MASTER

Modelling variability in a simulated printer for improved robustness testing of embedded control software

Okwudire, C.G.U.

Award date:
2012

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Modelling Variability in a Simulated Printer
for Improved Robustness Testing of Embedded Control Software

Okwudire, C.G.U.

August, 2010
Modelling Variability in a Simulated Printer for Improved Robustness Testing of Embedded Control Software

A thesis submitted in partial fulfilment of the requirements of the degree of M.Sc. in Embedded Systems at Eindhoven University of Technology

By:

Okwudire, C.G.U.
Eindhoven, The Netherlands

Supervisor:

dr. L.J.A.M. Somers

Embedded Software Group
Research & Development Department
Océ-Technologies B.V.
Venlo, The Netherlands
lou.somers@oce.com

Software Engineering & Technology Group
Faculty of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
l.j.a.m.somers@tue.nl

August, 2010
Abstract

The development of complex embedded systems is a challenging process typically involving multidisciplinary teams of engineers. The traditional design trajectory involves a largely sequential process in which outputs from one step serve as inputs for another. With this approach, there are very limited opportunities for parallel development, especially at early stages. Furthermore, it hampers the time to market which is crucial for maintaining a competitive edge over other industrial players.

A paradigm shift to model-based design (MBD) techniques offers a feasible solution to this problem since it enables parallel development and facilitates a common understanding of design artefacts across disciplines. Software-in-the-loop (SIL) simulation is one of such MBD techniques proposed in the literature. In the context of embedded system development, it entails simulating a model of the (non-existent) hardware interfaced with the actual control software via a modified input/output layer, thereby allowing for software design, implementation and verification in parallel with hardware development.

One of such SIL simulation frameworks has been successfully developed at Océ-Technologies B.V. where it is currently used for testing the embedded control software that drives their document management systems, in the absence of real machines. However, the simulated machine in its present form is ideal in the sense that variability in all modelled devices is ignored; given a particular input stimulus, the same results are always produced. The goal of this project is therefore to develop an adequate and concise tolerance model which captures system variability for improved robustness testing of the embedded software.

This thesis presents the design and implementation of a tolerance model integrated into the existing SIL framework. We show that modelling variability in the different devices comprising the simulated machine is feasible. In particular, we provide tolerance models for motors, pinches, actuators, sensors, segments and sheets. We also take sheet behaviour in bent segments into account. Furthermore, we provide a level of abstraction that facilitates understandability and usability of the model.

The results of case studies conducted show that the SIL simulation framework, extended with tolerances, enables the specification and verification of scenarios which were hitherto impossible or very difficult. Test cases using existing control software and realistic tolerance ranges buttress the added value of this work, namely facilitating more extensive robustness testing which will ultimately result in better products. These test cases also provide a positive indication of the usability and adequacy of our model. We equally show that our extension does not incur any significant performance penalties in terms of execution time, memory usage and log file size of the simulator.

Other contributions of this project include: Designing and implementing interfaces for tolerance specifications using existing XML files and via external tools for automatic regression testing and visualization. We also provide a command-line tool for validating XML specifications. This is a positive move towards more effective model-based design in which erroneous specifications are earlier (and automatically) detected, thereby preventing unexpected behaviour in subsequent steps and avoidable simulator crashes.

Finally, beyond the immediate scope in which this work has been performed, we make a contribution to MBD in the domain of embedded systems in general. This is because, at the time of this document, we do not know any work that reports SIL simulation with tolerance modelling for the development of complex document management systems. We anticipate that our approach is applicable to other domains where (embedded) mechatronic systems are designed.
Acknowledgements

I wish to thank the management of Océ-Technologies B.V. for sponsoring my master education and for giving me the opportunity to conduct my thesis project in the research and development department at Venlo, The Netherlands. I will especially remember Océ Venlo for the conducive working environment, people’s willingness to help and genuine interest in my work (especially by my stakeholders). Thank you all for being there when I had problems, for sharing my joys and for giving me your valuable feedback. In particular, many thanks to Lou Somers, my supervisor, for guiding me through my work without placing undue pressure on me. I will miss our weekly Friday morning progress meetings. I express my heartfelt appreciation to Henri Hunnekens, Rudi Huismans, Harald Schwindt, Sander Hulsenboom, Ralph Woltering, Sidney Laracker, Jeroen Lind and Joost Janse who facilitated my successful completion of this work in different capacities which space will not permit me to elaborate upon.

I express my sincere gratitude to Reinder J. Bril and Mike Holenderski from the System Architecture and Networking (SAN) expertise group of Eindhoven University of Technology for guiding me through an earlier internship. The technical and non-technical skills I gained really helped me carry out my thesis project methodologically and effectively. I equally owe Martijn van den Heuvel (also from SAN) a debt of appreciation for taking the time to peruse earlier manuscripts of this thesis.

Last, but by no means least, I acknowledge the loving support and prayers of my family and friends. Without you, my life is not complete. Most importantly, I glorify God Almighty “in whom I live, move and have my being” for being my Father in every sense of the word.

Chidiebere G.U. Okwudire
August, 2010
Dedication

In memory of my dad, G.O. Okwudire, Esq., who lost his life in a fatal accident on November 8, 2008 - barely three months after I came to The Netherlands for my master studies. He instilled in me the value of trusting God, working hard and paying attention to details; I know he would have been proud to read this work.
2 Requirement Analysis

2.1 Sources of Variability

2.1.1 Classification of Variability

2.2 Main Stakeholders

2.3 Use Cases

2.4 Project Requirements

2.4.1 High-level Requirements

2.4.2 Low-level Requirements

2.5 What to Model

2.6 Chapter Summary

3 Design

3.1 The (Existing) SIL Simulator Architecture

3.2 Tolerance Specification: Design Issues

3.2.1 What to Specify

3.2.2 Where to Specify Tolerances

3.2.3 How to Specify Tolerances

3.2.4 Input Probability Distribution

3.2.5 Default Tolerance Mode

3.3 Modelling Tolerance in the Simulated Plant

3.3.1 Tolerance in Motors

3.3.1.1 Updating Sheet Speed and Position

3.3.1.2 Modelling Tolerance in Motor Speed

3.3.1.3 Speed Tolerance in FPGA_StepperMotor

3.3.1.4 Speed Tolerance in VarSpeedStepperMotor

3.3.1.5 Speed Tolerance in StubbedMotor

3.3.1.6 Speed Tolerance in ControlledMotor

3.3.1.7 Speed Tolerance in SteeredMotor

3.3.1.8 Speed Tolerance in HIL_StepperMotor

3.3.1.9 Further Remarks

3.3.2 Tolerance in Pinches

3.3.3 Tolerance in Actuators

3.3.4 Tolerance in Sensors

3.3.5 Tolerance in Segments

3.3.5.1 Segment Length Tolerance

3.3.5.2 Sheet behaviour in Bent Segments

3.3.6 Tolerance in Sheets

3.3.6.1 Sheet Length Tolerance

3.3.6.2 Delaying Sheets

3.3.6.3 Sheets as Part of the Simulated Plant

3.4 Provided XML Interface for Tolerance Specification

3.4.1 Specifying Tolerance Ranges
3.4.2 Specifying Tolerance Values for Test Cases .................................. 44
  3.4.2.1 Specifying a Seed for Test Case Reproduction ......................... 44
  3.4.2.2 Injecting Tolerance ....................................................... 45
  3.4.2.3 Disabling Tolerance ...................................................... 45
  3.4.2.4 Specifying Sheet Length Tolerance .................................... 47
  3.4.2.5 Special Usage ........................................................... 47
  3.4.2.6 Enforcing Correct Tolerance Specification ............................ 48

3.5 Provided Interface Methods for Tolerance Specification ...................... 50

3.6 Test Cases for Verification ...................................................... 54
  3.6.1 Motors and Pinches ......................................................... 54
  3.6.2 Sensors and Actuators ...................................................... 55
  3.6.3 Sheets and (Bent) Segments .............................................. 57

3.7 Chapter Summary ................................................................. 57

4 Implementation ............................................................................ 58
  4.1 (RoseRT) Notation for Sequence Diagrams .................................... 58
  4.2 Handling XML Tolerance Specifications ...................................... 59
  4.3 Handling Tolerance Specifications via the Provided Interface Methods .. 60
    4.3.1 Implementing Sheet Delay ................................................ 62
    4.3.2 Timing in SIL ................................................................. 64
  4.4 Other Significant Changes/Additions to SIL .................................. 67
    4.4.1 Interactive Mode ............................................................ 67
    4.4.2 Plotting Default Graphs via a Provided Interface Method ............ 67
    4.4.3 The Segment Node Class .................................................. 68
    4.4.4 XML Schemas and a Tool for Validating Tolerance Specifications .. 69
  4.5 Chapter Summary ................................................................. 70

5 Validation and Verification .......................................................... 71
  5.1 Requirement Validation .......................................................... 71
    5.1.1 Motivation for Skipping Skewness and Lateral Displacement Models 73
    5.1.2 Unifying SIL Layout and MoBasE Machine Descriptions ............... 74
  5.2 Test Case Results ................................................................. 74
  5.3 Performance Analysis of SIL .................................................... 79
    5.3.1 Test Setup and Procedure ................................................ 79
    5.3.2 Results ........................................................................... 80
      5.3.2.1 Execution Time ........................................................... 80
      5.3.2.2 Memory Usage ........................................................... 80
      5.3.2.3 Log File Size ............................................................. 82
  5.4 Chapter Summary ................................................................. 84
6 Conclusion

6.1 Case Studies
   6.1.1 Testing a Flipping Wheel Error Routine
   6.1.2 Verification of a Skewness Detection Error Routine
   6.1.3 Debugging a “False Alarm”

6.2 Project Reflection
   6.2.1 Challenges
   6.2.2 Helpful Practices
   6.2.3 Design/Implementation Choices in Retrospect

6.3 Project Summary

6.4 Future Work
   6.4.1 More Input Probability Distributions and Multiple Seeds
   6.4.2 Error Status Levels in SIL
   6.4.3 Visualization of Variability using SILVis
   6.4.4 Statistical Analysis of SIL Extended with Tolerances
   6.4.5 Modelling Skewness

References

A Tolerance Specification Examples
   A.1 Specifying Tolerance Ranges
   A.2 Specifying Tolerance Values for Test Cases
   A.3 Specifying Tolerances via the Interface Methods

B Additional Test Case Results
List of Figures

1.1 Overview of "in-the-loop" simulation ................................................. 4
2.1 An example paper path ........................................................................... 12
3.1 Logical view of SIL showing interfaces with other components ............... 22
3.2 Logical view of typical control software architecture .............................. 23
3.3 Some sources of sensor tolerance .......................................................... 37
3.4 Potential sensor problems ....................................................................... 39
3.5 Expected results for test cases to verify tolerance in motors (and pinches) .. 54
3.6 Expected results for test cases to verify tolerance in sensors (and actuators) . 56
4.1 Handling XML tolerance specifications ................................................. 59
4.2 Handling immediate POI commands .................................................. 61
4.3 Handling delayed POI commands ......................................................... 62
4.4 Implementing sheet delay ...................................................................... 63
4.5 Timing in SIL ....................................................................................... 65
4.6 UML class diagram for sheets showing the new segment node class ....... 69
5.1 Motor speed tolerance test case results ............................................... 75
5.2 Pinch speed tolerance test case results ............................................... 76
5.3 Sensor position tolerance test case results .......................................... 77
5.4 Actuator delay test case results ............................................................ 77
5.5 Comparison of execution times of SIL for different test cases .................. 81
5.6 Comparison of private memory usage of SIL for different test cases ......... 82
5.7 Comparison of SIL log file sizes for different test cases ....................... 83
5.8 Comparison of SIL log file sizes for test cases without print jobs ............. 84
B.1 Snapshot of SIL showing tolerance logging ......................................... 100
List of Tables

2.1 Identified sources of variability .............................................. 13  
2.2 Project high-level requirements ........................................... 17  
2.3 Project low-level requirements ........................................... 18  
3.1 Kinds of tolerance modelled ............................................... 44  
3.2 Attribute for seed specification .......................................... 45  
3.3 Attributes for tolerance injection via XML specification .......... 46  
3.4 Handling ambiguous XML tolerance specifications ................. 50  
3.5 Overview of SIL’s provided interface methods for tolerance specification .......... 51  
5.1 Validation of project requirements ...................................... 72
List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Using optional “transition” attribute in actuator tolerance specifications</td>
<td>45</td>
</tr>
<tr>
<td>3.2 Specifying sheet length tolerance</td>
<td>47</td>
</tr>
<tr>
<td>3.3 Injecting tolerance in all but one motor</td>
<td>47</td>
</tr>
<tr>
<td>3.4 Injecting tolerance in all but one pinch</td>
<td>48</td>
</tr>
<tr>
<td>A.1 Specifying tolerance ranges in layout.xml</td>
<td>94</td>
</tr>
<tr>
<td>A.2 Specifying tolerance values in SILSimulation.xml</td>
<td>96</td>
</tr>
<tr>
<td>A.3 Specifying tolerance values via SIL’s provided interface methods</td>
<td>98</td>
</tr>
</tbody>
</table>
## Abbreviations and Terms

The page numbers at the end of each entry refer to the first occurrence and/or definition of the abbreviation or term.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boderc</td>
<td>Beyond the Ordinary: Design of Embedded Real-time Control: A research project for multi-disciplinary design analysis of high-tech systems conducted in collaboration with Océ and other partners.</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer-aided Software Engineering refers to computer-based tools and methods for software analysis, design and development.</td>
</tr>
<tr>
<td>Controller</td>
<td>In general, this is the part of a system which monitors and reacts to changes in the system’s state. In SIL simulation, it refers to the (embedded) control software that executes with simulated hardware (plant).</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic-link Libraries are executable files which allows programs to share resources. They are Microsoft’s implementation of a shared library in Windows operating systems.</td>
</tr>
<tr>
<td>Finisher</td>
<td>A finisher (module) is the part of a printer which handles the sheet right after printing is completed until delivery at a paper output module. It performs any post-printing operations like stapling, making perforations and binding.</td>
</tr>
<tr>
<td>FPGA</td>
<td>A field-programmable gate array is a (re)configurable integrated circuit.</td>
</tr>
<tr>
<td>Happy Flow</td>
<td>An ideal model of the simulated machine in which only desired behaviour of sheets and of all other parts are modelled. It was developed in the context of the Boderc project.</td>
</tr>
<tr>
<td>MBD</td>
<td>Model-based Design/development technologies promote the use of models (and simulation) for faster and more efficient design of systems.</td>
</tr>
<tr>
<td>MoBasE</td>
<td>Model-based Engineering: An acronym for the on-going effort at Océ aimed at developing an interdisciplinary model template which describes the essential elements of a printer.</td>
</tr>
<tr>
<td>MoSCoW</td>
<td>A prioritization technique used in business analysis and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement. It is also known as MoSCoW analysis.</td>
</tr>
<tr>
<td>MRE</td>
<td>Machine-recoverable Error: An error that the machine can handle by itself, e.g. by rebooting.</td>
</tr>
<tr>
<td>ORE</td>
<td>Operator-recoverable Error: An error that requires external intervention, e.g. an operator opening a tray to remove a jammed sheet.</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional - integral - derivative control is a generic feedback mechanism popularly used in industrial control systems.</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>PIM</td>
<td>Paper Input Module: The software and/or hardware module responsible for injecting sheets into a paper path.</td>
</tr>
<tr>
<td>Plant</td>
<td>In general, this is the part of a system which is controlled by the controller. In SIL simulation, it refers to the model of the hardware which is simulated with real control software. In this context, it is also known as the simulated machine or simulated plant.</td>
</tr>
<tr>
<td>POI</td>
<td>Point-of-Interest commands are used in SIL to specify actions that must be performed at the specified segment position. A POI may or may not be guarded by a condition.</td>
</tr>
<tr>
<td>PPH</td>
<td>Paper Path Handling: The software and/or hardware module responsible for transporting sheets along a paper path after it is injected at the PIM.</td>
</tr>
<tr>
<td>RC</td>
<td>Remote Control is a tool used for testing embedded control software of a printer’s main node.</td>
</tr>
<tr>
<td>RoseRT</td>
<td>Rational Rose Real Time is a UML-based CASE tool which supports the development of complex, reactive, real-time software.</td>
</tr>
<tr>
<td>Schematron</td>
<td>A rule-based language for making assertions about the presence or absence of patterns in XML documents.</td>
</tr>
<tr>
<td>SiLEST</td>
<td>Software-in-the-Loop for Embedded Software Test was a project focused on developing an automated testing process for embedded software within a simulated environment.</td>
</tr>
<tr>
<td>SILVis</td>
<td>SIL Visualization is a tool used for visually presenting sheet transport through a paper path.</td>
</tr>
<tr>
<td>Simulink</td>
<td>It is a tool developed by The Mathworks for multidomain simulation and model-based design for dynamic and embedded systems.</td>
</tr>
<tr>
<td>TE</td>
<td>Test Executor is a tool used for automated regression testing of embedded control software. It interfaces with other tools like Universal Tester (UT), Remote Control (RC) and visualization (SILVis).</td>
</tr>
<tr>
<td>UT</td>
<td>Universal Tester is a tool used for testing embedded control software of a printer’s subnode(s).</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition: An XML-based language for describing the structure of an XML document, allowing for easy validation of XML specifications. We sometimes use XSD to refer to the schemas themselves.</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformations is a declarative XML-based language used to transform XML documents into other formats.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Context and Motivation

As a result of rapid advancements in technology, embedded systems have become all the more pervasive in today’s world. It is estimated that up to 98% of processors developed by major vendors do not find their use in desktop computers, but in embedded systems\(^1\). Whether in miniature electronic gadgets, handheld devices, household equipment or extremely complex control systems for automotives, air- and spacecrafts, embedded systems continue to revolutionize the ways in which we do things, and provide more and more functionality to the delight of an ever-demanding market. However, the design and development of these systems pose new challenges to the software engineering industry.

The use of models (i.e. abstract representations of systems with emphasis on some particular aspects) has been successfully applied in virtually all engineering disciplines \([29]\) to manage system complexity and to bridge the conceptual gap between application requirements and implementation. Model-based design/development (MBD) technologies have recently received more attention by embedded-software developers as a means to address various quality aspects related to embedded software development such as safety, real-time constraints, time-to-market, robustness, reliability and development cost.

As one of the world’s leading providers of document management and printing systems for professionals, Océ-Technologies B.V. (hereafter also referred to as Océ or “the company”) applies cutting-edge technologies in developing multifunctional systems. In addition to printing, these systems feature other functionality like scanning, copying and finishing (e.g. adding cover pages, stapling, binding, and making booklets) with outstanding image quality, speeds and performance in terms of the number of sheets processed per unit time. For example, the Océ VarioPrint\(^{®}\) 6000 Ultra Line\(^2\) was the first cut-sheet printer ever to be capable of producing over 300 8.5” x 11” duplex prints per minute.

Developing these complex mechatronic systems is undoubtedly a daunting task and involves collaboration of individuals from different disciplines such as electrical, mechanical and software engineering \([10]\). To reduce the time-to-market, it is desirable to have concurrent development in such multidisciplinary projects. This involves a shift from the traditional, sequential, mono-disciplinary process to a multidisciplinary approach based on multidisciplinary models. To this end, models of other parts of the system need to be developed that replace the unavailable components and allow for parallel development. For example, for concurrent development and verification of the embedded control software while the hardware it will control is still being manufactured, a model of the missing mechanical components is required. Moreover, such a model must mimic the behaviour of the real hardware. This approach differs from the

\(^1\)This estimate is based on quarterly statistics provided by the Semiconductor Industry Association (SIA); see \url{http://www.sia-online.org/cs/papers_publications/statistics} (Courtesy: Jim Turley).

\(^2\)\url{http://www3.oce.com/ultra/}
1.2 Problem Description

Although the existing SIL framework allows for testing of embedded control software under ideal conditions (the so-called “happy flow” printing [10]) and provides support for simple error injection, all modelled components such as motors, sensors, actuators, pinches and sheets are “ideal”. In other words, their behaviour is completely deterministic and static, and every simulation run with the same input parameters always yields the same results. However, in order to test the robustness of the control software, it is necessary to take variability in the parameters of the modelled devices into account. This variability results from effects such as wearing, friction, different paper types, inherent physical properties of various devices, coupling and assembly tolerances, etc. No doubt such factors are present in the real printers. Thus, in order to perform reliable robustness testing, such deviations from the ideal situation must be captured in the hardware model. Hence, we aim at identifying what tolerance factors need to be taken into account for different devices and how to express these in the existing SIL framework without incurring unacceptable performance penalties.

1.3 Project Goal

Given the problem identified in the previous section, the goal of this project is therefore to extend the existing SIL simulation framework by modelling variability in different devices which affect sheet transport along a printer’s arbitrary paper path. The scope of this study is delimited to cut-sheet printers which are designed to print separate sheets and does not cover printers which handle continuous streams of paper, i.e. continuous feed printers.

1.4 Related Work

In this section, we provide an overview of related literature. We begin in Section 1.4.1 with a context for the project by reviewing MBD paradigms with emphasis of which approaches are being used at Océ and in what capacity. Next, we review the literature on the applications of software-in-the-loop simulation in Section 1.4.2 thereby providing the basis for addressing the evolution of SIL simulation within the company in Section 1.4.3.

1.4.1 Approaches to Model-Based Software Verification

Embedded software validation and verification are compulsory steps in any good development process and provide an avenue for the application of MBD techniques. Whereas validation focuses on ensuring that the system does what it is designed for, namely that it meets a set of specified requirements, verification
addresses the question of whether the implementation matches the design [29]. In the context of MBD, “in-the-loop” testing has emerged as a useful means of embedded software validation and verification with increasing popularity and tool support especially in the automotive, aerospace, medical and consumer electronics domains. Four different configurations are identified in the literature, viz. model-, software-, processor- and hardware-in-the-loop (MIL, SIL, PIL and HIL) [30, 17, 29], each representing a distinct step in the migration from design to implementation.

MIL testing is usually performed at an early stage by the system engineers to check the validity of the proposed solution with respect to the (functional) requirements of the system. Generally, a model consists of two parts: a plant and a controller. The plant refers to (models of) the parts of the system which need to be controlled (e.g. motors, pinches, actuators) as well as enablers (e.g. sensors). The rest of the system is then the controller which realizes the desired system behaviour by steering the plant. Broadly speaking, the plant and controller may respectively be viewed as the hardware and software of a system.

In the MIL phase, models of both the plant and controller are typically developed and simulated using suitable MBD tools on a general-purpose computer commonly referred to as the host computer. Although UML-based models are very common, other domain-specific modelling languages may also be employed to describe sophisticated systems for which the general-purpose semantics of UML do not suffice [10].

The main goal of the MIL step is to establish functional correctness of the system. Nevertheless, it also serves the purpose of generating reference test scenarios and results to be used as a baseline in subsequent steps. RoseRT, the de-facto software development tool at the company, supports model execution. Since the controller software and the SIL simulation framework are both developed using RoseRT, MIL testing is, in principle, possible. Nevertheless, it is not currently performed. The reason for this could be that during the preceding system analysis phase, the happy flow model is used for design space exploration as described in [2]. In particular, this model is used to investigate alternative layouts of the paper path and for preliminary analysis of sheet schedulability. We remark that the plant model used in happy flow differs from the one used for SIL simulation.

The next step in the trajectory is SIL testing. During this step, the controller model in MIL is replaced by the actual control software to be deployed in the final system. This software is compiled into executable object code that can run on the host computer. Ideally, the MBD tool should support partial or full auto-generation of controller software from the controller model. This feature implicitly requires that the tool provides a means to capture the software architecture in the controller model developed in the MIL phase [29]. The same test cases developed for MIL testing are run and the results are compared against the baseline.

The purpose of SIL testing is to detect and correct possible arithmetic errors such as overflow, underflow or division-by-zero resulting from faulty design choices of sizes of data variables and structures. In addition to enhancing validation and verification, SIL testing provides the first opportunity for robustness testing of the control software. SIL testing is also performed on the host computer. Furthermore, a modification of the input/output (I/O) layer between the software and underlying hardware may be required to run the real control software on the simulated plant. In this work, we focus on SIL testing. We particularly aim at improving the existing SIL framework developed at the company [33] by adding tolerances to the simulated plant with the goal of verifying the robustness of existing (and future) printer control software.

PIL testing provides the first step toward migrating to the target platform, i.e. the processor on which the control software will actually be deployed in the finished system. In this test, the controller model is cross-compiled for the target system and executed on an experimental hardware which contains the same processor as the target system with additional facilities for storing test data and results. The simulation tool, running on the host computer, then communicates with the deployed control software typically via a serial communication interface and the test scenarios are executed in a non-real-time simulation.

The objectives of PIL testing are to uncover any non-timing related bugs caused by inadequate target-
Figure 1.1: Overview of "in-the-loop" simulation
specific code [29], to verify the code behaviour on the target platform and to measure code efficiency [30]. However, PIL testing is currently not performed at the company. Although the reason for this is not clear, it may be related to the fact that RoseRT supports code generation for different platforms and therefore handles the typical platform-related errors that emerge at this stage.

In the final phase, HIL testing is employed to provide real-time verification of the control software now coupled to real hardware. In other words, the plant model is replaced by a dedicated (preferably programmable) hardware emulator such as an FPGA board which generates physical signals to mimic the real I/O interface between a controller and plant. Although HIL features real software and hardware, (a part of) the simulator is still required to supply inputs to the system, for instance. Thus, in order to correctly interpret the results obtained, the overheads resulting from all components which are not present in the real machine must be taken into account. A HIL framework has also been developed at Océ. [31, 22] describe the design and implementation of a real-time step motor emulator for this HIL setup.

Figure 1.1 summarizes the four steps of model-based software verification described above. MIL testing is preceded by an analysis of the system’s requirements. The test cases generated in the MIL phase are used to verify functional behaviour after which control software code is manual or (semi-)automatically generated and used in the SIL step for further verification as well as for software robustness testing. If successful, SIL simulation is followed by PIL testing in which the control software is deployed on the target platform for further testing. The last step, HIL simulation, features a replacement of the hitherto simulated plant with an emulator connected to real hardware. As shown in the figure, test results from each step are compared to those from earlier steps and if unsatisfactory, an iteration of one or more of those steps is required. Similarly, test inputs at each stage may be adapted and/or extended by adding test cases specific to the step being performed. What follows is a prototype of the system, leading to a finished product ready for the market. A traditional approach to development and verification involves a huge step from the requirements (possibly via MIL testing) directly to a prototype. Whereas this may be successful in simple systems, for relatively complex systems, a lot of effort is spent on iteratively identifying and solving problems which would more easily surface if an MBD approach is adopted. This observation serves as a strong motivation for model-based software development.

In Figure 1.1, we employ colors to distinguish simulated components (in blue) from real ones (in white) and their deployment - white for the host computer, yellow for the target platform and yellow with black stripes for a dedicated hardware emulator. The figure captures the progression from both simulated plant and controller using a MBD tool on a host platform in MIL; real control software with a simulated plant still executing on the host platform in SIL; real control software deployed on the target platform and interfaced to the simulated plant on the host in PIL; and finally dedicated hardware emulation of the plant using real software on the target platform in HIL.

At this point, it is expected that the reader now has a good overview of the different “in-the-loop” methodologies. Furthermore, we remark that in actual embedded software development trajectories where MBD techniques are applied, one or more of these steps may be skipped while still producing robust and well-verified software. For instance, as earlier observed, MIL and PIL testing do not feature in Océ’s current MBD trajectory. However, following through the process step-by-step is expected to improve the efficiency of the overall development process both in terms of development time and cost as faults discovered at earlier stages are generally cheaper and faster to rectify.

1.4.2 Applications of Software-in-the-Loop Simulation

The application of software-in-the-loop (herein referred to as SIL and otherwise known as SWIL [5], SITL [19], SiL [18]) techniques in various domains have been reported in the literature. In what follows, we provide an overview of the related work.
1.4. Related Work

Collins and George [5] apply SIL methodology in designing and analyzing job scheduling algorithms for heterogeneous computing environments. Apart from the actual job scheduling algorithms under investigation, all other subsystems such as the Resource Management System (RMS), network monitor, daemons, database, job creation client, etc. and the machines on which they would normally run (i.e. the computer network) are simulated. Using this framework, experiments are conducted to analyze the performance of the proposed algorithm and to compare it with existing algorithms. The authors present four case studies which illustrate the benefits of using SIL simulation rather than setting up a real heterogeneous computer network notable among which is the fact that jobs could simply be scheduled and not executed, resulting in faster experiments and enabling the use of larger input data sets than would be practical in a real test scenario.

In Demers et al. [9], SIL is compared to traditional simulation methods in the domain of ad-hoc computer network applications. The use of SIL is motivated by the desire to obtain a solution where model fidelity and simulation speed are not competing requirements, model validation is not an issue, and code reusability across design and testing phases is maximized. In addition to meeting these goals, the authors emphasize that SIL offers additional advantages of possible speed up during design, testing ease and repeatability of results. On the other hand, they also highlight some of the challenges of SIL which include a need to modify (albeit slightly) the software to allow it fit seamlessly into the simulation environment and timing issues resulting from running a time-driven software in an event-driven simulator (OPNET). In the two case studies presented, the authors describe methods employed to overcome these challenges.

