k,r} \). The existence of a set \( \mathcal{D}_{\text{seq}} = \{ S_{k,l} \mid N_{k,l} = N_{k,r}, S_{k,l} \in C_k \cap S_{M_i}, l \neq r, i \neq j \} \) might be the result of multiple versions of the same model in \( \mathcal{R}_{sr\text{id}} \) (i.e., \( M_i \cong M_j \)), with \( S_{k,r} \) not being a recurrent modeling pattern. Figure 4.13 shows subsystem Determine Undefaulted Alternative Fuel Tank Pressure, which appears multiple times in a repository containing multiple versions
4.4 Subsystem Name Usage Analysis

of the industrial repository studied in chapter ch:EmpiricalAnalysis, however the subsystem is unique in topology and design semantics.

• For the previous scenario, the existence of a set $D_{eq} = \{S_{k,l} | N_{k,l} = N_{k,r}, S_{k,l} \in C_k \cap S_{Mi}, l \neq r\}$ could suggest that $S_{k,r}$ is a recurrent component that implements a modeling concept which can be applied to different sections of a Simulink model, hence, it is a plausible candidate to conform or be part of a pattern. Figures 4.14a and 4.14b, in section 4.4, show an Event Scheduler subsystem having similar topology and possibly close semantics, and therefore they could be instances of a common event scheduler pattern.

• Let us maintain the same scenario. If $D_{neq} \neq \phi$ and $D_{eq} \neq \phi$, then we can apply some domain contextualization and build a stronger interpretation of subsystem $S_{k,r}$ as representing the partial or complete encapsulation of a modeling pattern.

![Figure 4.13: Recurring Subsystem Due to Multiple Versions in Repository](image)

Figures 4.14a, 4.14b and 4.15 depict a typical result produced by module SNUA, where the name Event Scheduler is used for subsystems of different sizes (i.e, 3B1L, 4B2L and 45B41L), topology and most likely design semantics. These subsystems belong to three different models of
4.4 Subsystem Name Usage Analysis

the industrial repository studied in chapter ch:EmpiricalAnalysis, and may indicate an overloaded
use, by different modelers, of the event scheduling concept in the same application domain.

(a) Subsystem Event Scheduler of Size 3B1L

(b) Subsystem Event Scheduler of Size 4B2L

Figure 4.14: Event Scheduler Instantiated for Subsystem Sizes 3B1L and 4B2L

Figure 4.15: Event Scheduler Instantiated for Subsystem Size 45B41L
Chapter 5

Model-Pattern Inference Processes

This chapter describes the semi-automated processes supported by the tools of the Simulink model-pattern discovery framework we propose in this dissertation. We use the MoSART interactive interface to manipulate and analyze the cluster classes that the spSearch workflow generates in an automated manner. The combined operation of both tools allows the implementation of inference processes that help practitioners identify Simulink semantic model-patterns. These patterns represent interesting, prescriptive or descriptive, modeling modules in the design or analysis of real-time embedded systems. To provide context, we describe a typical pattern discovery workflow using our experimental prototypes spSearch and MoSART. As shown in figure 5.1, we conceptualize the use of our tools to elicit patterns as a two-stage endeavour:

1. **Automated Pattern Detection.** Execution of the spSearch tool on the source code definitions of entire repositories $R_{srid}$ of Simulink models, in order to identify all their subsystems, compute their core topological properties and cluster them by size into a set of addressable classes (i.e., phase I in figure 1.1).

2. **Interactive Pattern Inference.** Use of the MoSART tool to query and perform relational operations on the cluster classes generated by spSearch, to select and visualize the content of specific classes and subsystems, in order to infer more abstract relations between these classes and refine
5.1 Model-Pattern Discovery Workflow

subsets of them into semantic model-patterns, that is, into Simulink design patterns applicable in targeted use cases (i.e., phases II and III in figure 1.1).

Figure 5.1: Simulink Pattern Discovery with MoSART’s Browse View

5.1 Model-Pattern Discovery Workflow

Our experimental work with the spSearch has shown that the tool efficiently scales up to cluster the entire scope of large sets of Simulink models within relatively small elapsed times (e.g., processing time on a desktop computer is under 3 minutes for the industrial repository studied in chapter ch:EmpiricalAnalysis, which contains 5.2 million lines of source code for 319 Simulink models and a total of 26,692 subsystems). In light of these results, we have concluded that it is viable, in practice, to re-apply spSearch numerous times on either different versions of the same Simulink repository or completely different ones. This capability of spSearch facilitates iterating on repositories from different perspectives, based for example on identified use cases, to generate solution-specific sets of classes. Furthermore, we can selectively merge and/or filter these sets of classes, using the size of the subsystems (i.e., mbnL code) as the merging or selection key, to generate any number
of derived ones (refer to section 3.5, ancillary utilities spClassMerge and spClassSelect). These operations expand the pattern inference possibilities to sub- or super-sets of classes without having to reprocess the source code of the original repositories. Our MoSART tool can process any derived sets of cluster classes, as long as they are expressed in the format of the DCD data structure defined in section 3.3.4. Therefore, we can use MoSART’s class querying and visualization facilities to study configurations of subsystems and infer patterns from a wide variety of sets of classes: 1) the ones that multiple iterations of spSearch generate (i.e., base sets), and 2) the ones derived from the base sets through class set reorganizations.

Based on the capabilities describe above for our tools, we propose the pattern discovery workflow depicted in figure 5.1. We consider it to be a generic template for the processes required to analyze repositories of Simulink models with the purpose of inferring model-patterns. Since our main objective is to optimize the use of the results that the combined actions of spSearch and MoSART can bring about, we conceptualize this workflow as having the following functional steps:

1. **Assemble repositories of Simulink models** $R_{srid_k}$ ($k = 1, ..., n$) according to grouping criteria of interest to analysts. All processing with spSearch and MoSART, for each $k$ repository, is linked to its $srid_k$ (Simulink Repository Identification).

2. **Create a set of subsystem-cluster classes** $C_{srid_k}$ ($k = 1, ..., n$) with spSearch, based on the size of each subsystem, for each repository $R_{srid_k}$.

3. **Create sets of derived classes** $C_d$ ($d = 1, 2, ..$) using either of these operations:
   - Merge two or more $C_{srid_k}$ on subsystem sizes (i.e., $mBnL$ key).
   - Filter any $C_{srid_k}$ using a set $\{mBnL\}$ of keys, that is, create $C_d \subset C_{srid_k}$.
   - Merge and filter any $C_{srid_k}$.

4. **Import to MoSART** any base class sets $C_{srid_k}$ or derived class sets $C_d$, to query and visualize their subsystems.
5.2 Interactive Elicitation of Model-Patterns

5. Analyze the statistical reports produced by spSearch for each $R_{srid_k}$. The core reports are 1) subsystem distribution for each set of classes, 2) name usage by class size and 3) chart sequence profiles.

6. Visualize classes and sub-classes of subsystems, in a class set $R_{srid_k}$ or $C_d$, with the selection, querying and visualization facilities in MoSART.

7. Infer model patterns using subsystem similarity and frequency of occurrence, as well as relations among subsystems uncovered during the visualization process (e.g., subsystem correspondence, hierarchy, natural containment, etc.)

8. Evaluate the set of inferred patterns for usefulness, perceived value or general interest. An evaluation indicating that more patterns could be inferred or refined from the current $C_{srid_k}$ and $C_d$ sets of classes, would trigger further class and subsystem visualizations on MoSART. Alternatively, appropriate reorganizations of the $R_{srid_k}$ repositories, and subsequent reprocessing with spSearch, could help resolve signs of saturation in the pattern inference information that can be extracted from visualizing the elements of $C_{srid_k}$ ($k = 1, ..., n$) and $C_d$ ($d = 1, 2, ..$) on MoSART.

5.2 Interactive Elicitation of Model-Patterns

We believe that the process of eliciting model-patterns from repositories of Simulink models is more effective and likely to produce positive results when it employs a series of cumulative model-knowledge-acquiring steps. Furthermore, the effectiveness of the results generated by the later steps of this process depend on proper domain knowledge and analysis skills. Specialists make decisions about what constitutes an interesting semantic-model pattern based on systemic concerns, project schedules and development practices. We propose that experts can use our high-performance tools spSearch and MoSART, along with the pattern discovery approach depicted in figure 1.1, to expedite this process, obtain a variety of useful indicators subsystems, enhance the knowledge about the
models in their application domain, and in the end, decide what entities they will extract as useful modeling patterns.

The Browse view of MoSART, shown in figure 3.5, supports a flexible and integrated environment to re-configure, query and visualize the cluster classes generated with spSearch. It supports the model-pattern inference work of practitioners, through progressive stages of abstraction and verification, by allowing them to manipulate and visualize the subsystems of existing Simulink repositories to:

1. Present full and partial views of the sets of subsystem classes produced by repeated runs of spSearch.

2. Query or manually select cluster classes from a set, and the contained subsystems, for visualization and identification of relations.

3. Drill-down into the diagrams of models and subsystems for the selected cluster classes.

4. Save and retrieve configurations of cluster classes and subsystems manually assembled or obtained as a result of queries on the attributes (e.g., class, topology hash codes, model path, physical location) of the subsystems listed on a dynamic grid.

MoSART implements the Browse view’s model visualization capabilities as a client application that interfaces with the processing engine and graphical tools of the Matlab-Simulink software acting as a server [54, 56]. Figure 5.2 shows the architecture of the client/server implementation for our experimental work. The Matlab-Simulink server side manages the storage, retrieval, display and visual navigation of models and subsystem diagrams. MoSART as a client 1) using the Browse view, saves, retrieves, queries and displays a set $C_{srid}$ of cluster classes for a repository $R_{srid}$, 2) controls the communication with the Matlab-Simulink server side, and 3) requests from the server side the display of models and subsystems selected on the Browse view.
5.2 Interactive Elicitation of Model-Patterns

The main objective of the framework and approach we propose (refer to figure 1.1) is to facilitate the elicitation of Simulink model-patterns through the iterative analysis and synthesis of increasingly specific derived classes of subsystems. Within this context, MoSART complements the automated pattern-detection work of spSearch and enables the use of quantitative and qualitative subsystem similarity and connectedness techniques such as: 1) inspections of the statistical distributions of subsystems across a system of classes \( C_{srid} \), 2) examination and correlation of values for subsystem properties such as size, names and topology hash codes, and 3) clustering of subsystems under specific relations. Furthermore, it allows highly-focused drill-down operations, on the diagrams of any model and subsystem in a repository, to understand or resolve very fine points existing in these models (e.g., embedded C-code, function triggers, optional execution and tables).

We can perform a large number of different pattern-inference operations with our framework, which we can likely delimit using specific pattern-application goals and making system-driven decisions about ways of using the modeling knowledge inferred from repositories of Simulink models. In this study, we have concentrated on capturing patterns from repositories that express the modeling ideas of functional encapsulation and hierarchy. We have also striven to uncover model-subsystem trends that have the potential to make the pattern analysis/synthesis process more productive. For this study, we have considered variations of two generic types of pattern inference operations to
5.2 Interactive Elicitation of Model-Patterns

show pattern analysis and synthesis processes on the base set of classes $C_{srid}$ generated by spSearch:

- **Class Consolidation (CC).** Given a set of classes $C_s = \{C_i\}$ such that $C_s \subseteq C_{srid}$, then a class consolidation operation $\text{CC}_{P_{CC}}(C_s)$ on $C_s$, based on a set of properties $P_{CC} = \{p_j\}$, produces a new class:

  $$C_{cons} = \{S_k \mid \exists C_i \in C_s \text{ such that } S_k \in C_i, \forall p_j \in P_{CC} \Rightarrow p_j(S_k) = TRUE\}$$

- **Sub-Classing (SC).** Given a class $C \in C_{srid}$, then a sub-classing operation $\text{SC}_{P_{SC}}(C_s)$ on $C$, based on a set of properties $P_{SC} = \{p_j\}$, produces a new class:

  $$C_{sub} = \{S_k \mid S_k \in C, \forall p_j \in P_{SC} \Rightarrow p_j(S_k) = TRUE\}$$

Overall, operation SC is a special case of CC (i.e., $C_s$ is a single class), however, the property sets $P_{SC}$ and $P_{CC}$ are often of a different nature. $P_{SC}$ usually aims to find subtypes of candidate patterns, while $P_{CC}$ seeks to expand or generalize those patterns through composition of subsystems (or portions of them) or inclusion of more subsystem instances, respectively. We can implement these operations in our pattern inference framework by performing one or a combination of the following tasks:

1. Manually explore the properties of the set of classes $C_{srid}$, produced by spSearch, with the class selection, subsystem selection and subsystem visualization capabilities of the MoSART’s *Browse* view. This search paradigm considers each class exploration a pattern inference step that can be saved, recalled and integrated into other steps to build up a pattern discovery workflow.

2. Interpret the cluster class, name usage and statechart sequences analysis reports produced by spSearch, and manually select specific cluster classes and subsystems that can support pattern inferences based on those reports.

3. Execute queries on the subsystem properties displayed on the *Browse* view of MoSART (refer to section 5.3) to select specific cluster classes and subsystems that can support cumulative pattern inferences.
5.2 Interactive Elicitation of Model-Patterns

The following are examples of types of CC and SC operations on $C_{sr_id}$ that we can emulate with the MoSART-supported tasks above.

- **CC or SC by subsystem frequency.** These operations aim to visualize statistical patterns in the frequency of occurrence of the subsystems in a repository that would indicate specific granularity levels in the modeling approach. Examples of these patterns are 1) generic small subsystems of high occurrence, 2) large hub subsystems of low occurrence, 3) core-functionality subsystems with a known estimated occurrence and 4) correlation of the subsystem occurrences within the same class of $C_{sr_id}$, etc. The empirical evidence that we have extracted with the tools of our pattern discovery framework from industrial and public domain repositories (refer to chapter 6), confirms the existence of these types of subsystems.

- **CC or SC by subsystem location.** In the design of large embedded systems, analysts have to partition the modeling effort and produce an assortment of Simulink models that usually contain similar or repeated subsystems. Subsystem location operations on $C_{sr_id}$ attempt to visualize the scope and nature of this commonality of subsystems across a repository. They also help to visualize hierarchical and compositional relationships between subsystems in abstraction from their logical and physical locations.

- **CC or SC by type- or name-connectivity.** These operations help visualize finer similarities or dissimilarities between subsystems, determined by their specific topologies, which may lead to refinements or more accurate descriptions of previously-inferred modeling patterns.

- **CC or SC by topology of stereotypes.** Subsystems in the same class of $C_{sr_id}$ encapsulate the same number $B$ of blocks and $L$ of data flows, however, their topology may be different. The name- and type-connectivity hash codes help identify groups of subsystems within a class with identical topology. Visualization of the subsystem topology in these classes, resulting from SC operations, and cross-referencing it with portions of larger subsystems, resulting from CC operations, will help infer topology stereotypes likely indicative of Simulink modeling patterns.
5.2 Interactive Elicitation of Model-Patterns

- **CC by property ranges.** Subsystems whose type-connectivity hash code values are within a narrow range, or whose sizes are similar, or whose frequencies of occurrence are correlated within a small variation range, tend to be very similar or tightly related. Visualizations of these subsystems, which CC operations filter and classify using ranges of the mentioned properties, have a strong possibility of uncovering modeling patterns.

- **CC by relationships.** The identification of hierarchical (e.g., containment, precedence, etc.) and/or referential (e.g., event triggers, complementary functionality, etc.) relationships, either by using previous domain knowledge or subsystem visualizations, can support CC operations on $C_{srld}$ leading to the formation of highly heterogeneous subsystem classes. Their visualization could result in the inference of complex and sophisticated patterns.

- **CC by Stateflow-chart sequences.** Usually, the design of large systems involves the modeling of repeating complex behaviours, composed of one or more primitive ones, all associated with the same system. Analysts model this situation in Simulink as related sets of basic Stateflow charts, possibly connected to one or more master Stateflow charts, that repeat in different places. Performing CC operations on $C_{srld}$, based on patterns of behaviour identified as sets of Stateflow charts (refer to section 4.1, algorithm getChartSequencePatterns), and later visualizing the resulting classes, can lead not only to the confirmation of behavioural patterns, but also to the inference of complex structural or architectural patterns.

