KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Contract-Guided Architecture Synthesis for Mechatronic Systems

César Augusto Ribeiro dos Santos

Supervisors:
Prof. dr. ir. T. Schrijvers
Dr. ir. M. Nicolai

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

March 2025

Contract-Guided Architecture Synthesis for Mechatronic Systems

César Augusto RIBEIRO DOS SANTOS

Examination committee:
Prof. dr. ir. E. Van Lil, chair
Prof. dr. ir. T. Schrijvers, supervisor
Dr. ir. M. Nicolai, supervisor
Prof. dr. ir. B. Jacobs, secretary
Prof. dr. M. Deneckers
Prof. dr. H. Vangheluwe
(University of Antwerp)
Prof. dr. dr. h. c. E. Ábrahám
(RWTH Aachen University)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

March 2025

© 2025 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, César Augusto Ribeiro dos Santos, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

I became enamored with computers very early on in life, all the way back in the 90s when my father bought our first computer: an Intel 486-powered computer running MS-DOS. It may be a bit unorthodox to thank a machine in the acknowledgements of a PhD thesis, but without the experiences I had growing up with it you would not be reading this, so thank you Intel 486-powered computer running MS-DOS.

But in all seriousness, the first thank you must go to my father, who is sadly no longer with us, and to whom I dedicate this thesis. He always cared a lot about my education, and probably cared more about me getting this PhD than I do myself. It is a shame you will not get to see me cross the finish line, but this one's for you dad.

Another thank you goes to my mom, who started teaching me letters and numbers before I even went to pre-school. That no doubt had an impact on how well I did in school and, in turn, in this PhD being writen.

I must also thank my uncle for effectively being a second father to me growing up. If my father could not be there for me for whatever reason, he was.

A massive thank you goes to my supervisor, Tom Schrijvers, who somehow didn't give up on me despite all the delays this PhD had, provided plenty of support throughout, and reviewed this manuscript many many times.

Another massive thank you goes to my other supervisor and former manager, Mike Nicolai, who hired me, gave me the chance to do this PhD, and created the original version of the algorithm that this work expands upon.

I also thank Mike, Herman Van der Auweraer and Johan Vanhuyse for helping me write my Baekeland proposal. It was thanks to their assistance that the proposal was good enough to be accepted by VLAIO, and I'm very thankful to VLAIO for funding this work.

I also cannot forget about Bart Jacobs and Marc Denecker, not only for agreeing to be part of my supervisory committee and examination committee, but also for always making sure I explained my work properly.

I thank Hans Vangheluwe and Erika Ábrahám for agreeing to be part of my examination committee and for the wonderful but sadly few discussions I got to have with them at Siemens.

I must also thank Henri Karhula and Amr Saleh for greatly improving my knowledge of SAT modelling and Constraint Programming and for reviewing this manuscript. Without all the discussions I had with them this work would not be what it is. I also thank the following people for their reviews: Andrei Muntenau, Haik Silm, and Daniil Chivilikhin.

Last but certainly not least I thank my wife, Solange, for her support throughout this whole endeavor. She somehow stuck by me and managed to keep me sane when my life was in shambles and I had a major burnout. Without her this PhD would never have been finished. I also thank my daughter, Leonor, for making it all worth it.

Abstract

Systems engineering is a discipline focused on the design, integration and maintenance of complex systems over their life-cycle. To lower the risks and costs involved with the design and implementation of complex systems, it is necessary to evaluate the quality of the system's design as soon as possible in the design process.

The use of simulation software is the most common solution for early design evaluation. Not just simulation of detailed 3D models that take into account the physical structure of the system, but also lower fidelity models that simply consider the logical structure of the system, i.e., its *architecture* (meaning how its internal components are connected).

Generative Engineering (a form of Computational Design Synthesis) enables the evaluation of system designs even earlier in the design process, through the use of artificial intelligence techniques to automate the exploration and evaluation of potential architectures. Generative Engineering can be applied as soon as the hierarchical decomposition of the system into sub-systems and sub-components is known.

However, state-of-the-art Generative Engineering tools have various trade-offs that limit their applicability for certain use-cases. Some tools are highly effective for a particular domain but can only be applied to problems in that domain. Other tools are very general, but rely on brute-force simulation of candidate architectures, limiting their scalability, or they require formal requirement specifications, which are often not available. Industrial tools tend to favor the former approach (simulation), while academic tools tend to favor the latter approach (formal requirements).

In this thesis we present a synthesis algorithm that is a best-of-all-worlds solution: If formal requirement specifications are not available, it has state-of-the-art performance for generation of candidate architectures. If formal requirement specifications are available, they are used to guide the search towards valid

architectures, avoiding the scalability issues of brute-force simulation.

Our algorithm also supports a novel synthesis mode where formal requirement specifications are used to generate test-cases that can then be used to guide the search process in a very efficient manner. This mode was inspired by how simulation verification is done in the real-world, but the encoding turned out to be similar to that of some domain-specific synthesis tools, particularly those used in the synthesis of logic circuits.

For problems with small input and output domains, like logic circuits, this mode allows our algorithm to find all the valid architectures in milliseconds, whereas its other modes and other general-purpose generative engineering tools are unable to, even when given a one hour time limit.

For problems with larger domains, this mode can still be used to reduce the search space and be combined with the other modes for completeness.

We evaluate our algorithm on three use-cases and using various backend Constraint Programming (CP) and Satisfiability Modulo Theories (SMT) solvers. We show that CP solvers have a major advantage on candidate architecture generation performance, and we also show that the native quantifier reasoning of SMT solvers has unexpected suboptimal performance.

In addition, we contrast our raw architecture generation performance against the closest general-purpose generative engineering tool used in industry, Simcenter Studio, and show a substantial performance lead when using a CP solver as a backend. We posit that the reason for the advantage is due to the use of "blocking clauses" in Simcenter Studio, which was also an issue for our tool when using certain SMT solvers as a backend.

Beknopte samenvatting

Systeemengineering is een discipline die zich richt op het ontwerp, de integratie en het onderhoud van complexe systemen in de loop van hun levenscyclus. Om de risico's en kosten die gepaard gaan met het ontwerp en de implementatie van complexe systemen te verlagen, is het noodzakelijk om de kwaliteit van het systeemontwerp zo vroeg mogelijk in het ontwerpproces te evalueren.

Het gebruik van simulatiesoftware is de meest gangbare oplossing voor vroege evaluatie van ontwerpen. Dit omvat niet alleen simulaties van gedetailleerde 3D-modellen die rekening houden met de fysieke structuur van het systeem, maar ook modellen met een lagere precisie die enkel de logische structuur van het systeem beschouwen, oftewel de architectuur (hoe de interne componenten met elkaar verbonden zijn).

Generatieve Engineering (een vorm van Computationele Ontwerpsynthese) maakt het mogelijk om systeemontwerpen nog eerder in het ontwerpproces te evalueren. Dit gebeurt door gebruik te maken van kunstmatige intelligentie-technieken om de exploratie en evaluatie van mogelijke architecturen te automatiseren. Generatieve Engineering kan toegepast worden zodra de hiërarchische decompositie van het systeem in subsystemen en subcomponenten bekend is.

Echter, de huidige Generative Engineering-tools kennen verschillende afwegingen die hun toepasbaarheid in bepaalde gebruiksscenario's beperken. Sommige tools zijn zeer effectief binnen een specifiek domein, maar kunnen alleen worden toegepast op problemen binnen dat domein. Andere tools zijn zeer algemeen, maar vertrouwen op brute-force simulatie van kandidaat-architecturen, waardoor hun schaalbaarheid beperkt is. Daarnaast vereisen sommige tools formele specificaties van vereisten, die vaak niet beschikbaar zijn.

Industriële tools hebben doorgaans een voorkeur voor de eerste aanpak (simulatie), terwijl academische tools de laatste aanpak (formele specificaties) verkiezen.

In deze thesis presenteren wij een synthese-algoritme dat een “best-of-all-worlds”-oplossing biedt: Als formele specificaties van vereisten niet beschikbaar zijn, levert het algoritme state-of-the-art prestaties bij de generatie van kandidaat-architecturen. Als formele vereisten wél beschikbaar zijn, worden deze gebruikt om de zoekopdracht te sturen naar geldige architecturen, waardoor de schaalbaarheidsproblemen van brute-force simulatie worden vermeden.

Ons algoritme ondersteunt ook een nieuwe synthese-modus waarbij formele specificaties van vereisten worden gebruikt om testcases te genereren. Deze testcases kunnen vervolgens worden gebruikt om het zoekproces op een zeer efficiënte manier te sturen. Deze modus is geïnspireerd op hoe simulatieverificatie in de praktijk wordt uitgevoerd, maar de implementatie vertoont gelijkenissen met die van bepaalde domeinspecifieke synthese-tools, met name die gebruikt worden voor de synthese van logische circuits.

Voor problemen met kleine invoer- en uitvoerdomeinen, zoals logische circuits, stelt deze modus ons algoritme in staat om binnen milliseconden alle geldige architecturen te vinden. Andere modi van ons algoritme en andere algemene Generatieve Engineering-tools kunnen dit niet, zelfs niet met een tijdslimiet van een uur. Voor problemen met grotere domeinen kan deze modus nog steeds worden gebruikt om de zoekruimte te verkleinen en kan deze worden gecombineerd met de andere modi voor volledigheid.

We evalueren ons algoritme op drie verschillende toepassingen en met diverse Constraint Programming (CP) en Satisfiability Modulo Theory (SMT) solvers als backend. We tonen aan dat CP-solvers een aanzienlijk voordeel hebben bij de generatie van kandidaatsarchitecturen en dat de native kwantorredenering van SMT-solvers onverwacht suboptimaal presteert.

Daarnaast vergelijken we onze ruwe architectuurgeneratieprestaties met de dichtstbijzijnde generieke Generatieve Engineering-tool die in de industrie wordt gebruikt, Simcenter Studio. Hierbij demonstreren we een aanzienlijke prestatievoorsprong wanneer een CP-solver als backend wordt gebruikt. Wij stellen dat deze voorsprong te danken is aan het gebruik van “blocking clauses” in Simcenter Studio, wat ook een knelpunt vormde voor onze tool bij het gebruik van bepaalde SMT-solvers als backend.

List of Abbreviations

1UIP First Unique Implication Point.

AADL Architecture Analysis and Design Language.

ACEL Architecture-Centric Exploration Language.

AG Assume/Guarantee.

AGREE Assume Guarantee Reasoning Environment.

AI Artificial Inteligence.

ALU Arithmetic Logic Unit.

API Application Programming Interface.

B&B Branch-and-Bound.

B&C Branch-and-Cut.

BCP Boolean Clause Propagation.

BDD Binary Decision Diagram.

BDD Block Definition Diagram.

BFS Breath-First Search.

BMC Bounded Model Checking.

CAD Computer Aided Design.

CBD Contract-Based Design.

CDCL Conflict-Driven Clause Learning.

CDS Computational Design Synthesis.

CEGIS Counter-Example Guided Inductive Synthesis.

CFD Computational Fluid Dynamics.

CLTL Constrained Linear Temporal Logic.

CLTLoc Constrained Linear Temporal Logic over clocks.

CNF Conjunctive Normal Form.

CP Constraint Programming.

CPU Central Processing Unit.

CSP Constraint Satisfaction Problem.

DIMACS Center for Discrete Mathematics and Theoretical Computer Science.

DPLL Davis–Putnam–Logemann–Loveland.

DSL Domain-Specific Language.

EPS Electric Power System.

EUF Equality and Uninterpreted Functions.

FOL First Order Logic.

GAN Generative Adversarial Networks.

HRELTL Hybrid Linear Temporal Logic with Regular Expressions.

IBD Internal Block Diagram.

IC3 Incremental Construction of Inductive Clauses for Indubitable Correctness.

ILP Integer Linear Programming.

LCC Life-Cycle Costs.

LCG Lazy Clause Generation.

LLM large language model.

LP Linear Programming.

LTL Linear Temporal Logic.

MAC Maintaining Arc Consistency.

MaxSAT Maximum Satisfiability.

MBSE Model Based Systems Engineering.

MILP Mixed Integer Linear Programming.

MINLP Mixed Integer Non-Linear Programming.

MIP Mixed Integer Programming.

MITL Metric Interval Temporal Logic.

MTL Metric Temporal Logic.

NLP natural language processing.

OCRA Othello Contracts Refinement Analysis.

ODE Ordinary Differential Equation.

OEM Original Equipment Manufacturer.

OSATE Open Source AADL Tool Environment.

PDR Property Directed Reachability.

PLTL Past Linear Temporal Logic.

QBF Quantified Boolean Formula.

RAGE Requirement Augmented Generative Engineering.

RAM Random Access Memory.

RELTL Linear Temporal Logic with Regular Expressions.

ReLU Rectified Linear Unit.

RFLP Requirement, Function, Logical, Physical.

SAT Boolean Satisfiability.

SCALe System Contract Assignment Language.

SCL SCALe Constraint Language.

SLS System-Level Simulation.

SMT Satisfiability Modulo Theories.

STL Signal Temporal Logic.

SyGuS Syntax Guided Synthesis.

SysML Systems Modeling Language.

SysMLv2 Systems Modelling Language version 2.

TLSF Temporal Logic Synthesis Format.

UML Universal Modelling Language.

VSIDS Variable State Independent Decaying Sum.

WSL Windows Subsystem for Linux.