The SiLEST project\(^3\) (Software-in-the-Loop for Embedded Software Test) focused on developing an automated testing process for software of embedded systems of all kinds within a simulated environment. According to Rebeschieß et al. [23], “attention focuses on identifying the possible applications and limits of the software-in-the-loop (SiL) test method with the aim of conducting ‘in the loop’ simulation at an early stage on a cost-effective, reproducible, retraceable and automatic basis. This process is also compared with the established hardware-in-the-loop (HiL) test method in order to reveal and contrast the benefits and drawbacks associated with both methods”. Several papers were published during the course of the project including: an overview of automated closed-loop testing of embedded engine control software [23, 24] and control functions [25], a description of the XML format for automated SiL testing [14, 15], and a comparison of SiL and HiL as alternative approaches to testing control unit software [17]. A final report summarizing the results of the project is also available (in German) [18].

Using the test bed developed in SiLEST, Maibaum [16] describes two case studies in which SIL is employed to test the robustness of control software in the presence of simulated electrical flaws, e.g. noisy or lost sensor signals, jerky actuators, etc. The test software are the attitude control software (ACS) of the BIRD micro satellite from the space domain, and an engine control unit (ECU) from the automotive domain respectively. The goal of this study was to investigate the adequacy of SIL for evaluating the robustness of control software compared to existing approaches using HiL.

Of the literature surveyed, the SiLEST project is the most related our project objective since it directly applies SIL techniques for robustness testing by injecting faults in the simulated plant. Error injection is currently a feature of the SIL framework at Océ but is limited mainly to lost sensors. Noisy sensors and jerky actuators which are considered in SiLEST are tolerance factors of the respective devices. In this project, we consider this problem in the domain of embedded printer control software for which we aim at developing a model to capture such variability in the simulated printer for better robustness testing of the software. If applicable, useful lessons will be drawn from [18] with proper acknowledgment.

In summary, we conclude from our literature review that the SIL methodology has been successfully applied in several domains such as:

\(^3\)http://www.silest.de/
• embedded control software development [5, 33, 10]
• design and analysis of algorithms for job scheduling in heterogeneous computer networks [5]
• design, analysis and integration of ad-hoc computer networks [9]
• performance evaluation of a large and complex Army communications network [19]
• robustness testing of space control and automotive ECU software [16]
• safeguarding of automatic model-based code generation for embedded software development in the automobile industry [30]

We end this section by highlighting some of the key advantages and disadvantages of SIL simulation mentioned in the literature and/or identified by SIL users at the company:

Advantages of SIL Simulation

1. Ease of testing and possibility for software development without a prototype [9].
2. Code reuse across multiple project phases, e.g. design and testing [9] (see also Figure 1.1).
3. Speed up in development time resulting from higher levels of abstraction of lower layers in the model [9].
4. More extensive and repeatable testing which may be impossible or difficult in real set-ups [5, 9].
5. Unlike HIL, SIL allows for non-real time testing which can significantly improve development and/or integration phases of a project [16].
6. Improved time-to-market [10] (as a result of (1) and (3)).
7. Less risk of damage caused to prototype during testing.
8. Improved quality of final software [16] (as a result of (1) and (4)).
9. Reduction in development cost [16] (due to (1), (2), (3) and (7)).
10. Additional debugging facilities, e.g. extra logging, graphs of motor profiles and other device characteristics, etc.

Disadvantages of SIL Simulation

1. Software may require some modifications to fit into simulation environment [9], e.g. I/O layer [33].
2. Timing failures may be masked as a result of a change in timing when using SIL simulation [16] and/or using time-driven software in an event-driven simulator such as OPNET [9]. These (sometimes subtle) effects may affect the validity of (i.e. introduce errors in) the model.
3. Even with SIL simulation, scalability issues may arise for large scale applications where setting up the other machines “in-the-loop” and keeping software synchronized between all these machines may be challenging [9].

From the foregoing, it is clear that the benefits of SIL simulation outnumber the drawbacks. For the company in particular, the only applicable drawback is the modification of the I/O layer as already mentioned. However, this is elegantly handled using macros. Currently, scalability is not an issue in SIL simulation. Furthermore, implementing SIL in the same software development environment used for developing actual embedded software was in part motivated by a realization of the potential risk of the second disadvantage highlighted above as will be seen in the subsequent section where we provide a brief history of SIL simulation within the company.
1.4.3 Evolution of SIL Simulation at Océ

The Boderc project marked the beginning of a pivotal shift to model-based development at Océ. Conducted in collaboration with the Embedded Systems Institute, Eindhoven, The Netherlands, other industrial partners and university research groups, the main goal of this project was to investigate the use of multidisciplinary design interactions to enable fast, model-based design space exploration for predicting system performance even at early phases of a project. The hypothesis was that such an approach should create significant value in terms of meeting business objectives like time-to-market and design effort per cost. The results of Boderc are summarized in a book [10] which largely accounts for the discussion in this section.

As already observed in Section 1.1, models are invaluable tools for capturing system characteristics and highly facilitate interdisciplinary collaboration in any multidisciplinary development effort such as Boderc. SIL simulation was employed to test embedded software within a simulated (hardware) environment. The three different SIL simulation frameworks investigated within the context of Boderc project are the subject of this section. However, to provide a context, we begin with an overview of the tool used for embedded software development at Océ: RoseRT.

1.4.3.1 Rational Rose RealTime (RoseRT)

Rose RealTime is a UML-based computer-aided software engineering (CASE) tool developed by IBM Rational that supports the development of complex, reactive, real-time embedded software. It supports the real-time object-oriented modelling (ROOM) methodology [28] and is the de-facto software development tool at Océ. Software is built using a combination of active entities called capsules and passive entities corresponding to regular data classes in any object-oriented language like C++. Capsules are unique in that they communicate by message passing via ports. The behaviour of a capsule is modelled by means of a state diagram. According to [6], “the advantage of message-based interfaces is that a capsule has no knowledge of its environment outside of these interfaces, making it much more flexible and robust than regular objects.” Furthermore, capsules represent logical threads of control which facilitate software decomposition and analysis.

RoseRT also features automatic code generation for different target platforms. This is enabled by a Service Layer which abstracts from the target platforms while providing general services such as timing facilities, and controllers for message queuing and delivery between capsules. Moreover, RoseRT supports two ways of validating its UML models: One option is to execute the model stepwise according to the behaviour specified in the state diagrams of the interacting capsules/classes. Each step is associated with processing the next message of highest priority and terminates when all actions that result from this message have been performed. Alternatively, code can be auto-generated from the model and executed. RoseRT supports code generation for C, C++ and Java programming languages.

Both approaches were investigated in developing a SIL simulation framework in the context of Boderc. At this juncture, we remark that the SIL simulation framework at the company does not have any particular name; it is simply referred to as SIL. In fact, only a part of the framework is typically inferred when the term “SIL” is used viz. the simulated plant. This point is elaborated in Section 3.1 when discussing the SIL simulator architecture. In the rest of this work, we stick to that convention and distinguish the simulation framework in use at the company from the generic SIL methodology by explicitly referring to the latter as SIL simulation.

4http://www-01.ibm.com/software/rational/
1.4.3.2 SIL Simulation using RoseRT and Simulink

The first SIL simulation framework built featured a coupling of RoseRT and Simulink. Simulink is a tool developed by The Mathworks for “multidomain simulation and model-based design for dynamic and embedded systems” [20]. It was used to model the simulated plant consisting of a mechanical layout of the printer’s paper path. This continuous model of the physical dynamical system was then coupled with an event-driven control algorithm in RoseRT. A common notion of simulated time was required and chosen to be that of Simulink. An intermediate multidisciplinary coupling tool was used to interface the simulated plant and controller. In order to ensure correct behaviour and time synchronization, the user had to specify the duration of transitions in the RoseRT-based controller to capture the delay associated with performing the (sometimes computationally intensive) actions upon transitions.

The approach was successfully implemented and served as a proof-of-concept for SIL simulation in the context of Boderc. Nevertheless, the dependence on RoseRT transition durations which are, in practice, often unknown and difficult to estimate as well as the absence of suitable Simulink models that could be easily coupled to the RoseRT models limited the use of this simulator within the company and necessitated a refinement in favor of code generation over executing a RoseRT model as done here. The resulting simulator is described next.

1.4.3.3 Integrating Embedded Software in Simulink using TrueTime

As mentioned earlier, RoseRT supports auto-generation of code. To overcome some of the limitations of the previous simulator, C-code was generated from the RoseRT models and simulated by means of TrueTime, a Simulink toolbox. The simulated plant in Simulink together with the periodically executed code interfaced with the plant via TrueTime constituted the new simulator. In addition, a visualization module was included to animate sheet transport along the paper path.

This framework was used within a project at the company where it served, among others, to illustrate the hypothesized reduction in software development time by using SIL simulation. Unfortunately, just like the previous approach, effective use of this framework was hampered by the high dependence on Matlab/Simulink skills. The frequent updates of the simulated plant (as expected especially in early stages of development for which the simulator was targeted) were difficult for software engineers thereby making their progress dependent upon the availability of Simulink experts.

1.4.3.4 A Simulated Plant in RoseRT

Given the challenges of the previous approaches, it was finally decided, for the purpose of improving usability and maintainability, to implement the simulated plant using RoseRT. Given the fact that software engineers at the company are familiar with RoseRT, understanding and extending the simulator was expected to be much easier. In this refinement, the behaviour of the simulated plant was also extended and some earlier simplifying assumptions were lifted. For instance, motor models were developed which could be controlled by software as opposed to previous highly simplified models in which motor profiles from happy flow models were directly used (the interested reader is directed to [10], ch. 6 for details of happy flow). This framework gained more acceptance, quickly became the standard SIL simulator within the company, and is currently used in different projects with steps towards extending it to other printer kinds such as the wide-format series.

This SIL simulation framework, although more extensive than previous ones, still adheres to the basic principle of the happy flow model viz.: an ideal model in which only desired behaviour of sheets and of all other parts are modelled. In other words, it abstracts from disturbances and variations of the modelled components as much as is possible while maintaining fast and reliable results. Nonetheless,
SIL also features simple error injection by stopping sheets and overruling sensors such that they fail to toggle their state upon sheet detection. The framework in this form was the starting point of our project with the goal of further extending SIL with tolerance modelling. In doing so, we still bear in mind the success factors of happy flow notable among which were a good level of abstraction and ease of understanding/use. Therefore, we aim at an extension which captures essential system variability without incurring unacceptable penalties in simulation performance (speed and/or memory usage) and remains conceptually easy to understand. The next section highlights the approach we adopt towards this end.

1.4.4 Remarks about Robustness and Fault Modelling

So far we have clearly stated that one of the main drivers for this work is to enable more extensive robustness testing of embedded control software. According to [1], robustness refers to a system’s dependability with respect to external faults. It characterizes a system’s capability to handle a specified class of faults [32]. Investigating robustness therefore requires the definition of a fault-tolerance model which captures these faults and specifies the expected system behaviour in terms of preventing, detecting and/or handling them. Robustness is then measured in terms of how well the system fulfils the requirements of the pre-defined fault-tolerance model.

In the context of the control software of document management systems, such a fault-tolerance model will include aspects like detection and reporting of paper jams, open doors, empty trays, out-of-range temperatures, etc. Furthermore, the defined response may (for OREs) or may not (for MREs) involve interaction with the user of the system.

In this work, we do not address fault-tolerance modelling. Instead, we focus on providing the means to capture variations in physical components that form part of a printer’s paper path. With respect to the embedded control software, the simulated plant is also part of its “external” environment. Hence, these variations will be particularly useful in simulating expected (and unexpected) error situations resulting from variability in the plant, e.g. actuator delays and sensor tolerances. SIL, extended with tolerances, can also be used to refine the existing fault-tolerance model. Thus, we conclude that the embedded control software is robust if it provides the specified response to all of these faults.

1.5 Approach

To address the problem identified in Section 1.2, namely modelling variability in a simulated machine, we adopt a requirement analysis, design, implementation and verification (RADIV) methodology. The requirements are largely driven by the needs of stakeholders (mainly SIL developers, maintainers and users within the company). The design phase focuses on specifying a solution that meets the identified requirements given the architecture and the existing SIL models. After this design is implemented in RoseRT, verification involves ensuring that the implementation matches the design and performing validation of both against the requirements.

1.6 Contributions

The main contributions of this project to research and development work at Océ are as follows:

1. Designing and implementing tolerance models to capture variability in the following components of a simulated printer: motors, pinches, actuators, sensors, (bent) segments and sheets.
2. Designing and implementing an XML interface for tolerance specification using existing XML files in the SIL simulator.

3. Implementing a python library to enable tolerance specification during automated regression testing with python-based regression testing tool, Test Executor (TE).

4. Providing support for tolerance specification via the existing visualization tool, SILVis.

5. Developing XSD and Schematron schemas, and a command-line tool, XMLValidator, for validating correct XML tolerance specifications.

6. Investigating overlaps in machine layout specifications and providing recommendations for improvements.

1.7 Organization of the Document

The rest of this document is organized as follows: Chapter 2 identifies the project requirements. These are driven by the needs of the identified stakeholders and are captured by several use cases in which SIL is insufficient in its current form to meet those needs. An analysis of the requirements is also presented to clearly delimit their scope.

Chapter 3 presents the design that realizes the outlined requirements. This design covers the tolerance models themselves as well as complementary interfaces for tolerance specification. We analyze the major design options and adopt the ‘best’ choice among these options with motivation. We also pay attention to the existing implementation and, therefore, to the SIL architecture in order to produce a design that does not impose avoidable challenges on the implementation.

Chapter 4 presents the main implementation details. Additional sections are provided to give the interested reader an overview of main mechanisms in SIL such as timing and handling of interface commands from external components. However, the chapter is laid out in such a manner that these sections may be skipped by other readers without a loss of flow.

Chapter 5, revisits and validates the project requirements against both the design and implementation. We also perform verification of the implemented tolerance model by presenting results of test cases defined during the design stage. In addition, we analyze the effect of our extension on the performance of SIL in terms execution speed, memory usage and log file size.

Chapter 6 contains case studies to illustrate how our work applies in real software development. Other contents of this chapter include a general summary of the work done and results obtained, a reflection on the project in retrospect, and possible directions for future work.

Each chapter begins with a preface with indications of which parts may be skipped during a quick (initial) reading and ends with a short summary of the chapter. Borrowing from the notation for describing processes in formal methods research\(^5\), we remark that although the document has a requirement analysis • design • implementation • verification structure, the actual work was conducted by an iterative process in which a preliminary design and implementation was completed for only a small part of the project (tolerance in motors) and then extended to other components. In other words, requirement analysis • (design • implementation • verification)\(^\star\) more closely reflects the project execution.

Finally, to facilitate reading, we distinguish glossary terms using this font. Similarly, methods, classes, parameters, device and file names are denoted using a different font. Occasionally, we also emphasize important points by making them bold.

\(^5\)A dot (•) indicates a sequential composition of steps whereas a star (⋆) indicates one or more iterations over a number of steps.
Chapter 2

Requirement Analysis

This chapter presents and analyzes the project requirements. In Section 2.1, we first identify and classify some sources of variability in different devices. Together with use cases (Section 2.3) derived from the project objective outlined in Section 1.2 and the needs of different stakeholders identified in Section 2.2, these sources of variability serve as key drivers for the requirements subsequently outlined in Section 2.4. Sections 2.4.1 and 2.4.2 address the requirements which are subdivided into high-level (functional) and low-level (implementation) requirements respectively. The goal of this chapter is to answer the question: “What should be modelled?” (Section 2.5)

Recommendations for Initial Reading: Sections 2.1 (except 2.1.1), 2.4, 2.5 and 2.6.

2.1 Sources of Variability

![Figure 2.1: An example paper path (adapted from Figure 19 in [33])](image)

As a sheet travels along a paper path, it may be affected by variability resulting from one or more kinds of (hardware) devices. These devices are considered to be sources of variability with respect to the sheet. Some sources of variability are presented in Table 2.1. They were identified based on investigation of the known characteristics of these devices as well as from discussions with SIL developers, users and domain experts. We note that the characteristics presented in Table 2.1 are not exhaustive but reflect what were perceived to be most important and/or a suitable level of abstraction from the underlying electromechanical characteristics of the devices.

An arbitrary paper path in SIL comprises several hardware devices including those listed in Table 2.1. Figure 2.1 shows an example of a paper path with some of the elements modelled in SIL. The following is
## 2.1. Sources of Variability

<table>
<thead>
<tr>
<th>Device</th>
<th>Characteristic(s) of Interest</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motors</td>
<td>Velocity</td>
<td>In reality, motor velocity may vary within a range unlike the constant value assumed in the current SIL model.</td>
</tr>
<tr>
<td>Sensors</td>
<td>Response time</td>
<td>The detection time of (the leading and/or trailing edge of) a sheet may vary due to sensitivity, placement, wear, dirt, etc.</td>
</tr>
<tr>
<td>Pinches</td>
<td>Diameter, force on paper, pinch slip</td>
<td>Variations may result from production and/or wearing over time, paper type (for pinch slip).</td>
</tr>
<tr>
<td>Solenoids (switches)</td>
<td>Response time</td>
<td>Switches that control paper routes have a worst-(and best-) case response time which should be modelled.</td>
</tr>
<tr>
<td>Clutches and motor enablers</td>
<td>Response time</td>
<td>Clutches and enablers used to (de)activate pinches and motors respectively have a worst- (and best-) case response time which should be modelled.</td>
</tr>
<tr>
<td>Segment of the paper path</td>
<td>Length</td>
<td>Length of paper path segments may vary due to production, coupling and/or use. Paper path length variation may also be used to abstract sheet behaviour at bends (e.g. sheets taking an inner or outer bend).</td>
</tr>
<tr>
<td>Sheet</td>
<td>Length</td>
<td>Factors like humidity, temperature and production may affect sheet length.</td>
</tr>
<tr>
<td>Sheet</td>
<td>Lateral displacement and skewness</td>
<td>Lateral displacement and skewness correction performed along the paper path of real machines are currently not modelled in SIL.</td>
</tr>
</tbody>
</table>

Table 2.1: Identified sources of variability

A brief description of the main “sheet logic” devices as they are commonly referred to in SIL terminology⁴.

**Motors**: The rotation of motors, under the control of software, provides the force that drives sheets along a paper path. The number and kinds of motors used vary from one machine to another, notably among which are stepper motors and (closed-loop) controlled motors. We remark that in SIL, speed is conventionally used to refer to velocity (though they are technically not the same). We adhere to this usage in the rest of this document for the sake of uniformity.

**Sensors**: Paper path sensors are used to detect the arrival of sheets at the specified locations, namely the position where the sensors are placed.

**Pinches**: The speed of a motor is transferred to the sheets via pinches which make direct contact with the sheet and convert their rotational motion into linear displacement of the sheets by rubbing against them.

**Switches**: An active switch may be required to direct sheets into one of multiple possible outgoing paths. This occurs at points where more than two segments meet.

**Clutches and enablers**: They are used to decouple pinches and motors from sheets. Clutches are used to (de)activate pinches. Similarly, motor enablers are used for motors.

**Segments**: A segment represents a piece of the paper path. Segments may contain one or more sensors, pinches, actuators and/or other devices.

---

⁴The term “sheet logic” may have originated from the fact that the devices in the simulated plant are responsible for sheet transport and therefore constitute the logic that governs it.
Sheets: These are the entities that are transported in the paper path. SIL simulation at the company focuses on modelling their behaviour as they travel through an arbitrary paper path under control of the embedded software and as they interact with other sheet logic devices.

2.1.1 Classification of Variability

Independent of the device kind (cf. Table 2.1), we broadly classify variability into two groups, namely static and dynamic variability. In this section, we provide a general overview of these two kinds of variability.

**Static variability:** This refers to those forms of variability that are unchanging in time. In other words, their value is determined once after which it remains constant. We further sub-divide this group into two:

- **Production variability:** Due to batch production of machine parts, two pieces of the same part produced in different batches may have slight variations. For example, two pinches with the same specifications but manufactured in different batches may vary in their diameter. If these pinches are installed in two different printers, it results in variations between the two machines. However, each pinch has a fixed size which is measurable once it has been produced. Typically, these manufacturing-process-dependent tolerances can be calculated and are known in advance. For example, the largest and smallest possible pinch diameter is typically stated on the datasheet provided by the manufacturer.

- **Assembly variability:** When a machine is assembled, different components might be slightly displaced from machine to machine. This kind of variability also depends on the technology used in manufacturing and assembling the machines. Nevertheless, in general, once a machine has been assembled, the positions of components such as sensors, actuators, pinches, motors and segments are fixed. Again, bounds on these tolerances are typically known a-priori based on the assembly process.

Regardless of the source, static variability may lead to having machines of the same configuration/model with different behaviour due to time-independent variations. Although each tolerance effect may be negligible by itself, the combined effect may result in unexpected behaviour in the so-called “Monday morning machines”\(^2\).

**Dynamic variability:** This is refers to the kinds of variability which change in time and/or according to statistical or spatial measures.

- **Time-dependent variability:** This accounts for wearing and tearing due to usage. An example of a time-dependent variability is the wearing away of pinches due to friction over time as the number of sheets they transport increases thereby leading to a change in their diameter. Another example is the reduced range of a sensor as it gets covered by tiny particles with use. Generally, devices degenerate in performance over time.

- **Variability due to device characteristics:** Other dynamic tolerances may result from the physical characteristics of devices. For instance, two concurrent sensor readings under identical conditions may differ owing to inherent sensor properties. The same is applicable to actuator switching delay and motor speeds. These characteristics may also have time dependency.

---

\(^2\)A Monday morning machine is one composed of the worst components, assembled in the worst manner (e.g. on Monday morning at the company after a weekend in which The Netherlands lost the FIFA World Cup final) such that things go wrong when control software is not robust.
It is not always easy or even possible to express dynamic variability in an equation or as a statistical (including a random) distribution. In these cases, approximations based on experiments may be used to derive tolerance bounds. Furthermore, we observe that although the above classification is sufficient to group most dynamic effects, there exist subtle effects which do not fit into this classification, e.g. the change in sheet length due to heating/cooling effects in regions of the paper path.

In the context of this project, the above classification is of little or no consequence; what is relevant from the standpoint of software verification is that regardless of the kind of variability, the embedded control software should be able to handle the best- and worst-case scenarios as well as anything between. In what follows, the exact source of variability will therefore seldom be of interest.

This observation also implies that, when necessary, we will abstract from the different kinds of variability and define a single tolerance measure for a device kind which represents the dynamic and static variability components combined. Nevertheless, such a classification may be useful for the person who has to decide upon the weights of each tolerance effect in the cumulative tolerance factor.

### 2.2 Main Stakeholders

Different individuals use SIL for different purposes. Thus, depending on the context of their work, certain aspects of a tolerance model may be more useful to them than others. With regard to this project, these individuals are all stakeholders given their common interest in an extended SIL framework with some form of tolerance modelled. In this section, we briefly identify our stakeholders as follows:

- **Software developers (SD)** use SIL to verify the functionality and robustness of their software code preventing their bugs from being discovered by others at a later stage or passing through undiscovered and resulting in unexpected machine behaviour.

- **Software integrators (SI)** use it to detect incorrect functional behaviour while integrating software from different developers, allowing for earlier correction, i.e. before deployment on the real (test) machine.

- **System engineers (SE)** have the same interest as software integrators except at a higher level. For them, the presence of SIL has advantages of reduced number of tests on the real machine which might be especially useful when there are limited number of prototype machines.

- **(Regression) Test engineers (RTE)** at different levels use SIL to uncover bugs in the system. The opportunity for overnight testing has the advantage of possibly shortening the duration of test procedures.

- **SIL maintainers (SM)** are responsible for the architecture of the simulator among other things. Any change therefore requires their knowledge and consent.

The abbreviations in parenthesis are used to identify stakeholders’ interest in particular requirements outlined in subsequent sections. We also distinguish software developers working on the main printer paper path from those working with the finisher module when their interests differ.

### 2.3 Use Cases

Whether presented as kindergarten-level actors (with a circular head and stick body) interacting with each other or textually as done below, use cases provide a convenient means to capture scenarios
which highlight the expected functionality of a system. They are sometimes equally employed to clarify (functional) requirements. Here, we use them to illustrate who should be able to do what with the SIL framework extended with tolerance modelling. These use cases therefore serve as a motivation for the requirements outlined later. We observe that the first two use cases relate to normal operation of the existing SIL framework and do not require any tolerance modelling.

1. A software integrator performs periodic regression test in which (s)he runs a number of pre-defined automated test cases and analyzes the results.

2. A software developer prepares and runs a test to verify that his/her operator-recoverable error (ORE) and/or machine-recoverable error (MRE) routines work correctly.

3. A software developer investigates the robustness of his/her paper handling routine by specifying in a test case that a particular motor runs at its maximum (minimum) speed or 10% faster (slower).

4. A software integrator varies the response time of active switches in a duplex loop to determine if the control software is robust against collisions at the turn station given the length of the duplex loop, the speed of the sheets and other parameters, and assuming the tolerance ranges of these switches are known.

5. A test engineer specifies that the diameters of all pinches randomly vary bidirectionally by 5% of the nominal value and runs several test cases to investigate robustness of the embedded control software to this variability.

6. A test engineer, upon running a (random) test that triggers an error, analyzes the results and extracts the exact device parameter values that caused the error. Using the same values, (s)he is always able to reproduce the same results.

7. A software developer investigates the effects of different kinds of paper on the detection time of sensors along the paper path by specifying that a given sensor detects a particular sheet as early as possible and the next sheet as late as possible.

8. A system engineer checks the robustness of the embedded control software to wear and tear by varying the diameter of pinches, response times of clutches, detection time of sensors, etc. in order to determine the conditions under which the software fails and to check if they are outside the fault-tolerance model ranges of these devices.

9. Using an automated testing framework, a test engineer specifies tolerances for devices of interests, runs test cases and obtains identical results as when the same tests are done manually.

10. A software developer adds a new simulation element which is specific to his/her project, specifies tolerances for these devices and uses the results (e.g. motor profile graphs) to fine-tune and/or develop robust software.

### 2.4 Project Requirements

Having outlined some use cases in the preceding section, we shift our attention to the project requirements. These requirements are divided into two, namely high- and low-level requirements. Whereas the former capture the functional requirements of the extended SIL framework, the latter focus on implementation (i.e. non-functional) requirements. Put differently, the high-level requirements place direct constraints on the design while the low-level requirements do the same for the implementation. For the purpose of separation of concerns, we address them separately in Sections 2.4.1 and 2.4.2 respectively. Each set of
requirements is immediately followed by an analysis. We also assign priorities to the requirements based on the MoSCoW prioritization method\(^3\).

### 2.4.1 High-level Requirements

The high-level (i.e. functional) requirements are summarized in Table 2.2. They were identified following discussions with stakeholders and consideration of the relevant use cases from the previous section.

<table>
<thead>
<tr>
<th>ID</th>
<th>Brief Description</th>
<th>Priority</th>
<th>Stakeholders</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR_1</td>
<td>To ensure that the extended SIL framework is \textit{backward compatible} and that the changes required to implement the desired functionality are as localized as possible</td>
<td>Must</td>
<td>ALL</td>
</tr>
<tr>
<td>FR_2</td>
<td>To implement the framework using the \textit{existing} modelling software at the company (IBM Rational RoseRT) and according to the existing software development guidelines</td>
<td>Must</td>
<td>ALL</td>
</tr>
<tr>
<td>FR_3</td>
<td>To provide a means to specify the variability in one or more device characteristics which affect the \textit{sheet transport} along an arbitrary paper path</td>
<td>Must</td>
<td>SD, SE</td>
</tr>
<tr>
<td>FR_4</td>
<td>To ensure that results are \textit{accurate} and \textit{reproducible} and identical for all executions of the same test case</td>
<td>Must</td>
<td>ALL</td>
</tr>
<tr>
<td>FR_5</td>
<td>To allow \textit{case of specification} of variability, e.g. options to set a given characteristic to its worst-case (or best-case) value for a subset of devices in a single step</td>
<td>Should</td>
<td>SD (printer)</td>
</tr>
<tr>
<td>FR_6</td>
<td>To model the detection and correction of “\textit{z-displacement}”(^a) and \textit{skewness} along the paper path</td>
<td>Could</td>
<td>SD</td>
</tr>
<tr>
<td>FR_7</td>
<td>To model \textit{delays} in the (worst-case) response times of sensors and actuators especially segment switches</td>
<td>Must</td>
<td>SD (finisher)</td>
</tr>
<tr>
<td>FR_8</td>
<td>To unify the \textit{layout specification} XML file with the MoBasE schema to remove unnecessary ambiguity and redundancy</td>
<td>Should</td>
<td>SD, SM</td>
</tr>
<tr>
<td>FR_9</td>
<td>To model sheet buffering effects such as blousing in a paper path.</td>
<td>Won’t</td>
<td>SD, SM</td>
</tr>
<tr>
<td>FR_10</td>
<td>To model sheet delay at an arbitrary point of interest along the paper path.</td>
<td>Could</td>
<td>SD (finisher)</td>
</tr>
</tbody>
</table>

Table 2.2: Project high-level requirements

\(^a\)“z-displacement” is a term used to describe the lateral displacement of a sheet relative to the segment edges. It is actually a misnomer as this phenomenon takes place in the x-y plane. Thus, “y-displacement” may be a more appropriate name.