- **CC by subsystem name and topology.** In general, modelers label subsystems with names that reflect their functionality, role or logical structure. The semantic accuracy of these assignments varies and frequently is not a reliable indicator of subsystem similarity. CC operations on $C_{srld}$ that include subsystem names and topology produce classes that help validate the structural and behavioural similarities of subsystems with common names. Visual inspections of these classes on MoSART’s Browse view allow one to confirm or dismiss these names as similarity indicators for subsystems.
Overall, the end goal of this subsystem management and visualization framework is to obtain a set $M_{srid}$ of derived classes, through a series of CC and/or SC operations, where each class can undoubtedly be interpreted as a Simulink modeling pattern of interest, at a desired abstraction level. Further segmentation, correlation and abstraction of the elements in $M_{srid}$ could greatly facilitate the elaboration of a language of interest to describe or apply those patterns in specific use cases.

Analysts can enhance the accuracy of the pattern visualizations with some adjustments to visual interpretations, which take into account exceptions applied when computing subsystem properties. These factors are not significant in evaluating subsystem similarity, however awareness of their effects helps clarify any potential false-positive or -negative pattern situations. Examples of these exceptions are: 1) disconnected blocks in subsystems do not contribute to the computation of the name- and type-connectivity hash codes, 2) the block-type for linked-blocks is provided by the model or library sourcing the block and may be undefined.

### 5.3 Class Querying System

As described in section 3.3.1, module PClu clusters by size all the subsystems in a repository $R_{srid}$ of Simulink models, into a set of classes $C_{srid} = \{C_{m,n} \mid \forall \text{ subsystem } S_{p_{m,n}} \in C_{m,n} \text{ encapsulates } m \text{ blocks and } n \text{ dataflows}, \ m, n = 0, 1, 2\ldots; \ p = 1, 2, \ldots\}$. This primary clustering, although broadly identifying similar sizes for the encapsulations of components, partitions a repository $R_{srid}$ into a set of classes, where each class is uniquely identified by an easy-to-visualize and query subsystem property. Furthermore, subsystem size functions very well with other subsystem properties (e.g., topology, location and names), allowing one to build very powerful queries on the set $C_{srid}$ of cluster classes. The Browse view of MoSART, shown in figure 3.5, implements an interactive facility to build queries, run them on the classes and subsystems listed on a dynamic grid $G$, display their results on $G$, as well as save them to and retrieve them from permanent storage. The mechanism for building queries is characterized by:
5.3 Class Querying System

- A query template that allows one to build queries according to the generic expression:

\[ C_g \land (P_1 \lor P_2 \ldots \land P_{n-1} \lor P_n) \ldots \] (5.1)

where,

- \( C_g \): Specifies a group of classes from \( C_{srid} \). Each class is represented by an identifier with the format \( mBnL \).
- \( P_k \) (\( k = 1, \ldots, n \)): regular expressions [59]. Each regular expression \( P_k \) acts upon a string that represents the value of subsystem property \( k \), respectively. Table 5.1 lists typical regular expressions applied on subsystem property value strings.
- \( \land, \lor \): Logical conjunction and disjunction, respectively.

- A class management and selection scheme to configure arbitrary subsets of \( C_{srid} \) and dynamically update the grid \( G \) with the attributes for the subsystems in the selected classes.

- A query management scheme to create, save and recall queries compliant with expression 5.1. This facility also updates the grid \( G \) with the results of a query.

- A scheme for manual selection of subsystems on the grid \( G \) for display and navigation of the diagrams they encapsulate. This includes a mechanism for saving and retrieving these selections at a later time.

Using queries compliant with expression 5.1, we can simulate operations such as sub-classing, class cross-referencing and intersection of derived classes, based on properties other than subsystem size, for example, subsystem topology encoded in hash codes, hierarchy encoded in subsystem nesting paths and models’ physical locations encoding specific use-case-oriented groups. By considering the subsystem size as the key index for these queries, we can select broad or refined groups of interesting Simulink subsystems in a very straightforward manner, due to the characteristic distribution of values for this property. For example, complex subsystems tend to have larger sizes, while template and high-abstraction-level subsystems tend to have small sizes, and common-concept subsystems tend to have sizes that group around specific mean values. These operations, in
5.4 Entailment and Abstraction of Model-Patterns

combination with the visualization of subsystems, the analysis of statechart sequences and the reuse of previously saved pattern candidates, build a comprehensive aid for iteratively refining Simulink model-patterns.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Regular Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCHC</td>
<td>3BD7B3C</td>
<td>3BD7B3C</td>
<td>Select subsystem if TCHC = 3BD7B3C.</td>
</tr>
<tr>
<td>TCHC</td>
<td>Not Null</td>
<td>.+</td>
<td>Select subsystem if it contains at least two blocks linked by a dataflow.</td>
</tr>
<tr>
<td>TCHC</td>
<td>Not Null</td>
<td>.*</td>
<td>Select subsystem regardless of its topology of blocks and dataflows.</td>
</tr>
<tr>
<td>Subsystem Path</td>
<td>Model/.../SubsysName</td>
<td>SubsysName$</td>
<td>Select subsystem if its path ends in SubsysName.</td>
</tr>
<tr>
<td>Model</td>
<td>ModelPath/ModelName</td>
<td>ModelName$</td>
<td>Select all subsystems in model ModelName</td>
</tr>
<tr>
<td>Model</td>
<td>ModelsFolder_latest/ARPR.../ModelName</td>
<td>(?!.<em>_Lib).</em>_latest/ARPR</td>
<td>Select all subsystems, in the latest version of an ARPR set of models that are not libraries</td>
</tr>
</tbody>
</table>

Table 5.1: Regular Expressions on Subsystem Property Strings

5.4 Entailment and Abstraction of Model-Patterns

Section 4.1 describes how spSearch takes advantage of the nested syntactical structure of Simulink models to selectively parse subsystems, compute a number of their interesting properties, and cluster them by size, measured as the number of blocks and dataflows they encapsulate. In many situations, this initial measurement of similarity allows us to provide clear assertions about a subsystem or group of subsystems being a pattern, by quickly browsing, querying, contextualizing and applying domain knowledge to the classes generated by spSearch. For example:

- If a repository contains a high number of identical subsystems (i.e. same name, size and topology hash codes), we will find a cluster class with a large number of elements whose visualizations on MoSART also look identical. We can run additional queries to obtain the usage context of these subsystems within the different models of the same repository. Finally, combining context and visualization, we will be in a position to describe the pattern for subsystems of the current type
and, most likely, using our domain knowledge, assign design meaning to this pattern.

- If the same repository contains other non-identical subsystems of the same size as the ones identified in the previous example, we can query the same class on either subsystem name or type-connectivity hash code to isolate sub-classes of identical subsystems. We use the name-connectivity hash code, displayed for each subsystem on MOSART, to finely isolate the different sub-classes. Thereafter, we visualize, contextualize and assign design meaning to each sub-class as we did for the identical-subsystem class.

- If a repository contains subsystems with the same name and different sizes, we can use the name-usage report, generated by the SNUA module (refer to section 3.3.3), to find those names. Thereafter, we can query on MoSART all the subsystems with a given name to isolate the classes and sub-classes that cluster them. Finally, we can visualize the subsystems in the latter data structures, contextualize them, and apply our domain knowledge to describe potential patterns with an appropriate design meaning.

- If a repository contains subsystems that are Stateflow charts, we can use the statechart-sequences report, generated by the PClu module (refer to section 4.2) to identify the models and subsystems that contained them. The sequence patterns in this report give a primary idea of structural relations between these subsystems. Thereafter, we can query these subsystems multiple times, on MoSART, to identify further relations among them and possibly build a structural pattern. The visualization of the Stateflow charts involved will give us further information about the nature of the behaviours implemented in those subsystems. Additional queries, to determine where those related subsystems are embedded in a model, will likely give us enough information to describe a pattern and use our domain knowledge to assign an overall design meaning to this pattern.

- If a repository contains subsystems that always go together in a model, through containment or interfacing (i.e., a scheme of dataflows), we can use MoSART queries to initially identify n-tuples of classes or subclasses having the same number of subsystems. We can then query the
5.4 Entailment and Abstraction of Model-Patterns

subsystems in a single n-tuple, for each n-tuple, to determine whether they appear in repeating groups of n-subsystems. Next, we can query the subsystems in each repeating group to identify further contextualization properties. Finally, we visualize the subsystems in the group to describe a pattern and use our domain knowledge to assign an overall design meaning to this pattern.

The previous examples show a pattern in the approach we use to build inferences about Simulink model-patterns with 1) the cluster classes and reports produced by spSearch, and 2) the query and visualization facilities of MoSART. We propose in this dissertation a conceptual view of the interactive model-pattern inference workflow, depicted in figure 5.3, that takes advantage of the features in MoSART and consists of the following steps:

1. Analyze the quantitative aspects of the subsystem cluster classes from the spSearch reports and MoSARTs class display facilities: high count of subsystems, large/small subsystems, small counts of large subsystems, high counts of small subsystems, frequency of names, and others, to identify more specific structural classes of subsystems. With the same goal, use the reports generated by spSearch to analyze any special cases, such as subsystems that are Stateflow charts or subsystems with the same names and different sizes.

2. Query specific groups of classes identified in the previous step to find sub-classes of either identical, similar or related-by-property subsystems. Visualize representative exemplars of these broad groups and describe, if enough information is available, a pattern at this stage. Apply domain knowledge to assign a design meaning to the pattern. Otherwise, continue with the model-pattern elicitation process.

3. Contextualize the sub-classes identified in the previous step to refine the description of patterns. Apply domain knowledge to assign a design meaning to the patterns. Otherwise, continue with the refinement process.
5.4 Entailment and Abstraction of Model-Patterns

4. Reorganize the subsystem-cluster classes, corresponding to Simulink repositories currently processed for patterns, to analyze different configurations of sets of classes on MoSART. This action allows analysts to i) focus on specific types of models, ii) incorporate models from other repositories that can enhance the pattern inference process on the current one, especially in the more abstract stages, and iii) study the variability of models and subsystems among the different repositories (i.e., systems). Unless new models have been added to the original set of Simulink repositories, we can complete this reorganization just by reshaping the original sets of cluster classes with our utility programs spClassSelect and spClassMerge (refer to section 3.5). Otherwise, it would be necessary to reprocess these repositories with spSearch before analyzing the new class sets with MoSART.

![Figure 5.3: Interactive Model-Pattern Inference Process with MoSART](image)

Our empirical work with industrial and public domain Simulink models has revealed that the effectiveness of our tools, spSearch and MoSART), is fairly unaffected by the size and content of the Simulink repositories. For various repository sizes and types, we still have short processing times, the cluster classes generated cover all the models and subsystems in each repository, and our tools allow an approach to eliciting model-patterns that is repository agnostic. With these features,
5.4 Entailment and Abstraction of Model-Patterns

we are able to reduce the accidental complexity in the pattern search processes to much lower thresholds, and our results are mostly influenced by the quality of the design knowledge embedded in the repositories. In addition, we provide some powerful repository indicators, embedded in the classes and subsystem properties computed by spSearch (e.g., localization of complex subsystems, distribution of small subsystems and typical block-dataflow links), that experts can easily gather in order to reason about patterns. Finally, they can apply their domain expertise to make the ultimate decisions about the usefulness of the model-patterns inferred in their domain.

In general, the larger and more domain-oriented repositories found in industry tend to drive more prolific and realistic model-pattern discovery efforts. Furthermore, the practitioners analyzing these repositories are generally very proficient domain experts, who are best prepared to provide optimal descriptions of patterns and assess their usability potential. They usually possess the systemic knowledge required to make decisions about the operational importance, effect on integrated systems, and general usefulness of potential model-pattern candidates. Their expert knowledge allows them to decide about the conclusive abstraction levels of model-patterns, for example, cross-domain, product-line-specific, or project-specific. Our tools, spSearch and MoSART, focus on helping to significantly reduce the complexity involved in completing these highly conceptual tasks.

As proof of concept, we have applied our Simulink model-pattern discovery to an industrial repository from the automotive domain. We describe the model-patterns elicited from this repository in chapter 6.
Chapter 6

Empirical Analysis of an Industrial Repository

In this chapter we describe our experimentation and the results obtained after applying the algorithms, processes and tools of our model-pattern discovery framework, to an industrial repository of Simulink models. spSearch implements the algorithms, described in chapter 4, to search, cluster and compute the topological properties of model-subsystems, and also to evaluate structural and relational properties of Stateflow charts. MoSART allows analysts to query, reorganize and save the cluster classes generated by spSearch, as well as visualize the Simulink diagrams for any groups of subsystems in these classes. Using the cooperative work that these tools perform 1) to significantly reduce the search space for model-patterns with clustering subsystems by size and computing key properties (i.e., spSearch), and 2) to quickly recall and visualize any configured groups of subsystems (i.e., MoSART), practitioners can implement iterative and interactive pattern discovery processes, as described in section 5.2, compatible with the approach we propose in this thesis (refer to figure 1.1). In the following sections, we show instantiations of these processes and describe the model-patterns we were able to extract from the repository of our case study.

We also applied our tools and processes to public-domain repositories to evaluate them with
incomplete, inconsistent and unrelated sets of Simulink models. We provide comparative structural and pattern results between the industrial repository and the public-domain model sets.

### 6.1 Model-Pattern Elicitation

We obtained the repository of Simulink models for our experimentation from an industrial partner in the automotive sector. This industry is characterized by regularly manufacturing new products, namely automobiles, that are incrementally-upgraded versions of existing ones. Therefore, successful modeling for either software development, system simulation, or system testing, in this product-line environment [70], is usually the result of focused efforts to 1) abstract the core entities for the domain in such a way that they present good re-usability properties for a range of use cases, 2) consider aspects of the system that can maximize the outcome of engineering design efforts and answer specific design concerns, and 3) assemble all entities within an architectural model (e.g., Plant-Control model shown in figure 2.2a) that offers a convenient integration of modeling aspects.

We completed a preliminary exploration of this repository with spSearch and MoSART, in order to understand core subsystems, large classes of subsystems, libraries, and other modeling entities and structures. We observed the existence in this repository of subsystem hierarchies, aspect-modeling features, architectural attributes and modeling objectives (e.g., control loops, simulation, system integration, software generation) described above. Based on this preparation work, we instantiated the templates for model-pattern discovery processes, explained in section 5.2, on this repository. We describe these processes in the following sections, including the extracted model-patterns.

#### 6.1.1 Subsystem Size and Class Consolidation (CC).

The clustering by size of this repository, produced by spSearch, consists of 442 classes containing 26,692 subsystems. Table 6.1 shows more details about the distribution of sizes. A quick
6.1 Model-Pattern Elicitation

visualization of these classes and subsystems in MoSART, along with an analysis of the subsystem-size distribution across classes, showed that the models in this repository contain large virtual Simulink subsystems to 1) graphically encapsulate complex or multiple functionality into modules with easy-to-manage interfaces, and 2) centralize the visualization of common operations. Typical exemplars of subsystems built using these approaches are function consolidation entities, task distribution hubs, and generic interfaces within a system architecture (e.g., AUTOSAR [8]).

<table>
<thead>
<tr>
<th>Subsystem Description</th>
<th>mBnL Value (B = block, L = dataflow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Largest size</td>
<td>758B589L</td>
</tr>
<tr>
<td>Smallest size</td>
<td>0B0L</td>
</tr>
<tr>
<td>Most frequent</td>
<td>3B1L (18545 subsystems of this size, 69.5% of total)</td>
</tr>
<tr>
<td>Least frequent</td>
<td>182 classes with 1 subsystem (size in range 1B1L - 758B589L)</td>
</tr>
</tbody>
</table>

Table 6.1: Industrial Repository Case Study - Subsystem Size Statistics

An inspection of the repository’s entire set of classes with MoSART class management panel, allowed us to isolate specific groups classes, by size ranges, and confirm the following trends:

- **Classes of large subsystems** contain very few elements (i.e., 1 - 3 subsystems), confirming the approach of encapsulating complex functionality and multiple operations in subsystems with simple interfaces. The exceptions are classes containing multiple occurrences of subsystem types that model components of a virtual powertrain operating system (i.e., set of Simulink models that emulate the dynamics of an automobile’s powertrain). The following are exemplars of this type of subsystems:

1. A subsystem type that models the operational states and high-level signals of a running car engine (e.g., subsystem name = *Engine State Machine*, TCHC = 33F3ED4, NCHC = 7CE8F, size = 50B41L).