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations x

List of Symbols xi

Contents xi

List of Figures xv

List of Tables xix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 10
  - 1.2.1 Contribution 1: The RAGE Algorithm . . . 11
  - 1.2.2 Contribution 2: Test-Set Guided Generative Engineering 12
  - 1.2.3 Contribution 3: Custom Domain-Specific Language . . . 13
  - 1.2.4 Contribution 4: Experimental Evaluation . . . 14
- 1.3 Outline . . . 14

2 Background 17
- 2.1 Model-Based Systems Engineering . . . 18
  - 2.1.1 Systems Modeling Language . . . 18
- 2.2 Computational Design Synthesis . . . 24
  - 2.2.1 Constraint Satisfaction Problems . . . 25
- 2.3 Formal Requirement Verification . . . 44

3 System Contract Assignment Language 51

3.1 System Contract Assignment Language . . . 52
3.1.1 SCALe Syntax . . . 58
3.1.2 SCALe Semantics . . . 61
3.2 SCALe Constraint Language . . . 64
3.2.1 SCL Syntax . . . 64
3.2.2 SCL Semantics . . . 67

4 Requirement Augmented Generative Engineering 69
4.1 Generation . . . 70
4.1.1 Ensemble Generation . . . 70
4.1.2 Architecture Generation . . . 72
4.1.3 Symmetry Breaking . . . 75
4.2 Verification . . . 77
4.3 Synthesis . . . 82
4.3.1 Counter-Example Guided Inductive Synthesis . . . 84
4.3.2 Test-Set Guided Synthesis . . . 85
4.4 Summary . . . 90

5 Experimental Evaluation 91
5.1 Redundant Sensors . . . 94
5.1.1 Summary . . . 99
5.1.2 Redundant Sensors (Broken) . . . 99
5.1.3 Summary . . . 102
5.2 Half-Adder . . . 102
5.2.1 Summary . . . 105
5.3 Full-Adder . . . 105
5.3.1 Summary . . . 108
5.4 Comparison with Simcenter Studio . . . 108
5.5 Summary . . . 111

6 Related Work 113
6.1 Contract-Based Synthesis . . . 113
6.1.1 OCRA . . . 114
6.1.2 PyCo . . . 114
6.1.3 Nuzzo et al. . . . 116
6.2 Structural Synthesis . . . 117
6.2.1 Simcenter Studio . . . 117
6.2.2 Dessia . . . 120
6.3 Contract-Based Verification . . . 121
6.3.1 Stimulus . . . 121
6.3.2 AGREE . . . 121
6.3.3 OCRA . . . 123
6.4 Tool Overview . . . 123

6.5 Reactive Synthesis 126

7 Conclusions and Future Work **127**
7.1 Future Work 129
7.1.1 Additional Structural Constraints 129
7.1.2 Facilitating Requirement Modelling 130
7.2 More Expressive Logics 134
7.3 Optimization 135
7.3.1 Optimizing Static Properties 136
7.3.2 Optimizing Dynamic Properties 136
7.3.3 Optimizing Soft Requirements 138
7.4 Improved Solver Performance 139
7.5 Improved Symmetry Breaking 141
7.5.1 Component-Level Symmetry Breaking 141
7.5.2 Alternative Symmetry Breaking Constraints 141
7.5.3 Abstract Connections 142
7.5.4 SAT Modulo Symmetries 144

Bibliography **145**

List of Publications **159**

List of Figures

1.1 Cumulative Life-Cycle Costs (LCC) vs Committed Costs at each stage of the product lifecycle. Data taken from the Incose Systems Engineering Handbook [68] 2
1.2 V-model following an RFLP structure. 3
1.3 V-Model with the verification and validation steps "shifted-left" through the application of simulation models. 5
1.4 Comparison of the winners of every SAT competition (2002–2020), running on the same hardware, and tackling the same set of problems. Plot taken from fmv.jku.at. 7
1.5 A 1-bit full-adder circuit made out of 9 nand-gates. Simcenter Studio is unable to find this logical architecture in a reasonable amount of time due to the massive number of possible (albeit invalid) configurations. There is only 1 valid configuration, not counting symmetries. Image taken from commons.wikimedia.org 9

2.1 Illustration comparing the classic document-based approach to systems engineering to the modern MBSE approach. Icons from freepik.com 19
2.2 SysML BDD of a distiller, taken from the Enterprise Architect SysML model gallery. Source: https://sparxsystems.com 21
2.3 SysML IBD of a distiller, taken from the Enterprise Architect SysML model gallery. Source: sparxsystems.com 21
2.4 SysML Requirement Diagram of a distiller, taken from the Enterprise Architect SysML model gallery. Note how the original top level requirements are decomposed into more component-level requirements. The link between these requirements and the specific components is not depicted in the image. Source: sparxsystems.com 22
2.5 All possible solutions to the 8-queens puzzle, without symmetries. 28

2.6 A chessboard of boolean variables for the 4-queens problem. The number in the top-right corner of the square is the variable in DIMACS format. 31
2.7 The 4-queens problem in SAT (DIMACS) form. Note that in an actual DIMACS file the literals have to be sorted, but here they are placed according to the original formula for readability. The official format only allows comments before the `p cnf` line at the start, but some solvers support inline comments, and they are added here for readability as well. 33
2.8 The boolean cardinality encoding of the 4-queens problem in SMT-LIB. The *at-most-1* constraint is an extension of the standard which is supported by some SMT solvers. The same encoding used for SAT could also be used here instead. 36
2.9 An alternative integer-based encoding of the n-queens problem, where each q_x variable represents a queen at column x and the value of the variable represents the row the queen is in. 37
2.10 The integer encoding of the 4-queens problem in MiniZinc. Solvers typically do not support this format directly, using the lower level FlatZinc instead. 39
2.11 The boolean cardinality encoding of the 4-queens problem in the CPLEX LP format [14]. LP (which stands for linear program) is one of many formats supported by MIP solvers. Unlike the other schools the standards for MIP are not as clear. The closest to a de-facto standard is the MPS format [3], but like DIMACS it is not exactly human readable. 42

3.1 The system decomposition of the redundant sensor. 52
3.2 The SCALe specification of the redundant sensor system. . . . 55
3.3 The SCALe specification of the internal sensor. 57
3.4 The SCALe specification of the variance monitor. 57
3.5 The SCALe specification of the difference monitor. 57
3.6 The SCALe specification of the selector. 58
3.7 The SCALe grammar. 59
3.8 The internal structure of a SCALe refinement. Refinements contain component groups, connections and connection constraints. Component groups contain component instances. Component instances contain port groups. Port groups contain port instances. Connections contain pairs of port instances. Connection constraints contain connections. 62
3.9 The SCL grammar. 65

4.1 The possible components instances for the redundant sensor use case. Each block depicted in the figure maps to a boolean variable representing the presence (or lack thereof) of that component instance in the system. Components with a dotted outline may or may not appear in the architecture, while the others are mandatory. 71
4.2 The possible connections of the redundant sensor example. Using more specific port types removes alternatives from the problem. 74
4.3 Subset of the constraints generated by Algorithm 6, in this case those related to the input ports of the sensor. Each constraint enforces that, if present, the port must have one and only one connection. 76
4.4 Two symmetric architectures and their respective graph representations. The variability monitor instances have been swapped, with VarMonitor 1 now connecting to Sensor 0 and vice versa. The two graph representations are identical. 76
4.5 The four modes of operation supported by RAGE. Each mode, with the exception of the generation mode that simply disables verification, is explained in its own subsection. The $\mathbb{T}$ in the bottom-right mode refers to the requirements/contract of just the system itself. 84
4.6 Subset of constraints produced by the CEGIS mode for the verifier. The various loop variables and related constraints are standard from BMC. The presence of each connection implies that the connection endpoints (the port instances) share the same value at every point in time. Then we require that the assumptions of the system hold alongside the guarantees of every subcomponent (hidden under the ellipsis), and that the guarantees of the system do *not* hold. This is because in CEGIS mode we are searching for counter-examples. 86

5.1 A valid solution to the redundant sensor specification, using a single difference monitor. 102
5.2 Comparison between Simcenter Studio (a) and RAGE-G (b) on their architecture generation performance over time. The dashed line shows the number of generated architectures at the 1 minute mark. 110

6.1 An OCRA version of the redundant sensor system and the selector. Connections must be explicitly defined, for space only 2 are shown. 115
6.2 The Simcenter Studio UI, showing a hierarchical decomposition of a hybrid vehicle system. 118

6.3 An ACEL version of the redundant sensor system and the selector. Unlike SCALe, output ports are only allowed to connect once, so to achieve a similar result the output port of the selector and the input port of the system have to be replicated multiple times. The use of the keyword group makes the ports symmetric. Requirement specifications are not supported. 119
6.4 The Stimulus UI, showing a set of requirements. While the requirements look like natural language, they are actually formal STL specifications. 122
6.5 The OSATE UI, showing an example with the AGREE annex. Taken from the AGREE documentation. 122
6.6 An AADL + AGREE version of the redundant sensor system and the selector. The arrow operator (`->`) defines a transition from the first step to the rest. Not to be confused with the implication operator (`=>`). 124

List of Tables

2.1 Overview of Constraint Solving Approaches 43

5.1 Solver Capabilities 94
5.2 Redundant Sensors use case using RAGE-G. 95
5.3 Redundant Sensors use case using RAGE-G&T. 96
5.4 Redundant Sensors use case using RAGE-EF. 97
5.5 Redundant Sensors use case using RAGE-CEGIS. 98
5.6 Redundant Sensors (Broken) use case using RAGE-G. 100
5.7 Redundant Sensors (Broken) use case using RAGE-G&T. . . . 100
5.8 Redundant Sensors (Broken) use case using RAGE-EF. 101
5.9 Redundant Sensors (Broken) use case using RAGE-CEGIS. . . . 101
5.10 Half-Adder use case using RAGE-G. 103
5.11 Half-Adder use case using RAGE-G&T. 103
5.12 Half-Adder use case using RAGE-EF. 104
5.13 Half-Adder use case using RAGE-CEGIS. 104
5.14 Half-Adder use case using RAGE-TS. 105
5.15 Full-Adder use case using RAGE-G. 106
5.16 Full-Adder use case using RAGE-EF. 106
5.17 Full-Adder use case using RAGE-CEGIS. 107
5.18 Full-Adder use case using RAGE-TS. 107
5.19 RAGE-G vs. Simcenter Studio on the Redundant Sensor case . 109
5.20 RAGE-G vs. Simcenter Studio on the Full-Adder case 111

6.1 Overview of Synthesis Tools 125

Chapter 1

Introduction

"Complexity comes with innovation. However, trying to reduce complexity is not an option, because that means limiting the ability to innovate. The goal must be to turn complexity into a competitive advantage."

—Jan Leuridan, former CEO of Siemens DISW

1.1 Motivation

Systems engineering is a discipline focused on the design, integration and maintenance of *complex systems* [81] over their life-cycle [68]. From minuscule computer chips to enormous spacecraft, systems engineering is the discipline responsible for ensuring all aspects of the system are taken into account and organized into a synergistic whole.

Complex systems are comprised of many interconnected and interacting components, leading to nonlinear and emergent behavior, feedback loops, and other complications that make it difficult to reason about the system as a whole. Not only are these systems hard to reason about by nature, but the complexity of human-made systems is becoming unmanageable. As an example, the Airbus 380 has more than a hundred thousand wires, connecting more than a thousand different functions, totaling over five hundred and thirty kilometers of wiring [59].

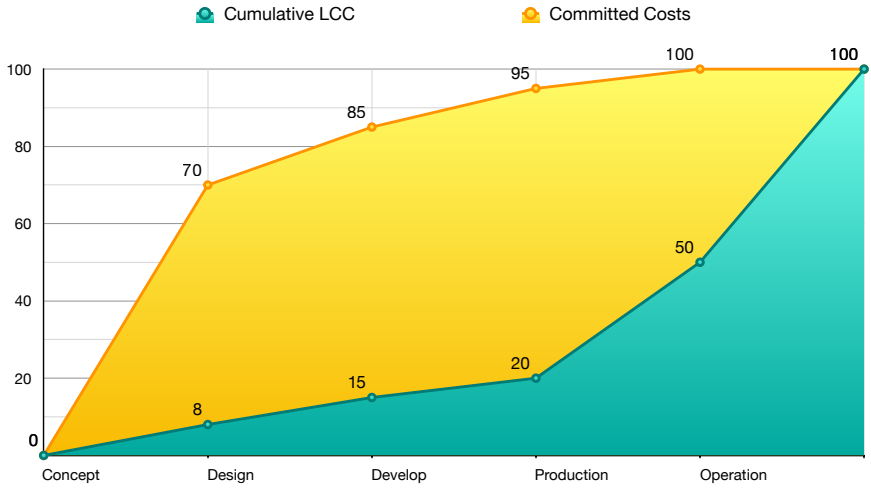


Figure 1.1: Cumulative Life-Cycle Costs (LCC) vs Committed Costs at each stage of the product lifecycle. Data taken from the Incose Systems Engineering Handbook [68]

Due to the size and complexity of the problem, systems are partitioned into subsystems and components that are then allocated to various specialized teams, who must maintain a consistent view of the system throughout its life-cycle. Requirements are often conflicting or ambiguous, and each engineering domain has its own lexicon and tools, which makes a consistent view difficult to maintain [130]. Coordination is necessary to ensure all artifacts describing related aspects of the system are kept up to date and remain consistent with one another.

The state of the art approach used in industry to ensure consistency is Model-Based Systems Engineering (MBSE) [88]. In MBSE, every design artifact links to, or is automatically derived from, a formal central model called the *system model*. The system model makes it possible to track changes to the system's design over time, and to compute the impact of those changes, i.e., which artifacts are affected and what relevant parties need be informed.

Tracking changes and their impact facilitates early detection of issues in the system's design. However, many issues can only be detected late in the design process. As an example, it might only be possible to detect that the system consumes too much energy once a working prototype of the system is built and tested.

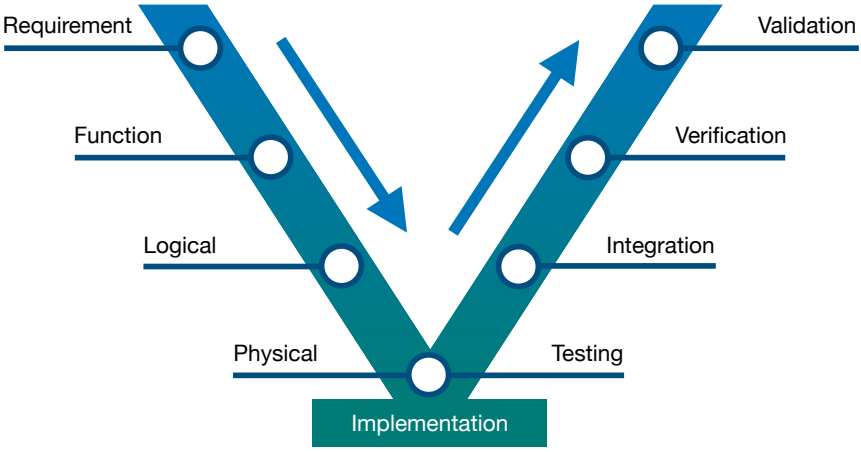


Figure 1.2: V-model following an RFLP structure.

On average, over 70% of a system's life-cycle costs are committed to during the conceptual-design phase, despite only around 8% of development resources being allocated to that phase [68]. Committed costs include contracts, purchase orders, and other financial obligations that have been agreed upon at that point in the development process but that have yet to be paid. Depending on the stage of the development process where an issue is detected, these committed costs will either have been paid already for a design that cannot be used, or will have to be renegotiated (possibly with penalties).

Figure1.1 constrasts the committed costs at each design phase against the amount of resources invested at that point in time. The later a design problem is detected, the costlier it is to fix, up to a thousand times more [68].

The typical system development life-cycle follows a v-shaped structure, as illustrated by the V-Model [125]. The steps depicted on a V-Model vary, but all V-Models move from requirements to implementation on the left side, followed by verification and validation on the right.

Figure 1.2 shows a V-model following a Requirement-Function-Logical-Physical (RFLP) structure [76], where the left-side of the V-model is split into four steps corresponding to each of the letters in the acronym:

- In Step 1, the requirement step (**R**), the system's requirements are collected, covering everything from business and legal demands to technical

constraints and objectives.