### Analysis of the High-level Requirements

FR_1 Backward compatibility is required to allow the execution of already existing test cases. Localization is desirable because it makes it easier to identify the changes that have been made during integration into the mainstream software tree by SIL maintainers.

FR_2 Familiarity with RoseRT within the company, ease of integration and reuse are the main drivers for this requirement.

FR_3 This requirement underlies the main goal of this project, i.e. to enable the analysis of robustness of control software developed at the company. It does not specify exactly what needs to be modelled. These will be derived from the sources of variability outlined in Table 2.1 and finalized after considering the question of how to model variability.

\(^3\)MoSCoW “is a prioritization technique used in business analysis and software development to reach a common understanding with stakeholders on the importance they place on the delivery of each requirement - also known as MoSCoW prioritization or MoSCoW analysis” (see http://en.wikipedia.org/wiki/MoSCoW_Method).
2.4. Project Requirements

Requirement Analysis

FR_4 This requirement is inherited from the original requirements for SIL. A direct consequence of this requirement is that sufficient information to regenerate the test scenario must be logged.

FR_5 Ease of specification is aimed at making the testing process less tedious and thus, causing the extended framework to gain faster acceptance among the targeted stakeholders.

FR_6 This requirement is motivated by a desire to make the simulated plant more representative of the actual machine where “z-displacement” and skewness correction are performed.

FR_7 Proper inter-sheet timing is essential for performance analysis and improvement. Modelling the response times of sensors and actuators facilitates the process of validating the existing (and future) control software against these timing requirements.

FR_8 MoBasE stands for model-based engineering. Part of the goal of this effort is to develop an interdisciplinary model template which describes the essential elements of a printer prototype and can be instantiated for various projects at the company [27]. There are overlaps in elements specified in different XML files required by SIL and those in the corresponding MoBasE specification for the paper path. This requirement aims at identifying and removing such overlaps and the undesirable ambiguity they create.

FR_9 Similar to FR_6, this requirement is driven by creating a more representative model of the printers developed at the company. However, this requirement will not be implemented in this project owing to time constraints and the fact that the problem is already being investigated by others.

FR_10 Currently, it is possible (at run-time) to stop or remove sheets at points of interest via the provided interface from SIL to external components such as TE. However, to verify particular error handling routines around the flipping wheel in a new finisher module, it is necessary to delay sheets (and thereby reduce the distance between them until an error is triggered so the error handling routine can be checked). In principle, such delay can be modelled using the tolerances (to be modelled) in motors, pinches and other components. However, the calculations required to translate the tolerances in these components to delays in sheets of interest are likely to be involved and undesirable. Therefore, the SD (finisher) stakeholder involved desires a direct interface to delay sheets.

2.4.2 Low-level Requirements

The low-level requirements are related to implementation. In particular, they define the constraints on the interface provided to different users for tolerance specification. This is important because the different stakeholders interact with SIL in different ways as will be elaborated upon in Section 3.1 of Chapter 3.

<table>
<thead>
<tr>
<th>ID</th>
<th>Brief Description</th>
<th>Priority</th>
<th>Stakeholders</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR_11</td>
<td>To provide interfaces for introducing variability into SIL which are compatible with the existing framework for automated regression testing (i.e. TE)</td>
<td>Must</td>
<td>SD (printer), SE</td>
</tr>
<tr>
<td>IR_12</td>
<td>To support the specification of tolerance values for components of interest without explicit use of TE, e.g. in a(n) (XML) file</td>
<td>Must</td>
<td>SD (finisher)</td>
</tr>
</tbody>
</table>

Table 2.3: Project low-level requirements
Analysis of the Low-level Requirements

IR_11 This requirement is motivated by the fact that TE is currently being used for automatic regression testing and provides a convenient means for extensive testing (e.g. overnight tests). This framework is familiar to the software engineers and they desire to continue using it with the extended SIL framework. Furthermore, the requirements implicitly demands that the interface methods added should be similar to the already existing one and should allow for specifying variability either in a single component or in a group of similar components, e.g. a specific motor versus in all motors.

IR_12 Other tools such as UT exist which can be used independent of TE. It is desirable to maintain support by allowing tolerances to be specified in predefined files which can then be overridden by TE if necessary, depending on the configuration decided upon.

2.5 What to Model

Having identified and analyzed the requirements (summarized in Tables 2.2 and 2.3), we now address the issue of what to model. The characteristics of interest identified in Table 2.1 suggest that a notion of tolerance can be defined for every kind of sheet logic device currently modelled in SIL with a direct or indirect consequence of sheet transport along an arbitrary paper path. However, the requirements neither explicitly state that tolerances should be modelled in all sheet logic devices nor identify a single device whose variability has the greatest impact on sheet transport. We investigated two options, namely:

Option I: To lump all tolerances in a single device kind

Advantage(s)

1. Ease of implementation: Expressing tolerances in only one device minimizes the changes to be made to SIL and the attendant risk of introducing bugs.

Disadvantage(s)

1. Converting variability in one device to another may be non-trivial: For instance, suppose we assume all variability is expressed as tolerance in motor speed, it become rather challenging to convert the delay of a sensor or actuator into an equivalent speed parameter (we return to this issue in Section 3.3.1.9).

2. It will be difficult to localize a problem to a particular device when robustness tests fail: For example, if a test case fails when motor ‘X’ is run say 90% of its nominal speed (where this factor is a weighted sum of tolerance effects from all device kinds), it would be extremely difficult to decide whether the failure should be attributed to late sensor detection, delayed actuators, the motor itself, and/or some other device(s). As a result, improving the control software to handle the scenario also becomes rather challenging if not impossible.

3. Deciding on which device is best to express the single tolerance factor will require analysis of tolerance specification in all device kinds.

Given the significant drawbacks of option I, particularly the third disadvantage, we considered another option:

Option II: To investigate tolerance specification in all devices identified and to determine (at least) one tolerance measure for each device kind
2.6 Chapter Summary

**Requirement Analysis**

**Advantage(s)**

1. No conversion will be needed between variability of different kinds (although an abstraction from different tolerance effects in the same device kind may still be required).

2. The analysis of tolerance specification for all device kinds will enable a sound decision on what is best, e.g. whether multiple tolerance factors per device kind are necessary.

3. Localizing a problem in the event of an error will be relatively easier at least to the level of a particular device kind.

**Disadvantage(s)**

1. More extensive and careful design and implementation will be required.

**Decision:** To investigate tolerance specification in all kinds of devices in the simulated plant.

Based on the analysis above, we decided to adopt option II. Therefore, with the exception of the force on paper by pinches (which is related to paper buffering and out of the scope of this project, cf. FR_9), we considered all other characteristics of interest for modelling.

2.6 Chapter Summary

The focus of this chapter was to define and analyze the project requirements. We achieved this by identifying the stakeholders and defining use cases that capture their various needs/expectations. We divided the requirements into functional (high-level) and implementation (low-level) requirements. The importance of each requirement was expressed by assigning a suitable priority to it. This facilitates a common understanding among all stakeholders of the expected deliverables as well as a yardstick for assessing success at the end of the project. Following analysis of the different sources of variability alongside the requirements, we decided to investigate tolerance modelling in each device kind in the simulated plant in order to determine what to model. The next chapter elaborates on the design, highlighting the major choices/abstractions involved in developing the tolerance models presented there.
Chapter 3

Design

In this chapter, we discuss the major design choices made in order to realize the project requirements which were presented in the preceding chapter. We provide answers to two important questions, namely: “Where should tolerance be specified?” and “How should it be specified?”. We then apply the answers to these questions in developing a tolerance model which forms the basis of the implementation in Chapter 4. To provide a background, we first consider the current architecture of SIL in Section 3.1. After addressing some general design issues in Section 3.2, we present our tolerance models for different device kinds in Section 3.3. This is followed by the design of an XML interface for tolerance specification in Section 3.4 and the corresponding interface methods for use by external tools which interact with SIL in Section 3.5. Finally, we define test cases for verifying the implementation in Section 3.6.

Recommendations for Initial Reading: All sections except 3.3, 3.4.2.6 and 3.6.

3.1 The (Existing) SIL Simulator Architecture

The architecture of the SIL simulator is presented in Figure 3.1. The simulator comprises two main components, namely a simulated machine and the embedded software. In the literature, these components are more generally referred to as the plant and controller respectively [29, 12]. The simulated machine models the different hardware components which make up a real machine such as motors, sensors, switches, clutches, and pinches. It mimics the behaviour of an actual machine and interacts with the embedded software via a modified I/O layer. This software is the actual control software that runs in the real machine and gives rise to name of the methodology software-in-the-loop, i.e. real software in a loop of simulated (hardware) components. Within the company, the term “SIL” is conventionally used to refer to only the simulated machine and not to the embedded software. We maintain that convention in this document and will use the term “SIL simulator” when making explicit reference to the whole simulation set-up as depicted by the corresponding block in Figure 3.1 and “SIL simulation” when referring to the software-in-the-loop methodology.

Test Executor (TE) is an in-house tool for automated (regression) testing of embedded software. It interfaces with other components such as SIL, UT, RC, and SILVis to perform automated test runs of the control software. TE communicates with these components via TCP/IP socket connections.

One way TE interacts with the embedded software is using another tool known as the Remote Control (RC). It does this by generating scripts (FScript files) which are executed by RC. The actual commands generated and sent to the embedded software depend on the functionality being tested. The software executes these commands and generates a dprintf log which TE then checks for an (un)expected pattern signalling success or failure of the executed test(s).
A direct socket connection from TE to SIL allows for simple run-time manipulation of the simulator. With this interface, it is possible to send so-called points-of-interest (POI) commands which cause SIL to perform certain actions either immediately or when the specified conditions are met. Currently, actions are limited to stopping/injecting/removing sheets at specified positions on the paper path, changing the length of new sheets to be injected, and setting or reading sensors/actuators/motor speeds. Using these commands, simple error injection can be performed, e.g. simulating a faulty sensor by forcing it not to toggle. Therefore, this connection is primarily used for error injection.

**Universal Tester (UT)** is another in-house tool which is also used for testing embedded software. Similar to RC, it connects to the control software and not directly to SIL. However, unlike RC, UT realizes communication with the embedded software using an emulated serial (RS232) interface.

Both RC and UT are commonly used (and supported) by engineers within the company. These tools are interfaced with TE for automated testing but they may also be used without it. Moreover, the choice between these two tools is related to the level at which testing is being performed which, in turn, is a function of the layered software architecture of the printers depicted in Figure 3.2. Apart from the central control node (main-controller) which receives print jobs from the external world, a printer may
have one or more sub-controller nodes which handle different aspects of the job. For instance, in the scenario shown in Figure 3.2, a job to print a document is decomposed by the main-controller which instructs the paper input module (PIM) to inject a sheet from a particular input tray at the desired time. In order to realize this, the PIM interacts with one or more actuators and sensors via a hardware-dependent I/O layer. Similarly, the paper path handling (PPH) module is responsible for transporting the sheet through the paper path. Once printing is completed, the finisher module may then be required to perform post-printing tasks like stapling before the printed document is output.

![Figure 3.2: Logical view of typical control software architecture (adapted from Figure 6 in [33])](image)

RC is used when testing the main controller node with one or more sub-controller nodes. Nevertheless, the sub-controller software can be tested independent of a main controller using UT. We remark that the embedded software layer in SIL (Figure 3.1) corresponds to the main- and sub-controller nodes of a real printer (Figure 3.2). However, the I/O layer and hardware are simulated by the modified I/O layer and simulated machine respectively in SIL. Furthermore, we note that the architecture presented in Figure 3.2 abstracts from specific details and therefore may differ from that of an actual printer. However, the figure suffices to illustrate the layered software architecture and the mapping to the SIL simulator.

SIL can also be connected to a visualization tool, SILVis, which gives a visual representation of the paper path of the machine under consideration. State information is periodically sent from SIL to the visualization tool over a socket connection thereby allowing the user to visualize the flow of sheets in the paper path. This tool also uses SIL’s provided interface methods for injecting errors at desired points of interest, via the same commands used by TE. Furthermore, the visualization tool features a widget which may be used to manipulate the simulation speed. This functionality is realized using the Time Manipulator dynamic-link library (DLL) file.

At this point, we note that the description of the SIL simulator so far abstracts from fine details; we only provide what is necessary to gain an understanding of how it works. In the actual implementation,
there are other DLLs which are used to enable some of the functionality hitherto described. For example, the main and sub-controller nodes are included as DLLs. Also, communication between these nodes is achieved over a CAN bus which is modelled using pipe servers in the simulator and for which another DLL (ntCan.dll) is required. All these are hidden within the embedded software block in Figure 3.1. In principle, it is also possible to extend the simulator with more functionality using custom DLLs or executables.

Currently, SIL reads information from three separate XML files, all depicted in Figure 3.1:

1. The topology of the machine (e.g. name, position and characteristics of different motors, sensors, actuators, paper path segments as well as their interconnections) is contained in the file layout.xml. It is possible to specify a different layout file as a command-line parameter to SIL. In the rest of this document we refer to this file as the layout file or the machine description file.

2. Information related to particular test cases such as the duration of a test case, the update frequency of the simulated machine, which nodes to load, etc. is specified in the file SILSimulation.xml. This file can contain multiple test cases, each having a unique identifier. Unlike the layout file, however, the name of this file is currently hard-coded. Thus, multiple test case files are not (yet) supported by SIL. We will refer to this file as the test case (specification) file.

3. Another file, graphs.xml, is used to specify parameters for plotting graphs for different device parameters, e.g. motor speeds, sensor and actuator states, sheet displacement, etc. Parameters that can be specified in this file include the colors, styles, and offsets for different plots. The information in this file can then be used to generate custom (default) graphs at run-time. For this purpose, the graphing tool DPlot\(^1\) is used.

In addition to the messages printed in the user interface console, SIL also generates one or more log files per test case; the number and contents of these log files depend on the mode in which SIL is run (e.g. silent versus normal mode). These are collectively represented in Figure 3.1 as SIL logging. Other log files may be generated by the simulator notable among which is the aforementioned application-specific dprintf logging which is also used by TE to verify test results. The dprintf logging is the same as that produced in the real printer (because the actual embedded software is used in the simulator). Bearing this overview of the architecture of SIL in mind, we now consider a number of design issues.

### 3.2 Tolerance Specification: Design Issues

In this section, we address the subject of specifying tolerances in SIL by analyzing the following issues: (1) what to specify; (2) where; (3) how to specify it; (4) what kind of input probability distribution to use; and (5) whether or not to allow tolerances by default. Although they are quite orthogonal design issues, we present them in an order that illustrates how insights from one affect the other(s).

#### 3.2.1 What to Specify

Devices vary in the way their tolerance characteristics are specified. For example, whereas a motor’s speed may be characterized by a minimum and maximum deviation from its nominal value, the response time of a solenoid may be characterized by its best-case and worst-case values. However, regardless of the device in question, we assume that a lower and/or upper bound on the tolerance for a given characteristic exist(s). How this is specified, e.g. an absolute value or relative to the nominal value is a separate issue (addressed in Section 3.2.3).

---

\(^1\)http://www.dplot.com/index.htm
3.2. Tolerance Specification: Design Issues

We identify generic attributes \texttt{minTolerance} and \texttt{maxTolerance} to represent the lower- and upper-bound of the tolerance characteristic in question respectively. To be able to model tolerance in a given device, we require that both parameters must be specified. If this tolerance range is not specified, the device is assumed to be ideal, i.e. it has no tolerance with respect to the characteristic under consideration. Also, we note that the actual names of these two generic attributes in the implementation may vary. Furthermore, we remark that these parameters are inherently machine properties since they depend on the characteristics of the device in question.

According to use case 3, the user should also be able to specify a particular tolerance value from a device’s tolerance range. In addition, it should be possible to specify different values for different test cases. Thus, for each device modelled, the necessary interface(s) to realize this functionality must be provided (cf. FR_3) although they may vary from device to device.

To meet the requirement of reproducibility (FR_4), it is necessary to provide (an) interface(s) for specifying all parameters needed to regenerate a particular test case (cf. use cases 6 and 9). When specific tolerance values are enforced by the user, storing these values is sufficient. However, for the scenario depicted in use case 6 using randomly generated tolerance values, extra information must be stored. For example, if a (pseudo-)random number generator is used to generate tolerance values from devices’ tolerance ranges, the seed and/or any other information necessary to reproduce the same sequence of tolerance values must be stored.

To conclude, we itemize the parameters that need to be specified as derived from the use cases, requirements and discussion in this section:

1. Tolerance ranges bounded by a minimum (\texttt{minTolerance}) and maximum (\texttt{maxTolerance}) value with the requirement that both of these parameters must be specified for non-ideal devices. Tolerance ranges are device characteristics.

2. The exact tolerance per test case may be any value from the tolerance range and determined from the test case specifications. The interface(s) necessary to realize this functionality in all devices considered must be specified. Tolerance values should be test-case specific.

3. Information required to reproduce every test case despite possible ‘randomness’ in the tolerance values generated must be stored.

3.2.2 Where to Specify Tolerances

In the previous section, we identified that two types of information need to be specified, namely: tolerance ranges and test-case-specific values from these ranges. We considered four options of where to specify this information. These options are highlighted in Figure 3.1 as numbered circles. We now analyze each of them. Before that, however, we observe that the decision made when SIL was initially designed to use XML files for specification is upheld for the same reasons outlined in the original design report (see Section 6.1 of [33]). Therefore, the alternatives considered in this section are all based on XML files.

Option I: To specify all tolerance information in a new XML file

\textbf{Advantage(s)}

1. The same file can be read both by \texttt{TE} and \texttt{UT}.
3.2. Tolerance Specification: Design Issues

Disadvantage(s)

1. Including yet another file makes test case specification more tedious and results in a more-complicated architecture.

2. Tolerance information logically fits into the machine specification and/or test case files.

3. This approach may make the requirement of migrating to MoBase machine description (FR_8) more difficult to achieve.

4. It may involve undesirable and avoidable changes to the TE interface to SIL (cf. FR_1(b)).

Option II: To specify all tolerance information in the test case specification file

Advantage(s)

1. All tolerance information will be contained in a single (existing) file.

2. The number of XML files to be parsed by SIL remains unchanged, i.e. the current architecture is retained.

Disadvantage(s)

1. As seen in Section 3.2.1, some tolerance information, e.g. the tolerance ranges for devices, are characteristics of the machine and not test-case-specific. Having such information in test case file is counter-intuitive and will result in unnecessary repetition of the same information for each test case.

Option III: To specify all tolerance information in the machine description file

Advantage(s)

Same as for option II.

Disadvantage(s)

1. In Section 3.2.1, we noted that it will be desirable to inject specific tolerance values for particular test runs. Arguably, such information is test-case-specific and should not logically be included in the machine description file.

Option IV: To split tolerance specification between the test case and layout files

Advantage(s)

1. Retains the advantages of option II above.

2. The specification is logical and intuitive. This will both facilitate the process of specification as well as debugging. For the latter, unexpected behaviour which is specific to a particular test case will be debugged in the test case specification file and vice versa.

3. Utility of the test case information file increases (since it currently contains little information compared to the layout file).
Disadvantage(s)

1. If care is not taken, ambiguity may result for instance from overwriting tolerance ranges within the test case specification file (see Section 3.4.2.2 for how this problem is avoided).

Decision: To specify tolerance ranges in the layout file and test-case-specific values in the test case file.

From the analysis above, options II and III are quite counter-intuitive. Option I includes the possibility of altering the existing TE interface which is avoidable by choosing one of the other options. That leaves option IV which is intuitive and combines most advantages from the other approaches. Notice that we do not consider graphs.xml at all, the reason being that this file is hardly used and has the same characteristics as options II and III.

Therefore, we decided to specify the tolerance ranges and/or worst-case values in the machine description file; other parameters identified in Section 3.2.1 will be contained in the test case specification file. Any new attributes to be defined in the future should be assigned to one of these two files by the same logic, i.e. to the file where it most intuitively fits.

3.2.3 How to Specify Tolerances

Tolerances can be specified as fractions (percentages) relative to the nominal value or in absolute terms. For example, the tolerance range of a motor’s speed may be specified as [0.99, 1.01] where this is understood to mean that the speed varies between from 99% to 101% of the nominal value. Alternatively, it could be specified in absolute terms. Suppose, the motor’s nominal speed is 10 mm/s. Then the corresponding tolerances will be [9.9, 10.1].

The advantage of using relative values should be obvious: It does not require a knowledge of the nominal values. Furthermore, in the event the tolerance range in a device’s datasheet is specified in absolute terms, the conversion is trivial. Thus, we decided to specify tolerance as fractions relative to a nominal value.

Decision: To specify tolerances as fractions relative to the nominal value.

In line with this decision, we placed no restrictions on the minimum and maximum percentages to be specified. This is because for software robustness testing, it might be desirable to specify wide ranges that possibly exceed the actual tolerance range of the device.

Nevertheless, in implementation, we required that the following precondition be met:

If the tolerance range for a device characteristic is specified, i.e. [minTolerance, maxTolerance] is defined, then minTolerance ≤ nominal value ≤ maxTolerance, where nominal value is the ideal value of the characteristic in question.

3.2.4 Input Probability Distribution

When a specific tolerance value is not assigned, the value of a parameter of interest (e.g. motor speed) must be selected from the tolerance range according to a pre-defined distribution. In [13], different discrete probability distributions are identified. These include: Bernoulli, uniform, binomial, geometric, negative binomial and Poisson. Assuming the availability of input data, means to determine the most
3.2. Tolerance Specification: Design Issues

Design suitable input probability distribution along with validation techniques are proposed. While it is possible to apply the ideas presented there in order to determine the most suitable distribution, the availability of data and time constraints made this option infeasible. Thus, we decided to use a uniform distribution. In other words, each value within the tolerance range has an equal chance of being selected.

Furthermore, we decided to use the pseudo-random number generator available in C(++) rather than attempting to implement one as explained in [13], for instance. The granularity of the generated numbers is defined by the range of the random number generator. This is given by the constant `RAND_MAX` defined in `<cstdlib>`. Its default value may vary between implementations but it is granted to be at least 32767. We assume random values not larger than the size of an integer (which is also system-dependent; typically it is 4 bytes).

Decision: To use a (discrete) uniform input probability distribution generated using the pseudo-random number generator implemented in C(++).

We take advantage of the inherent determinism in pseudo-random number generators to enable easy reproducibility of test cases as follows: Given the fact that a for a particular “seed” value and a given range, the random numbers generated by method `rand()` in C++ is completely deterministic, it is sufficient to store the seed for any particular test case in order to reproduce the sequence of values generated. Since the random number generator is implemented in a standard C library and not as a C++ class for which multiple instantiations would have been possible (e.g. one instance per device kind), a system-wide (i.e. global) seed is assumed. Options for specifying multiple seeds and different input probability distributions are highlighted in Section 6.4 as possible directions for future work.

3.2.5 Default Tolerance Mode

Another issue of interest is whether generating random tolerance values should be enabled by default in the extended SIL framework. This decision may affect the interface (both XML and methods) defined for tolerance specification. For instance, if tolerance is on by default, there may be no need for an interface method to turn it on and vice versa.

Having tolerance specification turned on by default has the advantage that it more closely models a real machine where variability is present by default. However, this approach may have a serious drawback: Generating random tolerance values by default may lead to wrong interpretation of results. Consider a scenario where a SIL user forgets to turn off tolerance in all devices except the one of interest. (S)he then runs a test case in which an unexpected behaviour occurs due to the combined effect of the tolerance (s)he is investigating and the ‘hidden’ tolerance (s)he inadvertently failed to turn off. Such a subtle situation may be very annoying and difficult to uncover, possibly leading to a wrong conclusions.

On the other hand, turning tolerance off by default causes the user to be completely in control. Any tolerance specification is explicit and the user is therefore forced to think about what (s)he is doing. Secondly, this approach reflects the expected software design trajectory in which the engineers first test their software for functional correctness on an ‘ideal’ simulated plant before performing robustness tests using the tolerance interface.

Decision: To turn tolerance in all devices off by default regardless of whether their tolerance range is specified or not.

In the next section, we investigate tolerance models for the different device kinds currently modelled in SIL.
3.3 Modelling Tolerance in the Simulated Plant

In this section, we address the main subject of this project, namely the actual tolerance models for different devices in the simulated machine.

The tolerance models described next were developed iteratively. In the preliminary design step, all tolerance was abstracted into one device kind, namely the motors. However, this abstraction was seen to be insufficient to meet some of the project requirement thus necessitating tolerance modelling in other device kinds. Since this project focuses on extending an existing framework, a good design should take into account the existing implementation. In other words, while considering the properties of these devices, we bear in mind options that enhance concise yet correct implementation. Therefore, the discussion includes brief explanations of the existing SIL models to provide a context for the proposed extensions for tolerance specification.

Preliminary Design

The goal of this project is to capture the essential tolerance parameters which affect sheet transport. To achieve this, we begin by considering tolerance in motors. In other words, we assume that all tolerance in the system can be expressed in terms of variability in motor speed. The choice for motors is motivated by the fact that they provide the torque that drives pinches which move the sheets. Successfully modelling motor speed tolerance will also provide useful experience with RoseRT and a means to reason about the problem at hand without dealing with the complexity of the whole system at once. Furthermore, a model must be as concise and simple as possible. Therefore, it is possible that such a model could be a sufficient level of abstraction to express system variability.

3.3.1 Tolerance in Motors

An electric motor is an electromechanical energy conversion device which takes in electrical energy as an input and produces mechanical energy (rotation) as an output. Without motors, sheet transport along a paper path will be practically impossible. Several kinds of motors are used in the company’s printers. The details of these motors are out of the scope of this work. Instead, we focus on the different motor models in SIL.

Currently, five kinds of motors are implemented in SIL. They are: (1) stepper motor; (2) variable speed stepper motor; (3) controlled motor; (4) stubbed motor; and (5) steered motor. Before discussing their characteristics, we first explain how sheet position and speed are currently calculated in SIL. This will be useful when deciding upon how to model tolerances in motors.

3.3.1.1 Updating Sheet Speed and Position

Whenever a motor is updated, its new speed (in mm/s) and the corresponding linear displacement are calculated. If the motor is enabled, the speed and displacement of the connected pinches are updated to those of the driving motor. The actual speed (or displacement) of a pinch is then the product of the speed (or displacement) of the driving motor and the transmission ratio between the motor and pinch.

At a given time, a sheet may be under the control of several pinches. In reality, depending on the relative velocities of these pinches, interesting effects like blousing may occur. Blousing refers to the buffering of paper when the velocity of the pulling edge is less than the velocity of the pulled edge of the sheet (assuming that all pinches rotate in the same direction). These effects are not modelled in SIL yet.
Instead, each pinch is assigned a force and under the control of multiple pinches, the one with the largest force drives the sheet.

In the current implementation, a sheet’s speed equals the speed of the strongest pinch driving it but its displacement is a consequence of average pinch speed. This means that in order to determine sheet displacement, the average speed of the strongest pinches at the previous and current times is multiplied by the elapsed time, i.e. $\text{displacement}_s = \Delta t \cdot \frac{\text{speed}_p(t_i) + \text{speed}_p(t_f)}{2}$, where subscripts $s$ and $p$ refer to the sheet and strongest pinch respectively; $t_i$ is the previous time; $t_f$ is the current time; and $\Delta t = t_f - t_i$.

The averaging approximates the change in sheet speed between two consecutive samples either because the driving pinch changes or the same pinch is accelerated.

From the discussion so far, we conclude that the parameter of interest with respect to sheet transport is a motor’s speed which is transmitted to the sheet via pinches. Thus, we model tolerance in terms of motor speed. We express this speed tolerance as a fraction of the nominal motor speed and bound it by $[\min\text{Tolerance}, \max\text{Tolerance}]$. The actual speed of the motor is then given by the nominal speed multiplied by a tolerance value obtained from this range. A motor with no tolerance effectively has a single-point range bounded by $[1.0, 1.0]$.

The factors that contribute to the variability in motor speed vary from one motor kind to another. For example for a closed-loop controlled motor, speed variability may result from the precision of the motor’s encoder, power losses in the motor, etc. Nevertheless, we abstract from these and simply assume that a single (possibly weighted) tolerance value can be determined from all these factors, including tolerances from all other sheet logic devices. Moreover, since both sheet speed and displacement are indirectly calculated from motor speed, its variability affects sheet transport along the paper path. In what follows, we derive mathematical equations for a motor speed tolerance model assuming both constant and non-constant acceleration.