2. Subsystem types that model the filtering of synchronous signals and the generation of single events (e.g., subsystem name = *System Events One Shot*, TCHC = 43A2740, NCHC = AD0E8, size = 60B48L; subsystem name = *Engine Events One Shot*, TCHC = 621F560, NCHC = E0054, size = 84B70L).
6.1 Model-Pattern Elicitation

3. A subsystem type that models the synchronous scheduling of sequences of tasks to occur in a car’s powertrain system (e.g., subsystem name = *Task Scheduler*, TCHC = 8D55E8C, NCHC = 1004BC, size = 165B132L).

4. A subsystem type that emulates the concurrent and/or synchronous calling of multiple subsystems for execution (e.g., subsystem name = *Dummy OS*, TCHC = 11776EB8, NCHC = 4EBD1A, size = 243B241L).

5. A subsystem type that models the selection of specific groups of subsystems for concurrent execution (e.g., subsystem name = *Operating System Tasks*, TCHC = FB459C4, NCHC = 194971, size 362B241L).

- **Classes of small subsystems** tend to contain large numbers elements (i.e., over 50 subsystems), which correspond to instances of subsystems that model different types of functionality with the same numbers of blocks and dataflows. In section 6.1.2, we provide a sub-classing and frequency analysis for these classes. The notable exemplar of this situation is class 3B1L, which contains 68.5% of all the subsystems in the repository.

In addition, visualizations on MoSART of the multiple occurrences of the same subsystem in a specific size, allowed us to 1) confirm the homogeneity of the subsystem instances in the group, by verifying that their name- and type-connectivity hash codes are equal, and 2) inspect the functional blocks and dataflows, in the Simulink diagram for an instance of the subsystem, to formulate a first interpretation of its design intent or meaning. We used the following typical query, which we can emulate by manual class selections on MoSART’s *Manage Classes* panel or by issuing the following query with MoSART’s *Query Grid* panel (refer to expression 5.1):

\[
[BL-Code] \in \{class_k \mid k = 1, \ldots, n\} \text{ AND } \text{RE[Types Hash Code]} = .+ \text{ AND } \\
\text{RE[Subsystem]} = '.+ \text{ AND } \text{RE[Model]} = '.+
\]

where [BL-Code] is a cluster class code and RE[fName] means regular expression entered in field “fName”.

97
6.1 Model-Pattern Elicitation

Thereafter, we visually confirmed identical subsystems in the results of these queries by observing identical hash codes. We concentrated on interpreting the blocks and dataflows of these subsystems, as well as their connection schemes, to classify them with more semantic-oriented descriptions. Using this technique, we were able to assign a design-oriented semantics to the following types of subsystems:

1. **Task Hubs** concentrate the selective initiation of sets of synchronous and/or concurrent tasks by mapping *Goto* to *From* blocks. A typical exemplar in this group is subsystem **GMPT Operating System Tasks** (size = 362B241L), which concentrates the synchronous calls to the relevant subsystems in the Simulink models of the virtual powertrain operating system.

2. **Main Schedulers** concentrate the scheduling of events for:
   
   (a) Tasks that interface with the virtual powertrain operating system.
   
   (b) Tasks affecting the overall modeling of the car’s powertrain system (i.e., the engine and transmission).
   
   (c) Tasks specific to the control of a specific module of the powertrain system.

   Exemplars in this group are:
   
   - subsystem name = **MIL OS Tasks Multitasking**, TCHC = 1A192318, NCHC = 25CD6D, size = 587B396L.
   
   - subsystem name = **MIL OS Tasks Multitasking**, TCHC = 281052C, NCHC = 48FD5, size = 99B64L.
   
   - subsystem name = **Task Scheduler**, TCHC = 11776EB8, NCHC = 4EBD1A, size = 165B132L.
   
   - subsystem name = **Main Task Scheduler**, TCHC = 11776EB8, NCHC = 4EBD1A, size = 43B45L.

3. **Time-Base Generators** concentrate the generation of time signals for schedulers and task hubs. They use a common time-base (i.e., clock), and signal generation is state-based. Subsystem
6.1 Model-Pattern Elicitation

*Engine State Machine* (size = 50B41L) is in this group.

4. **Integrators** connect sub-modules of a logical control module into a single subsystem, encapsulating a topology of data flows that satisfies the functional and operational requirements of a higher-level function. Subsystems that integrate a control *ring* for a physical module are typical subsystems in this group. Figure 6.8 depicts subsystem *RING* (TCHC = B95A64, NCHC = 62C78, size = 25B20L), which integrates the processing of the alternative fuel rail temperature ring.

5. **Generic Interfaces** encapsulate dataflows, and the generation of the signals that they trigger, into a single subsystem, in order to integrate the modeling of a formal interface that could be either:

   (a) A set of signals compliant with the control architecture in which the modeled system is embedded (e.g., proprietary or AUTOSAR architecture).Subsystem *Support Formal Interface Outputs* (TCHC = 3BF9E80, NCHC = 82E03, size = 73B48L) uses signal conversion blocks to change the name and/or the type of a set of dataflows to pre-defined names and types for a standard interface.

   (b) A set of generic signals required in different sub-modules of the modeled system. Subsystems *System Events One Shot* (TCHC = 43A2740, NCHC = AD0E8, size = 60B48L) and *Engine Events One Shot* (TCHC = 621F560, NCHC = E0054, size = 84B70L) encapsulate the generation of single-pulse events for specific groups of signals that originated in task scheduler subsystems.

   (c) A multiplexer/demultiplexer of generic signals. Examples of these subsystems are:

   • Subsystem name = *Periodic Events*, TCHC = 55CECC0, NCHC = A1277, size = 52B67L. This subsystem ramifies an *enable schedule* and a *enable periods* signal to generate a set of periodic events at fixed-time intervals. It also multiplexes the generated periodic event signals to report the *sample time* for each event.
6.1 Model-Pattern Elicitation

- Subsystem name = Process Level System, TCHC = 3341E0, NCHC = AD5D, size = 7B4L. This subsystem ramifies a vectorized function call generated by an S-function into three separate calls.

6. Test Signal Concentrators encapsulate into a single subsystem the initiation and timing of a series of standard tests applicable to a virtual powertrain operating system and the modeled system. Example: subsystem name = Test Tasks, TCHC = 2376BBDC, NCHC = 411BD, size = 758B589L.

7. Testers encapsulate the modeling elements (i.e., ring integration subsystem, tasks, controller scheduling and relevant signals) required for the execution of a specific ring (i.e., control path) of the modeled system. Example: subsystem name = IntegrationModel.Test, TCHC = 10C3A04, NCHC = 6211F, size = 35B21L.

8. Evaluators encapsulate the signals and logic corresponding to the sensing and diagnostic of predefined states of the modeled system. Examples of these types of subsystems are:

   - subsystem name = Evaluate Device Control, TCHC = 36A2090, NCHC = 7A8A5, size = 59B64L.
   - subsystem name = Evaluate Timer Expired Conditions, TCHC = F3A318, NCHC = 2593F, size = 20B20L.

9. Logic Blocks encapsulate into a single subsystem the logic of complex functionality supporting the internal operation of the modeled system. Example: subsystem name = Calculate Crank CylEvents, TCHC = 8D8D80, NCHC = 17BF4, size = 14B9L.

6.1.2 Subsystem Frequency and Sub-Classing (SC) or Class Consolidation (CC).

A preliminary analysis of the distribution of subsystem occurrence frequency, among the different cluster classes, showed that small subsystems tend to frequently occur in the repository of our case study. We determined that the number of elements in class 3B1L is exceptionally large, close to
6.1 Model-Pattern Elicitation

70% of the total number of subsystems. Observation of the name- and type-connectivity hash codes for the subsystems in this class highlighted the existence of various large sub-groups of subsystems that are either identical, have identical topology of types but different names, or have different connectivity but use the same types. These observations supported the idea of sub-classing and consolidating the elements of class 3B1L and other similar classes to understand potential design approaches that practitioners might have used in the models of this repository. We issued the following typical query to isolate the abovementioned sub-groups:

\[ [\text{BL-code}]= mbnL \ AND \ \text{RE[Types Hash Code]} = \text{TCHC_string} \ AND \ \text{RE[Subsystem]} = \'.+ \ AND \ \text{RE[Model]} = \'.+ \]

where \( mbnL \) is a class code string and \( \text{TCHC_string} \) is a type-connectivity hash code value.

Our results indicated that the design of the modeled system includes 1) a number of typical non-virtual subsystems [56] of the Simulink language, 2) various typical signal manipulation, computation and event generation subsystems, and 3) a generic two-state Stateflow chart to model the enabled/disabled states of signals throughout the system. Cases 2) and 3) represent subsystems of small size, with a high occurrence, which encapsulate atomic abstractions of either a signal type or a very common and straightforward structure of events. Semantic analysis of the blocks and dataflows in the subsystems returned by our sub-classing operations allowed us to identify the following types of subsystems:

1. **Enabled Subsystems** that, when temporarily activated by an external dataflow:

   (a) Initialize an encapsulated subsystem or an external one with a constant value. Examples:
   
   1) subsystem name = *Clear Subinterrupt Array*, TCHC = 76F84, NCHC = 39E8, size = 3B1L, and 2) subsystem name = *Clear Crank CylEvent*, TCHC = F3F84, NCHC = 3602, size = 3B1L, occurrences = 63.

   (b) Implement and initiate function calls to activate optional-execution subsystems or load a
6.1 Model-Pattern Elicitation

constant value. This type of subsystems, usually named Enabled Function-Call Generator[0-9]*\(^1\), exist in the repository as 1) TCHC = F436C, size = 3B1L, occurrences = 4274 occurrences, and 2) TCHC = 25394C, NCHC = 5862, size = 4B2L, occurrences = 62.

2. Triggered Subsystems that react to input events such as digital signal’s rising/falling transition or previous function calls to:

(a) Initiate new function calls with or without delays. Example: subsystem name = Triggered Function-Call Generator[0-9]*, TCHC = F549C, NCHC = 3744, size = 3B1L, occurrences = 10902.

(b) Initialize signals with constant values and/or connect optional dataflows. Subsystems named like Set Trigger[0-9]*, of sizes 3B1L (TCHC = F549C, NCHC = 3336, occurrences = 1837) and 5B2L (TCHC = 1E9BF0, NCHC = 3DD8, occurrences = 90.

(c) Activate a single or a configuration of data flows. Example: subsystem name = Set Trigger1, TCHC = F4754, NCHC = 2FB7, size = 3B1L, occurrences = 90.

(d) Activate periodic tasks. Subsystems named Operating System Tasks (For Simulation) are exemplars in this group. Table 6.2 shows the distribution of sizes for subsystems with this name.

<table>
<thead>
<tr>
<th>Group No.</th>
<th>TCHC</th>
<th>Size</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>450074</td>
<td>6B5L</td>
<td>57</td>
</tr>
<tr>
<td>2</td>
<td>4EDA54</td>
<td>7B6L</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>58B434</td>
<td>8B7L</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>64C418</td>
<td>9B7L</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>7AACB0</td>
<td>9B8L</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>B0575C</td>
<td>12B11L</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>E60208</td>
<td>15B14L</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>11BACB4</td>
<td>18B17L</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>15161EC</td>
<td>20B20L</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6.2: Distribution of Sizes for Subsystem: Operating System Tasks (For Simulation)

\(^1\)\([0-9]\)* indicates 0 or more digits
6.1 Model-Pattern Elicitation

(e) Trigger the execution of custom-developed functional blocks (i.e., S-functions or Matlab functions [56]). Examples: 1) subsystem name = *GetDFIR_FaultActive*, TCHC = 27FBF0, NCHC = 6FAE, size = 4B2L, and 2) subsystem name = *Process Level Subsystem*, TCHC = 348744, NCHC = B318, size = 6B3L.

3. **Action Subsystems** that exist in multiple sizes. In this section, we will refer to small subsystems. Section 6.1.3 provides descriptions for all sizes of this type of subsystems. The characteristic that defines them is their activation from subsystems containing decision logic blocks (e.g., if / else block) to perform an action modeled as conditional or optional. Table 6.3 provides a summary of small size exemplars and the typical actions they perform.

<table>
<thead>
<tr>
<th>Action</th>
<th>Subsystem Name</th>
<th>TCHC</th>
<th>NCHC</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize signals with constant values.</td>
<td><em>If Action Subsystem</em></td>
<td>F3D2CL</td>
<td>DF6</td>
<td>3B1L</td>
</tr>
<tr>
<td>Encapsulate the optional processing and generation of a signals.</td>
<td><em>If Action Subsystem</em></td>
<td>F4754</td>
<td>F2C</td>
<td>5B3L</td>
</tr>
<tr>
<td>Encapsulate a portion of a hierarchy of decision logic.</td>
<td><em>If Action Subsystem1</em></td>
<td>619EA0</td>
<td>B74F</td>
<td>7B6L</td>
</tr>
<tr>
<td>Encapsulate the optional execution of special blocks and legacy code.</td>
<td><em>If Action Subsystem3</em></td>
<td>34D564</td>
<td>6489</td>
<td>10B5L</td>
</tr>
<tr>
<td>Connect optional data flows.</td>
<td><em>If Action Subsystem</em></td>
<td>F4754</td>
<td>F2C</td>
<td>3B1L</td>
</tr>
</tbody>
</table>

Table 6.3: Types of Small *If Action Subsystems*

4. **Signal Generators** that connect basic enabled, triggered and action subsystems, and special functionality blocks (e.g., integrators, Stateflow charts), to build more complex signals. Table 6.4 lists three types of signal generator subsystems.

<table>
<thead>
<tr>
<th>Type of Signal Generator</th>
<th>Subsystem Name</th>
<th>TCHC</th>
<th>NCHC</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special delays</td>
<td><em>1000ms</em></td>
<td>A9C6E4</td>
<td>19715</td>
<td>9B8L</td>
</tr>
<tr>
<td>Periodic signals</td>
<td><em>Periodic1</em></td>
<td>7FC628</td>
<td>18856</td>
<td>9B6L</td>
</tr>
<tr>
<td>Interpolated signals</td>
<td><em>Cam Event</em></td>
<td>1BD3D4</td>
<td>4E6C</td>
<td>3B2L</td>
</tr>
</tbody>
</table>

Table 6.4: Types of Signal Generators

5. **Signal Hubs** that multiplex/demultiplex and re-route sets of core signals in the modeled system. Example: subsystem name = *Determine Diag Condition*, TCHC = 312E550, NCHC = 93CD0, size = 55B55L.
6. **Signal Selectors** that select a given signal from a set, depending on the state of a control signal. Example: subsystem name = *Cam Event*, TCHC =1BD3D4, NCHC =4E6C, size = 3B2L.

7. **Function Integrators** that combine, using a given configuration of data flows, a small set of encapsulated operations (i.e., 2 or 3 subsystems) of the modeled system. Example: subsystem name = *Cam Event*, TCHC =4A84F4, NCHC =7CB3, size = 5B3L. This subsystem computes a crank angle from an engine’s r.p.m. signal and generates a cam event.

8. **Event Generators** that either compute system conditions from input signals or sense input events from other subsystems, and generate one or more output events. Example: subsystem name = *MedRes Events*, TCHC =AF3070, NCHC =12E8E, size = 11B8L. This subsystem receives a base event and generates two intermediate events based on the values of two input signals.

9. **Event Handlers** that enable or disable the generation of an event based on their current state and the values assigned to a set of binary variables. Example: subsystem name = *Event Handler*, TCHC =2C9994, NCHC =713E, size = 5B4L, occurrences = 124. This subsystem is a generic Stateflow chart with two states, namely *Disabled* and *Enabled*, and an *Event Enable* output port.