- In Step 2, the function step (**F**), the requirements are translated into high-level, user-facing features, and those features are then refined into *technical functions* that can be implemented by specific subsystems or components.
- In Step 3, the logical step (**L**), the technical functions are mapped to abstract components that are then connected to form the system's *logical architecture*. The logical architecture defines how all the parts of the system fit together without taking into account the actual physical sizing and positioning of the various parts.
- Step 4, the physical step (**P**), concretizes the system's logical architecture, first mapping the abstract components to particular implementations and then placing them in relation to each other in physical space.

Since redesign cycles are expensive, new designs are often based on pre-existing systems to minimize the likelihood that major changes will be necessary. One of the major advantages incumbents have against new entrants is the accumulated institutional knowledge they built around the dominant designs of the time. A novel architecture can completely upend this advantage, turning institutional knowledge from an asset to a liability [60].

To lower the risks and costs involved with the design and implementation of entirely novel architectures, it is necessary to evaluate the quality of the system's design as soon as possible in the design process. Engineers rely on simulation software [127] to test the system as much as possible before investing time and resources into building a prototype. Computational Fluid Dynamics (CFD) [30] software, for example, allows engineers to simulate how a three-dimensional model of the system interacts with fluids and gases.

But it is possible to evaluate certain aspects of the system even before its physical structure is known. System-Level Simulation (SLS) software, for example software based on bond-graphs [32], treats the system as a simplified model that defscribes every component in terms of Ordinary Differential Equations (ODEs) [19]. These models can then be linked together to approximate certain aspects of the system's behavior, without consideration to the placement and size of the components.

Simulation allows to "shift-left" the test and verification steps of the V-model, as illustrated in Figure 1.3. The further "to the left" test and verification can be shifted, the cheaper it becomes to explore novel designs. With SLS, evaluation of the design can begin at the early stages of the physical design phase.

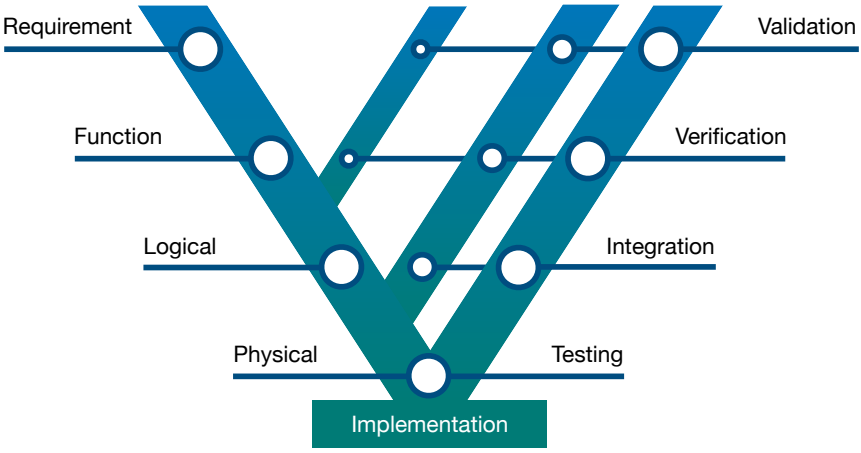


Figure 1.3: V-Model with the verification and validation steps "shifted-left" through the application of simulation models.

Generative Engineering, a particular kind of Computational Design Synthesis (CDS) [36], takes the idea of "shifting-left" even further, by allowing evaluation of the system's design before its logical architecture is fully defined. Starting from the system model, Artificial Inteligence (AI) techniques are applied to search for potential logical architectures automatically. If behavioral models of the system's components are available, for example because they are common parts or are reused from an older system, then the logical architectures can be translated to simulation models of physical architectures automatically as well. These physical architectures can then be simulated and evaluated to determine the optimal candidates to pursue, greatly reducing the costs associated with designing and implementing novel architecture designs.

To search for logical architectures, a generative engineering algorithm must know what *choices* it needs to make and what *alternatives* there are for each choice. As an example, a component's characteristics make up a set of possible choices. For a battery we can choose its capacity, voltage, density, weight, size, price, etc. At the architecture level, the presence of a particular connection (e.g., connecting component A to component B or C) is a choice.

For each choice, the set of possible alternatives makes up its *domain*. Domains can be a discrete (e.g., choosing between three different density classes) or continuous (e.g., choosing a particular density in a given range).

Note that making a choice has an impact on other choices. Increasing a battery's capacity will affect its weight, size or material. The relationships between choices *constrain* which alternatives can be taken together.

Formally, what is described above is a Constraint Satisfaction Problem (CSP). A CSP is a triple $\langle X, D, C \rangle$ where X is the set of variables (i.e., choices), D is the set of their respective domains (i.e., the alternatives), and C is the set of constraints (i.e., the relationships between choices).

Solving a CSP amounts to assigning a value to each variable, from its domain, such that every constraint is satisfied.

The most naive way to solve a CSP is to pick a random assignment and check if the constraints are satisfied. If they are not, try again with a different assignment. This approach might never find a solution if the same random assignment can be attempted more than once.

A slightly less naive but still brute force approach is to do a depth-first search where each combination of values from the domains of the variables is attempted in order. This approach is guaranteed to terminate, but it might take an extremely long time to do so.

The issue is that each additional variable multiplies, by the size of its domain, the number of possible assignments. The size of the problem grows exponentially with the number of variables. A problem with 64 binary variables has 18.446.744.073.709.551.616 possible assignments, all of which would have to be attempted in the worst case.

To put that number into perspective, a 3 GHz CPU that can increment a number every clock cycle would take around 97 years just to count up to that number.

Although CSPs are in theory very difficult to solve, in practice there are many algorithms that can solve complex real-world CSPs in a reasonable amount of time. Boolean Satisfiability (SAT) [29] solvers, for example, could solve problems with millions of constraints and hundreds of thousands of variables back in 2007 [103]. SAT solvers have seen substantial improvements every year, meaning there have been, at the time of writing, 17 years worth of research and improvements to those numbers.

Figure 1.4 contrasts the performance of the winners of every SAT competition [7] since 2002, running on the same hardware and tackling the same set of problems. Kissat [28], the winner of the 2020 competition, is able to solve over 100 benchmarks in a few minutes, something the 2007 winner is unable to do despite being given more than an hour to do so.

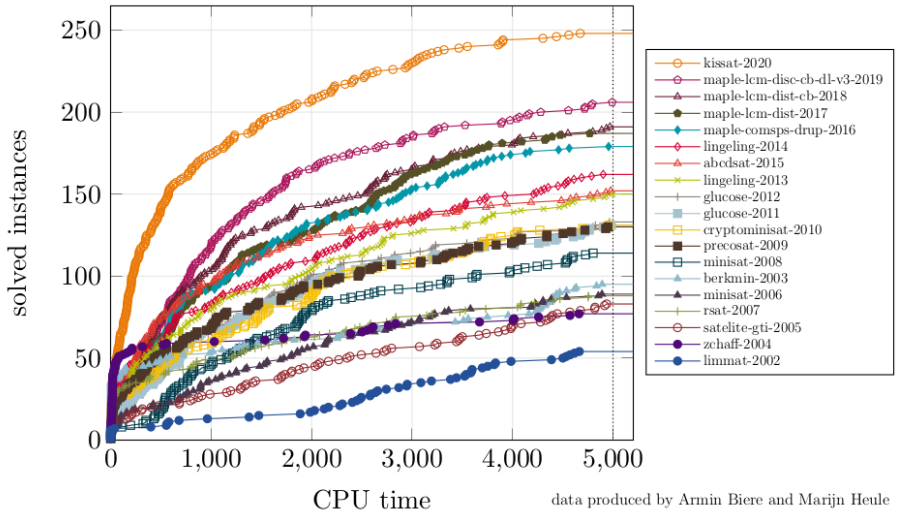


Figure 1.4: Comparison of the winners of every SAT competition (2002–2020), running on the same hardware, and tackling the same set of problems. Plot taken from fmv.jku.at.

Simcenter Studio [9] is a state-of-the-art generative engineering tool that relies on SAT solving to search for logical architectures. It works by translating a user-provided hierarchical decomposition of the system, plus constraints on what connections are allowed or not allowed, into a CSP. Solutions to the CSP can be translated to logical architectures, and each of those logical architectures can be translated to a simulation model for evaluation.

In RFLP terms, starting from the end of the Function step where the functions of the system have already been mapped to components, Simcenter Studio is able to automate (to a great extent) the Logical and Physical steps: it generates all possible logical architectures, translates them to physical architectures, and can evaluate them to find the best alternative.

Simcenter Studio was used, for example, to explore all possible 2-stage transmission designs consisting of 2 clutches and 2 brakes. It generated a total of 739.628 logical architectures, of which only 12 were found to reach 4-speeds after evaluation [73].

Only one 4-speed design had been previously patented. Simcenter Studio found **all** 4-speed designs that can possibly exist for that set of components. The complete design space was explored, meaning no other 4-speed designs exist

and 5-speed designs are impossible for that type of transmission.

How architectures are evaluated is a criticial design decision for Generative Engineering tools. Some industrial tools like Simcenter Studio and Dessia [1] are designed to be as generic as possible, relying on user-defined evaluation scripts and simulation to verify the generated architectures.

This generality makes Simcenter Studio applicable to a wide range of industrial problems, from transmission designs as above to hybrid vehicles [126] and electrical power systems in aircraft [94].

However, there is a cost: the search for architectures is "blind". Simcenter Studio suffers from a combinatorial explosion in the number of generated architectures because the CSP lacks information (in the form of constraints modelling the system's behavior) that would prevent the solver from generating architectures that cannot work in practice.

While engineers can explicitly forbid or enforce connections between components (based on domain knowledge), this may not sufficiently decrease the size of the design space, or may lead to design fixation where new designs follow existing designs too closely, a common phenomenon [70].

As an example of the combinatorial explosion issue, Simcenter Studio is unable to find the one logical architecture that exists for a full-adder circuit consisting of 9 nand-gates (see Figure 1.5) in a reasonable amount of time.

Full-adders are a common type of circuit where logic gates (e.g., nand-gates) are connected in such a way that the resulting circuit computes the sum of three input bits (one representing a carry bit from a previous addition). The output is two bits, one representing the carry. Multiple adder circuits can, in turn, be connected to compute sums of numbers with more bits. A CPU's Arithmetic and Logic Unit (ALU) includes many adder circuits for example.

Different types of logic gates can be used to build an adder circuit, but for this example we are using nand-gates because they are "functionally complete", meaning with just nand-gates it is possible to implement any Boolean function.

Although the system is very small and has only one valid architecture[1], the search space is massive. Every pair of ports that can be connected is one choice the solver needs to make; without constraints that amounts to 2^{240} possible assignments: the 3 inputs from the adder plus 1 output from each of the 9

[1]The architecure of the full-adder presented here is purely logical. A real hardware implementation must take into account physical aspects like electrical delay. We ignore the problem of electrical delay here because any concrete implementation of the system must respect the logical model, and transforming the logical model into a working hardware model can be done as a post-processing step.

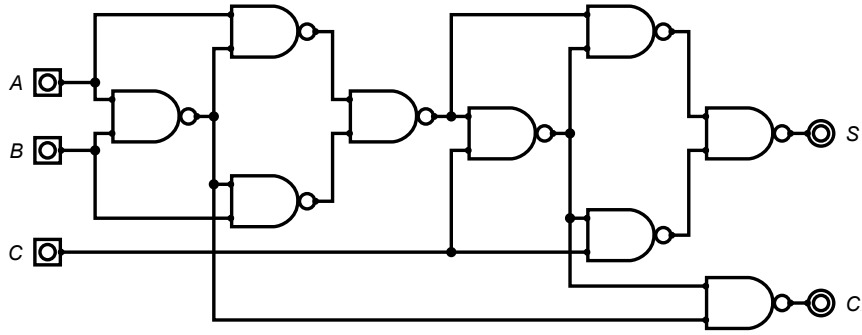


Figure 1.5: A 1-bit full-adder circuit made out of 9 nand-gates. Simcenter Studio is unable to find this logical architecture in a reasonable amount of time due to the massive number of possible (albeit invalid) configurations. There is only 1 valid configuration, not counting symmetries. Image taken from commons.wikimedia.org

nand-gates (12 ports total) can each connect to one or more inputs of the nand-gates (2 per gate, 18 total) plus the 2 outputs of the adder (20 ports total), where the presence of each connection ($12 * 20 = 240$) is a binary choice. With some constraints (e.g., that each port must connect at least once) there are still trillions of possible configurations.

Because the system consists of a single component type (nand-gates) there is no good way to constrain their connections without just modeling the architecture directly, meaning that nearly every valid solution to the CSP is invalid after verification.

On the other hand, the formal specifications of full-adders and nand-gates are well known: the output O of a nand-gate is $O = \neg(I_1 \wedge I_2)$ where each I_n is an input bit. The output of a full-adder is $C_o * 2 + S = A + B + C_i$, where C_o refers to the output carry bit and C_i the input carry bit. This information can be included in the CSP to reduce the search space.

Domain-specific tools, like logic circuit synthesizers, develop their search algorithms around the specific structural constraints of logic circuits (e.g., no cycles) and the behavior of relevant logic gates. They also use the intented truth table of the circuit to very effectively guide the search [77]. These tools are highly efficient but they are only applicable to their particular domain.

Some generic synthesis tools, like PyCo [66], require formal specifications to guide the search towards valid architectures. While there are restrictions on the sorts of specifications they support, these generic tools remain domain-agnostic,

and are applicable for many different kinds of systems if they can be described in terms of connected components and if formal specifications are available.

The main issue, and the main reason why generic industrial tools do not focus on formal specifications, is that outside of specific domains, formal specifications are often not available. Simulation models for common components are readily available [2], however, and since they can be used to evaluate the generated architectures in a post-processing step, formal specifications are not critical.

This causes a bit of a "chicken and egg" problem. On the one hand, generative engineering benefits greatly from the availability of formal specifications, as they greatly improve search performance, but such specifications are rarely available. On the other hand, because formal specifications are rarely available, industrial tools do not make use of them, so outside of safety critical domains there is little reason to invest time and effort into the development of formal specifications of common components.

If formal specifications of common components were as readily available as simulation models, generic generative engineering tools would be able to scale to much larger problems.

1.2 Contributions

The lack of support for formal specifications in industrial generative engineering tools is an understandable but unfortunate situation. Since formal specifications are rarely available, it is hard to justify the development of commercial tools that make use of them. This, in turn, means it is hard to justify the development of formal specifications of common components, since there are not many tools that can make use of them. This "chicken and egg" situation limits the potential of generative engineering, since formal specifications are necessary to guide the search and efficiently explore large design spaces.

In this thesis we hope to address this situation, and provide a path forward for general-purpose industrial generative engineering tools to make use of formal specifications.

Our goal is to develop an algorithm that can make use of formal specifications (when available) to guide the search and avoid the combinatorial explosion problem, but that otherwise retains the capabilities and performance of state-of-the-art tools.

By making formal specifications entirely optional, investment into formal specifications of common components can be done incrementally, on a

per-domain or per-use-case basis, greatly improving generative engineering performance for those domains and use cases over time, but without sacrificing the broad applicability of existing tools in the market.