### 3.3.1.2 Modelling Tolerance in Motor Speed

Assuming constant acceleration, the equations of uniformly accelerated linear motion directly give the relationships between acceleration, speed and position, namely:

$$
\begin{align*}
    v &= v_i + a \Delta t \\
    s &= s_i + v_i \Delta t + \frac{1}{2} a (\Delta t)^2 \\
    s &= s_i + \frac{1}{2} (v_i + v) \Delta t
\end{align*}
$$

where:

- $a$ is the motor’s (constant) acceleration
- $v_i$ and $v$ are the motor’s initial and current velocities respectively
- $s_i$ and $s$ are the motor’s initial and current positions, i.e. displacement $d = s - s_i$
- $\Delta t$ is the time interval between the initial and current states.

Note that Equation (3.3) is derived from Equations (3.1) and (3.2).

Let the change in the speed, i.e. $v - v_i$, in a time interval of $\Delta t$ be affected by a tolerance of $\delta$. Then $(1 + \delta)$ corresponds to the tolerance factor described in the preceding section, expressed as fraction of a motor’s nominal speed. The actual current velocity, $v'$, is then given by:

$$
    v' = v_i + \Delta v' = v_i + (1 + \delta) \Delta v = v_i + (1 + \delta) a \Delta t
$$

where $\Delta v$ is the change in velocity during time interval $\Delta t$ assuming no tolerance, i.e. $\Delta v' = (1 + \delta) \Delta v$. 

This equation expresses the fact that rather than attaining a final speed of \( v \), the actual speed attained, \( v' \), is affected by a tolerance of \( \delta \). Notice that when \( \delta = 0 \), Equation (3.4) reverts to Equation (3.1).

Substituting \( v' \) in Equation (3.4) for \( v \) in Equation (3.3), we obtain the new current position, \( s' \), resulting from a speed tolerance of \( \delta \) as expressed in Equation (3.5):

\[
\begin{align*}
  s' &= s_i + \frac{1}{2}(v' + v_i)\Delta t \\
  &= s_i + v_i\Delta t + (1 + \delta)\frac{1}{2}a(\Delta t)^2
\end{align*}
\]  

(3.5)

If acceleration is not constant, as is the case in some of the SIL motor models, the above equations are not directly applicable. Assuming acceleration is given by:

\[
a = a_i + j\Delta t
\]  

(3.6)

where \( a_i \) is the initial acceleration, \( a \) is the current acceleration and \( j \) is a parameter which is constant in the time interval \( \Delta t \), i.e. jerk. We know that acceleration is the derivative of velocity, i.e.

\[
\frac{d}{dt}v = a.
\]  

(3.7)

Substituting Equation (3.6) into Equation (3.7) and integrating, we obtain the velocity resulting from non-uniform acceleration as follows:

\[
\int_{t_i}^{t_f} \frac{d}{dt}v = \int_{t_i}^{t_f} a
\]

\[
v = v_i + a_i\Delta t + \frac{1}{2}j(\Delta t)^2,
\]  

(3.8)

where \( \Delta t = t_f - t_i \).

Similarly, we can derive position from velocity as given by Equation (3.9) as follow:

\[
\int_{t_i}^{t_f} \frac{d}{dt}s = \int_{t_i}^{t_f} v
\]

\[
s = s_i + v_i(\Delta t) + \frac{1}{2}a_i(\Delta t)^2 + \frac{1}{6}j(\Delta t)^3.
\]  

(3.9)

Similar to the case of uniform acceleration, the new current speed when tolerance is taken into account is given by Equation (3.10) below:

\[
v' = v_i + (1 + \delta)\Delta v = v_i + (1 + \delta)\left(a_i\Delta t + \frac{1}{2}j(\Delta t)^2\right).
\]  

(3.10)

Using Equation (3.10) for the final velocity and integrating, we obtain the corresponding position \( s' \) as follows:

\[
\int_{t_i}^{t_f} \frac{d}{dt}s' = \int_{t_i}^{t_f} v' = \int_{t_i}^{t_f} v_i + (1 + \delta)\Delta v
\]

\[
s' = s_i + v_i(\Delta t) + (1 + \delta)\left(\frac{1}{2}a_i(\Delta t)^2 + \frac{1}{6}j(\Delta t)^3\right).
\]  

(3.11)

Having considered the general equations governing motor motion, we now apply them to the specific kinds of motors modelled in SIL.
3.3. Modelling Tolerance in the Simulated Plant

3.3.1.3 Speed Tolerance in FPGA_StepperMotor

FPGA_StepperMotor models the stepper motors used in real machines. Among others, a stepper motor has the advantage that its speed and position can be precisely controlled without any feedback mechanism. It rotates in discrete steps which are triggered by external electrical input pulses.

In the SIL model, non-uniform acceleration is assumed and is given by Equation (3.6) where \( j \) is a parameter known as jerk. It is used to smoothen out the acceleration profiles generated for stepper motors for better sheet transport. However, in calculating speed and position, the relations derived above (Equations (3.8) and (3.9) respectively) are not used. Instead, the following relations are used.

\[
a = a_i + j \cdot \text{throttle} \cdot \Delta t \tag{3.12}
\]

\[
v = v_i + a_i \cdot \text{throttle} \cdot \Delta t \tag{3.13}
\]

\[
s = s_i + v_i \cdot \text{throttle} \cdot \Delta t \tag{3.14}
\]

The additional “throttle” factor is a time scaling factor used to speed (up or down) the whole machine without altering the displacement. For instance, in a profile with constant velocity, the distance covered by the sheet in a given time interval remains the same but the velocity is scaled by a factor of throttle. This means that the sheets arrive at a given point, say a sensor, earlier (for \( \text{throttle} > 1 \)) or later (for \( \text{throttle} < 1 \)). Clearly, the relationships expressed in Equations (3.6), (3.8) and (3.9), and those in Equations (3.12) to (3.14) are not the same. Nevertheless, the method employed in SIL is a model of what takes place in the actual FPGA-controlled stepper motors.

We retain the current SIL model for the stepper motor and include the speed tolerance by modifying Equation (3.13) to obtain Equation (3.15) below:

\[
v' = v_i + (1 + \delta) \cdot a_i \cdot \text{throttle} \cdot \Delta t. \tag{3.15}
\]

Based on Equation (3.14), we observe that position does not depend on the current value of speed. Thus, the speed tolerance is propagated to position at next time instant.

3.3.1.4 Speed Tolerance in VarSpeedStepperMotor

VarSpeedStepperMotor may be an initial (simplified) implementation of the stepper motor. It is not clear if this motor is still being used. However, for the sake of completeness, we also model its tolerance.

Although, this motor and FPGA_StepperMotor both inherit from the same StepperMotor base class in SIL, their mechanical models are very different. From the implementation, it is a constant acceleration device whose speed and position are modelled using Equations (3.1) and (3.3) respectively. Hence, we model speed tolerance by replacing these with Equations (3.4) and (3.5). The existing position calculation requires no change, since it directly uses \( v' \) (from Equation (3.4)) in place of \( v \) in Equation (3.3).

3.3.1.5 Speed Tolerance in StubbedMotor

This motor models a constant speed motor. However, it defines a throttle parameter, say \( T \) and a throttleChangeRate, say \( \Delta T \). The throttle continues to increase linearly until a targetThrottle value is reached. The original model is as follows:

\[
T = \begin{cases} 
T_i + \Delta T \cdot \Delta t & \text{if } T \text{ equals targetThrottle} \\
T_i & \text{otherwise}
\end{cases} \tag{3.16}
\]

\[
s = s_i + \frac{1}{2}(T_i + T) \cdot \Delta t \cdot S, \tag{3.17}
\]

\[
T = \begin{cases} 
T_i + \Delta T \cdot \Delta t & \text{if } T \text{ equals targetThrottle} \\
T_i & \text{otherwise}
\end{cases} \tag{3.16}
\]

\[
s = s_i + \frac{1}{2}(T_i + T) \cdot \Delta t \cdot S, \tag{3.17}
\]
where $T_i$ is the initial throttle at the beginning of the time interval $\Delta t$ and $S$ is the constant speed.

The product $S \cdot T$ is the effective velocity (compare Equations (3.3) and (3.17)). Thus, whenever activated, this motor attempts to attain and subsequently remain at a target velocity at a constant rate determined by its throttleChangeRate. As a consequence of this model, we model speed tolerance as throttle tolerance. We therefore replace Equation (3.16) with Equation (3.18).

\[
T' = \begin{cases} 
T' & \text{if } T' \text{ equals } \text{targetThrottle} \\
T_i + (1 + \delta)\Delta T \cdot \Delta t & \text{otherwise} 
\end{cases} 
\]  

(3.18)

Just like VarSpeedStepperMotor, the position calculation requires no change since $T$ is automatically replaced by $T'$ in Equation (3.17).

### 3.3.1.6 Speed Tolerance in ControlledMotor

This motor models a closed-loop controlled motor. In reality, this kind of motor has an encoder in a feedback loop to allow for corrections in the motor’s moved position and thereby increase the accuracy. This kind of motor is commonly used in the company. However, in some projects, it has been replaced by the stepper motor.

Essentially, the motor speed and position are calculated based on an input pulse-width-modulated (pwm) signal. By comparing the attained speed with the expected speed, the encoder then performs iterative error correction typically by means of a PID control algorithm. To model speed tolerance we simply replace the attained speed, say $v$ with $v' = (1 + \delta)v$, i.e. like other motors, tolerance affects the actual speed attained. Since position is determined directly from speed, this tolerance factor is correctly propagated to the motor’s displacement (as in the stubbed and variable speed stepper motors).

### 3.3.1.7 Speed Tolerance in SteeredMotor

The steered motor was added to SIL during the course of this project. It is a multi-purpose motor which is not controlled by a PID controller. In other words, there is no feedback to the application (cf. controlled motors). The motor can be used for a stepper, pulse-width-modulated, frequency-controlled or simple on/off motor. The tolerance model is the same as that of the controlled motor; the different motor types it may represent only determine what parameters are used to calculate motor speed. The speed tolerance model therefore defines the effective speed by $v' = (1 + \delta)v$.

### 3.3.1.8 Speed Tolerance in HIL_StepperMotor

An additional motor, HIL_StepperMotor, exists in SIL. As the name implies, this ‘motor’ is associated with hardware-in-the-loop (HIL) simulation. It is a stub that interfaces real hardware to the simulator since in HIL simulation, actual motors/emulators are used rather than simulated ones. Therefore, we do not model its tolerance. The same observation applies to all other HIL devices. These are present because the SIL simulator has been adapted to also support HIL simulation.

### 3.3.1.9 Further Remarks

The models for the different motor kinds have assumed tolerance in motor speed. The derived equations only affect time intervals in which speed is changing due to acceleration or deceleration (i.e. ramping). Two major problems with these models are as follows:
First, our goal was to express tolerances of all devices as motor tolerance. However, some tolerance effects do not just affect ramping but also the target speed. For example, a sheet being transported at constant speed (e.g. using a stubbed motor) may arrive at a set point late (or early) because of actuator delays and/or sensor tolerances. In terms of motor speed, this may translate to a lowering (or raising) of the constant speed so that the displacement within a given time interval is less (or more) than the expected displacement in an ideal plant. Thus, we conclude that for all motor kinds the target speed must be affected by the tolerance factor (where applicable). Furthermore, we once again emphasize that this factor represents the cumulative tolerance of the whole system and not necessarily motor variability. This is particularly significant for stepper motors which are known to be so precise that speaking of variability in their speed is rather bizarre. In this case, their tolerance factor is understood to be arising from variability in other devices.

Secondly, the insufficiency of this abstraction becomes obvious from the fact that the assumption of being able to express tolerance of all devices in terms of motor speed is not realistic. Although not impossible, the conversion of say actuator delay or sensor tolerance to motor speed is far from trivial and will require parameters such as sheet speed, motor speed, distance from the target set point, etc. Such a model involves complicated calculations. Moreover, expressing variability from the user’s point of view could be very cumbersome. Previous experiences at the company with the happy flow model (see Section 6.5 of [10]), whose success was largely due to the absence of the highlighted drawbacks, serve as a strong motivation for an extended design in which tolerance in each device kind is considered separately.

A final observation is that an alternative abstraction could be to include speed tolerance in the interface between the motors and pinches, e.g. by modelling speed tolerance in the set(...) and get(...) methods of the base motor class. Although this is easier and reduces the inherent risk of inadvertently distorting the functional behaviour of the existing models due to errors introduced into the models themselves, it has the disadvantage that the speed and position graphs of the motors will still reflect the ideal motor behaviour. Unfortunately, this is unacceptable as these graphs are usually used by software developers during development to understand and/or verify the behaviour of the system. In addition, changes made to the software to improve robustness will not reflect in the graphs. Moreover, verification of the correctness of the model is largely based on comparing these graphs (see the test cases in Section 3.6).

Although more challenging and risky, we therefore retain the motor model as described in this section, but equally investigate tolerances in other devices as explained in the subsequent sections.

**Extended Design**

Given the insufficiency of expressing all system variability in terms of motor speed tolerance as explained in Section 3.3.1.9 above, we now develop tolerance models for other devices in the simulated machine. Despite the consequent extended and improved models, we retain the original idea for motor tolerance. This means that an initial step in robustness testing may merely involve expressing system tolerance as motor (speed) tolerance to generate upper bounds. For this purpose, tolerance ranges can be based on educated guesses without the explicit conversion of tolerance in other device kinds into motor tolerance. In the next step, these tolerance values may be adjusted to reflect only motor speed variability (if any) and the models for other devices used to model their own variability. As already stated, tolerance specification for stepper motors may have no practical application in the latter case.

**3.3.2 Tolerance in Pinches**

Pinches (also known as pinch rollers) are free spinning wheels coupled to motors. They translate the rotary motion of the motor into linear motion of the sheet by maintaining direct contact with the latter.
SIL models the typical situation in the printers where a motor may be connected to one or more pinches via a transmission consisting of gears and/or pulleys. Furthermore, a motor may be disabled in which case it is decoupled from the connected pinches and does not drive them.

Assuming uniform circular motion\(^2\), the linear displacement, \(s\), and velocity, \(v\), of a rotating pinch are related to its radius, \(r\), as expressed in Equations (3.19) and (3.20) respectively.

\[
\begin{align*}
\text{(3.19)} \\
\quad s &= \Theta r \\
\text{(3.20)} \\
\quad v &= \omega r,
\end{align*}
\]

where \(\Theta\) is the pinch’s angular displacement in radians and \(\omega\) is its angular velocity in radians per second.

Assuming no slip, the velocity of the pinch and motor are equal, i.e.

\[
\begin{align*}
\text{(3.21)} \\
\quad v_p &= v_m \\
\omega_p r_p &= \frac{\omega_m r_m}{k}.
\end{align*}
\]

where subscripts \(p\) and \(m\) stand for pinch and motor respectively, and \(k = \frac{r_p}{r_m}\) is the transmission ratio between the motor and pinch\(^3\).

The diameter of a pinch may vary due to batch production and/or wearing due to friction as sheets rub over the pinch surface. Furthermore, there may be tolerance resulting from varying forces applied to the pinches or from the pinch-to-motor coupling. All these factors affect the pinch’s speed and invariably, the speed of the sheet it drives. We therefore model tolerance in pinch speed.

Suppose we define a single parameter, \(\delta\), to represent the tolerance in the pinch. The actual speed attained by the pinch, \(v_p\), is then given by:

\[
\begin{align*}
\text{(3.22)} \\
\quad v_p' &= v_p(1 + \delta) = \omega_p r_p(1 + \delta).
\end{align*}
\]

Like motor speed tolerance, we express the pinch tolerance as a fraction of the nominal pinch speed while abstracting from the actual sources of the tolerance. It is bounded by \([\min\text{Tolerance}, \max\text{Tolerance}]\) from which a value is chosen and multiplied by the speed received from the driving motor (and transmission ratio) in order to obtain the effective pinch speed. A pinch with no tolerance has a single-point range of \([1.0, 1.0]\).

We remark that this tolerance factor also covers variation in the transmission ratio since \(k = \frac{r_p}{r_m}\) is directly affected by variation in pinch diameter, \(d_p = 2r_p\). The same tolerance factor is also directly applicable to pinch displacement given the relationship expressed in Equation (3.24). Although sheet speed and displacement are currently calculated from pinch speed rather than pinch displacement (see Section 3.3.1.1), we include the tolerance model for pinch displacement for the sake of completeness. This ensures that the tolerance model remains valid in the event of a future change to displacement-based sheet updates.

\[
\begin{align*}
\text{(3.23)} \\
\quad s_p' &= v_p' \Delta t \\
\quad &= v_p(1 + \delta) \Delta t \\
\text{(3.24)} \\
\quad &= s_p(1 + \delta) \\
\quad &= \Theta r (1 + \delta)
\end{align*}
\]

\(^2\)It is implicitly assumed that the update frequency of the model is high enough, such that this assumption is valid.

\(^3\)The current practice in SIL where pinch speed is determined by multiplying motor speed with a transmission factor, i.e. \(v_p = \frac{v_m}{k}\), deviates from Equation (3.21). This ambiguity arises from propagating a motor’s linear velocity instead of its angular velocity as is more conventional.
We do not model pinch slip separately; it is included in the single tolerance factor for pinches. Moreover, pinch slip can be solved by replacing the faulty pinches rather than attempting to capture the dynamic, unpredictable and rather complex phenomenon of slip in a tolerance model.

Finally, we remark that pinch speed tolerance could arguably be expressed as part of the motor speed tolerance factor. However, this approach is hindered by the fact that a motor may be coupled to multiple pinches each having their own transmission factor, as earlier stated. A single factor expressing motor and pinch tolerance combined will make it impossible to express different tolerance values for individual pinches. Moreover, the graphs for motor speed will again be misleading since they will include pinch tolerance effects. These observations justify our decision to have a separate tolerance factor for pinches with the inherent advantage of separation of concerns.

3.3.3 Tolerance in Actuators

An actuator is a device that enables a system realize a reaction to (i.e. control) its environment. In other words, actuators create changes in a system (cf. sensors which detect changes). This change typically involves some kind of motion.

In SIL, several kinds of actuators exist which model corresponding devices in real printer paper paths. Among others, these include: active (segment) switches which are used to direct sheets between alternate output paths at the end of a segment; pinch lifts which are used to activate (or deactivate) pinches by raising (or placing) them from/on the sheet or clutches which realize the same functionality by other means; motor enablers for activating/deactivating motors; and sheet placement actuators used to model the injection of sheets done in the PIM of real machines.

Regardless of the functionality realized by these actuator, they share a common property, namely: their transitions from activated to deactivated states and vice versa are characterized by a finite delay. This delay may be due to the electromechanical properties of the actuator hardware, e.g. charging and discharging time of a capacitor in a solenoid and/or the time interval between when a command is sent to toggle the actuator to when it is actually received and processed by the actuator-control software. Whereas the former is typically provided in the datasheet of the device as worst- and/or best-case response-times, the latter must be calculated. Furthermore, the delay is not necessarily deterministic. In other words, the (de-)activation of an actuator will not always take the same time. This behaviour represents a tolerance that should be modelled. Such a tolerance parameter can then be used to express the cumulative effect of all factors that affect the response time of an actuator.

We define a delay tolerance range bounded by $[\text{minDelay}, \text{maxDelay}]$ from which the actual delay (in seconds) for each actuator instance is chosen. In reality, however, activation and deactivation delays may not be equal. An example is a solenoid where charging a capacitor may take more time than discharging it. In some cases, this asymmetry may be exploited to simplify control software algorithms. Hence, we define alternative bounds $[\text{minActivationDelay}, \text{maxActivationDelay}]$ and $[\text{minDeactivationDelay}, \text{maxDeactivationDelay}]$ to specify separate delays for activation and deactivation respectively. Depending on the actuator design, activation may correspond to opening or closing the actuator and vice versa.

In relation to the existing SIL implementation, we assume that activation delay affects the transition from idle to active states whereas deactivation delay affects the transition from active to idle states. Furthermore, we impose the constraint that activation- and deactivation-delay ranges must be specified together, but not in combination with the generic delay range. Thus, when the asymmetry in delay ranges is not of interest, the duplicity of two redundant parameter is avoided by simply using $[\text{minDelay}, \text{maxDelay}]$. Moreover, a device with no delay effectivly has a single-point range of $[0.0, 0.0]$.

SIL currently supports the specification of non-zero delays for clutches and pinch lifts. The implementation involves the use of the delayed callback functionality provided by the clock in SIL. Since the
functionality is defined in the actuator base class, extending it to other actuator kinds simply involves defining the necessary parameters for specifying their delay tolerance. Furthermore, since the existing implementation assumes the same delay for activation and deactivation, this must be updated to support asymmetric delays.

Lastly, we observe that going by the definition of actuators presented at the beginning of this section, motors and pinches are themselves actuators. However, given the variety (and relative complexity) of motors, they are considered separately in SIL. We simply adhere to this convention, without any loss of generality.

### 3.3.4 Tolerance in Sensors

Unlike actuators, sensors enable a system to control its environment by detecting a change in the environment and informing the system typically by a transition from off to on states or vice versa. Although different kinds exist, the primary purpose of sensors along the paper path is to detect the presence of a sheet (edge). The differences between these sensors generally relate to their principles of operation/properties which are beyond the scope of this work. The subsequently discussed tolerance model is only applicable to paper path sensors. For example, sensors associated with other devices such as encoders in controlled motors do not fall in this category; hence, their tolerance should be included in the cumulative motor tolerance.

Being on/off devices, all paper path sensors in SIL have a state attribute. An overruled mode is defined which, when enabled, prevents the sensor from toggling states. This mechanism was introduced to enable the simulation of simple error scenarios using faulty sensors. When a sensor is not overruled, it becomes active upon detecting the pulling edge of a sheet and goes back inactive after detecting the pulled edge.

When a sensor is placed on a segment, it becomes a “sensor instance” according to SIL terminology with an additional attribute which defines its position on the segment relative to the segment entrance, i.e. position 0.0. Nevertheless, in this document, the distinction is generally overlooked and a sensor instance is simply referred to as a sensor unless otherwise stated.

Tolerance in sensors may arise from several sources some of which are depicted in Figure 3.3 and briefly explained below. The interested reader is directed to [26] for more details on principles of sensor operation and some more sources of variability.

**Region of Interest (RoI):** This refers to the viewing angle of a sensor, i.e. how far from its nominal position that sheet detection is still possible. Sheet detection may take place at any location within this range typically in a non-deterministic manner.

**Depth of Focus (DoF):** The highest distance of a sensor from the segment surface for which it is able to detect sheets. Both RoI and DoF are affected by the sheet type, e.g. matte versus glossy sheets.
3.3. Modelling Tolerance in the Simulated Plant Design

**Latency:** This refers to the time between detection of and reporting the presence of a sheet. If the latency is high, the sheet will not be at the position reported by the time the control software is informed of the detection. Like actuator delay, this latency is bounded but variable within the defined bounds.

**Resolution:** This refers to the smallest change in sheet position which a sensor can detect. This affects the sampling frequency of the sensor by the control software and may vary from one sensor to another even from the same batch.

**Hysteresis:** A sensor may exhibit different behaviour depending on the direction from which the sheets arrive as a result of hysteresis [26].

**Mounting Tolerance:** When assembling a printer, sensors may be displaced relative to the specified position in the paper path layout.

All sensor tolerance factors affect sheets by changing the position at which they are detected. We can therefore model this in SIL by changing the position of the sensors on the paper path. Thus, early and late detection of a sheet are modelled by displacing a sensor relative to its nominal position on the paper path. The main motivation for this approach is the assumption that, given the sheet speed, each sensor tolerance factor can be converted to an equivalent sensor displacement. More importantly, this approach provides a simple yet useful means to capture several sensor tolerance effects despite the ‘limitations’ of SIL’s one-dimensional paper path model. For instance, it is not possible to directly express depth of focus since sheet transport is currently modelled as a one-dimensional motion of sheet entities defined by a length, a leading and trailing edge (both of which are points).

Contrary to other devices where we express tolerance as a fraction of a nominal value, sensor tolerance is expressed relative to the nominal sensor position and bounded by \([\text{minTolerance}, \text{maxTolerance}]\) for which the following relation holds:

\[
\text{minTolerance} \leq 0 \leq \text{maxTolerance}.
\]

Notice that \(\text{minTolerance}\) is negative since it is a displacement. The actual position of a sensor is determined by adding the tolerance value to the sensor’s nominal position (as opposed to multiplying in other devices). Typical sensor tolerance ranges are in micrometers. However, for uniformity, we adhere to the apparent convention in SIL of specifying distances in meters.

We model the typical situation where each sheet may be sensed at a different position by allowing sensor tolerance to change per sheet. In order to realize this functionality, the new tolerance value of a sensor must be set before the next sheet arrives because in SIL, the position of the sensor on the segment determines at what time it covers the sheet. This can be done for the next sheet when the sensor is turned off, since switching a sensor off signals that the sensor is no longer under the current sheet.

In reality, however, the pulling and pulled edges\(^4\) of the same sheet may be sensed at different positions. This can be modelled by changing the position of the sensor again immediately after it detects the pulling edge of the current sheet thereby setting the position at which the pulled edge of the same sheet passes under the sensor. Essentially, whereas the pulling edge tolerance of the next sheet is set when the current sheet leaves a sensor, its pulled edge tolerance is set as soon as the pulling edge is detected.

However, allowing a sensor’s position to change in the manner described above could potentially lead to late detection of sheets and/or multiple detection of the same sheet as described next.

\(^4\)While a sheet’s leading and trailing edges are fixed, the pulling and pulled edges may be the sheet’s leading or trailing edge depending on the direction of motion.
Late Sheet Detection

Late sheet detection is depicted in Figure 3.4(a). The nominal position of the sensor (with no tolerance) and its tolerance range are indicated in gray. At time $t$, sheet 1 is under the sensor. At time $t + 1$, both sheets have moved a distance $d$ in the direction indicated by the arrow. The displacement of sheet 1 causes it to fall outside the range of the sensor. Consequently, a new sensor position for (the pulling...
edge tolerance of) sheet 2 is determined randomly from the sensor’s tolerance range. However, since the new sensor position falls beyond displacement interval, $d$, of sheet 2, it misses the sensor. In other words, the sensor is “lost” relative to sheet 2. This problem arises if the sensor’s region of interest is wider than the minimum inter-sheet distance. In reality, this situation should not occur, i.e. a sensor’s range should not cover more than one sheet at any given time. However, modelling this scenario may have practical benefits. For example, it could be an easy way for embedded software developers to detect an unwanted situation of a single sensor over multiple sheets in SIL. It may also be used to simulate expected tolerance behaviour in a situation where inter-sheet distance is barely larger than sensor range, and the sensor actually toggles but detects the pulling edge of sheet 2 late. Thus, this late sensor must be detected and added to the sheet’s list of covered sensors. Yet, care must be taken that a late sensor is not reported when the new sensor position for the next sheet falls on the current sheet. Otherwise, this causes multiple falling-edge detections for the same sheet, which is undesirable. Given the fact that every sheet has a unique identifier, it is easy to distinguish cases when late sensors must be reported using the sheet ID.

### Multiple Sheet Detection

Figure 3.4(b) shows another possible problem that arises if a sensor’s position is toggled for the pulled edge and the sensor’s tolerance range exceeds the sheet’s length. At time $t+1$, the pulling edge of sheet 1 is moved by a distance $d_1$ such that it is no longer under the range of the sensor given the sensor’s position at time $t$. A new position for the pulling edge is generated which falls in front of sheet 1, because of the wide sensor tolerance range. Hence, at time $t+2$, sheet 1 encounters the same sensor again. In this case, triggering of the sensor should be prevented. One way to solve this problem is for each sheet to keep a history of sensors previously encountered (at least the last one).

Even if a sensor’s tolerance range is less than the sheet length, multiple detection is still possible, as shown in Figure 3.4(c). It arises from toggling a sensor’s position to set a sheet’s pulled edge tolerance. At time $t+1$, sheet 1 moves under the sensor 5. After moving the sheet by a distance of $d_1$, a new sensor position is immediately set for the pulled edge. However, because this position lies in the path of the pulling edge, multiple detection of the sensor occurs at time $t+2$. This problem can be solved by only adding sensor instances which do not already exist in the sheet’s list of spanned sensors.

Nonetheless, a careful analysis of the multiple detection scenarios highlighted above reveals that they both can be avoided by setting pulled edge sensor tolerance without toggling the sensor position. In other words, we still set the pulled-edge tolerance at the time the pulling edge gets covered by a sensor, but we do so without changing the sensor’s position. The tolerance is instead added to the position calculation which determines the time at which the pulled edge leaves the sensor. This solution is simpler, more elegant and preferable (prevention is always better than cure). Therefore, it is used for setting pulled-edge sensor tolerances. There is no walk around to late sensor detection. That notwithstanding, the proposed solution using the existing unique sheet IDs is equally straightforward, easy to implement and will be undertaken.