10. **Event Schedulers** that activate a reduced set of signals in the modeled system upon receiving a control event. Example: subsystem name = *Low Res Events Scheduler*, TCHC =7533E8, NCHC =133F7, size = 9B7L, occurrences = 63. This subsystem generates up to three function calls (i.e., events) on a rising-edge activation by an input signal.

11. **Event Counters** that produce sequences of pulses equally separated in time (i.e., events). Each execution of these subsystems increments a count by a fix step. When the count reaches a maximum value, they issue a pulse and reset the count to an initial value. Table 6.5 lists four types of event counters.
### 6.1.3 Subsystem Name and Class Consolidation (CC).

Our empirical work with the repository of our case study has allowed us to get an insight into approaches used by modeling experts in industry. For example, in this repository, occurrences of subsystems with identical or similar names and varying sizes, strongly correlate with the existence of common abstract views for the modeling of real-time embedded systems. Subsystem have names that either attribute them with a *functionality semantics* in the technical language (i.e., jargon) of the application domain or identify special constructs of the modeling language, Simulink in our case. It is common to find subsystems that encapsulate noticeably different configurations of blocks and data flows, however they model a similar *type of functionality*, therefore they are assigned identical or similar names.

Our spSearch tool, which clusters subsystems by size (refer to section 3.3), also identifies subsystems with the same name and different sizes, which therefore belong to multiple cluster classes. For each class and subsystems in this situation, it reports the subsystem names and all the other classes where those names are found. In order to understand these patterns of subsystem names, we cross-referenced the reports produced by this tool for the repository and executed on MoSART a series of queries with the following template:

\[
[\text{BL-code}] = \{mBnL | \forall m \forall n\} \ \AND \ \text{RE[Types Hash Code]} = .+ \AND \\
\text{RE[Subsystem]} = 'SubsystemName$ AND \ \text{RE[Model]} = '.+
\]

where:

i) BL-Code includes all the cluster classes.

ii) *SubsystemName* = *Name*[0-9]* or *SubsystemName* = [0-9]+*Name1*[0-9]+*Name2*.
6.1 Model-Pattern Elicitation

In general, using a number of variants of this query, first, we obtained on MoSART all the subsystems with a name similar to *SubsystemName*. Next, we clustered the resulting classes by subsystem size into three groups: small, medium and large sizes. Finally, within each group, we clustered the subsystems by their TCHC code. The following is a sample of the analysis of subsystem names for the repository of our case study:

1. **If Action Subsystem.** This is a widely-used name type in the repository (819 occurrences), to label subsystems that model conditional functionality. Encapsulating subsystems execute branching-block logic that will call this type of subsystems upon the results of one or more logical conditions. Our analysis work with spSEarch and MoSART revealed three major cases where this name is used:

   (a) Small subsystems that are optionally executed, activated from a decision block (e.g., *if* or *switch* Simulink blocks [56]) or another subsystem, to implement one of the following actions:

      i. Insert an optional data flow in the containing subsystem. Examples: subsystem names

         **If Action Subsystem** or **If Action Subsystem1**, TCHC = F4754, size = 3B1L.

      ii. Insert an optional initialization value or table to the logic of the containing subsystem. Examples: subsystem names **If Action Subsystem** or **If Action Subsystem**\(N(N = 1, 2, 3, 4, 5, 6)\), TCHCs = F3F84, F3D2C, F404C, size = 3B1L.

      iii. Optionally change the type of a dataflow in the containing subsystem. Examples: subsystem names **If Action Subsystem** or **If Action Subsystem**\(N(N = 1, 2, 3, 4, 5)\), TCHCs = 107480, F3D2C, F404C, size = 4B2L.

      iv. Optionally perform dual actions for the containing subsystem, that is, either insert a dataflow and initialize a value or initialize two values. Examples: subsystem names **If Action Subsystem** or **If Action Subsystem**\(N(N = 1, 2, 3, 4)\), TCHCs = 1E7F08, 1E8480, 12C708, 1E7A58, size = 5B2L.
6.1 Model-Pattern Elicitation

v. Optionally apply a binary operator to two signals. Subsystem names \textit{If Action Subsystem} or \textit{If Action Subsystem}_N (N = 1, 2, 3, 4), TCHCs = 3BF0EC, 3BE91C, 34DD98, 3BE91C, 3B3DA0, 2EA248, size = 5B3L.

(b) Medium-size subsystems (i.e., sizes 6B4L to 15B12L) that encapsulate optional, and potentially reusable, modeling of a triggered-selec­tion of subsystems or a calculation of a car engine’s variable. The following are the details of the operations modeled in these subsystems:

i. Optionally perform function-triggered single- or dual-selection of encapsulated actions for the containing subsystem. Example: subsystem name = \textit{If Action Subsystem}, TCHCs = 40BB18, 65F84C, 7BDDC4, sizes = 6B4L, 8B6L, 12B9L.

ii. Optionally compute sensed variables from a car’s engine. Examples: 1) subsystem name = \textit{If Action Subsystem}, TCHC = 9BE0B0, NCHC = EA6A, size = 12B9L, this subsystem computes a \textit{LoRes Period} variable; 2) subsystem name = \textit{If Action Subsystem1}, TCHC = BB2E5C, NCHC = 15CAA, size = 15B12L, this subsystem computes a \textit{t_NoEngMvmt} variable.

(c) Large-size subsystems (i.e., sizes 15B13L to 57B55L) that encapsulate the modeling of more complex optional selection schemes of subsystems in the operation and diagnostics of the modeled system. These subsystems are named either \textit{If Action Subsystem} or \textit{If Action Subsystem}_N (N = 1, 2, 3, 4). The following are typical exemplars in this group:

i. Complex selection schemes of optionally executed subsystems. Example: subsystem name = \textit{If Action Subsystem2}, TCHC = 3C51810, NCHC = 816FD, size = 57B55L.

ii. Selection of diagnostics in the modeled system. Example: Example: subsystem name = \textit{If Action Subsystem}, TCHC = F9987C, NCHC = 55B6F, size = 19B15L.

2. \textit{Subsystem}_N (N = 1, ..., 11). These names have been assigned to a series of identical subsystems that produce signals to trigger testing tasks in a virtual powertrain system of Simulink models, which is the hosting environment for the repository. Exemplars in this group exist in the
following two sizes: i) TCHC = C4D4AC, size = 11B11L, occurrences = 90, and ii) TCHC = ED3118, size = 11B12L, occurrences = 80.

3. $XpYms$ ($X = 3, 6, 12; Y = 5, 25, 125$). This naming scheme is assigned to subsystems that produce pulse signals with a period of $X.Y$ milliseconds. The following are exemplars in this group: i) subsystem name = $3p125ms$, TCHC = 93D04, NCHC = 16372 , size = 8B7L, occurrences = 62, ii) subsystem name = $12p5ms$, TCHC = A9C6E4, NCHC = 19715 , size = 9B8L, occurrences = 62, and iii) subsystem name = $6p25ms$, TCHC = 16EFB44, NCHC = 34C0F , size = 19B18L, occurrences = 62.

4. **System Events One Shot.** This name has been used to label subsystems that produce a group of single pulses as outputs to other subsystems, in response to a corresponding group of event-enable inputs. These subsystems issue a single event for each input dataflow in the group to indicate that it has been activated. The virtual powertrain Simulink model-set in the repository uses these subsystems to report status changes of core signals such as a propulsion system power-up and shutdown. Exemplars of this type of subsystems exist in three different groups: i) TCHC = 2774190, size = 35B28L, occurrences = 62, ii) TCHC = 43A2749, size = 60B48L, occurrences = 62, and iii) TCHC = 541ADC0, size = 72B60L, occurrences = 1.

## 6.2 Semantic Entailment of Model-Patterns

The main thrust of our pattern discovery framework, shown in figure 3.1, is the search of Simulink model-patterns applicable in use cases related to the design of embedded cyber-physical systems. The repository of our case study contains Simulink models for the control and testing of a module embedded in the propulsion system of a complex transportation system. Its models mainly address timing, interfacing and functional requirements typical of a hard real-time system. Furthermore, its models are sound and work correctly only in the context of a super-set of Simulink models that emulate a major virtual physical module. Consequently, it is reasonable to expect that this set of
models and its virtual environment contain a number of hidden modeling patterns, inherent to the development of modular systems within a specific architecture.

These patterns, expressed by hierarchical arrangements of Simulink blocks and dataflows contained in subsystems, represent varying degrees of abstraction in the design of an embedded system. So far, we have uncovered a large number of medium-abstraction level patterns with spSearch and MoSART, which we have described in sections 4.2 and 6.1, using automated and interactive processing. Although we could iterate a number of times this sequence of steps, as in the approach we describe in figure 1.1, inferring more abstract patterns requires applying 1) expert domain knowledge, and 2) conclusions from previous pattern elicitation efforts that express a more abstract semantics. For example, our analysis of the model-patterns uncovered so far indicates that the developers of this repository (and its virtual environment) delimited the size and nature of the Simulink subsystems using three core guidelines: a) potential for re-usability, b) appropriate reuse of the existing modeling environment, and c) compliance with system integration requirements.

Assuming these modeling guidelines, which have cross-domain validity in the case of embedded systems, we increase the abstraction level of our model-pattern analysis and state that, in this repository, Simulink subsystems encapsulate design patterns that are:

1. **Structural**, whose semantics can be denoted only by the description of a set of functional blocks and dataflows.

2. **Behavioural**, whose operational semantics is mainly dictated by Stateflow charts, embedded code (e.g., Matlab, C), and system-specific data (e.g., tables, constants, reference and calibration data).

3. **Hybrid**, which contain structural and behavioural elements. However, by applying domain knowledge, it is possible to decide which elements are more predominant and label the patterns as either **Hybrid-Structural** or **Hybrid-Behavioural**. We describe the former pattern in a denotational manner and the latter one in an operational manner.

Applying these considerations to the sub-classes of similar subsystems described in section 6.1,
6.2 Semantic Entailment of Model-Patterns

we reiterate our analysis of them to provide descriptions that we label as semantic model-patterns. We describe the denotational or operational semantics of one element of the class and extrapolate it to the entire class. Using MoSART, we query the cluster classes to identify containment and compositional relationships between the elements of the different sub-classes. We simulate the application of domain knowledge by using previous work experience with integrating real-time controls for automated railway systems. Where applicable, we aggregate or link subsystems to assemble modeling patterns of interest. These composite patterns (i.e., classes) have i) an overall semantics represented by an interpretation of the interactions between subsystems, functional blocks and dataflows, as well as the aggregations of these components, or ii) a more compact semantics derived from reductions, overlaps, and simplifications to either denotational or operational descriptions only, which we deem to be the result of expert domain knowledge. The following sections describe more abstract versions of the patterns we have found in the repository of our case study.

6.2.1 Optional Signals and Events Sources

Design patterns associated with event generation, initialization of signals and transient signals have been modeled with conditional-execution Simulink subsystems. The interface for this type of patterns consists of:

1. An input signal of type trigger, rise/fall edge, pulse or function-call which activates the execution of a subsystem.

2. While active, the subsystem outputs a preset value, event signal or waveform, or makes present and modifies a pass-through dataflow.

These patterns exist mostly in the classes for subsystems of size 3B1L, 4B1L, 4B2L, 5B2L and 5B3L (refer to figure 6.1). These subsystems produce an initialization value or event trigger signal when they are activated by an input signal that is either of type enable, trigger, transition, or selected action. Figures 6.1a and 6.1b show task triggering event subsystems enabled by task scheduler modules. Figure 6.1c shows an enabled subsystem which reports two conditions for
6.2 Semantic Entailment of Model-Patterns

a selected mode of operation in a component of the modeled system, and figure 6.1d depicts a subsystem which generates the asynchronous initialization signal for a hardware controller.

Figure 6.1: Enabled Signals and Events

6.2.2 Synchronous Events Generators

The models in the repository of our case study capture the real-time requirements of a control system with hard deadlines. Practitioners usually model these types of requirements with sets of core synchronized signals, which change or are updated within very small time scales and trigger regular events within strict deadlines. They make explicit the regularity, stability and relative timing of these signals, and most likely standardize their names and assemble them in reasonably portable formats, that is subsystems, for reuse in additional modeling projects. We have used this highly conceptual modeling approach to guide the use of MoSART to identify model-patterns with these
6.2 Semantic Entailment of Model-Patterns

characteristics, and have uncovered a number of exemplars in the studied repository.

Figure 6.2a shows the generation of three synchronized event signals from a common time baseline: (a) a hardware controller initialization event, (b) a power-up powertrain command event, and (c) a periodic event which can be distributed to other components of the modeled system. Figure 6.2b depicts the state-based processing of input conditions for the generation of two output signals which trigger events with a fixed relative frequency.

Figure 6.2 depicts subsystems which produce a set of synchronous events upon 1) the application of a common time-base to subsystems generating signals related by a fix timing structure (figure 6.2a), or 2) the processing of delay-separated input signals (e.g., In1 and In2) according to an internal state, maintained by a Stateflow chart, and the computation of event signals which respond to a set of input conditions determined by a subsystem insertion environment (figure 6.2b).

(a) Generation of Synchronous Events  
(b) Processing of Delay-Separated Signals

Figure 6.2: Synchronous Events

6.2.3 Optional Execution Enablers

In this pattern, modes of operation, special real-time environmental conditions, and computation of specific internal states can activate transient functionality and alternative system configuration in an embedded system. This on-demand type of functionality can be necessary to fulfill, for example, stringent requirements for a system’s enhanced safety, resilience functionality that operates a
6.2 Semantic Entailment of Model-Patterns

system in a degraded mode and high efficiency or throughput modes. An effective way to model these conditions, for easy identification and portability, consists of encapsulating the entities implementing the alternative types functionality in triggered, action or enabled subsystems. The type of subsystems named **If Action Subsystem**, explained in section 6.1.3), are typical exemplars of optional subsystems we have found in the analyzed repository.

Once the optional functionality has been modeled as described above, *optional execution enabler subsystems* encapsulate the decision blocks, function calls, event generators and other entities that model the logic for activating the optional subsystems. We have captured some of these conceptual patterns in the repository of our case study, for example:

Figure 6.3a shows the encapsulation of a controlled-execution subsystem which computes engine geometry-related signals based on a number-of-cylinders parameter. A Function-Call Generator block, whose call generation logic is implemented in an S-function [55], activates the execution of this subsystem. Figure 6.3b shows the encapsulation of the decision structure to execute alternative subsystems based on an input signal, which in turn is conditionally executed from a containing subsystem. Each alternative subsystem sets or resets a specific mode of operation.

![Figure 6.3: Optional Subsystem Execution](image)

(a) Function-Call-Driven Subsystem Execution

(b) Enabled Optional Subsystem Execution

Figure 6.3 shows encapsulation approaches of optional-execution schemes for subsystems. The *Process Level Subsystem* block in figure 6.3a executes only when it receives the activation signal
6.2 Semantic Entailment of Model-Patterns

from a function-call generator block. Subsystems SetOutputOn or SetOutputOff, in figure 6.3b, execute upon the value in an input signal and the activation of the container subsystem by an external signal of type block selection. This latter subsystem allows the assembling of hierarchies of signal-generation, because it is optionally executed, and the optional subsystems it activates can themselves contain optional-execution blocks.

6.2.4 Special Waveform Generators

The models in the repository of oout case study represent a typical exemplar of a cyber-physical module embedded in a major ensemble of mechanical and electronic components, which measures and reacts to a variety of physical stimuli. Simulation of these conditions requires the modeling and generation of generic or assembly-specific waveforms which will be applied in multiple system operation and testing scenarios. Encapsulation of the signal generation mechanism into one or more subsystems provides consistent, verifiable and reusable modeling of waveform generation entities.

The following are typical waveform generator subsystem found in this repository:

Figure 6.4: Special Waveform Generation

Figure 6.4a shows the encapsulated computation of a angle variable and the generation of an event, both depending on an rpm signal. Figure 6.4b shows the block logic for integrating an input
6.2 Semantic Entailment of Model-Patterns

signal varying between set maximum and minimal saturation levels.