We make the following four contributions.

1.2.1 Contribution 1: The RAGE Algorithm

The core contribution of this thesis is a novel algorithm for Generative Engineering called RAGE (Requirement Augmented Generative Engineering). The core feature of RAGE is the ability to work efficiently with or without formal specifications. When formal specifications are available, they are used to guide the search, avoiding the combinatorial explosion problem. When formal specifications are not available, RAGE can be used for architecture generation with user-specified evaluation scripts.

Architecture generation is encoded as an $\exists\forall$ (Exists/Forall) CSP. The existential ($\exists$) part of the CSP encodes the space of possible configurations while the universal ($\forall$) part of the CSP enforces that the configuration is a valid implementation of the system's requirements.

RAGE supports four different modes of operation:

1. In the first mode, **RAGE-G**, the requirement part of the problem is ignored and RAGE functions as a pure architecture generator.
2. In the second mode, **RAGE-EF**, the CSP is passed in its entirety to an $\exists\forall$-capable solver, generating only valid architectures directly.
3. In the third mode, **RAGE-CEGIS**, the CSP is split into two separate problems, each fed to a different solver, one to handle generation and one to verify each generated solution. Counterexamples found by the verifier are used to further constrain the search space of the generator.
4. In the fourth mode, **RAGE-TS**, two solvers are used in separate phases. The first phase solver generates a user-defined number of test cases from the system-level requirements. These test cases are then fed as constraints to the second phase solver which uses them to guide its search towards valid architectures.

The need for the first mode, **RAGE-G**, should be self-explanatory, it is how RAGE can be applied to use cases for which formal specifications are unavailable.

The second mode, **RAGE-EF**, was developed because, theoretically, combining the two problems into a single formula should allow an $\exists\forall$-capable solver to perform the search with perfect information, achieving the best possible performance.

Existing tools based on formal specifications rarely make use of such solvers, however, prefering instead to structure the solving process in a way that avoids the universal quantifier. This is also what we do in the third mode, **RAGE-CEGIS**.

1.2.2 Contribution 2: Test-Set Guided Generative Engineering

Our second contribution is the fourth mode based on test case generation, **RAGE-TS**. As far as we are aware this mode is novel in the context of general-purpose generative engineering tools.

Most tools based around formal specifications tend to focus on generating a single architecture, since any architecture they find is guaranteed to be a valid implementation of the system (barring issues with the specifications).

RAGE-TS, on the other hand, was inspired by how verification using simulation is done in industry, where the system is tested multiple times with many different inputs. If all the tests pass, then the implementation is assumed to be correct, with any undetected problems hopefully caught later in the design process.

Similarly, depending on the coverage of the system specification by the generated tests, **RAGE-TS** might allow invalid architectures to slip through. While this might seem like a major downside, it is not in practice.

Since RAGE must be able to function without formal specifications, it necessarily has to explore all possible architectures. In this setting **RAGE-TS** serves as a way to decrease the size of the search space, greatly reducing the number of architectures that need to be evaluated.

The main advantage of this mode, and why it is so important, is that the second phase solver only needs to deal with an existential problem (unlike **RAGE-EF**), and does not require continuous communication with another solver (unlike **RAGE-CEGIS**). Depending on the problem (i.e., how many tests are necessary for good coverage) this may result in massive performance gains. As an example, the full-adder circuit in Figure 1.5 only needs eight tests for complete coverage, making it a perfect use case for this mode.

While we found **RAGE-TS** to be novel in the context of general-purpose generative engineering tools, the resulting encoding is somewhat similar to

encodings used in Boolean circuit synthesizers [77], where each row of the circuit's truth table serves as the equivalent to a "test case" constraint in **RAGE-TS**. The main difference is that **RAGE-TS** computes relevant test cases automatically from the specification and is not limited to Boolean circuits.

1.2.3 Contribution 3: Custom Domain-Specific Language

Our third contribution is a novel Domain-Specific Language (DSL) [90] that combines the structural modelling features of industrial generative engineering tools with optional formal specifications.

Without formal specifications, generative engineering tools must exhaustively explore the complete design space to ensure that the best possible solution can be found after evaluation.

To generate all possible architectures efficiently without formal specifications requires the following: 1) the problem must be bounded, 2) symmetric architectures can be detected, and 3) powerful structural modelling features are available to allow engineers to encode their domain-knowledge.

If the problem is unbounded, then architecture generation will never terminate. While the process can be stopped after some amount of architectures have been generated, there is no guarantee that the best solution or even a valid solution is among them.

Each symmetry doubles the number of possible architectures, and those additional architectures will have the same evaluation results as their symmetric counterpart, making them useless. So detecting and avoiding symmetries is critical.

Tools that make use of formal specifications tend to have weaker structural modelling features and rarely take into account symmetries since any solution they find is guaranteed to be valid. This makes them unusable when formal specifications are unavailable.

Our DSL's structural modelling features are heavily inspired by Simcenter Studio, and like Simcenter Studio the problem is set up in a way that makes it feasible to do architecture generation efficiently without formal specifications.

Formal specifications in our DSL are modular Assume/Guarantee contracts, following the semantics defined by Benveniste et al. [24], using both propositional logic and Linear Temporal Logic (LTL) [115].

Because contracts are modular, component specifications are not tied to a particular system. This allows common components to be formally specified

once, independently of any given system, and reused across various systems and architectures.

1.2.4 Contribution 4: Experimental Evaluation

Our last contribution is an evaluation of all the modes supported by RAGE, on multiple use cases, and using multiple backend solvers of different kinds.

The backend solvers we evaluate have different capabilities. For example, some have native support for finding all solutions to a problem, while others support universal quantification. We want to determine the impact these capabilities have for Generative Engineering, determine what solver is the best option for each mode, and we want to confirm the following three assumptions.

Our first assumption is that guiding the search with formal specifications is substantially more efficient than simply generating and evaluating all possible architectures, even when using an encoding for architecture generation that does not require them.

Our second assumption is that **RAGE-EF** is superior to **RAGE-CEGIS** when using the same backend solver, since **RAGE-CEGIS** requires two solvers with incomplete information to communicate, while **RAGE-EF** allows a single solver to tackle the problem with perfect information. In the worst case **RAGE-EF** should be slightly faster, as the solver can simply use the same algorithm we implemented for **RAGE-CEGIS**, but without the communication overhead.

Our final assumption is that **RAGE-TS** is able to find the single valid solution to the full-adder use case faster than the other two guided modes, since a small number of test cases are enough to achieve complete coverage, but we do not know the magnitude of the difference.

1.3 Outline

The rest of this thesis is structured as follows:

1. **Chapter 2: Background** presents important methodologies for this thesis: MBSE, CDS, and formal requirement verification.
2. **Chapter 3: System Contract Assignment Language** presents the DSL developed for this thesis, including its syntax and semantics.

3. **Chapter 4: Requirement Augmented Generative Engineering** presents the novel RAGE algorithm developed in this thesis.
4. **Chapter 5: Solver Evaluation** presents an evaluation of various backend solvers, of different kinds, on a multitude of use cases, using RAGE's four modes of operation. We also compare our pure architecture generation performance with Simcenter Studio's.
5. **Chapter 6: Related Work** presents other projects that have similar synthesis capabilities and contrasts them to this work.
6. **Chapter 7: Conclusions and Future Work** presents the findings of this thesis and various ways in which this work can be expanded upon.

Use of Generative AI: I did use the generative AI assistance tool ChatGPT. ChatGPT was used to translate an example specification in the related work to a different specification language, and the translation was manually reviewed. The example is purely illustrative and has no bearing on the rest of the thesis.

The text/code/images in this thesis are my own (unless otherwise specified) and generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

Parts of this thesis were published in the following papers:

- C. A. R. Santos, A. H. Saleh, T. Schrijvers and M. Nicolai, "CONDEnSe: Contract Based Design Synthesis", *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Munich, Germany, 2019, pp. 250-260
- C. A. R. Santos, T. Schrijvers, A. H. Saleh, M. Nicolai, "Divide et Impera: Efficient Synthesis of Cyber-Physical System Architectures from Formal Contracts", *Formal Methods*, Beijing, China, 2021, pp. 776-787.

The first paper introduced our methodology and includes an early version of the DSL presented in this thesis (Contribution 3) and an early version of our RAGE synthesis algorithm (Contribution 1), mapping roughly to its first two modes of operation (RAGE-G plus filtering and RAGE-EF). That version of the algorithm only supported propositional logic and could not take advantage of counter-examples to guide the search.

The second paper implements some of the future work suggested in the first paper. The most important contribution is the addition of support for Linear Temporal

Logic specifications, extending both the DSL and the synthesis algorithm. Additionally, it also introduced an older version of RAGE-CEGIS (the third mode in Contribution 1), allowing counter-examples to guide the search, and an older idea for improving search performance we called "feasibility", which is not part of this thesis.

Feasibility required that every generated architecture must be capable of producing at least one valid input/output pair. This turned out to be a very weak constraint, but it evolved into RAGE-TS (Contribution 2) which is not published at the time of writing.

Other parts of this thesis, including improved symmetry breaking and the new experimental results, are also not published at the time of writing.

The ideas presented and most of the work are my own. The idea of combining Computational Design Synthesis with Contract-Based Design (but not the how) comes from Mike Nicolai, as does the original version of the pure architecture generation synthesis algorithm. A separate evolution of that algorithm is used in Simcenter Studio.

While the algorithms presented in the papers and in the thesis are my own, their design involved many discussions with Amr Saleh and Tom Schrijvers, who also helped with the text.

Chapter 2

Background

This section details the methodologies and technologies involved in this thesis.

Model-Based Systems Engineering (MBSE) is the focus of this work. Some of the models used in MBSE – particularly those that describe a hierarchical decomposition of the system and those that describe a mapping between requirements and the various components and subsystems – serve as input, while those that describe logical architecture models serve as output. In this section we go over those models and the language they are typically written in, the **Systems Modeling Language (SysML)**.

Computational Design Synthesis (CDS) is applied to find logical architectures given a hierarchical decomposition of a system. We achieve this through a translation of the problem into a **Constraint Satisfiability Problem (CSP)** which is then solved using various off-the-shelf solvers with varying capabilities in different configurations. In this section we go over different types of CSPs and their tradeoffs.

Formal Requirement Verification is used to guide the search towards valid solutions. There are various ways of formally modeling requirements and verifying them, but for this thesis we need the requirements to be of a specific form such that they can participate in the generation process. In this section we go over some approaches to formal requirement verification, particularly **Contract-Based Design (CBD)** which is the approach used in this thesis.

2.1 Model-Based Systems Engineering

Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification, and validation activities beginning in the conceptual design phase and continuing throughout development and later lifecycle phases [88].

In the traditional document-based systems engineering approach, system development is driven by natural-language documents and static diagrams that describe different aspects of the system. These documents serve as the primary means of communication among stakeholders.

Each discipline (e.g., mechanical, electrical, software, etc.) produces and maintains its own set of domain-specific documents, which must be cross-referenced and manually synchronized to ensure they remain consistent throughout the system's lifecycle. Maintaining consistency becomes increasingly difficult as the size and complexity of a system grows [68].

MBSE was developed as a response to this challenge. In MBSE, a *system model* serves as a central artifact that drives the entire engineering process. It serves as a formal, cohesive representation of the system across its entire lifecycle, integrating the various aspects of system development, from requirements to verification and validation. This is achieved by linking every artifact produced throught the system's lifecycle to the system model.

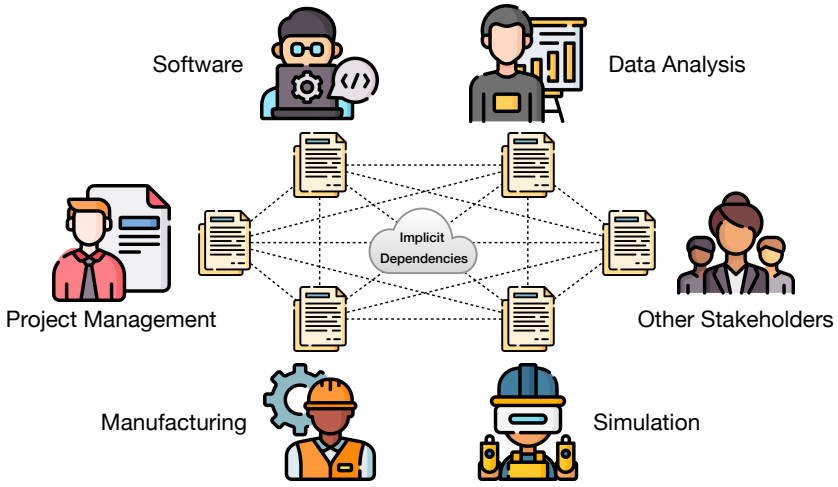
Figure 2.1 contrasts the traditional document-based approach 2.1a to the model-based approach 2.1b.

2.1.1 Systems Modeling Language

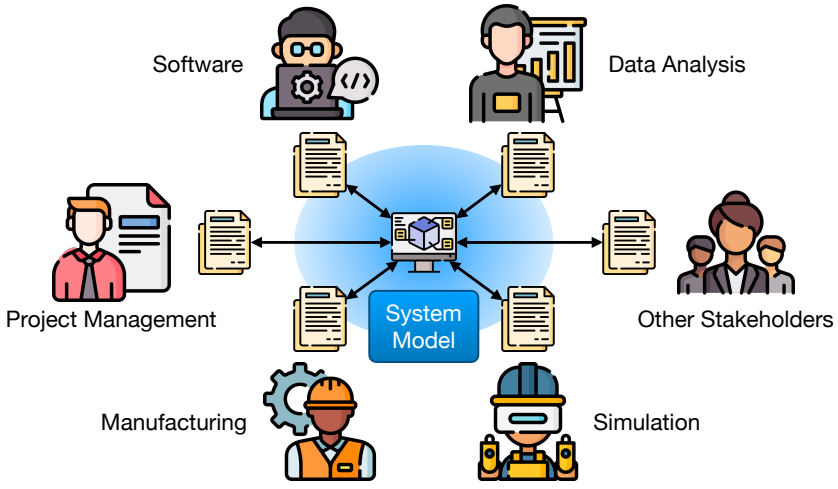
The de-facto standard language for systems level modeling is the Systems Modeling Language (SysML) [63]. SysML provides a graphical representation to describe the structure, behavior, requirements, and architecture of systems across different domains.

SysML supports 9 different types of diagrams:

- **Requirement Diagrams** capture the system requirements at different levels, allowing for example to map a particular component to the requirements it is responsible for fulfilling.
- **Block Definition Diagrams (BDD)** describe the hierarchical decomposition of the system



(a) The traditional document-based approach to systems engineering. Dependencies between the various artifacts produced throughout the design are implicit, making consistency difficult to maintain.



(b) The MBSE approach to systems engineering. Every document links to or is automatically derived from a central system model, allowing to trace the impact of changes to the affected artifacts.