### 3.3.5 Tolerance in Segments

Segments in SIL model the metal plates that enclose sheets as they travel through the paper path. Each segment has a length and direction. The start of a segment (i.e. position 0.0) is denoted as its “entrance” while the end (i.e. position = length) is marked as its “exit”. Using this notion, different paper path topologies can be described by specifying the interconnections between segments at their entrances and

---

5Although the sensor was previously at its nominal position at time $t$ in the scenario depicted in the figure, this condition is not a prerequisite for the problem to occur.
exit. An intersection is any point where more than two segments meet. Segments may contain pinches, sensor and/or other devices placed at specific positions along their length.

3.3.5.1 Segment Length Tolerance

Due to factors like production, assembly and climate, the lengths of paper path segments may vary. The total variation in the length of paper path is therefore the sum of the variation in each segment and affects the distance sheets have to travel along the paper path. We model this effect by specifying a length tolerance range \([\text{minTolerance}, \text{maxTolerance}]\) expressed as a fraction of the nominal segment length. The actual segment length is then obtained by multiplying the nominal length by a tolerance value chosen from this range. A segment with no length tolerance effectively has a range of [1.0, 1.0].

Referring to our earlier classification of tolerance as dynamic or static in Section 2.1.1, we observe that the variation in segment length is a static effect. Thus, the actual tolerance value of each segment should be specified once and subsequently remain fixed at this value during the course of any simulation run. Furthermore, this should be done before sheet transport begins, i.e. segment lengths should be initialized before any sheet starts moving along any segment of the paper path and should not be changed at run-time. These pre-conditions must be enforced by the SIL user.

3.3.5.2 Sheet behaviour in Bent Segments

The separation between the plates that constitute segments exceeds the sheet thickness. This allows for easy transportation of sheets. On the other hand, it may result in identical sheets following different paths. This phenomenon, which is particularly interesting at curves/bends in the paper path, is not currently modelled in SIL. In such regions, a sheet may take the inner bend, outer bend or an arbitrary path in-between, depending on factors such as bend length, sheet velocity, size and thickness. The path taken by a sheet may affect sheet transport because an inner or outer bend implies a varying displacement thereby affecting how fast the sheet passes through the bend. In some portions of the paper path such variation is acceptable and has no significant impact on sheet transport. However, in critical segments, such as the region surround the print head where images are fused onto sheets, high precision in sheet position is required and tolerance in sheet paths needs to be taken into account.

This effect may be viewed as a property of sheets or segments, i.e. variation in sheet or segment length respectively. However, we model it as a segment property. The motivation for this choice is twofold: First, it fits better with the existing implementation where segments are responsible for moving sheets lying on them. Secondly, it allows for reusing related concept of buffer areas defined in the concurrent extension of SIL for continuous feed printers.

Buffer areas are used to describe those segments in the paper path where buffering of paper occurs between consecutive pinches due to the conflicting forces they exert on the paper. Although defined to model blousing and bulging effects particularly in the context of wide-format printing, where they are used to control interactions between media input/output and the image creation function, a bend can be viewed as (part of) a buffer area in which sheets can take one of multiple paths (corresponding to different lengths of the buffer area). Buffer areas are characterized by a minimum (\(\text{minLength}\)) and maximum (\(\text{maxLength}\)) length. For bends, these correspond to the inner and outer paths respectively. These parameters default to the nominal length of the buffer area if unspecified.

Using the segment length parameters, we can model sheet behaviour in bends as follows: Upon entering a bend where tolerance is enabled, the sheet and segment dynamically ‘agree’ on what trajectory the sheet will take in the bend by selecting a length from the range \([\text{minLength}, \text{maxLength}]\). As far as the
3.3. Modelling Tolerance in the Simulated Plant

sheet is concerned, this is the (fixed) length of the bend which it must traverse\(^6\).

Another issue concerns how to describe these bends in segments. Generally, segments are delimited by intersection points in the paper path with no consideration about whether the segment is a straight piece of the paper path or whether it contains curves. To explicitly model the behaviour of sheets at curves, we specify bends as distinct segments which we refer to as “bent segments”.

**Decision:** To specify curves/bends in the paper path where sheet behaviour needs to be modelled as distinct segments.

The rationale for this decision is that segments are an existing mechanism in SIL to denote a portion of the paper path. Therefore, creating a new mechanism to specify curved segments is unnecessary and redundant. This decision is further motivated by the fact that it matches the concept of buffer areas which are required to be separate segments.

It is also necessary to distinguish regular buffer areas from bent segments. Normal buffer areas have dedicated sensors for modelling their behaviour. **Therefore, a bent segment is a buffer area with \([\text{minLength}, \text{maxLength}]\) defined but having no buffer area sensors.** This can be easily determined using the provided method `hasBufferArea()` . This method returns `true` if the segment instance it is invoked upon has a buffer area; otherwise it returns `false`. A straight segment or unspecified bent segment will have a single point bend tolerance range equal to the nominal sheet length, i.e. \([\text{length}, \text{length}]\).

### 3.3.6 Tolerance in Sheets

A sheet is the entity which is moved through a paper path and whose behaviour SIL provides a simulation framework to model. So far, we have identified and modelled tolerance in different devices which directly or indirectly affects sheet transport. We now consider a sheet parameter which may also vary, namely its length. Next, we address the requirement of enabling finite sheet delay.

#### 3.3.6.1 Sheet Length Tolerance

The length of a sheet exhibits some variability resulting from cutting. [11] gives the acceptable tolerances specified in the ISO 216:1975\(^7\) standard for common sheet sizes. We express this tolerance as a fraction of the nominal sheet length, bounded by the range \([\text{minTolerance}, \text{maxTolerance}]\). The effective sheet length is then the nominal sheet length multiplied by a value from this range. As in other cases, a sheet with no tolerance has a single-point range of \([1.0, 1.0]\).

Sheet length variability is a static tolerance factor. Therefore, it must be specified once for each sheet at injection into the paper path. Since sheet injection is dictated by the embedded software and is not done during SIL initialization, we specify the tolerance range in the sheet manager (which is created during SIL initialization). Upon creation of a new sheet, a tolerance value from this range is determined based on the user’s specification, including a default value of 1.0 for ideal sheets. In addition, when the length of a sheet (to be) injected is changed at run-time, a feature currently supported by SIL, it may be desirable to also update the tolerance range to match the new sheet length. To enable this functionality, we extend the existing interface method for changing the length of injected sheets at run-time by adding two new (optional) parameters, `minTolerance` and `maxTolerance` for setting the tolerance range of subsequent sheets. The new interface is as follows:

---

\(^6\)Note that since the buffer area length is specified in absolute terms (in meters), the tolerance is implicit, i.e. the chosen length equals the nominal length already adjusted by a tolerance factor.

\(^7\)http://www.iso.org/iso/catalogue_detail.htm?csnumber=4087
3.3.6.2 Delaying Sheets

Requirement FR_10 in Table 2.2 required an interface to delay sheets. It was motivated by the need to test an error routine in the finisher module for one of the new printers under development at the time of this project. The routine was developed to report an error whenever inter-sheet distance falls below a minimum threshold. This makes it impossible for the flipping wheel to flip the sheet and results in a request for operator intervention (ORE). Since inter-sheet distance is enforced earlier in the finisher paper path, the possibility of this error occurring is minimal. However, in the rare case when it occurs, robust control software must be able to correctly report it. Performing this test on the real machine is very difficult because a scenario leading to this situation is extremely hard to create.

Although this requirement does not directly concern tolerance specification it is related in the sense that if the described error ever occurs, chances are that it will be caused by unexpected behaviour resulting from tolerance in one or more devices.

An interface method already exists for stopping sheets at target locations in the paper path. We extend it with an optional duration parameter which, when specified, causes the sheet to be stopped for the stipulated duration (in seconds). Otherwise, the sheet is stopped indefinitely as before. The new interface is then:

```
stopSheet( int segmentId, double position, double duration = -1.0 ).
```

Similar to delay in actuators, sheet delay will be realized using the callback functionality of the clock in SIL.

3.3.6.3 Sheets as Part of the Simulated Plant

Although classified as one of the sheet logic devices in SIL, sheets do not belong to this group in a strict sense. Instead, they are the entities whose behaviour in the paper path is simulated in SIL using the (other) sheet logic devices. For this reason, no sheet specifications exist in the layout file. Rather, the sheet length is included when specifying the sheet injector for a particular test case within the test case specification file. Therefore, for sheets alone, the decision in Section 3.2.2 to specify tolerance ranges in the layout file does not apply. Instead, sheet tolerance ranges are specified in the test case file along with sheet length tolerance specifications.
3.4 Provided XML Interface for Tolerance Specification

In this section, we present the XML interface for specifying tolerance ranges in the layout file and tolerance values in the test case file.

3.4.1 Specifying Tolerance Ranges

Table 3.1 gives an overview of the different kinds of tolerance modelled based on the discussion in Section 3.3.

<table>
<thead>
<tr>
<th>Sheet Logic Device (Id)</th>
<th>Characteristics of Interest (Expressed as)</th>
<th>Parameters/Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bent segments (0)</td>
<td>Length of the bend (absolute length in meters)</td>
<td>$0.0 &lt; \text{minLength} \leq \text{length} \leq \text{maxLength}$</td>
</tr>
<tr>
<td>Segments (1)</td>
<td>Length of segment (a fraction of nominal segment length)</td>
<td>$0.0 &lt; \text{minTolerance} \leq 1.0 \leq \text{maxTolerance}$</td>
</tr>
<tr>
<td>Pinches (2)</td>
<td>Speed (a fraction of nominal pinch speed)</td>
<td>$0.0 &lt; \text{minTolerance} \leq 1.0 \leq \text{maxTolerance}$</td>
</tr>
<tr>
<td>Motors (3)</td>
<td>Speed (a fraction of nominal motor speed)</td>
<td>$0.0 &lt; \text{minTolerance} \leq 1.0 \leq \text{maxTolerance}$</td>
</tr>
<tr>
<td>Actuators (4)</td>
<td>Response time (absolute delay in seconds)</td>
<td>$0.0 \leq \text{minDelay} \leq \text{maxDelay}^a$</td>
</tr>
<tr>
<td>Sensors (5)</td>
<td>Position at which sheet is sensed (displacement in meters relative to nominal sensor position)</td>
<td>$\text{minTolerance} \leq 0.0 \leq \text{maxTolerance}$</td>
</tr>
<tr>
<td>Sheets (6)</td>
<td>Sheet length (a fraction of nominal sheet length)</td>
<td>$0.0 &lt; \text{minTolerance} \leq 1.0 \leq \text{maxTolerance}$</td>
</tr>
</tbody>
</table>

Table 3.1: Kinds of tolerance modelled

^aAlternatively, [\text{minActivationDelay}, \text{maxActivationDelay}] and [\text{minDeactivationDelay}, \text{maxDeactivationDelay}] may be used when activation and deactivation delays vary.

An example of how tolerance ranges can be specified in the machine description file is presented in Appendix A.1.

3.4.2 Specifying Tolerance Values for Test Cases

Elements `<injectTolerance>` and `<disableTolerance>` contain specifications for tolerance injection and disabling in all device kinds identified in Table 3.1. Specifications for each device are expressed as grandchildren of these nodes, each contained in a parent node that identifies the device kind. Next, we define attributes for injecting and disabling tolerance as well as for test case reproduction. An example showing how this interface can be used is presented in Appendix A.2.

3.4.2.1 Specifying a Seed for Test Case Reproduction

To meet the requirement of reproducibility of results, we store the seed used to initialize the random number generator. As already observed, this value alone is sufficient to reproduce any previous test under the current assumption of a single, global seed (see Section 3.2.4) and unchanged specification
expect for the included seed. The seed is specified as an attribute of nodes <injectTolerance> or <disableTolerance>. In case a seed is specified in both nodes, the first seed encountered is used. Unless reproducing a previous test, no seed is required and a random non-negative integer seed is generated during initialization. In all cases, the seed is stored in SIL logging for future reference.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Required</th>
<th>Type/ Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>Optional</td>
<td>Integer</td>
<td>Specifies the seed to be used to initialize the random number generator. When specified, it indicates test case reproduction.</td>
</tr>
</tbody>
</table>

Table 3.2: Attribute for seed specification

3.4.2.2 Injecting Tolerance

Table 3.3 summarizes the attributes used for tolerance injection via the test case specification file. In addition to the parameters common to all device kinds, namely target, name/id, type and value, attributes specific to a particular device kind are also defined. The attributes segmentId and nominalPosition for sensors as motivated by the fact that as observed in Section 3.3.4, a single sensor may have multiple instances. Therefore, it should be possible to specify tolerance for a particular instance uniquely identified by its position on a given segment. Similarly, the attribute transition allows for specification of variable activation and deactivation delay of actuators as discussed in Section 3.3.3.

The conditions in the “Required” column of Table 3.3 explicitly state when a given parameter must be specified. These conditions may be used to determine validation rules, e.g. in a corresponding XML schema and/or in defining error situations, e.g. failure to specify a required attribute should cause SIL to terminate with an error message (see Section 3.4.2.6, for example).

We remark that we provide no interface for changing tolerance ranges. The rationale for this decision is the fact that device tolerance ranges are machine characteristics. Changing them means defining a different machine. Hence, such changes should be limited to the machine description file. This choice also removes the ambiguity which may result from overshooting the tolerance range from a test case specification (see disadvantage of option IV in Section 3.2.2).

3.4.2.3 Disabling Tolerance

As already explained in Section 3.2.5, tolerances will be disabled by default in the extended SIL framework. However, <disableTolerance> is defined to provide an easy means of specifying “all-except” special cases where tolerance is injected in all devices of a particular kind except a few as illustrated in the next section.

For disabling tolerance, only parameter name/id as specified in Table 3.3 is required. In addition, transition may be used for actuators to disable tolerance in only one transition. For instance, to simulate a solenoid which is activated electromagnetically but deactivated via spring action for which the comparative delay of the latter is negligible, the specification in Listing 3.1 may be used.

Listing 3.1: Using optional “transition” attribute in actuator tolerance specifications

```
1 <disableTolerance>
2  <actuators>
3   <actuator name="%actuator_name%" transition="DEACTIVATION" />
4  </actuators>
5 </disableTolerance>
```
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Required</th>
<th>Type/ Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>target</strong></td>
<td>Unless target=&quot;ONE&quot; and the element node is not “sheets”</td>
<td>(ALL, ONE)</td>
<td>Determines the scope of specification, i.e. for a single device or for all devices of a particular kind. Defaults to ONE and can be omitted if a name/id is specified. However, for sheets, it is fixed at ALL and can be omitted.</td>
</tr>
</tbody>
</table>
| **name or id** | If and only if target is not specified or target="ONE" and the element node is not “sheets” | name: String id: Integer | The name of the device of interest. For segments an id is used instead. This attribute is not applicable to sheets.  
PRE-CONDITION: name/id must be valid and must exist in machine description file. |
| **type** | Unless type="INPUT" | (MAX, MIN, AVERAGE, INPUT, RANDOM, PER_SHEET, PER_EDGE) | MIN, MAX and AVERAGE are directly determined from the tolerance range where AVERAGE=(MIN+MAX)/2. If specified as INPUT, a value must be provided. Attribute type defaults to INPUT and can be omitted if a value is specified. RANDOM is used to generate a random value from the device’s tolerance range. PER_SHEET is allowed for sensors, sheets and bent segments only in which case a new random value is generated for the tolerance parameter for every new sheet. PER_EDGE allows sensor tolerance to vary for the leading and trailing edges of the same sheet and may NOT be used for any other kind of device. |
| **value** | If and only if type is not specified or type="INPUT" | Double | Specifies the tolerance value to be assigned when type is INPUT (or is unspecified) |
| **segmentId** | If and only if nominalPosition is specified and specification is for a single sensor instance | Integer | Used to identify an instance of a sensor on a segment with ID segmentId. Must be specified along with nominalPosition. Together, they allow for tolerance specification in each instance of a sensor (since one sensor can have multiple instances at different places) |
| **nominal Position** | If and only if segmentId is specified and specification is for a single sensor instance | Double | The nominal position of a sensor on a segment. Must be specified along with segmentId; both attributes are only applicable to sensors |
| **transition** | Optional attribute for actuators only | (ACTIVATION, DEACTIVATION, BOTH) | Specifies different delays for activation (i.e. transition from idle to active) and deactivation (active to idle) of actuators. Defaults to BOTH in which case the same delay is assumed for both transitions if a single range is specified. Otherwise, activation and deactivation delays are chosen from their respective ranges. |

Table 3.3: Attributes for tolerance injection via XML specification

This specification effectively turns off delay in the transition from active to idle states (assuming that an earlier injection specification enabled tolerance in both transitions, e.g. using a target="ALL" construct).

When tolerance specifications are disabled for a device, it behaves like an ideal device with respect to the characteristic of interest and its tolerance range is set accordingly as a single-point range. However, we remark that once disabled, tolerances can only be re-enabled at run-time using the provided interface defined in Section 3.5.
3.4.2.4 Specifying Sheet Length Tolerance

As already mentioned in Section 3.3.6.3, sheets are a special part of the simulated plant. First, their tolerance ranges are specified in the test case file. Similar to other devices, when no tolerance range is specified for sheets, they take their nominal value (with a single-point range of [1.0, 1.0]). However, sheet tolerance specification without a tolerance range is meaningless. Secondly, since sheet tolerance is always for all sheets, the attribute target is redundant (but allowed) and fixed at “ALL”; name/id is not allowed; and PER_SHEET is an acceptable value for the type attribute to specify a different random tolerance for every new sheet. Thirdly, an XML interface for disabling sheet length tolerance is not specified since tolerance is disabled by default. Hence, the only valid specification for sheets is as illustrated in Listing 3.2.

Listing 3.2: Specifying sheet length tolerance

```xml
<injectTolerance>
  <sheets target="ALL" type="{MIN, MAX, AVERAGE, INPUT, RANDOM, PER_SHEET}"
            value="<double>" minTolerance="<double>" maxTolerance="<double>" />
</injectTolerance>
```

Lastly, we remark that unlike other devices, sheet tolerance specification nodes have no children.

3.4.2.5 Special Usage

Specifications are read and executed sequentially. This gives the user the freedom to define special “all-except” cases. For example, in order to inject the same tolerance in all but one motor, the specification in Listing 3.3 may be used. The specification in line 4 causes tolerance in all motors (including MOTOR_A) to be set to their maximum value. However, the subsequent specification in line 5 overwrites this for MOTOR_A. Therefore, combined, we inject a tolerance of 99% in MOTOR_A and maximum tolerance in all other motors.

Listing 3.3: Injecting tolerance in all but one motor

```xml
<injectTolerance>
  ...  
  <motors>
    <motor target="ALL" type="MAX" />
    <motor name="MOTOR_A" value="0.99" />
  </motors>
</injectTolerance>
```

Similarly, the order of injection and disabling can be used for easier specification of tolerance. Listing 3.4 shows how to inject tolerance in all pinches except PINCH_B where it is disabled. The ‘normal’ specification will require injecting tolerance individually in all pinches except PINCH_B (which is disabled by default). Note that reversing the order of the disable- and injectTolerance nodes will result in injecting tolerance in all pinches since, in that case, the specification for disabling tolerance in PINCH_B will be overwritten.
3.4. Provided XML Interface for Tolerance Specification

Listing 3.4: Injecting tolerance in all but one pinch

```xml
<injectTolerance>
  ...
  <pinches>
    <pinch target="ALL" type="RANDOM" /> 
  </pinches>
</injectTolerance>

<disableTolerance>
  ...
  <pinches>
    <pinch name="PINCH_B" /> 
  </pinches>
</disableTolerance>
```

3.4.2.6 Enforcing Correct Tolerance Specification

Given the XML specifications in Table 3.3, an important question to be address is: To what extent and in what manner should adherence to these specifications be enforced?

Option I: To enforce specifications strictly, i.e. failure to meet conditions in the “Required” column of Table 3.3 will result in an error and SIL will terminate.

Advantage(s)
1. Results in well-structured documents.
2. Verification can be done using XSDs and free/online verification tools.

Disadvantage(s)
1. SIL users may desire some degree of freedom, e.g. ignoring superfluous attributes with a warning might be desirable over strict enforcement.

Option II: To enforce required fields/combinations but ignore superfluous attributes with warning messages logged.

Advantage(s)
1. May be more attractive to users.

Disadvantage(s)
1. Deviates from the idea of well-structured XML files and the use of XSDs for verification.
Decision: In implementation, to enforce required fields/combinations but ignore superfluous attributes and log warning messages using SIL logging functionality.

This decision is motivated by the fact that SIL users should have some freedom in specifying tolerances and SIL should not be forced to terminate except when a required value is missing. Otherwise, it may be very annoying, for instance, to discover that an overnight test (which is not uncommon at the company) failed simply because an optional attribute was specified, e.g. assigning a value when type is not “INPUT”. Although this combination is forbidden according to the specification, it should not abort the execution if a reasonable specification can be inferred. However, it is necessary to clearly state what SIL will do in such cases. This is the motivation for the next decision.

Decision: To explicitly specify how SIL reacts to ambiguous specifications and to include this information in the warning messages generated.

Although it might not be possible to capture all ambiguous combinations, Table 3.4 summarizes those identified and how SIL should respond (with motivation). For any cases not specified, we will make effort to mention these while discussing the implementation. Otherwise, the logged warning message can be used to verify what SIL actually does. The general idea is that if an attribute is specified with a wrong data type or a required attribute is not specified, SIL terminates with an error message. However, if an optional attribute is specified, a warning is issued but SIL continues according to a pre-defined rule which is generally that the default value for the superfluous attribute is assumed.

Decision: To provide an XML schema definition which can be used by SIL users to verify strict adherence to the specification requirements.

An XML schema is a template which describes an XML file. By providing an XSD for the tolerance specification XML file, it will be possible to check whether tolerances are correctly specified. This will allow for easy transition if it is decided in the future to migrate to strict XML implementation. Such XSDs already exist at the company in the context of MoBasE. A problem that may arise from this decision is that since there are other specifications in the test case file, the schema may have to be extended to the other elements of this file which implies defining validation rules for them. Nevertheless, this decision should not require a lot of additional work and arguably lies within the scope of FR_8.
### 3.5 Provided Interface Methods for Tolerance Specification

In this section, we describe the provided interface from SIL to external tools like TE and SILVis for tolerance specification (see Figure 3.1).

We aim at providing a generic interface that can be used for all kinds of devices in which tolerance is specified. This has the advantages of ease of use, fewer number of methods in the provided interface and relative ease when new device types are added. This objective is further motivated by the fact that, as seen in Table 3.3, the attributes required for tolerance specification in most of the devices are the same.

Table 3.5 summarizes the defined methods. Two dots within a method’s argument list signifies that one or more of its parameters have been ignored because they are not relevant for the discussion at hand.

Note that by separating the interface methods for specifying/disabling tolerance in one and all instances of a given device kind, we discard the attribute target in the XML specification along with the associated optional specification. In addition, we provide a method for reproducing test cases using a previously generated seed. When reproducing a test case, the seed must always be initialized before any other

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Required response</th>
<th>Motivation/Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>The device, identified by a name or id, is not found in machine description</td>
<td>Terminate simulation with an error message</td>
<td>Trying to simulate elements in a machine other than the one specified is not allowed</td>
</tr>
<tr>
<td>Any attribute is specified with a wrong data type or with a value outside the specified set of acceptable values</td>
<td>Terminate simulation with an error message</td>
<td>This is an unacceptable (error) situation</td>
</tr>
<tr>
<td>target, name/id, type or value is not specified when required (see column 2 of Table 3.3)</td>
<td>Terminate simulation with an error message</td>
<td>This is an unacceptable (error) situation</td>
</tr>
<tr>
<td>Tolerance is specified in XML file for a device without tolerance, i.e. whose default tolerance range is not specified in machine description.</td>
<td>Execute tolerance specification from single-point range with warning if specifications fall out of this range</td>
<td>There is no requirement to specify tolerance in all devices. Hence, this is considered superfluous (e.g. may have resulted from using target=&quot;ALL&quot;) and should be ignored</td>
</tr>
<tr>
<td>Specified tolerance value falls outside the tolerance range in the machine description</td>
<td>Clip to upper- or lower-bound of the range as appropriate with warning</td>
<td>Tolerance value must fall within specified range</td>
</tr>
<tr>
<td>name/id is specified when target=&quot;ALL&quot;</td>
<td>Warn and ignore target, i.e. assume default case target=&quot;ONE&quot;</td>
<td>name/id takes precedence over target</td>
</tr>
<tr>
<td>type is specified along with value</td>
<td>Warn and ignore type, i.e. assume default case type=&quot;INPUT&quot;</td>
<td>value takes precedence over type</td>
</tr>
<tr>
<td>Optional sensor attributes (segmentId and nominalPosition) or actuator attribute transition specified for another device kind</td>
<td>Ignore with warning</td>
<td>These optional attributes are superfluous for other device kinds and should be ignored</td>
</tr>
<tr>
<td>Conflicting specification (1) for the same device within an &lt;injectTolerance&gt; or &lt;disableTolerance&gt; (2) between elements in &lt;injectTolerance&gt; and &lt;disableTolerance&gt;</td>
<td>Read and execute all specifications in order, i.e. last specification takes precedence</td>
<td>XML file is be parsed sequentially</td>
</tr>
</tbody>
</table>

Table 3.4: Handling ambiguous XML tolerance specifications
# Method Description
1. `injectTolerance(..)` Used to inject tolerance in a single instance of a particular device kind
2. `injectToleranceAll(..)` Convenience method for injecting tolerance in all instances of a particular device kind
3. `disableTolerance(..)` Used to disable tolerance in a single instance of a particular device kind
4. `disableToleranceAll(..)` Convenience method for disabling tolerance in all instances of a particular device kind, i.e. devices become 'ideal' again
5. `initializeSeed(..)` Used to (re)set the seed for random number generator

Table 3.5: Overview of SIL’s provided interface methods for tolerance specification

tolerance specifications. When not reproducing a test case, `initializeSeed(-1)` may be used to randomly generate a seed for example to override the seed specified in the test case file; however, this is not required and in general results in unnecessary logging and longer execution time as the system is reinitialized.

We now elaborate on the interface methods defined in Table 3.5. Examples of their usage are also contained in Appendix A.3.

I. bool injectTolerance( int deviceKind, string name(int id), string type, double value = -1.0, int segmentId = -1, double nominalPosition = -1.0, string transition = "BOTH" )

Pre-conditions

1. All attributes must be of/convertible to the required data type.
2. `name/id`, `segmentId` and `nominalPosition` must match the corresponding attributes of a device specified in the machine description file.
3. `deviceKind`, `type` and `transition` must take one of their predefined values.
4. Whenever `type="INPUT"`, `value` must be specified.
5. Attributes specific to a particular device kind (i.e. `segmentId`, `nominalPosition` and `transition`) may only be specified for that device kind.

Post-condition Returns true if tolerance is successfully injected in the device of interest; otherwise false.

II. bool injectToleranceAll( int deviceKind, string type, double value = -1.0, string transition = "BOTH" )

Pre-conditions

1. All attributes must be of/convertible to the required data type.
2. `deviceKind`, `type` and `transition` must take one of their predefined values.
3. Whenever `type="INPUT"`, `value` must be specified.
4. `transition` may only be specified if the `deviceKind` corresponds to actuators.
3.5. Provided Interface Methods for Tolerance Specification

Post-condition  Returns true if tolerance is successfully injected in all instances of the device of interest; otherwise false.

III. bool disableTolerance( int deviceKind, string name(int id), int segmentId = -1, double nominalPosition = -1.0, string transition = "BOTH" )

Pre-conditions
1. All attributes must be of/convertible to the required data type.
2. name/id, segmentId and nominalPosition must match the corresponding attributes of a device specified in the machine description file.
3. deviceKind and transition must take one of their predefined values.
4. Attributes specific to a particular device kind (i.e. segmentId, nominalPosition and transition) may only be specified for that device kind.

Post-condition  Returns true if tolerance is successfully disabled in the device of interest; otherwise false.

IV. bool disableToleranceAll( int deviceKind, string transition = "BOTH" )

Pre-conditions
1. deviceKind must convertible to an integer and take one of its predefined values.
2. transition may only be specified if the deviceKind corresponds to actuators.

Post-condition  Returns true if tolerance is successfully disabled in all instances of the device of interest; otherwise returns false.

V. bool initializeSeed( int seed = -1 )

Pre-conditions
1. If specified, seed must be the integer value logged in the test case being reproduced.
2. All other specifications used in the original test case must be identical.
3. The method may be invoked at most once prior to any other tolerance specification.

Post-condition  Returns true if the random number generator is successfully initialized (using seed if specified); otherwise returns false.

The parameter deviceKind is a unique identifier of each device and corresponds to the device identifiers stated in the first column of Table 3.1. All other parameters are as defined for XML specification in Table 3.3. Also, all false return values are accompanied by warning messages which indicate the problem that occurred.