Figure 6.4 shows the encapsulated generation of special signals, in single subsystems, for easy reuse in the simulation of multiple control paths (i.e., rings) of the modeled system. In figure 6.4a, a Speed to Angle converter subsystem integrates an \textit{rpm} signal to produce discrete crank angle values, which in turn are interpolated/extrapolated by a table-lookup subsystem to approximate a continuous signal. Figure 6.4b shows a library subsystem, whose algebraic and signal threshold detection blocks work in conjunction to integrate an input signal, saturated at predefined minimum and maximum levels.

6.2.5 Task Schedulers

The \textit{de facto} device in the programmable control of the multiple electrical and mechanical modules of modern transportation vehicles is the Electronic Control Unit (ECU). These systems can have dozens of networked ECUs that sense a large number of physical variables representing timing, variation rates and physical limits. ECUs compute real-time responses to those sensed values and drive complex actuator mechanisms. The use of virtual modules (e.g., virtual powertrain models), commonly referred as plant models, is an effective way to model control software for the ECUs. This approach allows developers to capture and document the requirements for ECU software in models, which can be simulated and tested before generating the actual programming code.

In such an environment, it is very useful to abstract and encapsulate the timing and initiation of the tasks modeled for the virtual plant and the control modules into a Task Scheduler (TS) subsystem. The overall control of the tasks is inherently hierarchical. For example, in our case study, a high-level scheduler initiates tasks for the virtual powertrain (i.e., the plant models) and a more specialized scheduler initiates tasks for the control paths (i.e., rings) specific to the modeled system. Figure 6.5a shows a powertrain task scheduler subsystem embedded in a Simulink model intended for testing and simulation purposes. It emulates the initiation of system-wide events such as engine ignition and power request mode (i.e., gas fuel or electric motor) and transfers them to a more specialized scheduler. Figure 6.5b shows a subsystem that depicts the timing and nature of one-shot behaviors.
6.2 Semantic Entailment of Model-Patterns

and periodically-initiated tasks specific to the modeled system.

Figure 6.5 shows subsystems which implement two different levels of a hierarchy of task schedulers. The subsystem in figure 6.5a allows the manual emulation of a main task scheduler in a test environment. Figure 6.5b shows a specialized task scheduler for the modeled system as reused in the simulation of one of its control paths (i.e., rings).

6.2.6 Macro-Event Schedulers

A common paradigm in the modeling of a cyber-physical system involves encapsulating into a single entity (i.e., subsystem) 1) a number of concurrent or synchronous conditions for the generation of an event, 2) certain system state conditions necessary to trigger an event, 3) a series of computations that set the conditions that surround the event to trigger, and 4) a set of synchronous triggered signals, representing atomic events, which make up the generated event. Another view of the generation of this macro-event is to consider the encapsulating entity to be a task scheduler with state and reactive capabilities. In general, although the structure and properties of this entity may be complex, they are deemed sufficiently common in modeling different aspects of one or more targeted embedded systems, hence, they are good reusable designs. The subsystems in the repository of our case, which model the initiation of tasks in a virtual powertrain, encapsulate some complex event schedulers.

Figure 6.6 depicts a subsystem named Event Scheduler which models the generation of a macro-event by 1) processing a series of sensed signals from the engine, 2) updating some monitoring and driver signals which set the conditions for the atomic events to occur, and 3) generating a series of synchronous atomic events for the engine’s cylinders and other components. Figure 6.6 shows a subsystem which models the triggering of a macro-event, by computing input conditions, according to an internal state, and scheduling a series of synchronous atomic events.
6.2 Semantic Entailment of Model-Patterns

Figure 6.5: Task Scheduling

(a) Main Task Scheduler

(b) Specialized Task Scheduler
6.2 Semantic Entailment of Model-Patterns

spSearch computed 62 occurrences of this subsystem in the analyzed repository with the same property values, that is, NCHC = 39D927C, TCHC = 8DA0F, size = 46B43L. We confirmed, by querying all the repository’s cluster classes on MoSART, that this subsystem was embedded in every control ring of modeled system and generated common events for a virtual powertrain set of models.

![Event Scheduling](image)

Figure 6.6: Event Scheduling - Macro Event Generation

6.2.7 Component Testing and Integration

An effective approach to model-based testing and integration of real-time embedded systems consists of encapsulating all the essential waveforms, as well as their scheduling, which will be required in one or more classes of simulation scenarios, into one or more reusable entities (i.e. subsystems) with high portability properties. Additionally, integrating interfaces with hardware elements and
6.2 Semantic Entailment of Model-Patterns

other participating software modules (e.g., proprietary software architecture, AUTOSAR [8], etc.) into common high-level subsystems, greatly facilitates the visualization, simulation and testing of open- or close-loop paths of sensing and control. It supports, for example, the sound isolation and diagnostic, during simulation, of key physical variables which enable the proper operation of a system. Furthermore, it allows cleaner validation paths for designed modules, which interact with each other, involved in managing those variables. The repository of our case study contains subsystem that address these concerns.

Figures 6.7a and 6.7b show a waveform generation module for the different tests of the analyzed repository, and the insertion of the generated signals into a main task scheduler, respectively. Figure 6.8 depicts the sensing and control path (named ring in the figure) for monitoring and reporting physical signals under specific operation conditions. Figure 6.7 shows subsystems for the generation and injection of test signals into a main task scheduler subsystem.
6.2 Semantic Entailment of Model-Patterns

Figure 6.8: Physical Variable Sensor Ring
6.3 Analysis of Additional Results

The Signal Builder block in figure 6.7a is an interactive interface for the configuration of event test signals. Figure 6.7b shows the injection of these signals to a main task scheduler, for subsequent transfer to a specialized task scheduler. Figure 6.8 shows a subsystem which models the integration of all the components in the control path corresponding to a Physical Variable Ring. In addition to the components that build up the control functions in this ring, the subsystem also encapsulates a Function Scheduling, which provides initialization and periodic events for the ring’s simulation.

6.3 Analysis of Additional Results

While completing our model-pattern elicitation and entailment processes on the repository of our case study, in addition to the patterns identified in sections 6.1 and 6.2, we captured some general trends in its subsystems in relation to model-pattern inferences and potential reuse scenarios. They refer to the form, properties and frequency of the subsystem occurrences. We discuss in the following sections 1) general trends of our findings, 2) potential reuse scenarios for the inferred model-patterns, and 3) comparisons about the ability to observe patterns with our tools on the industrial repository of our case study and two public domain sets of Simulink models, namely, a generic model set for the control functions in an automobile (i.e., AVS = Advanced Vehicle Simulator) and a model set of miscellaneous exemplars released with a version of the Matlab-Simulink software (i.e., R2015a).

6.3.1 Subsystem Size Trends in the Model-Patterns

The industrial Simulink repository we have analyzed is the result of the modeling practice, in a structured development organization, to capture the functional, testing, simulation and implementation requirements of a realistic system. The models have been developed under a traditional plant-control architecture (refer to figure 2.2a), where the plant models implement a well-developed virtual powertrain [47] set of model-subsystems. The models in this repository are compliant with
6.3 Analysis of Additional Results

and use many of the subsystems that are part of this virtual powertrain. The subsystems follow func-
tional decomposition approaches common in the modeling of software for embedded systems [65].
In summary, the regularity of this environment greatly increases the expectation of finding sound Simulink design patterns for embedded systems in this model set. Using SpSearch and MoSART, we have identified correlations between the size of its subsystems and the type of system function-
ality they model or the intended support they provide to the modeling environment. They reflect planned modeling efforts with considerations about subsystem reuse and integration. The following are types of these subsystems that by themselves represent patterns of modeling activities:

**Block-Only.** These subsystems have a size-template of \( mB0L \). They exist in sizes 0B0L to 29B0L and encapsulate text-only, functional blocks only and subsystems without any connecting dataflows. They play central roles as modeling artifacts to:

1. Provide explanatory notes in a Simulink model and link to external documentation.

2. Serve as a single access-point to custom-code (e.g., S-functions, load calibration data).

3. Integrate control subsystems (i.e., rings) and their execution environment (i.e., task schedulers) for testing, simulation and diagnostic purposes. Provide single-block encapsulation for code generation.

4. Provide libraries of custom blocks for special functions such as calibration, diagnostic, special constants, signal delays, timers and counters, discrete filters and signal manipulators (e.g., bus, mux, selector).

5. Provide libraries of functional blocks for mathematical functions, logical operators, arithmetic operators, integrators and accumulators.

6. Channel read/write operations from/to common data stores.
6.3 Analysis of Additional Results

**Action-Reaction.** These subsystems model an optional or temporary action that when activated results in triggering or applying a previously-defined reusable reaction. Table 6.6 lists types of elementary reactions modeled in subsystems of size = 3B1L, which make 69.5% of the 26,692 subsystems in the repository. Implementing micro reactions, in small subsystems, supports well the modeling-by-reuse of the powertrain entities that operate large numbers of independent signals in an identical manner. These small-size reaction subsystems are easier to aggregate in the Simulink diagrams representing entities that can increase in size without changing their intended functionality or meaning, for example task schedulers. They are exceptionally useful, as reusable components, in implementing micro-steps in subsystems containing some complex decision logic, for example subsystems encapsulating the determination of state changes in the hybrid-fuel system.

<table>
<thead>
<tr>
<th>Type of Reaction</th>
<th>TCHCs</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate function call</td>
<td>F436C, F549C</td>
<td>18132</td>
</tr>
<tr>
<td>Read constant value</td>
<td>F3F84, F3D2C</td>
<td>276</td>
</tr>
<tr>
<td>Insert a dataflow</td>
<td>F4754</td>
<td>97</td>
</tr>
<tr>
<td>Load a subsystem with a constant value</td>
<td>12D2C, 13754, 38784, 76F84</td>
<td>25</td>
</tr>
<tr>
<td>Connect a subsystem</td>
<td>18A754</td>
<td>24</td>
</tr>
<tr>
<td>Read a data store</td>
<td>F404C</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.6: 3B1L Subsystems - Types of Reaction

**Virtual Powertrain.** These are large-size subsystems (i.e., sizes greater than 12B10L) that occur relatively frequently in the repository (i.e., more than 50 occurrences). They are the exception to the general trend existing in this repository, that is, small subsystems with high frequency of occurrence and large infrequent subsystems. Furthermore, they are complex subsystems and their sizes cluster around a specific number of occurrences, as shown in table 6.7. These occurrence-patterns highlight the embedding of these subsystems in every control ring of the modeled system and delineate the modeling of a common virtual powertrain (i.e., plant models) interacting with those rings. In summary, they model the scheduling of events for a virtual powertrain, as well as the sensing and actuation on standard signals for the control of an embedded system in a vehicle.

This insight into the size and frequency of occurrence of large subsystems in this repository, constructed with the query and subsystem visualization facilities of MoSART, and summarized in
6.3 Analysis of Additional Results

Table 6.7, allows us to visualize an architecture of model-subsystems (i.e., the virtual powertrain) which can be potentially embedded in a large variety of Simulink modeling projects related to embedded systems of the same kind.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Size</th>
<th>TCHC</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine Movement Detection</td>
<td>12B12L</td>
<td>11308FC</td>
<td>63</td>
</tr>
<tr>
<td>Engine Events One Shot</td>
<td>15B12L</td>
<td>BB2E5C</td>
<td>63</td>
</tr>
<tr>
<td>NVM Initi</td>
<td>17B22L</td>
<td>15367E4</td>
<td>62</td>
</tr>
<tr>
<td>6p25ms</td>
<td>19B18L</td>
<td>16EFB44</td>
<td>62</td>
</tr>
<tr>
<td>EngineState</td>
<td>23B22L</td>
<td>E60A3C</td>
<td>62</td>
</tr>
<tr>
<td>Cyl Index Generator</td>
<td>35B23L</td>
<td>21E6C28</td>
<td>62</td>
</tr>
<tr>
<td>System Events One Shot</td>
<td>35B28L</td>
<td>2774190</td>
<td>62</td>
</tr>
<tr>
<td>Engine Event Scheduling</td>
<td>43B77L</td>
<td>50E3C38</td>
<td>62</td>
</tr>
<tr>
<td>Event Scheduler</td>
<td>46B43L</td>
<td>39D927C</td>
<td>62</td>
</tr>
<tr>
<td>Engine State Machine</td>
<td>50B41L</td>
<td>33F3ED4</td>
<td>62</td>
</tr>
<tr>
<td>System Events One Shot</td>
<td>60B48L</td>
<td>43A2740</td>
<td>62</td>
</tr>
<tr>
<td>Engine Events One Shot</td>
<td>84B70L</td>
<td>621F560</td>
<td>62</td>
</tr>
<tr>
<td>Controller Init</td>
<td>20B19L</td>
<td>145C9F4</td>
<td>59</td>
</tr>
<tr>
<td>Task Scheduler</td>
<td>165B132L</td>
<td>8D55E8C</td>
<td>59</td>
</tr>
<tr>
<td>Dummy OS</td>
<td>243B241L</td>
<td>11776EB8</td>
<td>89</td>
</tr>
<tr>
<td>Operating System Tasks</td>
<td>362B241L</td>
<td>FB459C4</td>
<td>89</td>
</tr>
</tbody>
</table>

Table 6.7: Virtual Powertrain Subsystems

6.3.2 Pattern-Reuse Scenarios Analysis

The scope of the models in the repository of our case study is the emulation of a number of control paths for a major embedded system in a vehicle, with the ulterior purpose of generating ECU code (i.e. C-code) from them. In a broader context, these models are embedded-in and reuse the architecture of Simulink subsystems corresponding to a virtual powertrain system [47]. In this architecture, the subsystems of specific Simulink models:

1. Replace the algorithms and software to implement, for example, communication protocols, interfaces and operating system tasks running on hardware controllers (i.e., ECUs) and controller area networks (CAN).

2. Emulate, with an adjustable fidelity, the properties and behaviour of the physical variables in an automobile’s engine and transmission system (i.e., plant models).
6.3 Analysis of Additional Results

3. Implement interfaces between i) plant (i.e., virtual powertrain) and control (i.e., rings) models, and ii) plant-control and layered system architecture models.

In building this virtual powertrain with Simulink models, engineers have pursued improved quality, more cost-effective designs and a shorter time-to-market for vehicle propulsion systems. Practitioners and researchers working in this kind of environment generally agree that the use of Simulink models i) helps reduce the use of expensive Hardware-in-the-Loop (HiL) components and prototype vehicles by enabling the use of Software-in-the-Loop (SiL) and Model-in-the-Loop (MiL) components, ii) allows for more flexible measurement, calibration and diagnostic of physical variables, and iii) serves as a repository of easy-to-trace and verify graphical specifications for the generation of ECU code (e.g., C-code) [44, 47, 58, 81]. These guidelines are applicable to the analyzed repository and give us a reference for proposing reuse scenarios for the extracted patterns.

<table>
<thead>
<tr>
<th>Subsystem Names</th>
<th>TCHC</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled Function-Call Generator[n]</td>
<td>F436C</td>
<td>4274</td>
</tr>
<tr>
<td>Triggered Function-Call Generator[n]</td>
<td>F549C</td>
<td>13858</td>
</tr>
<tr>
<td>Clear Crank CylEvent</td>
<td>F3F84</td>
<td>154</td>
</tr>
<tr>
<td>If Action Subsystem[n]</td>
<td>F3D2C</td>
<td>112</td>
</tr>
<tr>
<td>Report Test Fail, Report Test Pass, If_CNG, SetOutputOff, SetOutputOn, OutputsNotDriven</td>
<td>12D2C</td>
<td>6</td>
</tr>
<tr>
<td>If Action Subsystem[n], Set Trigger1, Switch Case Action Subsystem, Function-Call Subsystem</td>
<td>F4754</td>
<td>97</td>
</tr>
<tr>
<td>Report Test Fail, Report Test Pass</td>
<td>18A754</td>
<td>24</td>
</tr>
<tr>
<td>Write Offsets, Write Zeros, If Action Subsystem1</td>
<td>38784</td>
<td>3</td>
</tr>
<tr>
<td>Clear Subinterrupt Array</td>
<td>76F84</td>
<td>1</td>
</tr>
<tr>
<td>If Action Subsystem</td>
<td>F404C</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.8: Sub-classes of the 3B1L Class

**Subsystem Encapsulation Reuse.** In this scenario, a small subsystem is inserted as a pseudo functional block in a containing subsystem to expand its structure, without changing its intended modeling meaning. These subsystems usually belong to a subclass of a cluster class, model functionality that is deemed “atomic”, and have a simple interface to other subsystems (e.g., one action
block and one output port, or one input and one output ports). Class 3B1L and its sub-classes, as shown in table 6.8, contain typical subsystems for this purpose.