Figure 2.1: Illustration comparing the classic document-based approach to systems engineering to the modern MBSE approach. Icons from freepik.com

- **Internal Block Diagrams (IBD)** describe how the various components and subsystems connect to each other.
- **Package Diagrams** organize the model into packages (collections of elements), showing their relationships and dependencies.
- **Use Case Diagrams** capture the functional requirements of a system by illustrating system actors (e.g., users) and their interactions with the system.
- **Activity Diagrams** describe control flows and data flows within a system.
- **Sequence Diagrams** describe component interactions over time.
- **State Machine Diagrams** define the various states the system and its components may be in and what occurrences trigger transitions between said states and under which conditions.
- **Parametric Diagrams** describe parametric constraints between system properties, such as performance, reliability, or cost.

Of the 9, the BDD, the IBD and the Requirement Diagram are the most relevant for this thesis. Figure 2.2 shows a SysML BDD for a distiller system, Figure 2.3 shows an IBD of the distiller, and Figure 2.4 shows a Requirement Diagram for the same distiller.

SysML's ability to model system requirements and link them to other models is one of its most important aspects, particularly in the context of this thesis. The requirements diagram enables engineers to capture system requirements in a formal structure: to define relationships between multiple requirements, and relationships between requirements and components or subsystems.

This allows for requirement traceability, which is critical to ensure that all system requirements are addressed in the design, and that verification and validation processes are aligned with the system's intended functionality.

Note, however, that the actual requirements are not necessarily formal, often the requirements are just natural language descriptions of the intended behavior. Only the diagrams themselves, meaning the relationships between requirements and components, are formalized.

SysML v1, the original version of the standard, extends and adapts the Unified Modeling Language (UML) for systems engineering, as UML lacks the constructs needed to describe the non-software components. A new version of the standard, SysML v2, is under development at the time of writing and is no longer based on UML [6]. SysML v2 diagrams are different from SysML v1 diagrams but

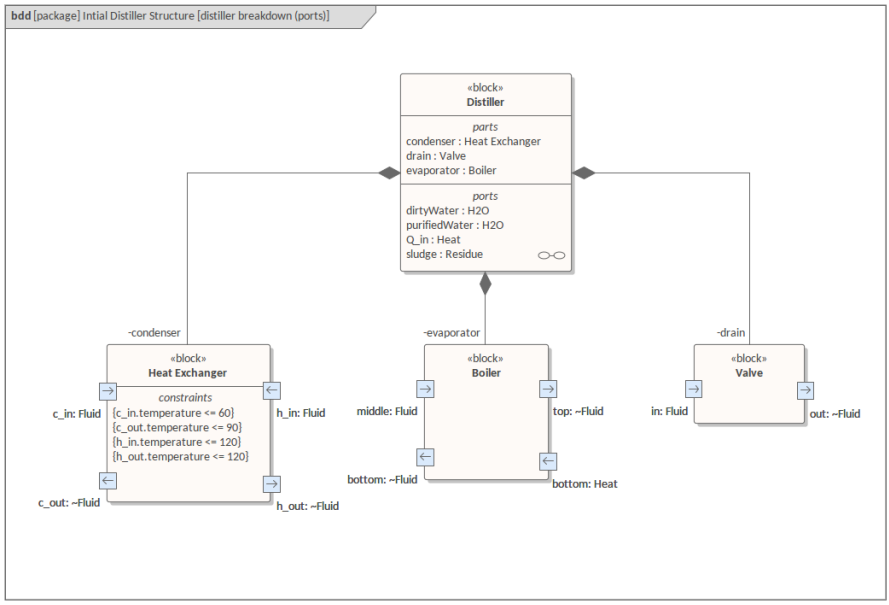


Figure 2.2: SysML BDD of a distiller, taken from the Enterprise Architect SysML model gallery. Source: https://sparxsystems.com

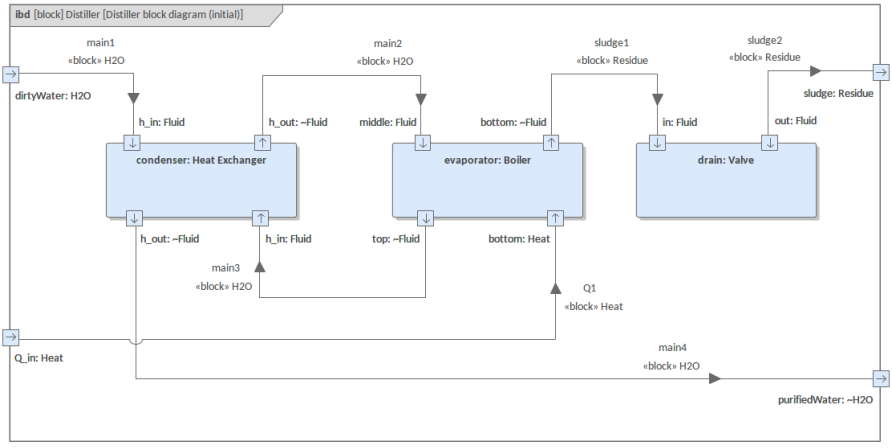


Figure 2.3: SysML IBD of a distiller, taken from the Enterprise Architect SysML model gallery. Source: sparxsystems.com

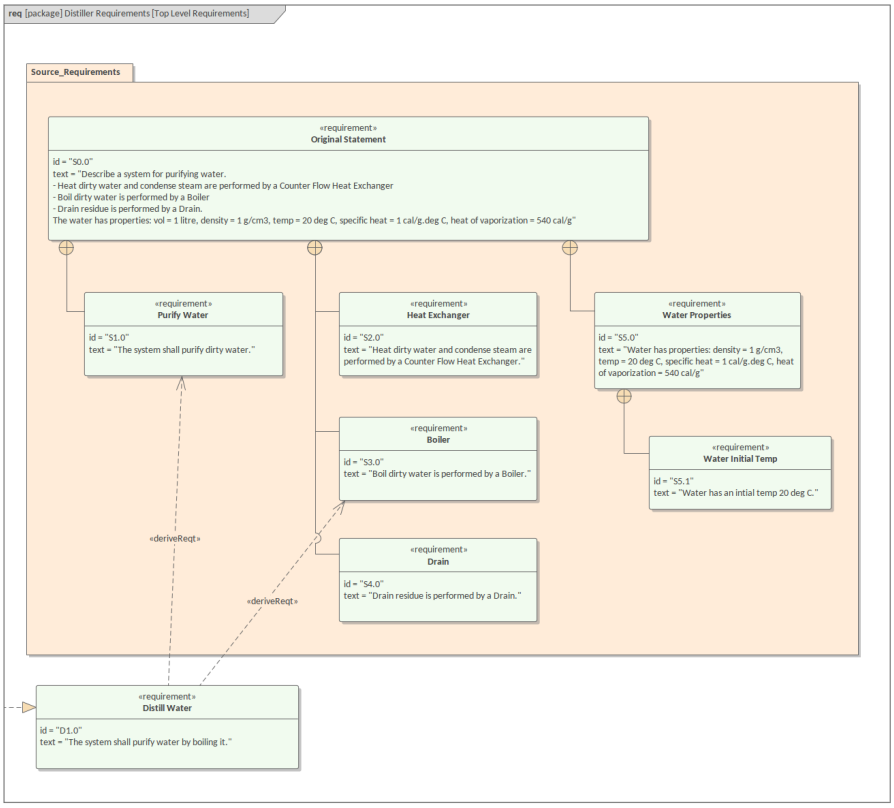


Figure 2.4: SysML Requirement Diagram of a distiller, taken from the Enterprise Architect SysML model gallery. Note how the original top level requirements are decomposed into more component-level requirements. The link between these requirements and the specific components is not depicted in the image. Source: sparxsystems.com

share similar purposes. In SysML v2 the Requirement Diagram is called the Requirement View, while BDDs and IBDs become General Views and Interconnection Views, respectively. It also supports a textual representation, providing a DSL for systems engineering.

System design is traditionally done following the V-Model process [125], illustrated previously in Figure 1.2. It describes the activities to be performed and their intended outcome throughout the system's development. The left side of the "V" goes from requirement gathering to a prototype implementation while the right side of the "V" verifies and validates that implementation.

There are multiple variants of the V-Model process, but all of them follow the same general idea. As an example, consider an automotive Original Equipment Manufacturer (OEM) who follows the V-Model development process to develop a new state-of-the-art hybrid car:

1. First, the system requirements are collected, identifying what is expected of the system. For example, a functional requirement might be: *the car must be powered by both gasoline and electricity*, while a non-functional requirement might be: *The car must cost less than thirty thousand euros.*

2. From the requirements, a set of functions are derived, e.g., *converting chemical energy into torque*, *converting electric power into torque* and *storing electric power.*

3. From the functions, a set of components that comprise the system is derived. In this case, the system would need to have a *combustion engine*, an *electric motor* and a *battery* at the minimum. This process culminates in the *system model*, with a BDD providing the hierarchical decomposition of the system into subsystems and components.

4. From the system model, various logical architectures can be derived. *Series*, *Discontinuous Variable Transmission* and *Continuous Variable Transmission* architectures are all available on the market [82]. Orthogonally, automotives can also be *front-wheel*, *rear-wheel* or *all-wheel* driven, giving at least 9 unique architectural alternatives. Each of these logical architectures corresponds to a different IBD. Typically only one of these architectures is further developed.

5. From the logical architectures, multiple detailed models of the system and its components are produced: fault tree analysis [116], Computer Assisted Design (CAD) [17] models, 1D [32] and 3D simulations [30], embedded controllers [91], etc.

In the process above, a single logical architecture had to be decided upon without any concrete data on its performance, beyond previously available historical data of similar implementations. The architecture can have massive impact on the performance of the system [126], and if the chosen architecture is found to be suboptimal, most of the models that were derived from it need to be redone. As discussed previously in the introduction, these redesign cycles are orders of magnitude more expensive the later in the development process they are found [68].

Exploring multiple alternatives in parallel is also costly as it diverts limited engineering resources. Each variability point in the logical architectures exponentially increases the number of designs that need to be considered to find the optimal solution, so even if the resources to explore multiple alternatives in parallel are available, there is a strict limit to the number of alternatives that can be manually explored.

2.2 Computational Design Synthesis

Computational Design Synthesis (CDS) [36] is a broad research field focused on leveraging computational methods to automatically or semi-automatically generate design solutions. Unlike traditional design approaches, where designers manually iterate through concepts to find and refine solutions, CDS allows for the systematic exploration of a design space to find designs that are optimal or entirely novel.

CDS provides two key advantages:

- **Efficiency**: Traditional design cycles can be lengthy due to the iterative nature of concept generation, analysis, and refinement. CDS reduces the time and effort required to explore a wide range of potential solutions by automating substantial portions of this cycle.
- **Novelty**: Human designers may be unable to conceive of all possible designs, a problem known as *design fixation* [70]. CDS can exhaustively explore large portions of the design space, potentially uncovering non-intuitive solutions that might otherwise not be considered.

To be able to computationally explore a design space, the design problem must be clearly defined and parameterized. This includes identifying relevant variables (design parameters), constraints (physical, geometrical, or material limitations), and performance criteria (such as weight or cost). These elements

are then structured into a formal problem statement that can be interpreted by an algorithm.

There are many CDS algorithms stemming from different AI fields: genetic algorithms [95], graph grammars [114], multiagent systems [119], generative adversarial networks (GANs) [56], reinforcement learning [72], constraint satisfaction [113], etc. For this thesis we are mainly concerned with constraint satisfaction.

The main advantage of constraint satisfaction over the other approaches is that it is *sound* and *complete*, meaning the constraint satisfaction solver is able to find (not counting bugs in its source code or a lack of sufficient computational resources) a valid solution to the problem if any exists, or prove that there are none. Constraint satisfaction can also exhaustively explore the entire search space when it is finite, finding *all* possible solutions. Most of the other approaches are heuristic in nature, they might find a solution more efficiently but they might not be able to find a solution at all and they may be unable to detect if there are no solutions.

2.2.1 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) is a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a set of the domains of those variables, and C is a set of constraints.

Solving a CSP amounts to computing a value for each $x \in X$, taken from domain D_x, such that all constraints in C are satisfied. If there is such an assignment, the CSP is *satisfiable*. If no assignment exists that makes all constraints satisfied, then the CSP is *unsatisfiable*.

Solving a CSP on a finite domain is an NP-complete problem in general [54]. Being in NP means that there is no known polynomial-time algorithm to find a solution to an arbitrary CSP. More concretely, the time requirxd to solve a CSP problem grows exponentially with the size of the problem (meaning the number of variables and possible values in their domain).

In practice, there are algorithms that solve various kinds of CSPs with hundreds of thousands of variables and millions of constraints in a reasonable amount of time [103]. Real world problems have an implicit structure that these algorithms are able to exploit to solve the problem efficiently.

There are many sub-fields or "schools" of research related to CSPs, with the most relevant for this thesis being: Boolean Satisfiability (SAT) [29], Satisfiability

Modulo Theories (SMT) [29], Constraint Programming (CP) [113], and (Mixed) Integer Programming (MIP) [129], all under the umbrella of *formal methods*.

SMT builds on top of SAT while CP may or may not use a SAT solver internally [103]. MIP follows an entirely different approach to SAT, but some SMT and CP solvers employ MIP techniques [74].

Complete algorithms for all four schools are based on backtracking search, though incomplete approaches based on local search are also possible. We will focus on the complete algorithms here, because the synthesizer developed in this thesis is designed to exhaustively find all solutions to a problem, which local search cannot do.

All these algorithms employ some form of preprocessing, inference, search, various heuristics, plus some learning mechanism to shrink the search space.

Each school has its pros and cons, and is more or less suited to different types of problems. The question then is which school is the best fit for the synthesis approach used in this thesis. There are a few properties of interest:

- **Expressiveness:** How easy is it to express complex properties? SMT and CP are both highly expressive. MIP struggles with disjunctions and disequalities. SAT only supports boolean propositions in Conjunctive Normal Form (CNF), though arbitrary boolean expressions can be converted to CNF through a Tseitin Transformation [123].
- **Performance:** How quickly can the solver set up the problem and find a solution? All four schools have impressive performance, though SAT and MIP tend to be the fastest because they are *lower level*. Of course this only applies if the problem has a reasonable encoding in SAT or MIP, otherwise SMT and CP solvers with their specialized algorithms will perform better [103].
- **Incrementality:** Can a problem be incrementally refined and solved again without needing to restart the solving process from scratch? Most SAT and SMT solvers excel at this, allowing for new constraints to be added after every solve call, while CP and MIP solvers tend to be more restricted in the incremental refinements they support.
- **Universality:** Is the solver able to encode universal properties, i.e., that some proposition must hold in all possible cases? Some SMT solvers can do this. SAT solvers cannot handle it directly, but Quantified Boolean Formula (QBF, an extension of SAT) solvers can [120]. CP solvers, if they can model universal properties, typically solve the problem by brute-

force unrolling every possible combination. MIP solvers do not support universal constraints.