Furthermore, we observe that this design allows optional attributes to take default values. This method is preferred over: (1) function overloading in C++ (i.e. multiple methods having the same name but
different parameters and/or return type) because overloaded functions make understanding a model more difficult in RoseRT since the differences between these methods having the same name are ‘hidden’ in their number/types of arguments; and (2) using variadic functions (i.e. printf-like functions which can take a variable number of parameters) because there is no way to automatically determine how many arguments are passed with an invocation and more importantly, because their behaviour is undefined when the expected number of arguments is not provided. In particular, when fewer arguments are provided, a crash may occur. As is the case in C++, we require that optional arguments are provided in the specified order and preceding arguments may not be skipped, e.g. in injectTolerance(..), a value (= -1.0 if type is not INPUT) must be specified before a segmentId or nominalPosition.

Remarks About Pre- and Post-conditions

No (formal) specification is complete without pre- and post-conditions. A useful way to view these is as forming a contract between the object and its client. The pre-conditions define a state of the program which the client guarantees will be true before calling any method, whereas the post-conditions define the state of the program that the object’s method will guarantee to create when it returns [21]. Therefore, the pre-conditions are required to be met upon invocation of the method. Otherwise, the behaviour of the system is not guaranteed. Except (explicitly) stated otherwise, e.g. in the inline code documentation, enforcement of any of the pre-conditions identified in this design is not required in the implementation.

Invariants for All Interfaces

The following invariants must hold for all the methods specified.

1. The number of devices of any type remains unchanged, i.e. instances of devices are neither created nor destroyed as a result of tolerance injection or disabling.

2. All other system variables remain the same except those required to realize the desired functionality and/or required to guarantee the specified post-condition(s).
3.6 Test Cases for Verification

In order to verify correct implementation of the design presented in this chapter, we define test cases in this section. In addition to outlining the test procedure, we also specify expected results which form a basis for assessing success or failure.

3.6.1 Motors and Pinches

The following test cases will be used to verify the implementation of tolerance in motor speeds. Since speed tolerance is also modelled for pinches, the exact same test cases can be reused for the latter. All tests assume time-based motor control as is typically the case.

**TEST CASE 1:** Using UT to create simple profiles of a motor with and without tolerance via XML tolerance specification.

**Steps**

1. Make a lua script that turns on the target motor and creates a simple profile as depicted in Figure 3.5(a).
2. In the test case specification XML file, specify a tolerance such that the motor’s speed is less (e.g. 67%) or greater (e.g. 133%) than its nominal value.
3. Run the simulation and plot the motor’s speed profile in time.
4. Repeat the simulation with tolerance disabled in the same motor and plot its new speed profile in time.

**Expected Result**

The result should be as depicted in Figure 3.5(b) showing slower (dotted line) or faster (dashed line) acceleration in the motor when tolerance is specified.

![Figure 3.5: Expected results for test cases to verify tolerance in motors (and pinches)](image)

**TEST CASE 2:** Using UT to create simple profiles of a motor with and without tolerance via XML tolerance specification with values selected randomly from the the motor’s tolerance range.
3.6. Test Cases for Verification

Steps

Same as in test case 1 above except that tolerance values are generated randomly, i.e. type="RANDOM" in the XML specification.

Expected Result

The result should be as depicted in Figure 3.5(b) for example, assuming a speed tolerance value of 0.67 (dotted line) or 1.33 (dashed line) was randomly generated.

TEST CASE 3: Using UT to reproduce a previous test case with randomly generated tolerance values.

Steps

1. Execute test case 2 above.

2. Retrieve the seed (from the SIL logging) and rerun the simulation with the specified seed.

3. Plot the new velocity graph.

Expected Result

The graphs from the original run and repeat should be identical. If DPlot supports this, verify point-wise equivalence.

TEST CASES 4 - 6: Repeat the test cases 1 - 3 using TE and the provided interface for tolerance specification. The results of the respective tests should be the same as those obtained using UT.

3.6.2 Sensors and Actuators

The following test cases will be used to verify the implementation of tolerance in sensor position and actuator delay. Currently, SIL supports only plotting of sensor and actuator statuses. For this reason, the same test cases can be used for both. All tests assume sheet-triggered sensor toggling (and software-controlled actuator toggling).

TEST CASE 1: Using UT to create simple profiles of a sensor with and without tolerance via XML tolerance specification.

Steps

1. Make a lua script that toggles the target sensor a few times and create a simple profile as depicted in Figure 3.6(a).

2. In the test case specification XML file, specify a tolerance such that the sensor’s position is less (e.g. 90%) or greater (e.g. 110%) than its nominal value.

3. Run the simulation and plot the sensor’s status in time.

4. Repeat the simulation with tolerance disabled in the same sensor and plot its status against time.
Expected Result

The result should be as depicted in Figure 3.6(b) showing earlier (dotted line) or later (dashed line) toggling of the sensor when tolerance is specified. Note that the actual mapping of sensor position to toggling time will depend on the speed of the sheets which trigger the toggling and may not correspond one-to-one.

![Figure 3.6: Expected results for test cases to verify tolerance in sensors (and actuators)](image)

**TEST CASE 2:** Using UT to create simple profiles of a sensor with and without tolerance via XML tolerance specification with values selected randomly from the sensor’s tolerance range.

**Steps**

Same as in test case 1 above except that tolerance values are generated randomly, i.e. type="RANDOM" in XML specification.

**Expected Result**

The result should be as depicted in Figure 3.6(b).

**TEST CASE 3:** Using UT to reproduce a previous test case with randomly generated tolerance values.

**Steps**

1. Execute test case 2 above.
2. Retrieve the seed (from the SIL logging) and rerun the simulation with the specified seed.
3. Plot the new status graph.

**Expected Result**

The graphs from the original run and repeat should be identical. If DPlot supports this, verify point-wise equivalence.

**TEST CASES 4 - 6:** Repeat the test cases 1 - 3 using TE and the provided interface for tolerance specification. The results of the respective tests should be the same as those obtained using UT.
3.6.3 Sheets and (Bent) Segments

SIL does not currently support any graphing options for sheets and (bent) segments. However, verification can be done by inspecting the log messages generated by SIL and specifying out-of-range tolerances that cause expected errors. The latter will depend on inputs from software developers and paper path layout experts. If these are unavailable, SIL logging inspection should suffice. Like other device kinds, the following three categories of tests should be performed using UT and TE: (1) specific tolerance values; (2) randomly generated tolerance values; and (3) test case reproduction.

3.7 Chapter Summary

In this chapter, we presented an extensive design of the proposed tolerance model. We began by considering general design issues regarding what to model, what tolerance parameters to specify, where and how to specify them, and the probability distribution for random tolerances. Next, we presented a preliminary tolerance model in which all tolerance was attributed to motors. We then extended our tolerance model to other sheet logic devices following the realization that the initial level of abstraction was insufficient. Subsequently, we defined interfaces for specifying tolerance both in XML and via the interface methods provided by SIL to external tools. Finally, we provided test cases to be used for verifying the implementation. The work in this chapter therefore forms a solid basis for the implementation addressed in the following chapter.
Chapter 4

Implementation

This chapter addresses the salient points of the implementation of the design presented in the previous chapter. UML sequence diagrams provide a convenient means to capture the flow of control and the interactions between different entities (e.g. classes and objects) in time. We therefore employ them to illustrate how tolerance specifications are handled in Sections 4.2 and 4.3. In addition, we address other significant changes made to SIL (Section 4.4). In order to facilitate understanding of the subsequent sections, Section 4.1 gives an explanation of the notation used in the sequence diagrams.

Recommendations for Initial Reading: All sections except 4.3.1, 4.3.2 and 4.4.

4.1 (RoseRT) Notation for Sequence Diagrams

RoseRT supports the use of UML sequence diagrams. However, the notation used does not strictly comply with UML specifications for sequence diagrams. Furthermore, we have adapted it to suit our context. Thus, we provide an overview of the notation used in the rest of this chapter; see Figure 4.1, for instance.

A solid line ending with a closed, filled arrowhead (→) on the receiver’s side indicates a blocking method call in which the caller waits until the receiver finishes execution and returns control to the caller. The return of control from such a call may be explicitly denoted by a dashed line from receiver to caller, ending with a closed, filled arrowhead (←) and appended with the returned value. Occasionally, the returned value is directly assigned to a variable during the blocking call in order to save space using the notation: returnedValue := methodName(). In all other cases, a void return type is implicitly assumed.

All returned status values are booleans and indicate success (true) or failure (false) of the method call. False is also returned for ambiguous specifications and out-of-bounds specifications (e.g. see Table 3.4). For instance, if minTolerance > nominalValue, it is clipped to the nominal value and the method returns false. The exact situation resulting in a failure status value can be verified from the SIL logging. In the future, status code may be introduced to distinguish different failure situations as stated in Section 6.4.2 when highlighting possible directions for future work. &status indicates that the returned status is a conjunction of the status of the called method and (some of) the intermediate methods it invoked during its execution.

We use two dots to indicate that a method takes one or more (additional) arguments which have been omitted because they are not quite relevant to the immediate discussion, e.g. methodName(a_Param, ..). This is typically done to save space and make the sequence diagrams more readable.

The naming convention for entities is as follows: name / role : type where the name identifies an instance (i.e. object) of class type. The role indicates the part this entity plays in the interaction. An entity with no name indicates a(n) (abstract) class on which static methods may be directly invoked.
Capsules are fundamental model entities in RoseRT already highlighted in Section 1.4.3.1. They are special classes with the following distinguishing characteristics: (i) All attributes and methods of a capsule are completely private except the public ports it exposes for communication; (ii) the sole means of communication with other capsules is by message passing (not method invocation) via these ports using protocols which define the messages that can be sent and received; and (iii) the behaviour of a capsule is defined and controlled by its state machine in which transitions are triggered by (asynchronous) signal events. Capsules are indicated by a symbol at the top right corner of the rectangle atop the entity’s lifeline. The rest of the notation such as focus-of-control, entity creation/destruction and comments, complies to standard UML notation for sequence diagrams.

Finally, messages have been numbered. These sequence numbers, prepended by “Msg” (i.e. message), will be included (in parenthesis) in the sequence diagram descriptions for better understanding. Unless otherwise stated, the method names used in the sequence diagrams correspond to those of the actual methods implemented in SIL.

4.2 Handling XML Tolerance Specifications

In this section, we explain how reading of XML tolerance specifications in both machine description and test case files is implemented in SIL. This discussion is particularly useful for persons who are interested in the internal workings of SIL and may be skipped by other readers without loss of flow.

Figure 4.1: Handling XML tolerance specifications
The sheet logic capsule implements the simulated plant including all the sheet logic devices, namely: motors, pinches, sensors, actuators, segments and sheets. Figure 4.1 shows the sequence of method calls involved in the reading and initialization of XML tolerance specifications under normal operation, i.e. assuming compliance to the template presented in Section 3.4. These operations are all performed during the initial transition of the sheet logic capsule’s state machine.

The device specifications are read from the machine description file, layout.xml, after which the device is created and initialized. This step involves a number of method calls collectively represented by Msg 1 in Figure 4.1 and corresponds to the original initialization phase in SIL before tolerance modelling was introduced. Following this step, the tolerance range of the device is read (Msg 2) and initialized (Msg 2.1). If no tolerance range is specified as is the case for an ideal device, the range is set to the nominal value. For example, an ideal motor’s speed tolerance range is set to [1.0, 1.0].

After initializing all devices including their interconnections, tolerance specifications for the test case being run are read from the test case file, SILSimulation.xml (Msg 3). If a seed has been specified, it is read and initialized. Otherwise, a random seed is generated for the test run. Since the same methods are used to read tolerance specifications for all device kinds, a unique identifier is required in Msg 3.1 to distinguish the device kinds. These identifiers were outlined in Table 3.1. Next, all specifications are read sequentially and checked for compliance to the conditions outlined in Section 3.4.2 for test case tolerance specification (Msg 3.1.1). The specifications may be for injection or disabling of tolerance in one or all devices of the kind under consideration. In Figure 4.1, we assume an injection specification for a single motor named a_Motor. A static method is invoked on the motor class (Msg 3.1.1.1) with the name of the device of interest, the type of tolerance, e.g. MIN, MAX, AVERAGE, etc., and a value when type is “INPUT”. The device is found and its tolerance value is set accordingly (Msg 3.1.1.1.1).

The status returned by Msg 3 is the conjunction of its status and those of all intermediate calls. For disabling tolerance, Msg 3.1.1.1.1 is still used with type set to “OFF”. In other words, all tolerance specifications are internally realized by invoking the same method. However, we separate tolerance injection and disabling in the XML specifications in compliance to the design in Chapter 3. This design allows anyone looking at the XML specifications to easily decipher what is being done which would not be the case if a single node, say <setTolerance>, were used in the test case file with specifications for tolerance injection and disabling intermingled.

For illustration purposes, we have considered reading and initializing motor tolerance specifications. Nevertheless, the same mechanism is applicable to all other device kinds. More importantly, we remark that Msg 3.1 and all its sub-calls could be moved to the base class of each device with the same results. In this case, however, virtually the same code would have to be duplicated in each device kind, thereby making maintenance more difficult. Hence, by employing generic methods in the sheet logic capsule for all device kinds, we reduce code size by avoiding unnecessary duplication and improve code maintainability albeit at the cost of an extra parameter to distinguish the device kinds and occasional short code blocks for attributes specific to (a) particular device kind(s). For instance, unlike all other devices, the tolerance range for sheets is specified in the test case file (Section 3.3.6.3). Hence, it is read and initialized in Msg 3.1. This implementation choice as well as implementing Msg 3.1.1.1 and all similar methods as static methods, undoubtedly results in less code.

4.3 Handling Tolerance Specifications via the Provided Interface Methods

In the previous section, reading of XML tolerance specifications was addressed. This section focuses on how tolerance specification via SIL’s provided interface to external tools (Section 3.5) is realized.
The ensuing discussion may again be skipped without loss of flow by readers who are not particularly interested in the internal workings of SIL.

Point-of-interest (POI) commands, POIs, are an existing mechanism for run-time interaction with SIL via TE or SILVis. These messages are received by a command interface capsule in SIL and forwarded to the sheet logic capsule where they are parsed and executed. POI commands may be executed immediately or alternatively, delayed until a specified condition is met. The condition is expressed in terms of the number of sheet rising or falling edges detected at the specified segment position. We begin by explaining the existing mechanisms for handling both immediate and delayed POI commands since they are directly applicable to handling tolerance specifications for all device kinds. Subsequently, we describe the changes which were made to enable the realization of sheet delay (FR_10 in Table 2.2).

**Figure 4.2: Handling immediate POI commands**

Figure 4.2 shows how immediate/direct POI commands are handled in SIL. When a command is received by the sheet logic capsule, it invokes one of its methods (Msg 1) which in turn performs a sequence of steps, the first being to determine whether the POI command is a direct or delayed one. Next, the received string is parsed and the name and parameters of the POI command are extracted (Msg 1.1). Subsequently, a new instance of the corresponding POI action class is created (Msg 1.1.1). In Figure 4.2, we use tolerance injection as an example. All the POI action classes inherit from a base PoiAction class in SIL (which also corresponds to the role specified for the POI entity in Figure 4.2). Once created, the required action is immediately performed (Msg 1.2) and the POI object is deleted. For tolerance injection, the action performed is to determine the device kind, say a motor, and then execute the same calls done during tolerance injection via XML specification described in the preceding section (i.e. Msgs 3.1.1.1 to 3.1.1.1.2 in Figure 4.1). In other words, the same underlying mechanism is used for tolerance execution regardless of whether they originate from XML or POI specifications. Although the methods for tolerance execution return a status value, these statuses are ignored; instead, void is returned (Msg 1.3) in compliance with the existing POI implementation. This illustrates one of the (few) situations in which adapting to the existing implementation took precedence over a more elegant solution.

Handling a delayed POI command is more involved as evidenced by the length of the sequence diagram in Figure 4.3 compared to that of the one for immediate POIs in Figure 4.2. Until Msg 1.1.2, the behaviour is the same as for direct POIs already explained. However, since the message is not executed immediately, a new PoiEdge instance is created (Msg 1.2) and added to the list of pending PoiEdges. Among other parameters, a PoiEdge stores a condition (edgeTarget) and a pointer (lp_PoiAction) to the PoiAction to be executed whenever this condition is met. Within the constructor of a new PoiEdge object, a pointer
4.3. Handling Tolerance Specifications via the Provided Interface Methods

Implementation

Figure 4.3: Handling delayed POI commands

to itself is added to a list of PoiEdge pointers which is maintained by the segment on which the target point-of-interest (position) lies. This is done because segments are responsible for detecting rising and falling edges as they moves sheets over the POI position and consequently for updating the edgeTarget accordingly.

One of the operations performed by the sheet logic capsule when it receives a tick from the clock is to iterate through the list of pending POIs (Msg 2), checking their execution conditions. If the condition of a POI has been reached, i.e. edgeTarget equals zero (Msg 2.1.1), the related action is then performed and a success status value (true) is returned (Msg_2.1.2). Once the action has been performed, the associated PoiEdge is deleted (Msg_2.2). Since the PoiEdge class derives from PoiAction, the compiler automatically frees the memory allocated to the related PoiAction.

All the methods in the provided interface for tolerance specification, therefore, have a D_PoiAction* class derived from the base PoiAction where * corresponds to the method names in Table 3.5. Moreover, as earlier stated, the mechanism discussed so far suffices for handling tolerance specification for all device kinds. Nonetheless, some changes were required to realize sheet delay as described next.

4.3.1 Implementing Sheet Delay

The motivation for delaying sheets and a means to achieve it were presented in Section 3.3.6.2. The approach proposed was to request a callback from the clock capsule to be sent when the duration of sheet delay elapsed following which the sheet would be unblocked. However, this could not be realized using the existing mechanism for handling POIs because the PoiAction object is deleted immediately the action (in this case, blocking the sheet) has been executed. Hence, at the time the callback is performed, the target PoiAction object no longer exists.

To tackle this problem, we defined a delayedAction flag in the base PoiAction class. Defining this flag in the base class allows for reusing the callback mechanism for other purposes if the need arises in the...
4.3. Handling Tolerance Specifications via the Provided Interface Methods

Figure 4.4: Implementing sheet delay
4.3. Handling Tolerance Specifications via the Provided Interface Methods

Implementation

When set, the associated PoiAction object is not immediately deleted. Figure 4.4 illustrates these modifications made for delayed PoiActions. Relative to Figure 4.3, these changes are easily identifiable by italicized text. When a non-zero duration is specified for the D_PoiActionStopSheet(..), sheet delay is implied. As before, the action (of stopping the sheet) is performed when the set condition is met, i.e. when the particular sheet to be delayed arrives at the target segment position. However, this action is extended by a callback request to the clock (Msg 2.1.1.1). An additional condition which checks the value of the delayedAction flag is included so that the PoiEdge object is not deleted immediately. When the duration of the delay elapses, the clock invokes a callback wrapper method on the PoiAction base class with the previously received pointer to the delayed PoiAction object (Msg 3). A timeout is then invoked on the object (Msg 3.1) during which the sheet is unblocked. The action has now been completed and the associated memory can be freed. This is executed by the base PoiAction class (Msg 3.2).

The idea is the same for sheet delay associated with direct POIs: With reference to the original mechanism for handling direct POIs in Figure 4.2, Msg 1.3 is now guarded by a condition that checks the delayedAction flag. The object is immediately deleted only if this flag is not set. Otherwise, deletion is performed by the base PoiAction class following the clock callback and unblocking of the delayed sheet. Thus, we do not provide a separate diagram to illustrate the change for direct POIs.

Although we employed its class callback functionality, no changes were made in the clock capsule. Nonetheless, for the sake of completeness, we present an overview of how timing is implemented in SIL in the next section.

4.3.2 Timing in SIL

The clock capsule generates the heartbeat that keeps the whole simulation running in synchrony. In addition to maintaining a global notion of time, the clock also implements the functionality of a scheduler - it determines which nodes to trigger and when to do so. These two functions are the subject of this section. To facilitate the reader's understanding of this discussion, we briefly explain the additional notation in Figure 4.5 with respect to the RoseRT notation earlier described in Section 4.1.

A solid line ending with an open, half-sided arrow head (→) indicates an asynchronous message between capsules. The sender immediately proceeds with its own logical thread of control after sending the message without blocking. A hexagon (idle) indicates the current state of the capsule identified by a name, e.g. idle. A state transition is usually triggered by a signal event generated upon receiving a trigger message. Such triggers are identified by the stereotype <<trigger>> for passive classes and by the name of the trigger message (e.g. <<run>>) for capsules. Variables are defined to the left of the capsule they belong to using the following syntax: variable name: datatype (= initial value). Down in the entity's lifeline, new values may be assigned to these variables. For easy identification, a variable whose value has changed is indicated in italics.

The clock (a_Clock in Figure 4.5) is connected to one or more capsules in SIL each of which models some functionality of the real machine. For instance, the sheet logic capsule (a_Sheetlogic) models the simulated plant. At least one capsule (e.g. a_Node) which defines the behaviour of the embedded software is needed. Depending on the kind of test being run and the architecture of the printer (see Figure 3.2), more than one sub-controller node and/or a main controller node may be present. Other nodes such as a pipe server to model the communication network between the controllers and a stub that models the modified I/O layer in Figure 3.1 are also present. The clock communicates with these nodes via its public ports (e.g. p_Outside).

During start up, each of the connected nodes registers itself by sending an informEvery(..) message to the clock (Msgs 1 and 2). The syntax for messages sent between capsules is: port name.message(...).send(). Notice however that the name of the same shared port may differ at the sender's and receivers' ends.
4.3. Handling Tolerance Specifications via the Provided Interface Methods

Figure 4.5: Timing in SIL
For example, the same port identified as p_Clock by the planf (Msg 1) is called p_Outside by the clock (Msg 4). Port names are simply chosen to be as intuitive as possible. Nevertheless, with the registration message to the clock, each ‘subscribing’ node indicates its priority and at what frequency it must be ticked. A tick is simply a mutually understood message that signals the transfer of control from the sender to receiver; in our case, from a_Clock to any of the other capsules that subscribes to it.

Upon receiving the subscription message, the clock adds the sender to its global schedule list and initializes the node’s next-scheduled time to \( t = 0 \text{ms} \). After all nodes have registered themselves, signalled by a run (trigger) message from the top-level capsule responsible for starting up and shutting down SIL (but not shown in Figure 4.5), the clock initializes the global current time \( (g\text{CurrentTime}) \) to zero and makes a transition to the running state where it remain until the simulation run ends.

In the running state, the clock continually loops through a series of actions which dictate timing in SIL. To explain these, we consider the simple scenario depicted in Figure 4.5 with only one sub-controller node and the sheet logic capsule with delay tolerance in actuators assumed. Upon entering the running state, the clock determines the next node to tick by scanning through its list of registered nodes and picking the one with the lowest next-scheduled time (Msg 3). For nodes with the same next-scheduled time, the one with the highest priority is ticked first. Initially both a_Sheetlogic and a_Node have next-scheduled times of 0ms but we assume that a_Sheetlogic has higher priority and hence, it is ticked first.

The clock maintains another list of callback requests. A capsule or passive class can, at any time, request for a callback at a future time expressed either in absolute terms, i.e. “simCallbackAt(...)” or relative to the current time, i.e. “simCallbackIn(...)”. Thus, before a_Sheetlogic is ticked, the clock checks for any pending callback requests scheduled for execution between the current time \( (g\_CurrentTime) \) and the scheduled time of the node to be ticked. At time \( t = 0 \text{ms} \), there are no callback requests so the clock sends a tick to a_Sheetlogic (Msg 4) after updating the current time, i.e. \( g\_CurrentTime = l\_NextTime = 0 \text{ms} \) and the next-scheduled time of a_Sheetlogic by its frequency, i.e. \( scheduledTimeP = 1 \text{ms} \). Since the clock’s p_Outside port is shared by multiple nodes, a sendAt(2) is used to address the particular receiver node bound with that port number (cf. a send() which broadcasts the message to all receivers). Note that the clock retrieved these port numbers when the nodes registered themselves earlier. The tick causes a transition in a_Sheetlogic during which it evaluates the state of all sheet logic devices, e.g. updating motor speeds, sensor and actuator states, sheet position, POIs, etc.

A tick indicates the transfer of control to the receiver to perform its operations. Thus, a_Sheetlogic must signal completion by sending a done() message back to the clock (Msg 4.1). This message triggers a repeat of the same operations to determine the next-scheduled node (Msg 5) and the Controller, a_Node, is ticked (Msg 6). During its execution, suppose that the embedded software toggles the state of one of the actuators, say a pinch lift (Msg 6.1). However, owing to the switching delay, this operation is not completed immediately. Instead, a callback request is issued to the clock by the actuator (Msg 6.1.1). In our example, a switching delay of 0.5ms is arbitrarily assumed. After a_Node returns control to the clock (Msg 6.2), the node with the lowest next-scheduled time is again determined to be a_Sheetlogic (Msg 7), i.e. \( \text{min}(1 \text{ms}, 2 \text{ms}) \). But now, there is a pending callback to be executed at \( t = 0.5 \text{ms} \). Hence, the current time is updated to 0.5ms after which the callback is performed (Msg 8) causing the actuator to go to an active state (Msg 8.1). We observe that the callback is performed by (a blocking) method invocation since the actuator is implemented as a passive class. After performing the callback, the current simulation time is updated to the current-scheduled time of a_Sheetlogic, i.e. \( g\_CurrentTime = 1 \text{ms} \). The next-scheduled time of a_Sheetlogic is then extended by its frequency (Msg 9), i.e. \( scheduledTimeP = 2 \text{ms} \). The simulated planf is ticked again (Msg 10) and the whole process repeats until the simulation runs to completion, or terminates upon user request or due to an error.

In the manner described above, a single global time is maintained while keeping separate scheduled times for each node. This mimics the typical situation in real printers where the nodes are distributed and
have their own separate (but synchronized) clocks. In SIL, the global time is stored in a shared memory, allowing the controller to access it directly when the need arises.

In the real machine, a time-slicing approach [4] is used in which the embedded software is ticked at a frequency of 2ms. Each of these 2ms time windows is subdivided into 20 equal slots of 1µs each. The embedded software schedules its operations within these slots. Real-time guarantees are trivially achieved by ensuring that the tasks allocated to these time slots can never exceed 1µs. Unused slots are idled away, i.e. they are not allocated to soft real-time tasks as proposed, for example in [7, 3]. The time-slicing scheduling approach has the advantage of being simple to analyze and implement albeit at the cost of reduced efficiency in time usage when compared to other scheduling approaches [4]. Finally, we remark that SIL simulation is not aimed at investigating timing requirements since the notion of time is based on simulated and not real time [29, 17]. This fact is also evidenced by timing model implemented in SIL.

4.4 Other Significant Changes/Additions to SIL

The aim of this section is to highlight some other changes we made to SIL. These include: realizing unimplemented specifications discovered while reverse engineering the SIL source code (silent mode); adding desirable features (interactive mode, plotting default graphs from provided interface); bug fixes (crash while attempting to plot custom graphs); and architectural changes in order to realize the design (segment node class).

4.4.1 Interactive Mode

SIL supports two modes: silent mode and normal mode. The silent mode is specifically for regression testing during which, according to the inline documentation, the simulation will not pause to indicate errors but will rather directly terminate. However, at the time of this project, this feature was not yet implemented and system("pause") statements still causes the simulation to pause even in silent mode.

In addition to implementing this documented yet non-existent functionality of a silent mode, we defined a third mode - the interactive mode. This new mode was motivated by the fact that in normal mode, SIL is only expected to pause upon warnings and just before termination in the event of an error (both warning and errors are typically highlighted in red within the console). However, when testing a new feature or when a SIL user creates a new test case (with tolerance specifications) and wants to verify that it does what (s)he expects, additional pauses are very helpful, e.g. at information messages after the seed is initialized or after tolerance has been successfully injected.

A new option INTERACTIVE defines the interactive mode and can be used wherever SILENT and NORMAL are valid for the silent and normal modes respectively. All system(string str) calls are replaced by my_system(string str, bool flag = false). The behaviour of this new (wrapper) method is as follows: Whereas in the silent mode, all calls do nothing, they are executed in the normal mode by calling system(str) with the provided string argument, e.g. “pause”. The optional flag must be set to true to enable additional pauses in the interactive mode; otherwise, this parameter defaults to false. In this way, silent and interactive modes are implemented while retaining the familiar behaviour of the normal mode (backward compatibility). The interactive mode was employed extensively in this project during the implementation and verification phases.

4.4.2 Plotting Default Graphs via a Provided Interface Method

In order to run the test cases using TE as described in Section 3.6, an additional method for plotting default/custom graphs via the provided interface was implemented. Default graphs are used for plotting...
4.4. Other Significant Changes/Additions to SIL

Implementation

graphs of predefined characteristics of specific devices, e.g. motor speed, sensor states, etc. They are defined in the file `graphs.xml` as already mentioned in Section 3.1 when addressing the architecture of SIL.

We note that this feature, while useful for test case verification, does not affect regression tests as TE is able to run multiple tests while a graph window is open. However, there is a caveat worthy of mention: DPlot currently allows a maximum of 32 different graphs; trying to print more graphs will result in an error which may hamper regression tests.

The interface for this function is specified as follows:

```c
void showDefaultGraph( int option )
```

**Description:** Used to plot custom graph via SIL’s provided interface to external components. Graph options are specified in `graphs.xml`.