**Subsystem Ensemble Reuse.** In this scenario, reusing a specific subsystem in a new model requires the mandatory insertion of a set of either encapsulated or interfacing subsystems. In more abstract terms, this situation refers to reusing the modeling of a complex entity, which has been partitioned into virtual subsystems either because the entity itself is reusing “atomic” functionality, it is easier to manage as an ensemble of virtual subsystems, or a portion of its modeling overlaps with another complex entity subsystem-wise. An exemplar of this type of reuse is subsystem *Engine State Machine*, size = 50B41L, TCHC = 33F3ED4, depicted in figure 4.6. This subsystem uses a series of Stateflow charts to 1) encapsulate the running-states of an engine in an *EngineState* chart, 2) selectively enable the reporting of engine-statuses with multiple instances of an *EventHandler* chart, and 3) determine an engine’s speed zone value using a *EngSpdZoneDtermination* chart encapsulated in an *Engine Speed Zone Determination* subsystem.

**Subsystem Architecture Reuse.** This is a pervasive reuse scenario in the analyzed repository, as every Simulink model for a control ring reuses the architecture of subsystems that models a virtual powertrain. The subsystems in table 6.7 are reused in every ring model of the repository, with the same relative containment and composition structure.

In general, the structure of subsystems found in the studied repository has been developed with architectural reuse in mind. The main approach used in the development of each of the control rings involves modeling it in a manner compatible with the subsystems of a virtual powertrain. Finally, the ring and powertrain subsystems are merged into a single Simulink model.

### 6.3.3 Comparison with Public Domain Model Sets

Our analysis of the industrial repository in our case study has uncovered a set of patterns, of varying abstraction level, that occur in a structured modeling environment. Many of these models are
6.3 Analysis of Additional Results

built within an architecture of subsystems that fosters re-usability, supports stringent system testing and integration activities, and serves as effective graphical documentation of requirements. Using spSearch and MoSART, we have elicited model-patterns that reflect the sound, industrial-strength, modeling and development of an embedded real-time system. In this section, the pattern results we have generated for this repository serve as a benchmark to evaluate the content of two public domain repositories, namely 1) the Advanced Vehicle Simulator (AVS) research prototype set of Simulink models for the development of a hybrid electric vehicle propulsion system (HEVPS) [51], and 2) and the Simulink Toolbox Release R2015a (STR-R2015a) model exemplars [56]. Table 6.9 shows core indicators, produced with spSearch and MoSART, that profile each of these three repositories. Using the values in this table, we provide a comparative analysis of these repositories applying the criteria 1) repository content, and 2) potential for Simulink model-pattern inference.

<table>
<thead>
<tr>
<th>PROPERTY</th>
<th>Industrial Rep.</th>
<th>AVS</th>
<th>STR-R2015a</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of SLOC</td>
<td>5,187,4896</td>
<td>828,560</td>
<td>1,620,708</td>
</tr>
<tr>
<td>No. of Models</td>
<td>319</td>
<td>72</td>
<td>908</td>
</tr>
<tr>
<td>No. of Subsystems</td>
<td>26,692</td>
<td>4,386</td>
<td>4,034</td>
</tr>
<tr>
<td>No. of Blocks</td>
<td>220,722</td>
<td>42,456</td>
<td>25,833</td>
</tr>
<tr>
<td>No. of Dataflows</td>
<td>155,007</td>
<td>34,154</td>
<td>19,101</td>
</tr>
<tr>
<td>No. of Cluster Classes</td>
<td>442</td>
<td>379</td>
<td>292</td>
</tr>
<tr>
<td>Largest Subsystem</td>
<td>758B589L</td>
<td>256B234L</td>
<td>70B68L</td>
</tr>
<tr>
<td>Smallest Subsystem</td>
<td>0B0L</td>
<td>0B0L</td>
<td>0B0L</td>
</tr>
<tr>
<td>Most Frequent Subsystem</td>
<td>3B1L 18,545 occurrences</td>
<td>2B1L 754 occurrences</td>
<td>0B0L 575 occurrences</td>
</tr>
<tr>
<td>Least Frequent Subsystem</td>
<td>758B589L 1 occurrence</td>
<td>Multiple sizes 1 occurrence</td>
<td>Multiple sizes 1 occurrence</td>
</tr>
<tr>
<td>No. of Stateflow charts</td>
<td>1655</td>
<td>0</td>
<td>400</td>
</tr>
<tr>
<td>Largest Stateflow Chart</td>
<td>67 states</td>
<td>N/A</td>
<td>38 states</td>
</tr>
<tr>
<td>Smallest Stateflow Chart</td>
<td>0 states</td>
<td>N/A</td>
<td>0 states</td>
</tr>
<tr>
<td>Max. Subsystem Nesting</td>
<td>9 levels</td>
<td>9 levels</td>
<td>6 levels</td>
</tr>
</tbody>
</table>

Table 6.9: Profiles of Simulink Repositories in Study

We use this industrial repository as a reference for the comparative analysis because:

- Our exploratory analysis of its content showed a trend of higher correlation between types of models and patterns, in comparison with the public domain repositories. This can be partly
6.3 Analysis of Additional Results

attributed, in addition to more problem-specific modeling practice, to its larger, cleaner and more stable (i.e., version-controlled) source code corpus of Simulink models.

- It contains a significant amount of structural and architectural reuse across subsystems. This is the result of planned and coherent modeling of an embedded system, subject to industrial-strength functional and non-functional requirements.

- It contains a higher amount of structural and behavioural modeling content. This reflects a more rigorous, architecture-based, requirement-driven, and complete approach to model an existing embedded system.

Our summarized analysis focuses on identifying the levels of subsystem occurrences, reuse and content coverage found in the public domain repositories in comparison to the same levels detected in our reference repository.

The AVS repository is, in general, a set of Simulink-block libraries to simulate specific physical variables of components used in the design of hybrid electric vehicle prototypes (e.g., engine, gearbox, fuel converter, battery, controls) [51]. Its models mainly contain elements of plant models in a plant-control modeling architecture (refer to figure 2.2a). They usually contain, at the root level, ensembles of connected subsystems that represent idealized interconnections of the components of an electric vehicle for simulation purposes, or abstract templates for encapsulating simulation or testing activities. They do not contain any Stateflow charts, i.e., all subsystems are stateless, and modes of operation and state of components are not modeled. Additionally, the following subsystem-size-related patterns are present in the AVS repository:

- The root subsystems of models cluster around sizes 1) 5B10L (12 occurrences), which are templates to encapsulate simulation and testing subsystems, and 2) 25B35L (18 occurrences), which are interconnection templates for subsystems modeling vehicle components. These templates reuse subsystems that encapsulate the computation of physical signals in idealized components of a hybrid propulsion vehicle. These patterns strive to provide users with open suggestions about
6.3 Analysis of Additional Results

modeling for this type of vehicle.

- The most frequent subsystems have size 2B1L (754 occurrences), and are used either to encapsulate a disconnected dataflow or write the data transferred by a data flow to a virtual storage (e.g., array, time series). Subsystems of size 0B0L (657 occurrences) are also frequent, and they encapsulate context-sensitive explanatory notes to be embedded in specific subsystems. These patterns emphasize the open prototype nature of the AVS repository, as many signals are left for a to-be-determined use and subsystem-usage explanatory notes are found in many subsystems.

- The largest subsystems have size 256B234L (2 occurrences), and encapsulate sets of disconnected subsystems, including their inputs and outputs, which compute subsets of measurements from sensed physical variables. This pattern, in conjunction with block-only subsystems of size mB0L, emphasize the open-library nature of the AVS repository.

In contrast, the root subsystems of the models in the industrial repository (sizes 12B7L to 54B48L) encapsulate the subsystems and signals required for a defined system control, testing, or integration task. The most frequent subsystems have size 3B1L, and encapsulate either an optional triggered event, initialization constant or dataflow. And the two largest subsystems, of sizes 758B598L (1 occurrence) and 587B396L (1 occurrence), encapsulate the initiation of multiple tasks that have to be scheduled in a modeled operating system environment. These subsystem-sizes and frequencies of occurrence are a reflection of 1) high subsystem reuse already embedded in this repository, 2) compliance with a pre-established architecture of subsystems, and 3) a completely different usage-focus for the models, that is software generation for ECUs, relative to the AVS repository’s objectives.

The STR-R2015a repository contains an assortment of Simulink-model exemplars from multiple domains and some model templates (e.g., documentation, empty subsystems). Among the domains addressed by these models are aerospace, automotive, heavy machinery, physics, mathematical modeling, signal analysis, weather modeling, and game simulation. Some subsystem-size-related patterns present in this repository are:
6.3 Analysis of Additional Results

- There exists a high number of root-subsystem-only or low nesting (i.e., maximum 2 levels of nesting) models (737 occurrences, sizes 2B1L to 63B34L). This confirms the tutorial focus of the model-exemplars in the STR-R2015a repository, where system decompositions and aggregations are not a priority. In this context, single-purpose subsystems encapsulate sets of atomic blocks, elementary subsystems, and dataflows to model a specific function and its output(s).

- Consistent with the previous pattern, the two largest subsystems, of sizes 69B68L and 70B69L, are second-level nesting subsystems encapsulating a large number of atomic components to emulate signal reorganization components. This is a common pattern for other large subsystems in this repository, in which relatively flat models contain unique exemplars that prototype a small and isolated modeling situation, and their subsystems are not representative of any architecture of models.

- The most frequent subsystem size is 0B0L (575 occurrences), where most of these subsystems are empty and not contributing to any visible modeling action. This indicates a low priority on any subsystem integration focus of the model exemplars in the R2015a toolbox.

In general, the amount of reuse already embedded in STR-2015a of, for example, composite or architectural components, is very low. This contrasts with the strict plant-control architecture of models existing in the industrial repository.

In conclusion, in this section we have highlighted the overall structural differences between an industrial repository and two public-domain model sets: AVS and STR-R2015a model sets (refer to table 6.9), in order to clarify their scope and overall purpose. Using the tools of our framework, we have extrapolated some semantic interpretations about the subsystems in each repository, in order to extract some initial knowledge from the public domain repositories that could then be applied to further understand the qualities of the industrial one. Since we have created the tools to merge classes and select specific cluster classes (i.e., spClassMerge and spClassSelect) from any number of repositories, this initial insight can be very useful to study potential patterns, for specific use cases, that may fall in preferential subsystem-size ranges.
6.3 Analysis of Additional Results

6.3.4 Comparative Evaluation of the Subsystem Clustering

Clustering of subsystems by specified metrics of size and topology is one of the central ideas in our framework to elicit Simulink model-patterns. We believe that a system of cluster classes with these properties enables the creation of powerful tools (i.e., MoSART) which support an extensive range of pattern inference modalities such as statistical trends, drill-down paths, predefined exemplar occurrences, stereotype subsystems, and others. The ability to implement closed operations (e.g., union, intersection, arbitrary aggregation, sub-classing) on those classes and their elements (i.e., subsystems) allows the derivation of classes with the same types of properties as their originators. This is a core concept in directing the elicitation process towards more abstract or semantic model-patterns. Experts can reduce any group of these classes, in multiple ways, to equivalent sets of classes that they can establish as appropriate representations of model-patterns in their domain.

Using these concepts as a reference, in this section we evaluate the performance, coverage and pattern-elicitation capabilities of our tools in comparison to clone detection tools. Researchers in this field have approached the challenge of discovering model-patterns in repositories of Simulink models by clustering subsystems into graph- or text-based clone classes. This type of class groups subsystems that are structurally similar (i.e., either graph-like or text-based similarity) within a certain percentage, which ranges between an established minimum and 100%. Direct interpretation of the subsystems clustered for each individual class provides a notion of a model-pattern. Operations between classes have not been implemented in any of these clone-detection-based tools to the present time.

Our evaluation includes comparisons of the empirical results produced, on one side by spSearch, and on the other side by the Simulink clone clustering tool SIMONE, which is an adaptation for the Simulink language of the text-based code clone detection tool NICAD [19]. The study in [4] compares the results produced by SIMONE against the ones produced by the graph-based clone detection tool ConQAT [22], and concludes that it 1) uses a well-defined structural granularity for finding clones (i.e., subsystems), which is more useful system-wise than detection of clones on
6.3 Analysis of Additional Results

structural-fragments performed by ConQAT, 2) produces a higher percentage of clone recall due to its ability to detect near-miss clones, and 3) has been demonstrated to detect Simulink clones in repositories of models much larger than the ones examined by ConQAT. These characteristics make SIMONE a reference for comparison much closer to the capabilities of spSearch.

In our evaluation scenario, we applied spSearch and SIMONE to the industrial repository (refer to table 6.9 for detailed statistics on this repository). Next, we applied a script to the cluster-class files output by either tool 1) to match classes and subsystems, on subsystem nesting path and subsystem name, and 2) to report on unmatched subsystems. This task generated a list of 4-tuples with the format (SIMONE Class ID, SIMONE Class Similarity Percentage, spSearch Class ID, Subsystem Nesting Path & Name), and a separate list of unmatched subsystems. Table 6.10 provides details of clustering results obtained from the application of each tool to the same repository and the analysis of the lists of matched and unmatched subsystems.

<table>
<thead>
<tr>
<th>DESCRIPTION</th>
<th>spSearch</th>
<th>SIMONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem Clustering Type</td>
<td>Subsystem-Size Metric</td>
<td>Subsystem-Code Clone</td>
</tr>
<tr>
<td>Cluster Class Identification</td>
<td>$mBnlL$ Identifier</td>
<td>Positive Integer</td>
</tr>
<tr>
<td>$m = \text{No. of Blocks}$</td>
<td>$n = \text{No. of Dataflows}$</td>
<td></td>
</tr>
<tr>
<td>Total No. of Subsystems in Repository</td>
<td>26,692</td>
<td>26,692</td>
</tr>
<tr>
<td>Clustering time</td>
<td>40 sec.</td>
<td>192 min. 20 sec.</td>
</tr>
<tr>
<td>No. of Cluster Classes</td>
<td>442</td>
<td>289</td>
</tr>
<tr>
<td>No. of Clustered Subsystems</td>
<td>26,692</td>
<td>2,502</td>
</tr>
<tr>
<td>Subsystem Clustering Coverage Percentage</td>
<td>100%</td>
<td>9.37%</td>
</tr>
<tr>
<td>Largest Class</td>
<td>3B1L</td>
<td>1</td>
</tr>
<tr>
<td>No. of Subsystems in Largest Class</td>
<td>18,545</td>
<td>94</td>
</tr>
<tr>
<td>Smallest Class</td>
<td>Multiple</td>
<td>Multiple</td>
</tr>
<tr>
<td>No. of Subsystems in Smallest Class</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>No. of Classes with Smallest Size</td>
<td>185</td>
<td>111</td>
</tr>
<tr>
<td>Largest Clustered Subsystem Size</td>
<td>758B589L</td>
<td>Operating System Tasks (Class ID = 1). (Equivalent to spSearch’s 362B241L)</td>
</tr>
<tr>
<td>Smallest Clustered Subsystem Size</td>
<td>0B0L</td>
<td>XofY_FOM_Lib (Class ID = 46). (Equivalent to spSearch’s 1B0L)</td>
</tr>
</tbody>
</table>

Table 6.10: Industrial Repository Case Study. spSearch and SIMONE Clustering Results
6.3 Analysis of Additional Results

In a comparative analysis of the results outlined in table 6.10, we can highlight the following points:

- The spSearch classes contain subsystems that are identical under a given size metric, whereas the SIMONE clusters contain subsystems whose textual definitions fall within a given similarity range. These differences in clustering approach generate the following differences in clustering coverage:

1. spSearch generates a set of classes that constitute a full partition of all the subsystems in the industrial repository, that is, the subsystem coverage in the clustering process is 100%. SIMONE generates a set of classes, where each class must have at least two subsystems, which shows a low subsystem coverage (i.e., 2,502 subsystems, which is 9.37% of the total of 26,692 subsystems).