- **All-Solution Search:** Can the solver find all solutions to a problem in a single query? Most CP solvers and some MIP solvers can do this. SAT and SMT solvers typically do not. The only mainstream SMT solver that supports all-solution search natively at the time of writing is MathSAT [39].
- **Optimization:** Can the solver find optimal solutions to the problem given an objective function? MIP is focused entirely on optimization, and almost all CP solvers support optimization as well. Some SMT solvers have added optimization support but performance is not yet on par with MIP or even CP. SAT solvers do not support optimization directly, though various SAT extensions like MaxSAT do [98].

MIP excels at optimization, CP at all-solution search and SMT at universality and incrementality. All of these properties are relevant to synthesis as we will show in Chapter 4, so evaluating and exploring the different problem and solver types is valuable.

Despite the differences in expressiveness, all four schools are NP-complete and, therefore, are all able to express the same problems with more or less effort.

To illustrate each school, we will use the classic "N-Queens puzzle" [25]. The choice of example is not arbitrary, as the generation part of the synthesis algorithm introduced in Chapter 4 is similar to the boolean encoding of the N-Queens puzzle.

The N-Queens puzzle is the problem of placing N queen pieces on an $N \times N$ chessboard without any of the queens being threatened by another queen. Queens can attack in any of the four cardinal directions or in a 45° diagonal, up to any distance. The queens must be positioned such that no other queen is in the threatened squares of another queen.

Encoding the N-Queens puzzle amounts to encoding the following constraints:

1. There is exactly 1 queen in each row.
2. There is exactly 1 queen in each column.
3. There is at most 1 queen in each \ diagonal.
4. There is at most 1 queen in each / diagonal.

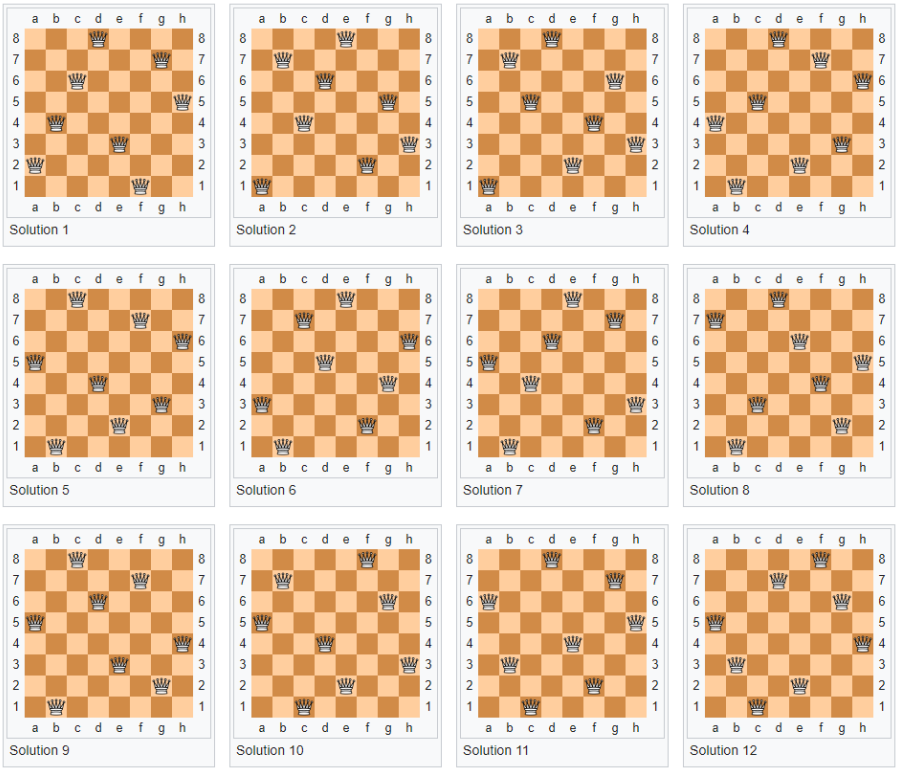


Figure 2.5: All possible solutions to the 8-queens puzzle, without symmetries.

Figure 2.5 shows all possible solutions to the 8-queens puzzle, not counting symmetric solutions (rotations and reflections of the board). For our examples we will use 4 queens on a 4×4 board instead, to keep the constraints small.

Boolean Satisfiability

Boolean Satisfiability (SAT) is the problem of determining if a propositional formula is satisfiable, specifically one in CNF [29]. A CNF formula is a formula of the form: $\bigwedge(l_1 \vee \ldots \vee l_n)$ where each l_m is either x_m or $\neg x_m$, x_m is a variable, and $\neg$ is the negation operator.

Consider the following CNF formula:

$$\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \tag{2.1}$$

The same formula can also be viewed in set form:

$$\varphi = \{\{x_1, \neg x_2\}, \{x_2, x_3\}, \{x_2, \neg x_4\}, \{\neg x_1, \neg x_3, x_4\}\} \tag{2.2}$$

In practice, SAT solvers accept problems in the DIMACS format [29]. That same formula in DIMACS would look like the following:

```
p 4 4
 1 -2
 2  3
 2 -4
-1 -3 4
```

As you can see SAT is very low level. The first line indicates the problem size. Each following line is a *clause* and each number in a clause is a *literal*. During solving, each variable can take one of three values: 0 meaning false, 1 meaning true, or *undefined*. Literals take the value of the corresponding variable, or $1 - value$ if they are negated.

Clauses, meanwhile, can be characterized as:

- Unsatisfied: All literals of that clause are set to 0.
- Satisfied: At least one literal in the clause is set to 1.
- Unit: All literals except one are set to 0 and the last literal is undefined.
- Unresolved: None of the above.

Unit clauses are specially important since the only course of action is to set the remaining literal in the clause to true, which then affects other clauses. This is called *unit propagation* and is one of the core procedures of Conflict-Driven Clause Learning (CDCL)-based SAT solvers, the state-of-the-art complete algorithm for SAT solving [29].

The goal of CDCL is to find an assignment to every variable such that every clause is *satisfied*. The input to the algorithm is the set of clauses φ and a set of assignments μ (a set of pairs of variables and values) which starts out empty. The pseudo-code for CDCL is shown in Algorithm 1.

CDCL is an improvement over the older Davis–Putnam–Logemann–Loveland (DPLL) algorithm [29], which also combines backtracking search with unit

Algorithm 1 The CDCL algorithm

```
function CDCL(φ, μ)
    dl ← 0
    if BCP(φ, μ) = CONFLICT then
        return Unsatisfiable
    end if
    while not CompleteAssignment(φ, μ) do
        (x, v) = Branch(φ, μ)
        dl ← dl + 1
        μ ← μ ∪ {(x, v)}
        if BCP(φ, μ) = CONFLICT then
            β = ConflictAnalysis(φ, μ)
            if β < 0 then
                return Unsatisfiable
            else
                Backtrack(φ, μ, β)
                dl ← β
            end if
        end if
    end while
    return Satisfiable
end function
```

propagation. The main improvements of CDCL over DPLL are the ConflictAnalysis(φ, μ) function and the non-chronological backtracking of Backtrack(φ, μ, β) [29].

Each function in the CDCL algorithm does the following:

- BCP (Boolean Clause Propagation) applies the unit clause rule until there are no more unit clauses or a conflict is detected (i.e., a clause becomes unsatisfied with every literal set to false). The detection of a unit clause is typically done through a 2-Watched-Literal scheme [29].
- Branch picks a "random" unassigned variable and sets it to true (1) or false (0). There are various heuristics to guide the algorithm on which variable to pick, the most well known being Variable State Independent Decaying Sum (VSIDS) [29].
- ConflictAnalysis analyses the current conflict and learns a clause that prevents a similar set of decisions from being taken again during the search. This is typically achieved by computing a First Unique Implication Point (1UIP) [29].

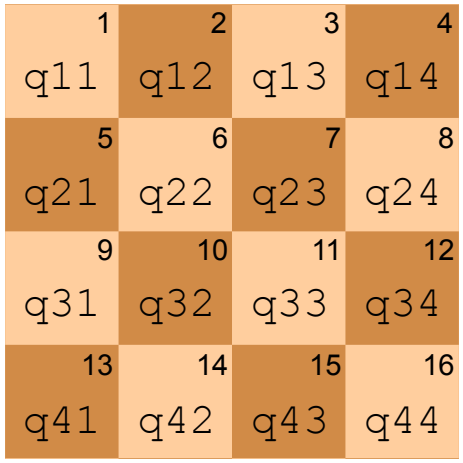


Figure 2.6: A chessboard of boolean variables for the 4-queens problem. The number in the top-right corner of the square is the variable in DIMACS format.

- Backtrack reverts every decision taken since the decision level β computed by ConflictAnalysis. In current state-of-the-art SAT solvers, Backtrack restarts all the way to decision level 0 after a certain number of iterations [29]. This prevents the algorithm from getting stuck in a bad area of the search tree. The learned clauses ensure that the algorithm terminates.
- CompleteAssignment checks if the all the variables have been assigned and terminates the search loop in that case, with the algorithm returning Satisfiable.

The inputs to the CDCL algorithm are passed by reference, and are mutated by the algorithm. The result of CDCL is not just Satisfiable or Unsatisfiable, but also the variable assignments stored in μ.

The N-Queens puzzle can be modeled in SAT as follows: Model each position in the chessboard as a boolean variable $q_{r,c}$ where r is the row and c is the column. If $q_{1,2}$ is true, then there is a queen at row 1 and column 2, otherwise the position is free. See Figure 2.6 for which variable corresponds to which position.

Then, we need to encode the cardinality constraints in CNF form. Constraints of the form ≤ 1 are called *at-most-1* constraints and there are many different encodings of such constraints into SAT [52]. Here we use the only encoding that

does not require additional variables, the *pairwise* encoding. A formula of the form $x_1 + x_2 + x_3 \leq 1$ can be encoded with the pairwise encoding as follows:

$$(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3) \tag{2.3}$$

These constraints enforce that for every pair of variables in the formula, at least one of those variables must be false. When all pairs are combined, the only way to solve the constraints is to set all variables to false or at most 1 to true.

To encode constraints of the form ≤ 1, it suffices to combine *at-most-1* with *at-least-1*, and the latter is a simple disjunction ($\vee$). The entire 4-queens puzzle in SAT form can be seen in Figure 2.7. The cardinality constraints are shown as comments, indicated by the `c` marker at the start of the line.

State-of-the-art SAT solvers include Kissat [28], CryptoMiniSAT [121] and CaDiCaL [27].

Satisfiability Modulo Theories

While SAT is powerful and can tackle all sorts of finite domain problems, it is not meant to be human readable. Most real world problems are defined in terms of higher level concepts like numbers and equations, not flat disjunctions of boolean variables.

That is where Satisfiability Modulo Theories (SMT) [29] comes in. SMT is an extension of SAT towards first-order logic plus a fixed interpretation of functions and predicates from background *theories*, for example the theory of Presburger Arithmetic [74].

There are two main approaches for developing SMT solvers, the *eager* approach and the *lazy* approach. The eager approach translates the complete SMT problem into a SAT problem, whereas in the lazy approach specialized theory solvers and an internal SAT solver coordinate to solve the problem [58]. Depending on the background theories, each approach might yield better results. Most SMT solvers support both approaches depending on the theories used.

The most used algorithm to solve SMT problems is CDCL($\mathcal{T}$), which is CDCL with the addition of theory solvers. The basic idea is to abstract away all the theory predicates into boolean variables which are solved by CDCL as normal. Once CDCL finds a satisfiable assignment, the theory solvers are queried for the satisfiability of the theory interpretation. The pseudo-code for CDCL($\mathcal{T}$) is shown in Algorithm 2.

```
p cnf 4 84
c 1 queen per row
c q11+q12+q13+q14 = 1
1 2 3 4
-1 -2
-1 -3
-1 -4
-2 -3
-2 -4
-3 -4
c q21+q22+q23+q24 = 1
5 6 7 8
-5 -6
-5 -7
-5 -8
-6 -7
-6 -8
-7 -8
c q31+q32+q33+q34 = 1
9 10 11 12
-9 -10
-9 -11
-9 -12
-10 -11
-10 -12
-11 -12
c q41+q42+q43+q44 = 1
13 14 15 16
-13 -14
-13 -15
-13 -16
-14 -15
-14 -16
-15 -16
c 1 queen per column
c q11+q21+q31+q41 = 1
1 5 9 13
-1 -5
-1 -9
-1 -13
-5 -9
-5 -13
-9 -13
c q12+q22+q32+q42 = 1
2 6 10 14
-2 -6
-2 -10
-2 -14
-6 -10
-6 -14
-10 -14
c q13+q23+q33+q43 = 1
3 7 11 15
-3 -7
-3 -11
-3 -15
-7 -11
-7 -15
-11 -15
c q14+q24+q34+q44 = 1
4 8 12 16
-4 -8
-4 -12
-4 -16
-8 -12
-8 -16
-12 -16
c max 1 queen per \
c q11+q22+q33+q44 <= 1
-1 -6
-1 -11
-1 -16
-6 -11
-6 -16
-11 -16
c q12+q23+q34 <= 1
-2 -7
-2 -12
-7 -12
c q13+q24 <= 1
-3 -8
c q21+q32+q43 <= 1
-5 -10
-5 -15
-10 -15
c q31+q42 <= 1
-9 -14
c max 1 queen per /
c q41+q32+q23+q14 <= 1
-13 -10
-13 -7
-13 -4
-10 -7
-10 -4
-7 -4
c q31+q22+q13 <= 1
-9 -6
-9 -3
-6 -3
c q21+q12 <= 1
-5 -2
c q42+q33+q24 <= 1
-14 -11
-14 -8
-11 -8
c q43+q34 <= 1
-15 -12
```

Figure 2.7: The 4-queens problem in SAT (DIMACS) form. Note that in an actual DIMACS file the literals have to be sorted, but here they are placed according to the original formula for readability. The official format only allows comments before the `p cnf` line at the start, but some solvers support inline comments, and they are added here for readability as well.

Algorithm 2 The $CDCL(\mathcal{T})$ algorithm

```
function CDCL(T)(φ, μ)
    if T-Preprocess(φ, μ) = CONFLICT then
        return Unsatisfiable
    end if
    φ^p = T2B(φ); μ^p = T2B(μ)
    loop
        T-Branch(φ^p, μ^p)
        loop
            status = T-Deduce(φ^p, μ^p)
            if status = Satisfiable then
                μ ← B2T(μ^p)
                return Satisfiable
            else if status = Conflict then
                β = T-AnalyzeConflict(φ^p, μ^p)
                if β = 0 then
                    return Unsatisfiable
                else
                    T-Backtrack(φ^p, μ^p, β)
                end if
            else
                break
            end if
        end loop
    end loop
end function
```

We will not go into detail on CDCL($\mathcal{T}$) since it is mostly CDCL with queries to the theory solver. Two notable functions are $\mathcal{T}2\mathcal{B}()$ and $\mathcal{B}2\mathcal{T}()$ which translate the input formula into a propositional formula and the translated propositional formula back to a first-order formula, respectively.