**Pre-conditions**

1. `option` is a valid identifier defined in `graphs.xml` in element `<customgraph option="option">...</customgraph>`.

**Post-condition** Prints the default graph if correctly specified; otherwise fails possibly with a warning.

An existing bug which caused SIL to crash when attempting to display default graphs was encountered while using SIL and fixed. It resulted from incorrect parsing of the XML specifications for default graphs in `graphs.xml`.

4.4.3 The Segment Node Class

In Section 3.3.5.2, sheet behaviour in bent segments was addressed. In the model presented there, each sheet may perceive the length of the segment differently, reflecting the path it takes in the bend. As mentioned previously, sheets are moved by segments. To enable this, each sheet maintains some (dynamic) information such as (pointers to) the segments it is currently spanning, the pinches and sensors it is currently under, etc. In particular, when a sheet is moved to a new segment (with the help of the segment that it was previously on), it adds a pointer to the new segment to its list of spanned segments. The length of this segment is used to determine the initial position of the sheet’s pulling edge on the new segment. However, given our tolerance model, the length of bent segments may now vary relative to each sheet. Hence, it will be wrong to use the nominal bend length for this purpose as already mentioned in the design. Therefore, for each spanned segment, the sheet stores its perceived length of that segment. Accordingly, the list of spanned segments no longer stores pointers to segments but to segment nodes encapsulating the segment pointer and the sheet’s perceived length of the (bent) segment. This same length is used to move both pulling and pulled edges of the sheet through the segment, as per design specifications.

The segment node class defined for this purpose is depicted in a UML class diagram in Figure 4.6. We remark that, for readability, not all relationships between the classes are captured in Figure 4.6; instead the figure shows only those interactions that directly involve the sheet class (D_Sheet). Some attributes of the segment class (D_Segment) are visible: `minLength` and `maxLength` define a bent segment or buffer area and default to `length` when not specified; `currentTolerance` specifies the tolerance in segment length and defaults to 1.0 for ideal segments. Furthermore, we stress the fact that, in general,
4.4. Other Significant Changes/Additions to SIL

Figure 4.6: UML class diagram for sheets showing the new segment node class

D_Segment::length differs from D_SegmentNode::sheetSegmentLength for sheets in bent segments. Moreover, we purposely did not call the latter parameter “length”, to avoid inadvertent reference to the wrong attribute.

4.4.4 XML Schemas and a Tool for Validating Tolerance Specifications

When discussing how to enforce correct XML tolerance specifications in Section 3.4.2.6, we decided to develop an XML schema against which SIL users can easily validate their tolerance specifications in SILSimulation.xml. Since schemas specify what elements and attributes are allowed in the corresponding XML file, they provide a convenient means for checking validity of XML data.

We benefited from the several XSD tutorials available online as well as from the existing MoBasE machine description schema within the company. Nevertheless, a challenging situation was encountered: Our XML specification was designed in favor of ease of specification (cf. FR_5). Therefore, we allowed users to omit optional parameters when it did not lead to any ambiguity. For example, when a valid input was specified, type=“INPUT” was implicit and could be omitted (see the second column of Table 3.3 for more details). This situation results in different combinations of acceptable attributes. Unfortunately, the XSD language does not provide any means for handling such specifications. This problem has been observed and reported on a few online forums. Generally, three possible solutions are presented. They are: (1) supplementing XSD with another schema language; (2) solving the problem programmatically using C(++), Java, Perl, or any other programming language; or (3) expressing additional constraints using an XSLT/XPath stylesheet.

We opted to supplement XSD with a separate rule-based grammar, Schematron\(^1\). This choice was motivated by the fact that Schematron is a simple yet powerful schema language for making assertions about the presence or absence of patterns in XML documents. Secondly, it can be integrated with XSDs either by embedding Schematron rules in the XSD or in a separate Schematron-based schema. We adopted the latter approach as it facilitates independent development and modification albeit at the cost of maintaining and validating two separate files. The idea is generally to specify as much as possible in the XSD and to capture the additional constraints which cannot be expressed in XSD using Schematron.

\(^1\)http://www.schematron.com/
We developed a simple command-line tool, XMLValidator, which realizes the functionality described above. It consists of two main parts: an XSD schema validator written in C programming language and a Schematron validator which uses SAXON\(^2\) as an XSLT processor. Both are wrapped together using a batch script. The validation rules were derived from the conditions in the second column of Table 3.3. Ambiguous specifications (Table 3.4) were also checked and reported. An XML specification is declared to be valid by this tool if it violates none of the specified rules; otherwise, validation fails typically with an indication of the non-conformant element(s). Along with the XSD and Schematron schemas, this tool forms part of the final project deliverables.

Another issue concerns the fact that SILSimulation.xml contains other specifications necessary to execute a test case, e.g. the update frequency of the simulated plant, the (sub-)controller node DLLs to load, etc. For the sake of completeness, we decided to validate the entire file. Hence, we included the specifications of these elements and their attributes in the schemas.

### 4.5 Chapter Summary

In this chapter, we discussed the implementation of the design presented in Chapter 3. Using sequence diagrams, we explained important mechanisms in SIL including how tolerance specifications in XML and via SIL’s provided interface methods are handled. For the latter, we saw that the existing mechanism for handling POI commands could be reused for tolerance execution. However, an extension was required for sheet delay, which we also discussed. Next to that, we presented other significant changes made to SIL in order to realize our design. In addition, we explained the tool we developed for XML validation using XSD and Schematron schemas. In the next chapter, we verify the implementation using the test cases defined in Section 3.6. We also validate both the design and implementation against the project requirements.

\(^2\)http://saxon.sourceforge.net
Chapter 5

Validation and Verification

Design and implementation cannot be complete without verification and validation. Thus, this chapter focuses on answering two important questions, namely: “Have we done the right thing?” and “Have we done it correctly?”. Whereas validation addresses the former, verification focuses on the latter. In Section 5.1, we validate both the design and implementation against the project requirements outlined in Chapter 2. Section 5.2 presents the results of the verification test cases defined in Chapter 3. Next, Section 5.3 follows with an analysis of the effects of our extension on SIL’s timing, memory usage and logging performance.

Recommendations for Initial Reading: Sections 5.1 (except 5.1.1 and 5.1.2), 5.2 and 5.4.

5.1 Requirement Validation

In this section, we validate the design (Chapter 3) and implementation (Chapter 4) against the requirements outlined in Chapter 2. Validation is very important because a design and/or implementation which fails to meet its requirements is of little or no use. The project requirements (high- and low-level combined) are reproduced in Table 5.1 with an extra status column to indicate to what extent each one of them was realized. In addition, we highlight specific design and/or implementation choices that enabled/facilitated the realization of these requirements including a motivation for undone (i.e. skipped) and uncompleted (i.e. in progress) ones.

FR_1 We enabled backward compatibility by: reusing existing XML files for tolerance specification (Section 3.2.2); providing support for the deprecated “delay” attribute of pinch lifts and clutches (Section 3.3.3) during implementation; and turning tolerance off by default (Section 3.2.5) and specifying tolerance ranges for ideal devices in such a way that any inadvertent tolerance specifications do not affect their behaviour (Section 3.3). Moreover, tolerance implementation was limited to sheet logic capsule and sheet logic device classes (localization of implementation).

FR_2 All tolerance implementation was done in RoseRT; coding style and naming of variables comply as much as possible to the rules in the internal RoseRT design and implementation guide and/or to the existing SIL convention. The validation tool was written in C because of availability of sample code in that language. Moreover, it is a simple tool of only a few lines of code for which the rich UML modelling features of RoseRT would obviously have been an overkill solution.
<table>
<thead>
<tr>
<th>ID</th>
<th>Brief Description</th>
<th>Priority</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR_1</td>
<td>To ensure that the extended SIL framework is backward compatible and that the changes required to implement the desired functionality are as localized as possible</td>
<td>Must</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>FR_2</td>
<td>To implement the framework using the existing modelling software at the company (IBM Rational RoseRT) and according to the existing software development guidelines</td>
<td>Must</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>FR_3</td>
<td>To provide a means to specify the variability in one or more device characteristics which affect the sheet transport along an arbitrary paper path</td>
<td>Must</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>FR_4</td>
<td>To ensure that results are accurate and reproducible and identical for all executions of the same test case</td>
<td>Must</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>FR_5</td>
<td>To allow case of specification of variability, e.g. options to set a given characteristic to its worst-case (or best-case) value for a subset of devices in a single step</td>
<td>Should</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>FR_6</td>
<td>To model the detection and correction of “z-displacement” and skewness along the paper path</td>
<td>Could</td>
<td>NOT DONE</td>
</tr>
<tr>
<td>FR_7</td>
<td>To model delays in the (worst-case) response times of sensors and actuators especially segment switches</td>
<td>Must</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>FR_8</td>
<td>To unify the layout specification XML file with the MoBosE schema to remove unnecessary ambiguity and redundancy</td>
<td>Should</td>
<td>NOT COMPLETED</td>
</tr>
<tr>
<td>FR_9</td>
<td>To model sheet buffering effects such as blousing in a paper path.</td>
<td>Won’t</td>
<td>NOT DONE</td>
</tr>
<tr>
<td>FR_10</td>
<td>To model sheet delay at an arbitrary point of interest along the paper path</td>
<td>Could</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>IR_11</td>
<td>To provide interfaces for introducing variability into SIL which are compatible with the existing framework for automated regression testing (i.e. TE)</td>
<td>Must</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>IR_12</td>
<td>To support the specification of tolerance values for components of interest without explicit use of TE, e.g. in a(n) (XML) file</td>
<td>Must</td>
<td>COMPLETED</td>
</tr>
</tbody>
</table>

Table 5.1: Validation of project requirements

FR_3 Our tolerance model defines a single tolerance parameter representing the combined (possibly weighted) tolerance effects in each of the following device kinds (as): motors (speed), pinches (speed), actuators (response time delay), sensors (position at which sheets are sensed), segments (length) and sheets (length). We equally modelled sheet behaviour in bent segments. Details of all these are extensively covered in Section 3.3.

FR_4 We checked accuracy by running several tests and by discussing expected behaviour with stakeholders. Reproducibility was enforced by logging the global seed used to generate random tolerance values (Section 3.2.4) in a deterministic manner. This seed can always be used to regenerate all tolerance values and, therefore, to reproduce test cases. Furthermore, all tolerance executions are written to the SIL logging with easily identifiable warning messages. We also added a new interactive mode in which users can more easily verify that their specifications are correctly executed.

FR_5 Our decisions to: provide injectToleranceAll(...) and disableToleranceAll(...) methods (Section 3.5); allow a target="ALL" option in <injectTolerance> (Section 3.4.2); read tolerance specifications sequentially, thereby allowing unrestricted special “all-except” specifications (Section 3.4.2.5); and allow omission of optional attributes in XML specification were all geared towards fulfilling this requirement. Also, we used intuitive attribute names and XML attribute values to enhance easy remembrance.

FR_6 We did not model skewness and “z-displacement” owing to limitations identified following analysis. This is the subject of Section 5.1.1 below.
FR_7 This requirement was covered by the realization of FR_3.

FR_8 This requirement could only be partially fulfilled as explained in Section 5.1.2.

FR_9 This requirement was not considered at all (i.e. “won’t” priority), motivation for which was already presented in Section 2.2.

FR_10 The design and implementation of sheet delay were discussed in Sections 3.3.6 and 4.3.1 respectively.

IR_11 In order to realize this requirement, we implemented the provided interface methods for tolerance injection in python for use with TE. We also tested these methods when running the verification test cases defined in Section 3.6. This python library forms part of the final deliverables of the project.

IR_12 Tolerance specification in the test case file using the XML interface described in Section 3.4.2 meets this requirement. We also verified the implementation by running tests with UT on finisher control software.

5.1.1 Motivation for Skipping Skewness and Lateral Displacement Models

A sheet is said to be skewed whenever its orientation (in the x-y plane) is at an angle relative to the expected orientation. In other words, the sheet’s edges are not straight relative to the segment on which it is moving. Skewness may occur as early as during sheet separation at the paper input trays. As a skewed sheet is transported along the paper path, its skewness may increase as a result of its interaction with other devices such as pinches and bent segments. If a skewed sheet proceeds to the print head, the image fused onto it also becomes unaligned to the sheet edges. To prevent this undesirable situation, skewness detection and correction is performed. Different mechanisms for skewness correction have been successfully developed in the company over the years. The most common approach involves using a number of pinches placed next to each other but controlled independently. With the help of skewness detection sensors, the skewness angle is calculated and correction is performed by adjusting the speeds of the pinches relative to one another.

The dynamics of sheet behaviour during skewness correction are known, from experience, to be difficult to capture in a mathematical equation. Likewise, the algorithms developed for skewness correction often require several iterations to calibrate them to the acceptable millimeter precision.

Modelling skewness detection and correction has the advantage of providing the developers of these control algorithms for new projects with a means to quickly verify and improve their control software leading to less iterations and more accurate control. Nonetheless, although modelling skewness detection is straightforward, a realistic model in SIL for skewness correction, given its complexity, was not feasible in this project for the following main reasons:

1. The equations for generating profiles for the FPGA stepper motors which control the skewness correction pinches have fourth and fifth order terms. However, the current stepper motor model in SIL only features a third order equation. Therefore, modelling skewness will require a non-trivial extension of the current stepper motor model, consultation with experts and a lot of time albeit at the risk of introducing errors into the model.

2. More importantly, the correction requires nanometer precision of the stepper motors. To achieve this in SIL, the update frequency of the sheet logic must be increased perhaps by several orders of magnitude. Consequently, this may result in unacceptably huge loss of performance in terms of SIL’s execution time.
5.2. Test Case Results

Simply extending SIL with skewness detection (which is relatively easy) without simulating the correction has no added value since skewness modelling is intended to facilitate the development and optimization of the correction algorithms. For these reasons, we did not model skewness.

Independent of skewness, a sheet may be closer to or farther away from the segment edges than expected. This lateral displacement, commonly referred to as “z-displacement”, is actually an effect in the x-y place and may be more appropriately called y-displacement. If not corrected, it results in a displaced image after printing, e.g., extra white spaces at the top/bottom of the sheet and/or truncated images. Hence, “z-displacement” is detected and subsequently corrected either by moving the sheet laterally or alternatively by moving the image to match the sheet’s position before fusion of the image and the sheet takes place at the print head. Interestingly, a careful consideration of this phenomenon revealed that it has no significant effect on sheet transport which takes place in the x-direction especially when correction is performed by adjusting the image. This is because lateral displacement relates to the y-direction and not the x-direction. The problem is therefore more linked to the process/data path aspect of a printing job than to the paper path. Since SIL currently models the paper path and this project specifically addresses the effects of variability in the simulated plant on sheet transport, detection and correction of lateral displacement fall outside the project scope. Hence, we did not model it.

5.1.2 Unifying SIL Layout and MoBasE Machine Descriptions

A number of concurrent issues and open questions regarding the best way to proceed with MoBasE made it an unrealistic target to immediately unify, i.e., merge overlapping specifications in layout.xml and MoBasE template for machine description. Rather, we identified overlapping properties and attributes and made recommendations on how they can be fixed. These recommendations are summarized in a separate (internal) document which forms part of the final deliverables of this project.

In the following section, we discuss the results of the verification test cases.

5.2 Test Case Results

In this section, we present the results of the test cases defined in Section 3.6 for verifying the implementation. These results were generated by running finisher and/or main controller software on the new SIL implementation extended with tolerances.

Figure 5.1(a) shows the results from a controlled motor, say MOTOR_A. Tolerance values of 90% and 110% of the nominal speed are shown along with the nominal speed profile. The results match the expectations expressed in Figure 3.5 - speed tolerance affects both ramping and target speeds.

In Figure 5.1(b), the results of stepper motor, MOTOR_B, are presented. In the deceleration region of the profile, the expected tolerance behaviour is observed. However, in the acceleration region that follows, we observe that when running at 90% of the nominal speed, the acceleration begins later. Explaining this behaviour requires in-depth understanding of the control software which is beyond our scope. We also remark that since stepper motors are known to be very precise, the control software may have been adapted to this fact. Hence, such tolerance behaviour may have little or no meaning in terms of the motor itself. However, as mentioned earlier, this effect is interpreted as the cumulative tolerance from all other devices. Both motor speed profiles were generated during start-up of the finisher.

Pinch speed tolerance is presented in Figure 5.2. The results are self-explanatory and match expectations for pinch speed tolerance (Figure 3.5). We only add that the motors and pinch in Figure 5.1 and Figure 5.2 respectively are unrelated, i.e., unconnected in the simulated plant.
5.2. Test Case Results

(a) Speed profile of Motor_A

(b) Speed profile of Motor_B

Figure 5.1: Motor speed tolerance test case results

We also investigated tolerance in sensors. During the test, a print job of a single sheet was executed and the status of a sensor along the path of this sheet was checked. A position tolerance of -10mm was assigned. This translates to an early detection of the pulling edge of the sheet as evidenced by the results presented in Figure 5.3. An interesting observation is that due to the delay, the software keeps the sensor in an active state for a longer duration. Moreover, the relationship between the sensor-position tolerance and actual toggling time is not one-to-one; 10mm maps to 23.5ms of activation delay in the
5.2. Test Case Results

Validation and Verification

(a) PINCH_1 showing complete speed profile

(b) PINCH_1 zooming in on deceleration region

Figure 5.2: Pinch speed tolerance test case results

test case results depicted in Figure 5.3. This observation holds in general for sensors, as earlier stated. However, the conversion of sensor delay to displacement should not be difficult, given information about sheet speeds. Moreover, in the case of mounting tolerance, no conversion is needed.

Actuator delay tolerance results are presented in Figure 5.4. The test was performed with the same single-page print job used for testing sensor tolerance. Toggling of this actuator served as a trigger for sheet injection. Therefore, since just one sheet was printed, Actuator_A toggled only once. A delay of
50ms was set for both transitions. The expected results in Figure 3.6 assumed that an actuator is toggled at fixed *time instants*. However, the software that controls this actuator keeps it active for a fixed *duration* of 141ms (verified by inspecting the logging). Upon receiving a request to toggle, the actuator request a callback in 50ms as explained in Section 4.3.1. Thus, a total delay of of 100ms for both transitions is observed. Different activation and deactivation delay are also supported by our model.

**Figure 5.3: Sensor position tolerance test case results**

**Figure 5.4: Actuator delay test case results**
Sheet and segment tolerance verification was done by inspecting console outputs and log files since SIL does not currently provide any graphing support for the device characteristics in question. A snapshot of SIL showing the associated log messages is presented in Appendix B for the interested reader.

The results presented in this section suggest that the implementation matches our design. A further verification step involves comparing the results of the original version of SIL without tolerances modelled and the new version of SIL with no tolerances specified. Ideally, both results should be identical. We could not perform this verification using log files due to varying time stamps and the multi-threadedness of the control software. However, each test suite comes with some so-called ‘smoke test cases’. These test cases may be used for the purpose of regression testing. Their success is a necessary condition for checking unexpected behaviour resulting from changes to the SIL or embedded software source code. We ran these regression test cases successfully on our implementation.

Having gotten an indication in Section 5.1 that our design and implementation fulfill the requirements, and in Section 5.2 that our implementation does what it is expected to do, we now analyze the performance of SIL following our extension in the next section.
5.3 Performance Analysis of SIL

Considering the changes made to SIL in order to realize the proposed tolerance model, an important issue to address is its effect on the performance of SIL. We do that in this section. We measure performance in terms of three parameters: execution time, memory usage and log file size.

**Execution Time:** The execution time of a simulation run (in seconds) depends on the kind of test being performed. Since timing in SIL is simulated, the actual measured time is typically smaller than the elapsed time in SIL. This is usually an advantage as tests can be run in a fraction of the time they would normally take on the machine. Execution time also depends on the processing capabilities of the host machine. Since most of the operations for realizing tolerance involved simple multiplications, we expect no significant change in the timing performance of SIL resulting from tolerance specifications; at most a few hundred milliseconds.

**Memory Usage:** SIL requires memory to store its state (variables), method pointers, et cetera. This is applicable to both data classes and capsules. Furthermore, there is an additional overhead arising from the functionality provided by the RoseRT, e.g. for realizing communication between capsules and state machine behaviour. Again, in our design and implementation, we avoided reinventing the wheel as much as possible as well as unnecessary code duplicity. Most of the tolerance execution mechanism was implemented using static (class) methods as explained in Section 4.2. Therefore, we except at most an increase in memory usage in the order of tens of kilobytes.

**Log File Size:** The main logging output of SIL is a text file called silentLog.txt. It contains all the information written to the console and a lot more ranging from device incarnation, state changes, time advancements to warning and error information. This log file is very useful for debugging in the event of unexpected behaviour. Given the need for reproducibility, effort was made to log as much information as concisely as possible to allow for easy tracking of tolerance execution. Sample outputs following tolerance specifications are presented in Appendix B. Generally, tolerance specifications are expected to be performed at the beginning of a simulation run (though run-time tolerance specification is possible via the provided interface methods). Each tolerance execution operation is typically accompanied by one to three lines of logging (excluding warnings and errors, i.e. assuming correct specifications). We therefore expect a fairly constant increase in log file size in the order of tens of kilobytes, independent of the test cases.

5.3.1 Test Setup and Procedure

Two kinds of test cases were used for the performance tests. The first one consisted of a standard printing job in which the printing of 10 single-sided A4-sized sheets was simulated. In the second test case, an ORE was simulated while printing 10 duplex A3-sized sheets. The ORE simulated was the occurrence and subsequent removal of a paper jam. We refer to these tests as simple print and duplex-ORE, respectively. The test cases were obtained from a test engineer and belong to the suite of regression test cases. For each test cases, we considered three scenarios: (i) SIL without tolerance; (ii) SIL with tolerance specified in the test case XML file; and (iii) SIL with tolerance injected via TE using the provided interface methods. Tolerance ranges of all devices defined in the layout file were assigned as follows:

- **Motors:** [0.999, 1.001], i.e. speed tolerance of 0.1%
- **Pinches:** [0.99, 1.01], i.e. speed tolerance of 1% (or equivalently diameter variation of 1%)
- **Sensors:** [-0.000001, 0.000001], i.e. position tolerance of 1µm
Actuators: [0.000001, 0.000001], i.e. delay of 1µs
Segments: [0.99, 1.01], i.e. 1% variation in segment length
Sheets: [0.99, 1.01], i.e. 1% variation in sheet length

Since no bent segments were defined in the layout file, bend tolerances were not specified. For each of the devices listed above, a random tolerance value was selected from its tolerance range.

The performance monitor available in Windows XP (\texttt{perfmon.msc}) was used to log execution time and private memory data at a sample rate of two seconds. The simulated time and log file size were equally logged via SIL. Microsoft defines this private memory (bytes)\textsuperscript{1} as “the memory that [a] process has allocated that cannot be shared with other processes”. Therefore, we believe that this is a fair performance indicator of the memory usage of SIL.

Each test case was repeated ten times in order to average out any discrepancies. Therefore, in total, sixty runs were performed, ten times each per test case under the three scenarios mentioned above.

All tests were run using \texttt{TE} on a Windows PC running Microsoft Windows XP Professional Service Pack 3 (Version 2002) on an Intel(R) Core(TM)2 CPU clocked at 1.86 GHz and having 0.99GB of RAM.

5.3.2 Results

Having described the performance criteria and test setup, this section presents the results obtained from the performance tests.

5.3.2.1 Execution Time

Figure 5.5 shows the execution times for each of the test runs of both print jobs as well as the average execution time. The results show that independent of the test case, the average difference between the execution times for SIL without tolerance and SIL with tolerance specified in the test case file is negligible. It is in the range of 100ms as predicted. The situation with tolerance injection via interface methods is more interesting. We observe an overhead of about 7 seconds when using \texttt{TE} for tolerance injection. During preliminary implementation tests, we uncovered an unusual situation in which some commands were lost or truncated when several commands were sent back-to-back from \texttt{TE} to SIL. This problem was reported to the SIL maintainers and is most likely due to a wrong socket implementation. The same problem had been detected and fixed for the socket interface between SIL and \texttt{SILVis} during some other project. As a temporary solution, we included one-second delays between consecutive tolerance command. This therefore accounts for the overhead of about 7 seconds since, in total, 7 interface commands were sent in either test case. When this error is fixed in SIL, the delays will be removed. Under this assumption, we can conclude that there is no significant overhead in the execution time performance of SIL resulting from the tolerance model. Furthermore, we observe that for the first test run, SIL without tolerance has a longer time than SIL with tolerance. Although this suggests a surprising improvement in performance due to the tolerance model, it is not the case. All tests were executed using \texttt{TE}. In some cases, there was a delay in terminating SIL after completing the test. Since we measured the total execution duration of the SIL process, these delays were invariably taken into account. Hence, we employed averaging over multiple (in this case 10) runs to even out such disturbances.

5.3.2.2 Memory Usage

Figure 5.6 shows the performance results in terms of memory usage. Most of the (statically allocated) memory is requested during start-up. Given our sample rate of two seconds, that phase was not logged.\textsuperscript{1}

\textsuperscript{1}Private memory bytes in Windows XP is renamed private working set in Windows Vista.
5.3. Performance Analysis of SIL

(a) Simple print

(b) Duplex-ORE

Figure 5.5: Comparison of execution times of SIL for different test cases

However, this fact is not important for the purpose of the present analysis as we are interested in the average and peak memory demand. For the same reason, we did not investigate the differences between the memory usage patterns for simple print and duplex-ORE. Figure 5.6 also shows that the transitions from low to high memory usage for both simple print and duplex-ORE differ for the three scenarios considered. Nevertheless, the average time difference between their respective transitions is of the same order of magnitude as their differences in execution times. Another observation is that the average execution time of the simple print is about half that of the duplex-ORE. This justifies our choice of them as examples of simple and complex test cases, respectively.

We conclude that the results show no significant difference in the average memory demands of SIL with or without tolerance. Again, this matches our predictions of a change in the order of at most a few kilobytes. We mention that only one of the ten test runs per scenario is shown in Figure 5.6, namely the one with the most number of memory usage samples. An average over the ten test cases would smoothen out the gaps during transitions in the figure. However, finding such an average was limited by the fact...
that the exact time instants at which samples were taken for each test run were generally different.

5.3.2.3 Log File Size

Figure 5.7 shows the results for log file size performance analysis of SIL. Just like the execution time doubles on the average for a duplex-ORE print job compared to a simple print (cf. Figure 5.5), the amount of logging approximately doubles as well. An interesting observation, however, is that whereas the log file size for SIL without tolerance remains fairly constant, this is not the case for the other two scenarios. Furthermore, the difference between the log files sizes is not fixed as would be expected if it is only due to tolerance-related logging. The explanation for this behaviour lies in the fact that owing to tolerances, system behaviour changes. For instance, when sheet arrival at a sensor is delayed, additional logging is produced both from the clock which reports the passage of time and from the embedded software itself, for instance, to report the delay as long as it is within the permitted range. Beyond this threshold, an error may be triggered as was the case with the failed tests (see the case study in Section 6.1.3 for more details). Therefore, the presence of tolerance leads to more logging from embedded software as well.
5.3. Performance Analysis of SIL

 Nonetheless, the average increase in log file size for both scenarios was only about 5%\(^2\).

To verify the claim above, we performed new tests in which tolerance was injected but no print jobs were given. The results in Figure 5.8 show that, as predicted, the size of the logging due to tolerance specifications is fairly fixed for all test runs. The few byte differences between successive test runs of SIL with tolerance are a consequence of different randomly generated tolerance values, e.g. “100.1” and “99.9” differ by one character. Furthermore, the difference between the logging resulting from XML and TE tolerance specifications accounts for the additional messages associated with interface commands such as the verification strings which may be used by TE in the future for postmortem analysis during automatic (regression) testing.

\(^2\)The reader’s attention is drawn to the fact that the vertical axis in Figure 5.7 does not start from zero (this remark is also applicable to other figures, e.g. Figure 5.8).
5.4 Chapter Summary

In this chapter, we validated the design and implementation against the project requirements with motivation for undone/uncompleted ones. We also used test cases to verify the implementation. The results showed that our tolerance model produced the expected behaviour for the cases investigated. These two observations (i.e. validation and verification) suggest that we have done the right thing correctly. However, they do not serve as proof, since the absence of known bugs does not guarantee their non-existence. An analysis of the effects of the tolerance model on SIL’s performance showed no significant degradation in terms of execution time, memory usage and log file size. These results indicate that we met our objective of an elegant solution at the right level of abstraction. The next (and final) chapter focuses on a general summary of the project and possible directions for future work.

Figure 5.8: Comparison of SIL log file sizes for test cases without print jobs
Chapter 6

Conclusion

This concluding chapter begins in Section 6.2 with three case studies which illustrate the practical use of the tolerance model developed within the company in Section 6.1. Next, we present a general reflection of the project. This includes the main challenges encountered, good practices which facilitated successful completion of the work, and more importantly a retrospective look at some major design decisions. Section 6.3 is a summary of the work done during the course of the project. Lastly, we highlight possible directions for future work are highlighted in Section 6.4.