2. 42% of the spSearch classes contain a single subsystem. These classes, by definition, will not be flagged in SIMONE. In many instances we have discovered a great deal of information in these classes, as its single subsystem, usually large, represents a core architectural or modeling entity in the modeled system. 38.2% of the SIMONE classes contain two subsystems. These dual-subsystem classes can rarely be extrapolated to patterns due to the difficulty in assessing their statistical relevance.

3. In general, the spSearch classes are much larger than the SIMONE ones. This indicates, in cloning metric, that a wider range similarity is accepted in spSearch for subsystems to become a member of a given class. For larger subsystems, this may require additional subclassing operations (which we implement in MoSART) to isolate patterns. However, for small subsystems, this similarity range is very narrow. A typical example of this last scenario is class 3B1L (refer to table 6.6), which contains 18545 subsystems, where the high frequency of subsystem-occurrence allows the elicitation of interesting statistically-supported modeling patterns.
6.3 Analysis of Additional Results

4. spSearch classes with more than one subsystem, in a wide range of subsystem sizes (i.e., 0B0L to 362B241L), contain subsystems that SIMONE does not cluster into any class. The notorious case is spSearch class 3B1L, where none of its 18545 subsystems is clustered by SIMONE.

5. The size-metric-based definition of the spSearch classes allows the implementation of closed union, intersection and sub-classing class operations (i.e., the results are also classes defined by the same metric) by matching subsystems on the $mBnL$ identifier. The SIMONE classes, identified by positive integers, are very difficult, and perhaps not possible, to process with the mentioned operations to obtain classes with the same metric or meaning. In this respect, the spSearch classes are more amenable to further model-pattern elicitation via class operations.

- spSearch clustered the subsystems of the repository in an elapsed time which was two orders of magnitude smaller than the time employed by SIMONE to complete its clustering of the same subsystems. This can be partially explained by the fact that, to generate the cluster classes, spSearch traverses the subsystems of Simulink repository, in a bottom-up fashion from most- to least-nested subsystems, using an algorithm whose time-complexity is $O(n \lg n)$ ($n =$ total number of SLOCs in repository). Whereas SIMONE (via NiCad) uses code fragment comparisons, which tend to have complexities approaching $O(n^2)$, to generate the cluster classes.

- SIMONE classes containing subsystems with a similarity percentage greater or equal than 90% are completely contained within a single spSearch class. SIMONE classes having subsystems with low similarity percentage (i.e., 70%), overlap with different spSearch classes. This latter SIMONE scenario generates clusters with noticeably different subsystems, whose structure may not be amenable to model-pattern elicitation. A visible example is SIMONE class 1, which contains subsystems with similarity 70% or higher. This class contains highly dissimilar subsystems which spSearch identified as belonging to classes 1B0L, 4B0L, 6B1L, 7B2L, 9B4L, 11B6L, 12B7L, 13B8L, 14B8L, 14B9L, 16B10L, 16B11L, 17B12L, 18B13L, 19B13L, 20B14L, 22B16L, 23B17L, 24B18L, 26B20L, 26B21L, 28B22L, 29B24L, 31B25L, 37B31L, 335B223L,
and 362B241L. Although we can carry out some pattern elicitation on this class, it becomes evident that some sub-classing operations have to be performed by visualizing the contents of each of its subsystems. The visual cross-referencing of subsystem components needed to determine the existence of a pattern could represent a large manual effort in this situation.

In summary, we believe that the spSearch clustering offers many advantages over the code-clone-based clustering for model-pattern elicitation such as:

- It covers 100% of the subsystems in a repository. The set of cluster classes that it generates are a full partition of a Simulink repository. This represents a 100% availability of the repository for pattern elicitation purposes.

- It generates a complete set of cluster classes in a much smaller time than code-clone-based tools. In practice, this is a requirement for implementing a feasible iterative approach for discovering model-patterns, which is what we propose in this dissertation. Its processing time is scalable to very large Simulink repositories, for which we could still apply iterative pattern discovery processes.

- It supports powerful closed class operations which have a straightforward implementation. This concept has not been implemented, and it is not a simple or feasible task, in code-clone-based tools. These operations are central to the powerful interactive pattern-elicitation capabilities that user interface MoSART implements. They also allow the arbitrary merging of cluster classes from different repositories. This feature enables the elicitation of model-patterns by using sub-systems from multiple repositories without having to merge the repositories’ content and process the resulting text corpus with spSearch.

### 6.4 Results Validity Analysis

Our framework focuses on finding structures of Simulink subsystems and Stateflow charts that could be interpreted as useful modeling patterns by practitioners. To meet this objective in an expedited
and efficient manner, it avoids automating the measurement and comparison of detailed properties of subsystems and statecharts (e.g., block properties, specific dataflow configurations, state transitions, guards). We have assumed that realistic repositories (e.g., industrial repository analyzed in this chapter) contain a significant number of “entity regularities”, typical of sound and structured systems modeling approaches, which we can capture into patterns without examining their detailed definitions. In line with this assumption, the initial clustering and characterization of objects for establishing similarities and relations is rather coarse, however, highly efficient in terms of the execution times from data to pattern-structures, level of repository coverage, and versatility of the cluster-classes generated. This approach enables the fast and iterative analysis of very large repositories, with widely-varying contents, in a repository-agnostic manner. However, we must ensure that, regardless of a repository’s content, the tools and processes we have implemented allow practitioners to include details and model-scopes with enough resolution to produce useful, refined and ready-to-classify Simulink patterns. The nature of this environment requires considering the following threats to the validity of the empirical pattern-results [66] that we can obtain with our toolchain, as well as elaborating on approaches to mitigate their effects.

1. Repositories of unrelated models with very shallow subsystem-nesting depths (e.g., repository R2015a), that is, one or two levels, having subsystems that contain a large number of atomic functional blocks and connecting dataflows, may make it hard to identify patterns of subsystems by clustering them by size and matching their topological properties using hash codes. This scenario would mask the effectiveness of our framework, as we would be attempting to identify patterns of subsystems and stateflow charts that most likely do not have the desired design value, or do not represent a common modeling approach of systemic entities, or their extrapolations to patterns would lead to unsound subsystems from a design viewpoint. To mitigate this effect, we cross-referenced and visualized the classes and subsystems in this situation in multiple ways with the querying and display facilities of MoSART and the class operations described in section 5.2. We also compiled counts of equal subsystem-size and hash code instances to estimate
6.4 Results Validity Analysis

the re-occurrence of subsystems as an indication of an existing model-pattern candidate. We considered that extending the analysis beyond this point for these subsystems would require exhaustively validating them against targeted use cases to, for example, justify abstracting them (i.e., incrementally altering boundaries and modeling intent) into higher-level model-patterns.

2. Our pattern discovery framework clusters subsystems into classes defined by a metric of subsystem size. These classes form the basis for the identification of model-patterns, and sub-classing operations on hash codes isolate the different topologies of dataflows that may exist in subsystems of equal size. Subsystem membership to any of these classes changes only if blocks or dataflows are added or deleted. From a pattern-detection viewpoint, changes to block properties do not change a subsystem’s membership. Furthermore, we have assumed that changes in these properties are not significant enough to change the modeling intent associated with a subsystem, which is only affected by object additions and deletions, hence, they do not trigger pattern changes. We provide extensive subsystem querying and visualization facilities in MoSART to mitigate any negative effects that this assumption might cause on the validity of the pattern elicited with our framework. Practitioners can quickly display and analyze with MoSART any subsystems and block properties in a repository, in groups selected by cluster-class, hash code, model and subsystem path, that might require extended resolution for the proper identification of model-patterns.

3. We have applied our framework to three types of repositories, namely, industrial, experimental and toolbox-style. Our results have been richer and more complete on the industrial repository, as expected, as it represents the expert modeling of a realistic system, in a specific application domain, including structural (i.e., subsystems and blocks) and behavioural (i.e., Stateflow charts) entities. The experimental repository does not contain any Stateflow charts and the toolbox-style repository does not represent the modeling of one specific system. Therefore, attempting to match pattern-results originating in any two of these repositories would likely fail. However, if we could hypothetically compare the pattern-results obtained from the industrial repository of
6.4 Results Validity Analysis

our case study with the ones resulting from a very similar repository, and we could not quickly identify any matches, then the portability of our framework would be in doubt. This situation could be the result of applying completely different pattern-elicitation workflows with our MoSART tool or generating completely different sets of cluster-classes for two very similar repositories. To mitigate any negative effects that this scenario could cause, we have implemented tools (i.e., spClassSelect and spClassMerge) to quickly and arbitrarily select and merge elements, on identical subsystem sizes, from classes in different repositories to build classes for subsystem comparative analysis. These tools expedite the processes to match results, backtrack on pattern-discovery workflows, reconfigure repositories and reiterate pattern searches with spSearch.
Chapter 7

Summary and Conclusions

This chapter concludes our exploration of algorithms, tools and techniques to discover Simulink model-patterns. We summarize our contributions, present a consolidated view of our results and elaborate on future lines of research on this topic.

7.1 Summary

This dissertation has introduced a framework, comprising efficient algorithms, processes, and a software toolchain, to discover modeling patterns from the source code definitions of large repositories of Simulink models. The framework is largely automated, facilitates the application of expert domain knowledge for making decisions about patterns, and supports an iterative approach to pattern discovery that users can conceptualize as having three major phases of model-pattern elicitation, namely, I) search/clustering, II) inference, and III) refinement. Two groups of software tools have been developed to allow practitioners to complete the model-pattern inference processes: 1) spSearch and a set of ancillary utilities which implement the automated tasks that define phase I, and 2) the MoSART user interface which supports the interactive completion of the pattern inference and refinement tasks corresponding to phases II and III.

The spSearch tool is a workflow that implements a series of algorithms to process the models’
source code and produce as output a set of subsystem-cluster classes, organized by subsystem-size, which are a full partition of a Simulink repository. To complete its work, spSearch first consolidates, annotates, and filters the source code definitions of all the models in the repository. Second, it traverses these preprocessed textual definitions in a bottom-up fashion, from most- to least-nested ones, to identify and parse each Simulink-model subsystem. Third, it computes the size and topology properties (i.e., name- and type-connectivity) of each subsystem. Fourth, it parses all the Stateflow charts to identify chart-sequence patterns. Fifth, it completes a name-usage and cross-referencing analysis for all the subsystems. Finally, it clusters by size, measured as the number of encapsulated blocks and dataflows, all the parsed subsystems. The resulting data structure is an addressable-by-size set of cluster classes which points to all the existing subsystems in the given Simulink repository. The list of algorithms implemented in the spSearch tool and the ancillary utilities includes:

- A repository traversal algorithm which processes a temporary, integrated and annotated version of the models’ source code in a bottom-up fashion. It conceptualizes each Simulink model in this data structure as a tree, whose nodes are subsystems and whose edges are subsystem-containment relations, and the entire structure as a forest of trees. Every time that it identifies, parses and evaluates the properties of a subsystem, it deletes its content from this temporary repository. The overall time-complexity of this algorithm is lower than $O(n \lg n)$, where $n =$ repository’s total number of SLOCs.

- A clustering algorithm which evaluates the size of each subsystem, defined as the number of blocks and dataflows it encapsulates, and labels it with a $mBnL$ ($m =$ blocks, $n =$ dataflows) identifier. This action automatically places the unique identification and properties of each subsystem in the appropriate cluster class.

- Hash functions NCHC and TCHC which compute name- and type-connectivity hash codes, respectively. These codes have negligible probabilities of collision associated with them, which minimizes the risk of obtaining false positives and negatives when assigning similarity properties
7.1 Summary
to subsystems.

- A statechart sequence detection algorithm which parses all the Stateflow charts, existing in a Simulink repository, to compute the number of states in each one. It then detects sets of Stateflow charts, having the same sequence of numbers of states, in different Simulink models of a repository. The detection of these sequences captures similar or identical patterns of behaviour embedded in the different models of a repository.

- A subsystem-name-usage algorithm which determines, for each subsystem of a given name and size, all the other subsystems in the same repository, with the same or different size, that have been assigned the same name. The objective is to determine semantic similarity of subsystems that have a slightly different structure, but which embody an identical or similar modeling intent.

- The ancillary tools spClassMerge and spClassSelect which allow users to merge and select, respectively, any subsets of cluster classes corresponding to the same or different repositories, making it possible to extend the analysis to any number and composition of Simulink repositories with minimal overhead. Class merge and selection are closed operations that use the class identifier \( mBnL \) as a processing key, i.e., they generate a set of cluster classes as output.

The MoSART graphical user interface provides 1) a class management system that allows users to manually select and store any sets and subsets of cluster classes for analysis, 2) a query facility that enables arbitrary selections of classes and subsystems based on their topological and location properties, and 3) a dynamic grid that lists the subsystems selected with either the class management system or the query facility, serves as a data container for drill-down queries, and plays the role of a look-up table for manually-selected subsystem visualizations. Users can keep a record of their model-pattern inferences processes, using MoSART, in saved queries, cluster class groups and subsystem-visualization lists. They can visually confirm patterns of subsystems by displaying their content and reading their associated properties listed on the dynamic grid.
7.1 Summary

The model-pattern elicitation process involves 1) the automated detection by spSearch of size-based cluster classes, subsystem topology properties, Stateflow chart sequences, and subsystem-name usage, and 2) the interactive model-pattern inference iterations that use the facilities of MoSART to query, visualize and compile statistics about subsystems. Users interpret subsystem properties and compiled statistics, cross-reference subsystem visualizations, and apply domain knowledge for making decisions about semantic model-patterns of interest.

The application of the pattern discovery framework and processes to an industrial repository of Simulink models identified various types of structural Simulink model-patterns. The use of domain knowledge and systems integration experience permitted the extrapolation of design meanings for those patterns. Semantic entailment processes identified patterns such as optional signals and events sources, synchronous events generators, special waveform generators, task schedulers and macro-event schedulers. Comparisons of the patterns inferred from this industrial repository with those obtained from two public domain repositories (i.e., AVS and STR-R2015a) showed that the former repository turned out a greater variety of structural, architectural, and behavioural patterns, non-existent in the latter repositories.

In summary, our pattern discovery toolchain 1) identifies all the subsystems in all the models of a Simulink repository, 2) characterizes them with hash codes for further manual or automated similarity processing, 3) clusters them using a metric of subsystem size, 4) identifies patterns of Stateflow chart sequences in models, and 5) provides user-friendly querying, visualization, configuration and persistence mechanisms for identifying derived or composite cluster classes of selected subsystems. All these features directly produce or strongly facilitate the elicitation of useful structural views of a repository, consisting of groups of subsystems and/or Stateflow charts, clustered only on topological subsystem properties, that practitioners can classify, manipulate and reapply in targeted use cases. The fine-tuning and semantic interpretation of those structural views as specific system design patterns reside in the cognitive abilities and domain knowledge exercised by practitioners. Our tools create the data structures, as well as implement the pattern search/reduction mechanisms and presentation facilities that greatly expedite and empower this cognitive process. The case study we
7.2 Future Work

have introduced in this thesis regarding the application of these tools to an industrial repository of Simulink models, has uncovered a set of basic structural modeling patterns whose descriptions we have enhanced into subjective semantic patterns. We deem that these latter patterns capture useful and reusable system designs in the involved application domain.

7.2 Future Work

Although we have obtained excellent pattern discovery results with the tools and processes discussed in this dissertation, we can envision some future research work directed at enhancing their effectiveness and scope, and also extending the automation of the search beyond syntactical patterns into targeted semantic patterns with reuse specifications in mind. The following are interesting lines of research to pursue:

1. Study of enhancements to the pattern search and subsystem-property computation algorithms that would allow the processing of extremely large repositories of Simulink models (i.e., billions of SLOCs), into pattern-classes with richer properties, within small time spans. Some of the studies that would be interesting to pursue are:

   • Research upgrades of the repository traversal and model-pattern search algorithms that would make them sensitive to subsystem nesting profiles (i.e., nesting path depth, nesting level width). Investigate the magnitude of the time-complexity reductions that can be accomplished with these changes.

   • Determine what additional properties of Simulink subsystems would help automate their classification and understanding into, for example, structural types or predefined semantic classes of subsystems. Investigate best approaches to compute these properties.