If the combination of predicates is not theory-satisfiable, the theory solver produces additional explanation clauses that tell CDCL which combination of abstracted boolean variables led to the problem.

Eventually either CDCL returns Unsatisfiable, or both it and the theory solver return Satisfiable. Not every theory is complete however, in which case the solver will eventually return Unknown. For example the theory of non-linear integer arithmetic is undecidable unless every variable has a finite domain [31]. Some theories of interest include:

- **Equality and Uninterpreted Functions (EUF)**: Sometimes called the "empty" theory because it does not have any actual interpreted symbols, it is nonetheless one of the most important theories in SMT, as many problems can be abstracted as uninterpreted functions. EUF satisfiability is typically handled by Ackermannization [29].
- **Presburger Arithmetic** The theory of Presburger Arithmetic interprets integer arithmetic over the symbols $(0, 1, +, -, \leq)$. It can model linear equations and it is typically solved using MIP techniques [74].
- **Real Closed Fields** Unlike integer arithmetic, non-linear real arithmetic is decidable, though very computational expensive to solve. Modern SMT solvers use a variety of techniques to tackle non-linear real arithmetic, including decidable methods based on *Cylindrical Algebraic Decomposition*, which has doubly exponential complexity [80]. Linear real arithmetic can be solved much more efficiently using the *Simplex* algorithm [74].
- **Fixed-width Bit-vectors** While the theories of Presburguer Arithmetic and Real Closed Fields are focused on pure mathematics with infinite domains, the theory of Bit-vectors is concerned with *machine arithmetic*, meaning mathematics as done on a computer, where numbers are restricted to certain bit sizes. Bit-vectors are typically solved by encoding the relevant binary circuits as SAT clauses (an eager encoding).

Combinations of theories can be handled by the Nelson-Oppen framework, Delayed Theory Combination or Model-Based Theory Combination [44].

The standard input format for SMT solvers is SMT-LIB [22]. The 4-queens problem can be encoded into SMT using the same approach as SAT, but there are other alternatives. Some SMT solvers have built-in support for cardinality constraints, so the *at-most-1* constraint can be encoded directly.

Alternatively, instead of one boolean variable per position of the board, the problem can be modeled as 1 integer variable per queen, with the value indicating which row the queen occupies. Then, a *distinct* predicate can be used to enforce that all values assigned to the queen variables (and hence, the rows) are different. For diagonals, the vertical distance to the other queens is used to check if they would land on a threatened square.

The two different encodings are shown in Figure 2.8 and Figure 2.9, respectively.

State-of-the-art SMT solvers include Z3 [45], cvc5 [21], MathSAT 5 [39] and Yices 2 [48].

```
(declare-const q11 Bool)
(declare-const q12 Bool)
(declare-const q13 Bool)
(declare-const q14 Bool)
(declare-const q21 Bool)
(declare-const q22 Bool)
(declare-const q23 Bool)
(declare-const q24 Bool)
(declare-const q31 Bool)
(declare-const q32 Bool)
(declare-const q33 Bool)
(declare-const q34 Bool)
(declare-const q41 Bool)
(declare-const q42 Bool)
(declare-const q43 Bool)
(declare-const q44 Bool)
; 1 queen per row
(assert (or q11 q12 q13 q14))
(assert ((_ at-most 1) q11 q12 q13 q14))
(assert (or q21 q22 q23 q24))
(assert ((_ at-most 1) q21 q22 q23 q24))
(assert (or q31 q32 q33 q34))
(assert ((_ at-most 1) q21 q22 q23 q24))
(assert (or q41 q42 q43 q44))
(assert ((_ at-most 1) q41 q42 q43 q44))
; 1 queen per column
(assert (or q11 q21 q31 q41))
(assert ((_ at-most 1) q11 q21 q31 q41))
(assert (or q12 q22 q32 q42))
(assert ((_ at-most 1) q12 q22 q32 q42))
(assert (or q13 q23 q33 q43))
(assert ((_ at-most 1) q13 q23 q33 q43))
(assert (or q14 q24 q34 q44))
(assert ((_ at-most 1) q14 q24 q34 q44))
; max 1 queen per \ diagonal
(assert ((_ at-most 1) q11 q22 q33 q44))
(assert ((_ at-most 1) q12 q23 q34))
(assert ((_ at-most 1) q13 q24))
(assert ((_ at-most 1) q21 q32 q43))
(assert ((_ at-most 1) q31 q42))
; max 1 queen per / diagonal
(assert ((_ at-most 1) q41 q32 q23 q14))
(assert ((_ at-most 1) q32 q21 q13))
(assert ((_ at-most 1) q21 q12))
(assert ((_ at-most 1) q42 q33 q24))
(assert ((_ at-most 1) q43 q34))
(check-sat)
(get-model)
```

Figure 2.8: The boolean cardinality encoding of the 4-queens problem in SMT-LIB. The *at-most-1* constraint is an extension of the standard which is supported by some SMT solvers. The same encoding used for SAT could also be used here instead.

```
(declare-const q1 Int)
(declare-const q2 Int)
(declare-const q3 Int)
(declare-const q4 Int)
; each queen can be in 1 of 4 rows
(assert (and (>= q1 1) (<= q1 4)))
(assert (and (>= q2 1) (<= q2 4)))
(assert (and (>= q3 1) (<= q3 4)))
(assert (and (>= q4 1) (<= q4 4)))
; each queen must be in a different row
(assert (distinct q1 q2 q3 q4))
; queens cannot be diagonal to each other
(assert (! (= (abs (- q1 q2)) 1)))
(assert (! (= (abs (- q1 q3)) 2)))
(assert (! (= (abs (- q1 q4)) 3)))
(assert (! (= (abs (- q2 q3)) 1)))
(assert (! (= (abs (- q2 q4)) 2)))
(assert (! (= (abs (- q3 q4)) 1)))
(check-sat)
(get-model)
```

Figure 2.9: An alternative integer-based encoding of the n-queens problem, where each q_x variable represents a queen at column x and the value of the variable represents the row the queen is in.

Constraint Programming

Constraint programming (CP) is the paradigm focused on solving finite-domain CSPs directly. Unlike SMT, CP does not require an underlying SAT solver, though some modern CP solvers make use of one. Like SMT, CP is highly expressive compared to SAT, though with a focus on *global constraints* rather than theories.

Global constraints are specialized predicates that encode common patterns in CSPs. They can be solved by decomposition into simpler constraints (e.g., down to boolean expressions), or by specialized *propagators*, which are similar to theory solvers in SMT but applied to a single constraint.

There is a huge catalog of global constraints [23], the most well known being the *AllDifferent* constraint, which enforces that a set of variables must not share a value. The *AllDifferent* constraint can be decomposed into a set of inequality constraints between every pair of variables, or it can be solved by a propagator implementing one of many specialized algorithms.

A propagator's job is to prune the search space by shrinking the domains of the variables in the constraint. For example, when a variable gets assigned the value 5, the *AllDifferent* propagator would remove that value from all the other variables in the constraint. This may, in turn, trigger other propagators to also change the domains of shared variables. This process is called *constraint*

propagation, a more general case of *unit propagation* in SAT.

Constraint propagation continues until every constraint reaches a state of *local consistency*, meaning the propagators cannot detect further inconsistencies between the variables in the constraint. There are different forms of local consistency, with different pruning power. The most commonly used form of local consistency is *generalized arc consistency* [87].

Complete CP solvers combine constraint propagation with some form of backtracking search. One of the most successful CP algorithms that does not rely on an underlying SAT solver is the Maintaining Arc Consistency (MAC) algorithm [113]. Pseudo-code for a simple version of the MAC algorithm, using the AC-3 algorithm for arc-consistency [87], is shown in Algorithm 3.

Algorithm 3 The MAC algorithm

```
function MAC(φ, μ)
    if CompleteAssignment(μ) then
        return Satisfiable
    end if
    v_i ← PickUnassignedVar(φ)
    for x_i ∈ OrderDomainValues(v_i, φ, μ) do
        if Consistent(x_i, μ) then
            μ ← μ ∪ {v_i, x_i}
            PropagateAC3(v_i, x_i, φ, μ)
            if MAC(φ, μ) ≠ Unsatisfiable then
                return Satisfiable
            end if
        end if
        μ ← μ \ {v_i, x_i}
    end for
end function
```

The MAC algorithm can also be combined with a *nogood* learning mechanism (same concept as conflict clauses in SAT) [113]. Examples include negative last-decision nogoods [85], increasing nogoods [86], or the use of an embedded SAT solver, similar to how SMT-solvers work. The latter is called Lazy Clause Generation (LCG) [103] and is the state-of-the-art in CP solving.

Unlike SAT and SMT, there are two separate standards CP solvers accept as input: FlatZinc [99] and XCSP3 [33]. FlatZinc is the flattened form (no nested constraints) of the constraint modeling language MiniZinc [99], which is a higher-level language meant for humans.

```
include "alldifferent.mzn";
var 1..4: q1;
var 1..4: q2;
var 1..4: q3;
var 1..4: q4;
constraint alldifferent([q1, q2, q3, q4]);
constraint abs(q1 - q2) != 1;
constraint abs(q1 - q3) != 2;
constraint abs(q1 - q4) != 3;
constraint abs(q2 - q3) != 1;
constraint abs(q2 - q4) != 2;
constraint abs(q3 - q4) != 1;
solve satisfy;
```

Figure 2.10: The integer encoding of the 4-queens problem in MiniZinc. Solvers typically do not support this format directly, using the lower level FlatZinc instead.

Figure 2.10 shows the integer encoding of the 4-queens problem written in MiniZinc.

State-of-the-art CP solvers include CP-SAT from Google OR-Tools [106], which is an LCG-based solver, and Choco [111], which is a MAC-based solver.

(Mixed) Integer Programming

Most CP solvers and some SMT solvers support optimization as well as satisfiability, but this is not the main focus of the algorithms. (Mixed) Integer Programming (MIP), on the other hand, focuses on optimization explicitly, with satisfiability being a side-effect.

The *mixed* part of the name refers to the fact that variables can be either continuous or integer. When all variables are continuous and every constraint linear, this is called Linear Programming (LP). When all variables are integer and every constraint linear, this is called Integer Linear Programming (ILP). When mixed and linear, it is called Mixed Integer Linear Programming (MILP).

There are many more possibilities for problem classification, with the most general being Mixed Integer Non-Linear Programming (MINLP).

Like SAT, MIP is rather low level, only supporting conjunctions of equations. Modelling disjunctions requires Big-M [13] formulations or similar encodings.

MIP solvers use a combination of techniques. The core algorithm of every MIP solver is the Simplex [105] algorithm for LP. For linear problems with only

continuous variables, applying the Simplex algorithm is enough. For MILP, solvers combine Simplex with a type of backtracking search.

When integer variables are introduced, the Simplex algorithm is used to solve a *relaxed* version of the problem as if every variable was continuous. If the solution found happens to have integer results for all the integer variables, the algorithm is done. Otherwise, this gives a lower bound to the problem, the real solution (if any) cannot be better than the result of the relaxed problem.

If the solution does not respect the integrality constraints on the variables, a Branch-and-Bound (B&B) [105] procedure is applied, where a heuristically chosen variable is rounded up on one branch of the search tree and rounded down on the other branch. This is similar to the backtracking search used in SAT and CP.

Furthermore, MILP solvers can also use *cutting planes* [105] to learn from conflicts detected during the search, similar to how SAT solvers learn conflict clauses and CP solvers learn *nogoods*. When cutting planes are added to the B&B algorithm, it is called Branch-and-Cut (B&C) [105], and it is the state-of-the-art in MILP solving. The pseudo-code for the B&C algorithm is shown in Algorithm 4.

Although MILP is focused on integer and continuous variables, we cannot effectively use the integer encoding for the 4-queens problem. The reason is that neither strict inequality constraints nor disjunctions are natively supported by MILP. That means the $\neq$ operator and correspondingly the *AllDifferent* constraint are very difficult to encode. In fact, modelling the *AllDifferent* constraint in MIP is often done with a boolean encoding [128].

The MILP, boolean cardinality encoding of the 4-queens problem is shown in Figure 2.11.

State-of-the-art MIP solvers include Gurobi [57], SCIP [16], and HiGHS [65], the first being a commercial solver.

Best technology for Synthesis

There are two main properties of interest to this thesis that a solver should support: all-solution search and universal quantification (e.g., $\forall$). The first is mainly available in SMT solvers while the latter is mainly available in CP solvers, so they are our focus.

We will not tackle optimization use cases in this thesis, so MIP does not make much sense to use. MIP solvers are heavily reliant on the objective function to

Algorithm 4 The B&C algorithm (assuming a maximization objective)

```
function BnC(φ, μ)
    l ← {φ}
    s ← −∞
    while l ≠ {} do
        p ← Dequeue(l)
        repeat
            φ_r ← LPRelax(p)
            μ_r ← LPSolve(φ_r)
            if not Feasible(μ_r) then
                continue
            end if
            s_r ← Solution(μ_r)
            if s_r ≤ s then
                continue
            else if IsInteger(μ_r) then
                s ← s_r
                μ ← μ_r
                continue
            end if
            vcps ← ViolatedCuttingPlanes(μ_r)
            if vcps ≠ {} then
                for cp ∈ vcps do
                    l ← l ∪ {LPRelax(p, cp)}
                end for
                continue
            end if
            bps ← BranchPartition(p)
            for b ∈ bps do
                l ← l ∪ {b}
            end for
        until vcps = {}
    end while
    return s
end function
```

```
Maximize
  \ The objective is unnecessary, but it is mandatory
  q11 + q12 + q13 + q14 +
  q21 + q22 + q23 + q24 +
  q31 + q32 + q33 + q34 +
  q41 + q42 + q43 + q44
Subject To
  \ 1 queen per row
  q11 + q12 + q13 + q14 = 1
  q21 + q22 + q23 + q24 = 1
  q31 + q32 + q33 + q34 = 1
  q41 + q42 + q43 + q44 = 1
  \ 1 queen per column
  q11 + q21 + q31 + q41 = 1
  q12 + q22 + q32 + q42 = 1
  q13 + q23 + q33 + q43 = 1
  q14 + q24 + q34 + q44 = 1
  \ max 1 queen per \ diagonal
  q11 + q22 + q33 + q44 <= 1
  q12 + q23 + q34 <= 1
  q13 + q24 <= 1
  q21 + q32 + q43 <= 1
  q31 + q42 <= 1
  \ max 1 queen per / diagonal
  q41 + q32 + q23 + q14 <= 1
  q32 + q21 + q13 <= 1
  q21 + q12 <= 1
  q42 + q33 + q24 <= 1
  q43 + q34 <= 1
Binary
  q11 q12 q13 q14
  q21 q22 q23 q24
  q31 q32 q33 q34
  q41 q42 q43 q44
End
```

Figure 2.11: The boolean cardinality encoding of the 4-queens problem in the CPLEX LP format [14]. LP (which stands for linear program) is one of many formats supported by MIP solvers. Unlike the other schools the standards for MIP are not as clear. The closest to a de-facto standard is the MPS format [3], but like DIMACS it is not exactly human readable.