Recommendations for Initial Reading: Sections 6.1, 6.2.3 and 6.3.

6.1 Case Studies

In Section 3.6 of Chapter 3, we presented test cases that verified the implementation. However, modelling variability in SIL was only the means to an end, namely to be able to perform more extensive robustness testing on the embedded control software. In this section, we consider three case studies in which (aspects of) the tolerance model were used during testing. In addition to being an extra validation and verification step, these case studies also provide some feedback on the usability of our tolerance model.

6.1.1 Testing a Flipping Wheel Error Routine

At the time of this project, a new finisher module was being developed for a new product. In developing such a system, a considerable amount of effort goes into error detection and correction routines (it is bad enough that errors occur but even worse when the exact problem cannot be deciphered). The finisher has a flipping wheel which is used to flips sheets when required. In order to perform this operation, consecutive sheets must not be closer to each other than a pre-defined minimum inter-sheet distance. A routine to detect and report a violation of this ORE was developed. However, testing this routine on the real machine was infeasible because earlier in the finisher’s paper path trajectory, the distance between sheets is adjusted to be larger than the minimum inter-sheet distance. From this point until the flipping wheel, inter-sheet distance hardly ever drops below the threshold except in the very unlike yet possible error situation being investigated. Therefore, creating a scenario that triggers the error on the real machine is almost impossible.

Using the finite sheet delay functionality added to SIL (see Section 4.3.1), the situation was easily simulated and the routine successfully tested. With help from the software engineer who wrote the error routine (e.g. to identify the point of interest and inter-sheet delay thresholds), the actual test lasted
6.1. Case Studies

Conclusion

for less than a quarter of an hour. This example underscores one of the advantages of SIL simulation highlighted in Section 1.4.2, namely that it can be used to perform tests that are very difficult, time consuming or even impossible on a real machine.

6.1.2 Verification of a Skewness Detection Error Routine

The second case study conducted focused on verifying an error routine included in the skewness correction algorithm. Skewness is typically detected using a pair of sensor placed next to each other, as explained in Section 5.1.1. The maximum amount of skewness which can be handled (i.e. corrected) by the algorithm is known in advance. When the skewness of a sheet exceeds this threshold, an error is triggered to report too great skewness. To verify that this error routine works correctly, we specified an appropriate tolerance range for one of the skewness detection sensors. Next, we injected a tolerance value which should trigger this error. Since our sensor tolerance model expresses tolerance in terms of sensor position, no conversion was necessary. Effectively, we simulated skewness of a sheet by displacing one of the detection sensors relative to the other. We ran a test and the error was triggered as expected. Again this is an instance of a test that would normally be very difficult to perform on the real machine but could easily be done using SIL. Moreover, we remark that, in the past, this test was successfully performed using SIL without tolerances. The idea was still to displace the skewness detection sensors but it involved changing the nominal position of the sensor each time. There was also no possibility to simulate a more realistic situation where the level of skewness for individual sheets varies. With our tolerance model for sensors in SIL, performing this test is easier and possible for the realistic scenario mentioned above.

6.1.3 Debugging a “False Alarm”

We conducted another case study to investigate the general robustness of control software. While generating data for the performance tests, tolerance ranges were defined for several devices as explained in Section 5.3. In total, tolerances were injected in 9 stepper motors, 3 stubbed motors, 30 pinches, 19 segments, 24 sensor instances, 40 actuators of different kinds, and in all printed sheets. Although quite arbitrary, we verified that the tolerance ranges specified for these devices (see Section 5.3.1) were actually realistic and for some devices even smaller than typical ranges. Therefore, we expected the results of these tests were expected to give a good initial impression of control software’s robustness.

During both standard print and duplex-ORE test cases, a number of test runs failed. The error indicated that (a) sheet(s) had arrived late at particular sensors. We chose one of the failed standard print tests for this case study: The first step was to reproduce the test several times and verify that the same error was always reported. We did this by retrieving the seed of the failed test from the SIL log file, silentLog.txt. The test was re-run ten times with the same error. Furthermore, a comparison of log files (both SIL logging and embedded software dprintf logging) further confirmed that the situation was indeed accurately reproduced in each re-run. However, we mention here that owing to the multi-threadedness of the embedded software, log files from different runs are not guaranteed to be identical. This remark also holds for SIL without tolerances. Thus, we do not speak of matching log files in terms of one-to-one correspondence of text output but in terms of equivalence of behaviour.

The next step was to debug the error. However, in order to do that, familiarity with the control software and tasks involved in the printing job were required. We sought the assistance of a domain expert who quickly inspected the log files and discovered the error was not due to unrobust software (good news for him). Instead, it was a result of a subtle oversight in our tolerance model. We allowed segment length to change albeit only at the beginning of a test run, i.e. before any sheet is injected. However, in the
scenario that caused the error, both segment and sensor were created during initialization. The sensor was subsequently placed 3mm from the segment’s nominal exit position (which is acceptable). Next, the segment got shortened by more than 3mm following injection of a random tolerance from its length tolerance range. Therefore, the sensor became ‘orphaned’ and was unable to sense any sheets, thereby triggering the error. In other words, it was a “false alarm”. Of course, an implicit precondition is that all tolerance ranges should be specified such that there are no possibilities for such conflicting conditions. Moreover, as already mentioned, this is a subtle situation which we did not check in our implementation.

Following this discovery, we included an explicit check for any orphaned elements in the implementation. This check is performed not just for sensors but for all ‘real’ devices placeable on a segment as well as ‘virtual’ points-of-interest and sheet blocks. Whenever a segment length changes due to tolerance, this check is performed and the detection of any orphaned element causes SIL to terminate with an error indicating which element(s) got orphaned. In this way, we improved the robustness of SIL to wrong specifications, although we again emphasize that enforcing pre-conditions is the responsibility of the user (see remarks at the end of Section 3.5). After updating the implementation, we repeated the same test case 40 times without recording any errors.

These case studies provided us with useful insights on important aspects of our tolerance model. First, we saw that reproducibility of test cases using the stored seed works as expected. Secondly, we assessed the time and steps needed to get SIL with tolerances running on a new workstation. Thirdly, we gladly observed that enough information was logged to enable quick debugging of an error, albeit a false alarm. This information was also useful in preparing a user’s manual. Lastly, we were able to improve our implementation to guard against future occurrences of the same situation.

In addition, they show that SIL with variability creates opportunities for new test cases which will ultimately result in better software. We could not investigate more specific test cases as they require a sound knowledge of the paper path and embedded software. Therefore, the long-term benefits of this work can only be projected but will largely depend on how fast the extended framework is made available to the engineers and how well they adopt it.

6.2 Project Reflection

This section provides a post-completion reflection on the project. It outlines the major challenges as well as helpful practices. It also includes a reflection on some design/implementation choices with suggestions for other feasible alternatives discovered in retrospect.

6.2.1 Challenges

The major challenges encountered during this project are as follows:

- **Inadequate documentation of SIL**: Important details about some features of SIL such as how timing works were not documented. This problem necessitated a lot of reverse engineering to understand the basics of SIL which was time consuming and further limited by the fact that the source code itself was also poorly documented. On the other hand, reverse engineering the code led to discovering and fixing some bugs such as the default graphs crash mentioned in Section 4.4.2, the specified yet unimplemented silent mode (Section 4.4.1) and others which are inline documented within the code deliverable of this project. Learning from this mistake, extensive inline documentation, an internal document summarizing major design choices and their rationale including a user’s manual, feedback on the existing SIL design document, updates of internal wiki pages, and...
an additional section in Chapter 4 on how timing works in SIL are some of the steps we took to improve the situation.

- **Rapidly evolving SIL software**: Within the six-month duration of this project, several changes were made to SIL. This was especially because a team was concurrently working on extending SIL to the company’s continuous-feed printer series where it was hitherto unused. In some cases, testing of our intermediate implementations was not possible without a (non-trivial) merge step since embedded software was also evolving. The merger between Océ and Canon equally implied changes in some modules. In other instances, the tolerance model implementation needed to be extended, e.g. to steered motors (Section 3.3.1.7) which were added to SIL after the motor tolerance model had already been implemented. Most importantly, an extensive merge with the latest SIL version became imperative for the performance analysis described in Section 5.3 since we did not consider an analysis using an outdated baseline to be a sound choice. For the latter, the experience gained during earlier merge steps became very handy.

- **Lack of experience with C++ and UML**: This was a perceived risk at the start of the project which turned out not to pose any serious challenges. Previously having completed only one term project in C++ about four years ago and lacking a knowledge of UML, we foresaw a possible hitch in the implementation phase. To address this challenge, online tutorials and reference books such as [8] were used. Sample code from the existing SIL source also proved helpful. Given the fact that most of the implementation focused on extending an existing framework, this risk did not hamper implementation.

### 6.2.2 Helpful Practices

Learning from previous experience and based on advise from others, the following practices were very helpful:

- **Version control**: Using TortoiseSVN\(^1\) for version control of the implementation enabled roll backs to earlier versions and provided an efficient means of managing the work.

- **Intermediate documentation**: Short reports of preliminary design and implementation steps enabled early feedback; keeping a log of meetings and design choices ensured that steps were easily trackable. These documents also facilitated the production of this final report.

- **Writing test cases during the design phase**: The benefit of this practice became obvious when an understanding gap in the behaviour and differences between stepper and controlled motors was detected by a reviewer based on the expected test case results. Furthermore, they provided a more objective basis for verifying the implementation.

- **Schedules and milestones**: Although rather obvious, making a clear schedule with dates and milestones and keeping track of progress contributed to a timely completion of the project. Weekly progress meeting were also very helpful in this respect. On a personal note, outlining daily tasks helped me check my tendency to overwork.

### 6.2.3 Design/Implementation Choices in Retrospect

In Chapters 3 and 4, several design and/or implementation choices were mentioned with motivation. In this section, we consider a few alternative approaches. In other words, we address the question: “What could we have done differently if the project were to be repeated?”.

\(^1\)http://tortoisesvn.net/
As already stated in Section 3.3, a preliminary design and implementation step was a major factor that contributed to the success of this project. The motivation for considering tolerance in motor speed first was also stated. Given the different kinds of motors, we decided that the tolerance model should be included in the devices themselves and not in the interface to the connected pinches. One of the major considerations for this was to ensure that tolerance was captured in the motor speed profiles (see Section 3.3.1.9). However, considering each motor separately, understanding its behaviour and modelling its tolerance was not trivial. An alternative could have been to consider tolerance in pinch speeds instead since as observed, motor speed is linearly translated via pinches to the sheets. The advantage would lie in the considerably simpler pinch model (there is only one kind of pinch). Furthermore, pinch speed graphs could then be used for debugging. Another possibility was to express tolerance in the motor-pinch interface but additionally, to separately record ideal speeds and speeds with tolerance included. In this way, plots of both ideal and ‘real’ situations could still be generated.

Another design issue relates to the choice to separate the interfaces for tolerance injection and disabling using `<injectTolerance>` and `<disableTolerance>` for XML (Section 3.4.2), and `injectTolerance[All](..)` and `disableTolerance[All](..)` for the interface methods (Section 3.5), respectively. As mentioned in Section 4.2, we abstracted from this difference in the actual implementation where the same underlying `setTolerance(..)` method was used for both injecting and disabling tolerances. Parameter `type` was set to “OFF” in order to realize the latter (though this option was not exposed to the external world). Arguably, a single `<setTolerance>` node in XML and `setTolerance[All](..)` method(s) in the provided interface would therefore suffice. The consideration for not adopting this approach was mainly a need to separate concerns and allow for more readable specifications. However, the advantage of the alternative approach, if adopted, would have been a reduction in the number of methods used from four to two and possibly a shorter learning curve for the stakeholders.
6.3 Project Summary

In this thesis, we presented the work of a six-month master’s project conducted within the research and development department of Océ-Technologies B.V. in Venlo, The Netherlands. The goal of this project was to develop a tolerance model that captures system variability. This model extends an existing in-house software-in-the-loop (SIL) simulation framework which is used for development and robustness testing of embedded control software in the absence of the real printer hardware. In other words, SIL mimics the hardware components that constitute a printer. An appropriate level of abstraction was needed, which would provide an adequate yet concise model that can easily be understood and used. Failure to meet this implicit requirement could significantly hamper the performance of SIL, lead to “false alarms” regarding software robustness and/or pose usability issues resulting from model complexity. After careful consideration, we decided to model tolerance in all main components of the simulated plant (or “sheet logic devices” as they are called in SIL terminology), viz.: motors, pinches, sensors, actuators, segments and sheets. For each one of these devices, we specified a single tolerance parameter which abstracts from the underlying sources of tolerance but represents the cumulative variability due to the device. For motors and pinches, we considered speed tolerances; for actuators and sensors, we expressed tolerances as response-time delays and displacement from a nominal position, respectively; we captured length tolerance for sheets and segments as static variability factors, relative to their nominal values. Furthermore, we modelled sheet behaviour in bends, which is known in practice to affect sheet transport within a paper path.

Based on the requirements from the different (internal) stakeholders of the project, we specified and implemented interfaces for tolerance specification using an existing XML test case file and via SIL’s provided interface to external components like SILVis (for visualization) and TE (for automated regression testing). The complete implementation was done using RoseRT, the standard CASE tool for software development within Océ, Venlo. We conducted test cases via both interfaces and used the results to verify correct implementation. In addition, we validated both design and implementation against the project requirements. Together, these validation and verification steps suggested that we did the right thing in the right way (without any unwanted side effects). SIL extended with tolerances remains backward compatible and all existing regression test cases yielded the same behaviour as before the extension. While the results obtained are an indication of correctness, we do not assert foolproofness since the absence of known faults is not proof of their non-existence. However, a formal verification is beyond the scope of this work.

Tolerance modelling was only a means to improve robustness testing of embedded control software. Hence, we conducted case studies to investigate the added value of SIL extended with variability. The results showed that even such a preliminary investigation as undertaken illustrated the benefits of this endeavour. In particular, the tolerance model provided a means to specify scenarios and verify error routines that were hitherto difficult or impossible using SIL and/or on prototype machines. We expect more interesting results as the extended framework is adopted by more developers, integrators and test engineers. Moreover, our investigation of the performance penalty resulting from including a tolerance model in SIL revealed no significant losses in terms of execution time, memory usage and log file size. This result further consolidated our claim that we indeed chose the right level of abstraction.

In addition to extensively documenting the work done, this document also serves as a reference for future tolerance modelling and maintenance in SIL. It will also serve as a useful resource for understanding the internal workings of SIL as important mechanisms such as timing, handling of point-of-interest commands and the callback functionality provided by the clock are explained with the aid of UML sequence diagrams.

Finally, the possible directions for future work outlined in the next section provide a good indication of further steps towards an even better SIL simulation framework. To this end, the project reflection which highlights challenges, alternative choices and helpful practices may provide some valuable insights.
6.4 Future Work

In this final section, we indicate possible directions for future work. Given their diversity, we group them in subsections as follows.

6.4.1 More Input Probability Distributions and Multiple Seeds

In Section 3.2.4, a discrete uniform input probability distribution was chosen for generating random tolerances from the tolerance ranges of different devices. For a start and in cases where the input probability is not known, this suffices. However, means to determine the input probability distribution given some input data exist [13]. In particular, when the input probability distribution is known, the randomness in the tolerance model for that device can be fine-tuned. Given the availability of all hardware components, we consider determining the input probability distributions for various sheet logic devices and implementing or importing libraries that generate random numbers according to these distributions as a possible direction of future work. In a related vein, using multiple seeds to decouple the tolerance values generated for different devices may be investigated and implemented.

6.4.2 Error Status Levels in SIL

We highlighted the benefit of returning status values upon method invocation when specifying the interface methods for tolerance injection in Section 3.5. Whenever possible, these boolean status values were used to check successful execution of the action embodied in a method. Nevertheless, in the event of a failure, the exact cause could sometimes only be determined by inspecting the SIL logging. An improvement on this situation can be to define several status values for instance, using integers instead of booleans. Unifying these statuses across SIL has the advantage of a common understanding of these well-defined status levels similar to the log levels currently in use.

6.4.3 Visualization of Variability using SILVis

Run-time visualization of (some of) the tolerance effects modelled is possible. For instance, sensor displacement per sheet/edge and range-of-interest, static sheet and segment length tolerances, sheet behaviour in bent segments, and tolerance in pinch diameter are some of the aspects of the tolerance model which can be incorporated in SILVis in the future. Similarly, additional graphs such as sensor displacement in time due to tolerance may also be added to SIL.

6.4.4 Statistical Analysis of SIL Extended with Tolerances

Although we have provided a means to specify variability in a simulated machine, we have not performed any statistical analysis. For example, given a simulated plant with tolerance specified for one or more of its devices such that a combination of inputs which produces an error is possible, we may like to know how many times a test case must be repeated before this error is unmasked. Such analysis is beyond the scope of the present work but can be addressed in the future.

6.4.5 Modelling Skewness

Finally, skewness detection and correction was not modelled for reasons outlined in Section 5.1.1. In the future, some of these constraints may be lifted, e.g. an extension of the current stepper motor model in SIL. Consequently, skewness can be modelled as it remains an important phenomenon and will definitely add value to SIL.
References

The page numbers where the references occur (i.e. back references) have been included at the end of each entry.


Appendix A

Tolerance Specification Examples

A.1 Specifying Tolerance Ranges

In Section 3.4.1, the interface for specification of tolerance ranges in XML was described. This appendix contains a sample specification developed for the purpose of illustration.

Listing A.1 shows how to specify the tolerance range for various devices. All other attributes except the tolerance ranges are from the original layout file used with SIL without tolerance modelled.

Listing A.1: Specifying tolerance ranges in layout.xml

```xml
<node>
  ...
  /* PINCHES */
  <pinches>
    <pinch id="0" name="PINCH_A" force="4.000000"
      minTolerance="0.99" maxTolerance="1.01" />
    <pinch id="1" name="PINCH_B" force="5.000000"
      minTolerance="0.8" maxTolerance="1.25" />
    <pinch id="2" name="PINCH_C" force="5.000000"
      minTolerance="0.99" maxTolerance="1.25" />
  </pinches>

  /* MOTORS */
  <motors>
    <simulatedMotors>
      <motor name="MOTOR_A" stepDistance="0.000204204" encoderStepDistance="0.66"
        minTolerance="0.9" maxTolerance="1.1" />
      <motor name="MOTOR_B" stepDistance="1" encoderStepDistance="1"
        minTolerance="0.9" maxTolerance="1.1" />
    </simulatedMotors>

    <controlledMotors>
      <motor name="MOTOR_C" inputType="position" inputSensor="SENSOR_C"
        outputActuator="ACTUATOR_A" maxSpeed="1" nrNmPerSlit="1000"
        minTolerance="0.9" maxTolerance="1.1" />
    </controlledMotors>
  </motors>
</node>
```
A few remarks about the specifications in Listing A.1:

- [line 15] Simulated motors are the stepper motors described in Section 3.3.1.3.
- [line 22] MOTOR_C is a controlled motor. Hence, it has an associated sensor/actuator pair used to model the encoder in its feedback loop.
- [line 32] Segment 2 is a bent segment since minLength and maxLength are defined. It could also be a buffer area though in which case special buffer area elements must also be defined.
- [line 41] SWITCH_B is a segment switch with different activation and deactivation tolerance ranges.
- [line 46] CLUTCH_A still uses deprecated delay attribute. Range is initialized to [delay, delay] with warning of new attributes (backward compatibility).
- [line 52] Tolerance for sensor is not in <sensors> but <sensorsAtSegments> because a sensor does not have a position until placed on a segment (i.e. sensor instance; see Section 3.3.4).
- [line 53] Notice that minTolerance takes a negative value because sensor tolerance is expressed as a displacement relative to a nominal position (see Section 3.3.4).
- [lines 55, 57] SENSOR_B has two different instances on segments 1 and 2 respectively. This is allowed in SIL.
A.2 Specifying Tolerance Values for Test Cases

In Section 3.4.2, the interface for XML specification of tolerance values in test cases was described. This appendix contains a sample specification developed for the purpose of illustration.

Listing A.2 shows examples of tolerance specifications in a test case file.

```
Listing A.2: Specifying tolerance values in SILSimulation.xml

    <SILSimulationSettings>
      <testCase id="100" duration = "9800.0">
        <injectTolerance>
          <segments>
            <bentSegment target="ONE" id="2" type="PER_SHEET"/>
            <segment target="ALL" type="MIN"/>
          </segments>
          <pinches>
            <pinch target="ONE" name="PINCH_A" type="INPUT" value="0.90"/>
          </pinches>
          <motors>
            <motor target="ALL" type="AVERAGE"/>
            <motor name="MOTOR_A" value="0.99"/>
          </motors>
          <actuators>
            <actuator name="CLUTCH_A" type="MAX"/>
            <actuator name="SWITCH_A" value="0.07" transition="ACTIVATION"/>
            <actuator name="SWITCH_B" type="MIN"/>
          </actuators>
          <sensors>
            <sensor target="ALL" type="MIN"/>
            <sensor name="SENSOR_B" type="PER_EDGE" segmentId="2" nominalPosition="0.1"/>
          </sensors>
          <sheets target="ALL" type="AVERAGE" minTolerance="0.95" maxTolerance="1.05"/>
        </injectTolerance>
        <disableTolerance seed="1024">
          <segments>
            <bentSegment id="2"/>
            <segment id="2"/>
          </segments>
          <motors>
            <motor name="MOTOR_C"/>
          </motors>
          <actuators>
            <actuator name="ACTUATOR_A" transition="DEACTIVATION"/>
          </actuators>
          <sensors>
            <sensor name="SENSOR_B"/>
          </sensors>
        </disableTolerance>
      </testCase>
    </SILSimulationSettings>
```
Some remarks about the specifications in Listing A.2:

- [line 4] Zero or more `<injectTolerance>` (and/or `<disableTolerance>`) can be specified in the same test case but only the first one is executed by SIL. However, this is not recommended and XMLValidator does not validate a test case file with multiple tolerance specifications.

- [line 6] `bentSegment` is used to distinguish tolerance in a sheet’s path in a bend (Section 3.3.5.2) from the tolerance in the actual length of the segment (Section 3.3.5.1).

- [line 6] Segment 2 is a bent segment. `PER_SHEET` means that for each sheet entering this segment, a random length is generated which is the sheet’s perceived length of the segment and used by it throughout its transport on that segment. It could be take the inner bend (minLength), outer bend (maxLength) or anything between. Note that the length, min- and maxLength of segment 2 are also affected by the specification in line 7!

- [line 7] Sets the tolerance in segment length (this is a static tolerance, e.g. due to manufacturing and/or coupling and should be done once before sheets are injected).

- [line 10] Specified tolerance is out-of-range and therefore clipped to minTolerance=0.99 with a warning (Table 3.4).

- [lines 13, 14] Special “all-except” usage (Section 3.4.2.5). Results in a tolerance of 0.99 in MOTOR_A and average tolerance in all other motors. See also lines 21 and 22.

- [line 16] Actuators are not nested according to their specific names, e.g. segmentSwitch, clutch, etc. They are all simply actuators! This observation also holds for motors and sensors.

- [line 18] By setting the optional parameter transition to "ACTIVATION", this specification models a switch with negligible deactivation delay. By default, actuator delay tolerance affects both transitions.

- [line 19] Since SWITCH_B has different activation and deactivation delay ranges (see Appendix A.1), this specification takes the minimum from the respective ranges namely: 0.5 for activation and 0.1 for deactivation.

- [line 23] This specification affects only the particular instance of SENSOR_B at the stated (nominal) position of segment 2 (recall that a single sensor can be at multiple places). `PER_EDGE` means that the sensor may detect the leading and trailing edges of the same sheet at different positions (this option is unique to sensors; see Section 3.3.4).

- [line 25] Sheets are not created during start up but by embedded software and they are not machine-specific. Hence, their tolerance range is specified here along with injection specifications (Section 3.3.6.3). Specification is always for all (subsequently generated) sheets. Thus, `<sheets>` has no child nodes. Also, attribute target is fixed at “ALL” and may be omitted without ambiguity.

- [line 28] The seed for test case reproduction is an attribute of both `<injectTolerance>` and `<disableTolerance>`. The first one encountered is used, if specified in both.

- [line 37] Only the deactivation delay is disabled using the optional transition attribute which is valid for only actuators.

- [line 40] This specification disables tolerance in all instances of sensor SENSOR_B, (on segments 1 and 2).
A.3 Specifying Tolerances via the Interface Methods

In Section 3.5, the interface methods for tolerance specification were described. The following are the tolerance specifications corresponding to the XML specifications in Listing A.2.

Listing A.3: Specifying tolerance values via SIL’s provided interface methods

```
/* INJECTION */
initializeSeed(1024); /* for new (random) seed, use initializeSeed(-1)*/
injectTolerance(0, 2, "PER_SHEET");
injectToleranceAll(1, "MIN");
injectTolerance(2, "PINCH_A", "INPUT", 0.90);
injectToleranceAll(3, "AVERAGE");
injectTolerance(3, "MOTOR_A", "INPUT", 0.99);
injectTolerance(4, "CLUTCH_A", "MAX");
injectTolerance(4, "SWITCH_A", "INPUT", 0.07);
injectTolerance(4, "SWITCH_B", "MIN");
injectToleranceAll(5, "MIN");
injectTolerance(5, "SENSOR_B", "PER_EDGE", -1.0, 2, 0.005001);
changeSheetLength(.., 0.95, 1.05);
injectToleranceAll(6, "AVERAGE");
/* DISABLING */
disableTolerance(0, 2);
disableTolerance(1, 2);
disableTolerance(3, "MOTOR_C");
disableTolerance(4, "CLUTCH_A", "DEACTIVATION");
disableTolerance(5, "SENSOR_B");
```

Some remarks about the specifications in Listing A.3:

- [line 1] Seed initialization (if necessary) must be done first before any other tolerance specification. When not reproducing a test case, it is still possible to generate a random seed (e.g. to overwrite one specified in the test case file when not reproducing a previous test case) using initializeSeed(-1).

- [line 12] In order to specify optional (sensor) attributes `segmentId` and `nominalPosition`, a sentinel for optional attribute `input`, i.e. -1.0, must be provided.

- [line 13] Before injection tolerance in sheets (line 14), it is necessary to first set the tolerance range. This is done using method `changeSheetLength(.., minTolerance, maxTolerance)` as described in Section 3.3.6.

- [line 17] This specification disables sheet behaviour in bent segments, i.e. all sheets use the same (fixed) segment length.

- [line 18] This restores all segments (including bent segments) to their nominal lengths.
Appendix B

Additional Test Case Results

In this appendix, we provide a snapshot of SIL to give an impression of the kinds of log messages that result from tolerance specification. It also serves as a means to verify sheet and (bent) segment tolerance specifications for which no graphing options currently exist (see Section 5.2).

Figure B.1 shows the console output of a dummy test run in interactive mode (Section 4.4.1); hence, the “Press any key to continue...” messages due to additional pauses. The first observation is that these tolerance specifications (made in SILSimulation.xml) are read during the initialization of the sheet logic capsule, i.e. before the clock starts ticking. No tolerance range was specified for PINCH_2 in the layout file. Thus, a message is issued, reminding the user that it will behave as an ideal pinch. This is enforced in the implementation by (re)setting its tolerance to OFF which effectively sets the speed to its nominal value.

Since a seed was specified in the test case file, it was used to initialize the random number generator. This is not required unless when reproducing a previous test case. Furthermore, we enforce that the seed is reset only once per test run using the interface method, initializeSeed(...). This is done for efficiency reasons as each time the seed is (re-)initialized, the random number generator is reset, XML tolerance specifications are read again and executed. This costs time and produces unnecessary logging.

The sensor tolerance logging indicates that SENSOR_A has two instances located on segments 1 and 2 respectively. Furthermore, their tolerance is set PER_SHEET, i.e. a new sensor position is set for the pulling edge of each sheet. This differs from that of SENSOR_B which is set PER_EDGE, i.e. sensor position is toggled both for pulling and pulled edges of a sheet.

Segment 2 is defined as a bent segment, i.e. by specifying a minLength and a maxLength in the layout file without any corresponding buffer area sensors. Therefore, in addition to the segment length tolerance injected in all segments, a fixed bend length to be perceived by all sheets is specified using type="INPUT" and a value relative to the nominal bend length and corresponding to 0.412928 meters. A random sheet tolerance is generated from the specified range.

Using the optional transition attribute for actuators, the deactivation delay of ACTUATOR_A is set to zero after bidirectional delay was injected for all actuators. Similarly, the tolerance in one of the two instances of SENSOR_A is turned off using the optional segmentId and nominalPosition attributes for sensors. One final yet very important remark is that all floating point numbers are displayed with the default precision of six decimal places. This is sufficient for realistic tolerance ranges of all the devices whose tolerances are modelled, the lowest being sensor tolerance which may be a few microseconds. Any smaller specification e.g. 100nm will be logged as 0.000000 and erroneously be read as a zero since it falls out-of-range.
MODELLING VARIABILITY
IN A SIMULATED PRINTER

Eindhoven University of Technology

Figure B.1: Snapshot of SIL showing tolerance logging