   • Investigate the effects caused by the inclusion of predefined Simulink syntactical constructs on the search and inference of model-patterns, as well as on the comprehension of Simulink repositories. Research pattern-matching algorithms that would optimize these tasks.
7.2 Future Work

- Investigate the effects caused by the inclusion of transitions, triggers and conditions in the analysis of Stateflow charts. Research algorithms to match predefined domain-specific encapsulated behaviours to syntactical definitions of those charts.

- Research hashing algorithms that would generate codes decomposable into sub-codes that could be mapped to the components encapsulated by subsystems. Investigate the degrees of similarity between subsystems that these decomposable codes could yield.

2. The architecture of both, spSearch and MoSART, is geared to allow modular expansions in the pattern-search capabilities of our toolchain. spSearch generates generic intermediate data structures that can be reused by additional pattern-detection modules. MoSART can have any number of views, in addition to the Browse view. Building upon these expandability properties of our tools, we can:

- Investigate intermediate data structures, in the spSearch workflow, that can provide strong support for the detection of predefined semantic model-patterns.

- Research the architecture and presentation of additional specialized views in MoSART. We envision that the pattern results that can be obtained with the Browse view could be placed in use cases and “what-if” scenarios with a view focused on single-pattern analysis and manipulation. Decisions about approaches to applying single or multiple model-patterns could be significantly enhanced with a view focused on assembling reuse scenarios. Exploration work towards formalizing interesting sets of model-patterns could be best handled with a view that supports specific pattern-language constructs.

- Investigate additional class- and pattern-querying languages. For example, we have used a language to query cluster classes on the Browse view, which is based on a fixed but configurable logical expression and regular expressions on subsystem-property strings. Although very flexible and powerful, this query language could be enhanced to hide the syntax of regular expressions and allow a variety of logical expressions and composite queries.
7.3 Conclusions

3. In this dissertation we have identified model-patterns by common topological properties of subsystems, statistics of subsystem occurrences and sequences of statechart sizes. Groups of subsystems that follow one or more of these rules of extraction become structural model-patterns. Domain knowledge has been applied to complement these rules of pattern identification. We could extend these pattern-definition rules for groups of subsystems or statecharts that are of special interest in a specific application domain. For this purpose, given a set of appropriate use cases, the research of domain-specific pattern-description languages would be a most interesting endeavour. Within the scope of such an initiative one could investigate how to capture subsystem properties, occurrences, sequences, and any other relevant pattern-definition criteria, into the syntax of a pattern-description language. This language would allow the description of existing model-patterns and the specification of emergent ones.

4. Our algorithms and tools could be adapted to discover patterns in models developed with other block-and-dataflow modeling languages similar to Simulink, for example UML-RT. Providing answers to this portability challenge would make it possible to abstract modeling patterns for embedded systems that are applicable in any of those languages.

7.3 Conclusions

We have described in this dissertation the design, implementation and results for a framework to discover Simulink model-patterns in very large repositories. Our main objectives for the design of the algorithms and tools in this framework have been to effectively address the multiple complexity-reduction, scalability and portability challenges arising when attempting to find patterns in graphical models. Once syntactical Simulink patterns, either structural or behavioural, are automatically or interactively identified, the final denomination and utility of semantic model-patterns resides in the knowledge and judgment of domain experts and modelers. To make effective decisions about patterns within short time spans, these practitioners need to deal with reduced search-spaces, simplified model properties, powerful information compacting operations, and friendly interfaces to
pattern data. We believe that the algorithms, data structures and interfaces embedded in our tools spSearch and MoSART provide extensive support to detect, infer and refine model-patterns of various abstraction levels, because:

- For Simulink models, the search-space reduction approach in the SpSearch tool is agnostic to a repository’s content. It summarizes its content into a pattern of cluster classes, organized by subsystem size, that accepts powerful closed class-oriented operations such as sub-classing, class merge, class cross-referencing, class union and intersection. This generic clustering, 1) often directly generates patterns of subsystems, 2) enables the merge and comparison of classes from different repositories, and 3) facilitates the analysis of specific model-subsets of interest in a given repository.

- The algorithms in the spSearch tool can process and reduce the content of very large Simulink repositories in minimized time, relative to the repository’s size. With time-complexities lower than $O(n \log n)$, where $n =$ total number of Simulink-model SLOCs, they can process entire repositories in times that are, on average, at least two orders of magnitude smaller than the times that $O(n^2)$ algorithms would spend for the same tasks. This property of spSearch makes it feasible to run multiple clustering iterations, on the same or different incrementally-modified repositories, to support a progressive analysis and refinement of patterns.

- The closed operations on cluster classes, supported by the class management, querying and visualization facilities of MoSART, enable a wide range of configurations of pattern inference steps. Practitioners can reduce, compose or focus the patterns of subsystems in multiple ways, as shown for the industrial repository case study in chapter 6, so they can endow them with modeling meaning applicable in specific use cases.

- The responsive behaviour of SpSearch and the flexible querying capabilities implemented in MoSART, make the coordinated use of these tools an effective alternative for the comprehension of Simulink repositories. Users can concentrate on identifying patterns of similar subsystems or
7.3 Conclusions

even complete Simulink models that occur multiple times, or focus on understanding any specific configuration of subsystems, Stateflow charts and models. In each scenario, both tools provide selective access, either directly to the subsystem level or indirectly to the block and property level, to any portions of entire repositories of Simulink models.

This dissertation has focused on the algorithms, tools and processes to elicit Simulink subsystem-centric model-patterns. Subsystem-cluster classes have been the central data structures used to reason about patterns; computed subsystem-topology properties such as size, name- and type-connectivity have provided the main indicators of similarity; frequencies of subsystem occurrence have served as references to infer more complex architectural patterns; and subsystems visualizations combined with domain knowledge have provided the context to assign modeling semantics to syntactic patterns. The experience with applying our framework to an industrial-strength Simulink repository has shown that, in practice, subsystems truly reflect modeling intent about hierarchy, decomposition, specialization, re-usability goals, and that they are not open to arbitrary changes. Breaking the nature, scope or interface of subsystems often results in a noticeable change to the nature of the Simulink models that contained them, with hard-to-predict results. These aspects have to be tightly controlled in models for embedded systems, which are usually subject to stringent timing, safety and reliability requirements.

In summary, we believe that our approach strives to extract quality model-pattern information that thoroughly covers entire repositories and can be reused with minimal side effects. We have attempted to establish that the effective parsing and analysis of Simulink models’ source code definitions can provide us with compact and reliable pattern-like data for which we can set up flexible and powerful pattern-inference processes. The combination of all these factors has allowed us to infer a wide variety of Simulink model-patterns and to comprehend entire repositories with manageable effort.
Bibliography


BIBLIOGRAPHY


Appendix A

Simulink Models Source Code

The Matlab framework maintains structured-text serialized versions of each Simulink model in “mdl” (older format) or “slx” (newer format) files. Conversion utilities make both file formats interchangeable. Either representation stores definitions, graphical representation and hierarchical relationships for all the entities in a Simulink model using a proprietary (i.e., “mdl” format) or an XML-based (i.e., “slx” format) markup language.

In our work, we have implemented algorithms that parse the “mdl” versions of Simulink models to identify basic classes of subsystems of equal size. Our approach instruments these textual representations with indexing fields, and later traverses entire repositories of these modified representations in a bottom-up fashion, from the innermost definitions all the way to the root entity of each model. Context-sensitive parsing of each entity, during this traversal, allows the identification of equally-sized subsystems. The following sections describe the syntactical elements involved in the description and traversal of models in the “mdl” format.

A.1 Simulink Model Syntax

The highest abstractions stored in a “mdl” file are either Model or Library. The syntactical structure of either one is: (1) Stag { Text definitions of contained Block/Dataflow (BD) entities }, where Stag
A.1 Simulink Model Syntax

Model or Library. All other BD entities in a Simulink model, with the exception of Stateflow charts, are stored as nested textual definitions within one of these two top entities using the (1) syntactical structure. The definition of a BD entity follows the structure: (2) Stag { Multi-line entity definition }. Stateflow charts are stored in a separate Stateflow entity using a flat structure of Statechart Object (SO) entities. The syntactical structure of each SO is: (3) Ctag { Multi-line entity definition }. Table A.1 provides a summary view of these textual representations. Tables A.3 and A.2 list the Stateflow and Simulink tags, respectively, detected in the industrial and public domain repositories examined in the empirical work for this thesis.

<table>
<thead>
<tr>
<th>Model or Library Structure</th>
<th>Block/Dataflow (BD) Entities</th>
<th>Statechart Object (SO) Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model or Library {</td>
<td>Stag {</td>
<td>Ctag {</td>
</tr>
<tr>
<td></td>
<td>Stag {</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>Entity Definition</td>
</tr>
<tr>
<td>Stateflow {</td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>Entity Definition</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Table A.1: Internal Representation of Simulink Models

<table>
<thead>
<tr>
<th>Stateflow Tag (Ctag) List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateflow, activeStateOutput, array, autogen, chart, data, debug, diagnostic, dst, eml, event, fixpt, instance, junction, layoutInfo, machine, noteBox, props, range, simulink, slide, src, state, stateFontS, statecell, subviewS, table, tableUIPrefs, target, transition, transitionFontS, transitioncell, transitionFontS, truthTable, type</td>
</tr>
</tbody>
</table>

Table A.2: Stateflow Tags - Detected in Test Repositories
A.2 Annotation of Models

To reduce the time and space complexity of the pattern search algorithms that we implemented in this work, we compacted the Simulink models of each repository under study into a single storage area $\mathcal{M}$. During this process, we added property description and indexing fields to each line of each model stored in this area. The following are descriptions of those fields:

- **Type** is a single character that identifies whether the current line belongs to a **Model** (Type = M) or **Library** (Type = L) entity. It also flags the start (Type = 0) and end (Type = 1) of a model, as well as the start (Type = @) and end (Type = #) of a library in $\mathcal{M}$.

- **ModelId** is a positive integer that acts as a pointer to the original Simulink model file that sourced the current line.

- **LineId** is a positive integer that identifies the location of the current line within the current model.

- **Level** is a positive integer that identifies the nesting depth of the current line, in reference to one of either “**Model** {”, “**Library** {”, or “**Stateflow** {” lines, which declare a root entity in the current model and are assigned a level 0.

### Table A.3: Simulink Tags - Detected in Test Repositories

<table>
<thead>
<tr>
<th>Simulink Tag (Stag) List</th>
</tr>
</thead>
</table>

A.2 Annotation of Models
A.2 Annotation of Models

• **SourceCode** is a string that stores the content of the current line as defined in the original “mdl” file defining a Simulink model.

With the inclusion of these annotation fields, each line of \( \mathcal{M} \) has the following layout of comma-separated fields:

\`
Type, ModelId, LineId, Level, SourceCode
\`
Appendix B

Pattern Persistence Formats

The spSearch tool and MoSART can produce patterns of Simulink subsystems that, once identified, have to be kept in a persistent and retrievable storage format for further analysis or reuse. The following sections provide a summary of these formats.

B.1 Formats in Pattern Files

The primary clustering of Simulink subsystems functionality in spSearch produces two files: (1) srid_pattern_Simulink_00.txt, and (2) srid_pattern_Simulink_01.txt.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem Size</td>
<td>Number of blocks and dataflows in a subsystem</td>
</tr>
<tr>
<td>Source Model</td>
<td>Pointer to the model sourcing a subsystem</td>
</tr>
<tr>
<td>Text Location</td>
<td>Pointers to the start and end lines of the text definitions of a subsystem</td>
</tr>
<tr>
<td>Subsystem Nesting</td>
<td>Integer greater than 0 indicating subsystem containment in reference to a root subsystem in a model</td>
</tr>
<tr>
<td>Subsystem Path</td>
<td>List of subsystems in the containment path from the root of a model for a subsystem</td>
</tr>
<tr>
<td>Type Connectivity</td>
<td>Hash code representing the topology of block types and dataflows for a subsystem</td>
</tr>
<tr>
<td>Name Connectivity</td>
<td>Hash code representing the topology of block names and dataflows for a subsystem</td>
</tr>
</tbody>
</table>

Table B.1: Content Description of File srid_pattern_Simulink_00.txt
### B.2 Primary Clustering Classes Display Format

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem Type</td>
<td>Subsystem is an encapsulation subsystem or a Stateflow chart</td>
</tr>
<tr>
<td>Source Model</td>
<td>Pointer to the model sourcing a subsystem</td>
</tr>
<tr>
<td>Subsystem Nesting</td>
<td>Integer greater than 0 indicating subsystem containment in reference to a root subsystem in a model</td>
</tr>
<tr>
<td>Text Location</td>
<td>Pointers to the start and end lines of the text definitions of a subsystem</td>
</tr>
<tr>
<td>Subsystem Path</td>
<td>List of subsystems in the containment path from the root of a model for a subsystem</td>
</tr>
<tr>
<td>Stateflow Chart ID</td>
<td>Identifier of a Stateflow chart within a model</td>
</tr>
<tr>
<td>States</td>
<td>Number of states in a Stateflow chart</td>
</tr>
<tr>
<td>Chart Text Location</td>
<td>Pointers to the start and end lines of the text definitions of a Stateflow chart</td>
</tr>
</tbody>
</table>

Table B.2: Content Description of File \texttt{srid\_pattern\_Simulink\_01.txt}

These files store computed properties, usable in model pattern discovery, for each Simulink subsystem in a repository $\mathcal{R}_{\text{srid}}$. Tables B.1 and B.2 describe the properties stored for each subsystem.

### B.2 Primary Clustering Classes Display Format

The class conversion stage of the spSearch workflow reads file \texttt{srid\_pattern\_Simulink\_00.txt} and generates file \texttt{srid\_pattern\_Simulink\_00.csv}, which serves as a persistent data structure for the proper display and manipulation of subsystem classes on MoSART. Table B.3 describes the display attributes captured in this file for each subsystem.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class ID</td>
<td>Encoded as a representation of the number of blocks and dataflows in a subsystem</td>
</tr>
<tr>
<td>Subsystem ID</td>
<td>Integer representing an index within a class for a subsystem</td>
</tr>
<tr>
<td>Subsystem Path</td>
<td>List of subsystems in the containment path from the root of a model for a subsystem</td>
</tr>
<tr>
<td>Model Path</td>
<td>Physical path to a Simulink model containing the current subsystem</td>
</tr>
<tr>
<td>Type Connectivity</td>
<td>Hash code representing the topology of block types and dataflows for a subsystem</td>
</tr>
<tr>
<td>Name Connectivity</td>
<td>Hash code representing the topology of block names and dataflows for a subsystem</td>
</tr>
</tbody>
</table>

Table B.3: Content Description of File \texttt{srid\_pattern\_Simulink\_00.csv}
B.3 Class Queries Storage Format

The *Browse* view of MoSART (refer to fig 3.5) allows users to assemble, execute, save and retrieve queries on the subsystems of a set of clustering classes displayed on a dynamic grid. These queries use the class identification codes and regular expressions on subsystem paths within models, physical paths of models and type connectivity hash codes of subsystems. Table B.4 describes the storage format for these queries.

<table>
<thead>
<tr>
<th>Section or Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Codes</td>
<td>List of the class identification codes selected in the query</td>
</tr>
<tr>
<td>End of Codes</td>
<td>&lt;BLCodes_END&gt;</td>
</tr>
<tr>
<td>Types Hash Code RE</td>
<td>Regular expression (RE) on the type connectivity hash code of subsystems</td>
</tr>
<tr>
<td>Logical Connector</td>
<td>AND</td>
</tr>
<tr>
<td>Subsystem Path RE</td>
<td>Regular expression (RE) on the containment path of a subsystem within a Simulink model</td>
</tr>
<tr>
<td>Logical Connector</td>
<td>AND, OR</td>
</tr>
<tr>
<td>Model Path RE</td>
<td>Regular expression on the physical storage path of a Simulink model</td>
</tr>
</tbody>
</table>

Table B.4: Class Queries Storage Format Description

MoSART’s *Browse* view can also save any group of subsystems, manually selected on the grid, into a persistent data structure with the format described in table B.3.