Table 2.1: Overview of Constraint Solving Approaches

	Incremental	Quantifiers	All-Solution Search	Optimization
SAT	Yes	QBF	No	MaxSAT
SMT	Yes	Some	No [1]	Some
CP	No	No	Yes	Good
MIP	Some	No	Some	Excellent

[1] There is one SMT solver with all-solution search support, MathSAT 5 [39], but we do not support it as a backend.

guide their internal search.

Table 2.1 shows an overview of the capabilities. For SAT, quantifier support and optimization support are extensions (QBF and MaxSAT, respectively).

All-Solution Search The generation part of the algorithm uses a boolean encoding similar to N-Queens, so SAT is an option. If the goal is to find a few solutions as quickly as possible, SAT is the best option due to its performance advantage on pure boolean problems.

While SAT solvers rarely support all-solution search, it is possible to take advantage of their incrementality to implement it. This is for example what Simcenter Studio [9] does, using the blocking-clause method [108].

The blocking-clause method is the most efficient method for finding all solutions to problems with large search spaces but few solutions, or when finding the first few solutions as quickly as possible is important [108]. Unfortunately, each new solution implies the addition of a corresponding blocking clause to the problem, increasing its size, and therefore making the problem harder and slower to solve over time.

CP solvers, on the other hand, can use depth-first search without restarts after pruning the search space with consistency algorithms like AC-3 [87]. This approach is guaranteed to find all solutions without the need for blocking clauses. The fixed-order of the search means that the CP solver might be stuck on a bad part of the search tree for a long time before finding a solution, so there is a trade-off. This can be mitigated with a custom search heuristic, but finding a good search heuristic is itself a challenge.

A similar depth-first search can be implemented for SMT by taking advantage of the assertion stack feature of SMT solvers [108]. Using the assertion stack

prevents the use of specialized algorithms within the SMT solver, so again there is a trade-off to doing this, on top of the fixed-order problem.

Universal Quantification In order to check if the system requirements are satisfied by a particular implementation, we need either universal quantification ($\forall$) to state that the requirements must hold for all possible states, or the ability to negate the problem and check for its unsatisfiability, meaning there is no state where a requirement is unsatisfied. The advantage of universal quantification is that the generation and verification problems can be combined into one.

Neither MIP nor CP can deal with universal constraints. It is possible to model universal constraints in some CP modelling languages like MiniZinc [99], but they are simply unrolled, they are not natively supported by CP solvers. For example, in MiniZinc, one can model $\forall x.\, p(x)$ with `forall(v in dom(x))(p(v))`. The `forall` in MiniZinc is unrolled into a conjunction $p(v_1) \wedge p(v_2) \wedge p(v_3) \wedge \ldots$ for every possible value v_i in the domain of x. This is only feasible when the domains of the universally quantified variables are small.

Negating the problem turns the conjunction of constraints into a disjunction, which MIP cannot deal with efficiently, and it causes global CP constraints to be decomposed, preventing the usage of more efficient propagation algorithms on the part of the CP solver.

SAT cannot deal with universal constraints natively, but extensions of SAT like QBF can. Many SMT solvers support universal quantification ($\forall$) natively. Both SAT and SMT are built to deal with disjunctions very efficiently, so negating the original problem and checking for unsatisfiability is no problem for these types of solvers as well.

2.3 Formal Requirement Verification

Formal requirement verification is a systems (and software) engineering process used to ensure that the specification of a system is *consistent*, *complete*, and *unambiguous*, and that an implementation follows that specification. By consistent we mean that there are no conflicting requirements; by complete we mean that there are no undefined behaviors that would allow for incorrect implementations; by unambiguous we mean that there is only one possible interpretation of the requirements.

Formal verification is particularly valuable in critical domains like aerospace, healthcare, and automotive, where mistakes in the requirements or in the system's implementation of the requirements may lead to catastrophic outcomes.

The "formal" aspect involves the use of mathematically rigorous methods, which means that specifications must be written in some *formal language*. There are many formal specifications languages, though most focus on software verification. Languages that focus on system-level specifications include Event-B [15], FORM-L [100], and SMV [37].

Given a formal requirement specification and a black-box simulation model of an implementation, the implementation's satisfaction of the requirements can be evaluated through test-case generation and monitoring [47]. This approach is *incomplete* in that while it may detect incorrect behavior, it cannot prove that the implementation satisfies the requirements for all possible states.

A more complete formal verification techique is *model checking*. Model checking requires both a formal specification (usually a temporal logic formula [115]) and a formal model of an implementation. Given a formal specification (p), a formal model of an implementation (M), and an initial state (s), model checking amounts to deciding if:

$$M, s \vDash p \tag{2.4}$$

If M is a finite-state machine, for example, model-checking can be achieved by exhaustively exploring all possible states. Much like with CSPs, the state space grows exponentially with the size of the problem, so more clever techniques are required in practice.

Symbolic Model Checking [37] translates the problem into a BDD or into a SAT (or SMT) problem which is then solved [37]. The latter means the same techniques employed to efficiently solve SAT problems can be used for model checking.

More concretely, SAT-based model checking is achieved through Bounded Model Checking (BMC) [83], and it works by unrolling and solving the formula at each time-step until an incorrect state is detected. By itself BMC is incomplete, since in the case of a correct implementation it will keep unrolling the formula forever. However, additional techniques like k-induction [118] and Property Directed Reachability (PDR) [51] / IC3 [34] can be used to ensure completeness.

For architecture generation, we need separate formal models for each individual component, rather than a model for the entire implementation (which would require an already complete architecture). Highly detailed models of the components are not necessary, we only need to know what the component is supposed to do, i.e., we only need the component's own *requirements*.

The state-of-the-art for modular requirement specifications is Contract-Based Design (CBD) [24], which has been proposed as a way to tackle the safe integration of independent designs. CBD allows for compositional reasoning, refinement, and a principled reuse of preexisting components.

In CBD each component (including the system itself) is associated with a contract, a formal description of its expected behavior (i.e., its requirements). Through CBD, it is possible to perform virtual integration testing using only contract specifications. Meaning if the integration of the components passes the test (based solely on their contracts), then *any* concrete implementation of a component compliant to its contract will not invalidate the outcome of the test.

A very useful result of these capabilities is that, at any level of the supplier hierarchy, the higher-level organization can (prior to contracting suppliers) analyze whether the various subsystem contracts pass the virtual integration test, and are sufficient to satisfy the system requirements. This fits perfectly with our intention of "shifting-left" the evaluation of the system's design.

There are many possible ways to define contracts, but this thesis is focused solely on *Assume/Guarantee* (AG) contracts [24]. AG contracts specify the input-output behavior of a component by defining what the component must do (the guarantees), provided that its environment obeys some conditions (the assumptions).

An AG contract (C) is defined as a tuple of Assumptions (A) and Guarantees (G):

$$C = (A, G) \tag{2.5}$$

Both the assumptions and the guarantees are conjunctions of logical formulas. These formulas constrain the supported environments (assumptions) and the possible implementations (guarantees) of this contract. Assumptions and guarantees are roughly equivalent to the concepts of pre-conditions and post-conditions used in software engineering.

The basic idea is to view each component as an independent system (ensuring modularity), compose their contracts, and then see if the composition is a refinement (i.e., valid implementation) of the system contract.

To accomplish this, CBD theory specifies, in a formal manner, a set of operations on contracts:

- **Composition** ($\otimes$) returns the contract of connected components.

- **Conjunction** ($\wedge$) combines multiple "viewpoints" (contracts) of the same component, for example safety and performance considerations.
- **Refinement** ($\preceq$) allows for independent implementation; a contract refines another contract if its assumptions are the same or less strict, and it provides the same or more guarantees.
- **Quotient** ($/$), given a system-wide contract C and a contract C_1 that specifies pre-existing components and their interactions, returns the part of the system-wide contract that still needs to be implemented. It formalizes the practice of "patching" a design to make it behave according to another contract.

The operations are defined as follows:

$$\begin{aligned}
(A_1, G_1) \otimes (A_2, G_2) &\coloneqq (G_2 \to A_1 \wedge G_1 \to A_2, G_1 \wedge G_2) \\
(A_1, G_1) \wedge (A_2, G_2) &\coloneqq (A_1 \vee A_2, G_1 \wedge G_2) \\
(A_1, G_1) \preceq (A_2, G_2) &\coloneqq A_1 \to A_2 \wedge G_2 \to G_1 \\
(A_1, G_1) \,/\, (A_2, G_2) &\coloneqq (A_1 \wedge G_2, (A_2 \wedge G_1) \vee \neg(A_1 \wedge G_2))
\end{aligned} \tag{2.6}$$

The quotient operation for AG contracts was unknown at the time of the original specification [24], but a solution was later discovered [67].

An important aspect of AG contracts is that assumptions (A) *must* hold. It is not enough that if they do not hold, then the corresponding guarantees do not hold. In CBD those are called *weak* assumptions and they are modeled as implications inside G. Assumptions in A are called *strong* assumptions and if they do not hold then the system implementation is invalid.

Additionally, there are two important contract properties:

- **Compatibility** means the assumptions of a contract do not result in an empty set (i.e., there is at least one environment where the contract is applicable).
- **Consistency** means guarantees of a contract are not contradictory (i.e., there is at least one component that can implement this contract).

These two properties, compatibility and consistency, are simply checks that assumptions and guarantees have at least one solution, respectively. If that is not the case, then the contract is either unusable or not implementable.

Before applying each operation, each contract must be converted to *saturated* form, where each G is first replaced by $A \to G$ [24].

For an example of composition, consider the following two components: Component `SQRT` returns the square root of its input, which must not be negative. Component `ABS` returns the absolute value of its input.

The contracts of the two components are as follows:

- $\texttt{SQRT} = (i \geq 0, o \times o = i)$
- $\texttt{ABS} = (\top, (i < o \to o = -i) \wedge (i \geq 0 \to o = i))$

First, we must convert the contracts to saturated form, but since `ABS` has no assumptions we only need to do this for `SQRT`:

- $\texttt{SQRT} = (i \geq 0, (i \geq 0) \to (o \times o = i))$

Second, we must rename their variables to distinguish the ones that belong to `SQRT` and those that belong to `ABS`. We will use a simple suffix in this example:

- $\texttt{SQRT} = (i_s \geq 0, (i_s \geq 0) \to (o_s \times o_s = i_s))$
- $\texttt{ABS} = (\top, (i_a < 0 \to o_a = -i) \wedge (i_a \geq 0 \to o_a = i_a))$

We can then compose the two contracts as follows:

$$
\begin{aligned}
(A_\otimes, G_\otimes) &= \texttt{SQRT} \otimes \texttt{ABS} \\
\texttt{SQRT} &= (A_s, G_s) \\
\texttt{ABS} &= (A_a, G_a) \\
A_\otimes &= (G_a \to A_s) \wedge (G_s \to A_a) \\
G_\otimes &= G_s \wedge G_a \\
A_s &= i_s \geq 0 \\
G_s &= (i_s \geq 0) \to (o_s \times o_s = i_s) \\
A_a &= \top \\
G_a &= (i_a < 0 \to o_a = -i_a) \wedge (i_a \geq 0 \to o_a = i_a)
\end{aligned}
$$

We must now unify any variables that are connected in the composition. In this case we are connecting the output of `ABS` (o_a) to the input of `SQRT` (i_s). We replace all instances of those two variables with a new variable c_{as} representing the connection. After some minor simplication the final contract is as follows:

$$A_{\otimes} = ((i_a < 0 \rightarrow c_{as} = -i_a) \wedge (i_a \geq 0 \rightarrow c_{as} = i_a)) \rightarrow (c_{as} \geq 0)$$

$$G_{\otimes} = (o_s \times o_s = c_{as}) \wedge (i_a < 0 \rightarrow c_{as} = -i_a) \wedge (i_a \geq 0 \rightarrow c_{as} = i_a)$$

Variable c_{as} is an *internal* variable, an implementation detail of the component. CBD makes no distinction between variables that are internal to the component and those that are externally exposed, but this variable will not be unified again. Notice also that the resulting assumption is a tautology: it is true for all values of i_a and c_{as}. The guarantees of `ABS` ensure the assumptions of `SQRT`.

In the case of refinement, the formula has to hold for all possible inputs and states. Proving that the refinement relation of AG contracts holds for all possible values can be done by hand through standard mathematical proof methods, but that obviously does not scale.

Thankfully the process can be automated through model checking, which in turn can be turned into CSPs. This is what enables the algorithm in this thesis to use requirements to guide the search for architectures. The results of the requirement verification CSP can be used to reduce the search space of the architecture generation CSP.

Chapter 3

System Contract Assignment Language

> *"The single biggest problem with communication is the illusion that it has taken place."*
>
> —*George Bernard Shaw*

Requirements in systems engineering are almost always written in natural language. For a computer algorithm to make use of requirements, they must be formalized. Going from natural language requirements to formal requirements is difficult, and for the purposes of this thesis we will assume that this translation is done at some point in the design process by an expert. We discuss technologies that help with this task in the future work (Chapter 7).

Another issue is that the systems we are concerned with are often *hybrid systems*, meaning systems that display a combination of discrete and continuous behavior. Verification of hybrid systems is undecidable in general [117], so for simplicity this thesis is only concerned with the discrete time case. Extensions towards more expressive requirements are also discussed in the future work(Chapter 7).

To perform generative engineering guided by requirements, we need a hierarchical decomposition of the system and formal specifications of the requirements for each component (including the system itself), which we refer to as their *contracts*.

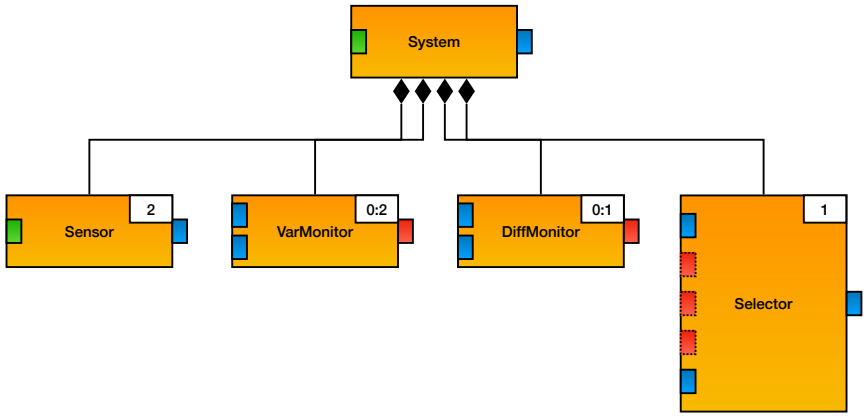


Figure 3.1: The system decomposition of the redundant sensor.

3.1 System Contract Assignment Language

We developed a language called SCALe (System Contract Assignment Language) to model both the system decomposition and the contracts. SCALe is inspired by Simcenter Studio's ACEL language [9] for the specification of the decomposition and by the OCRA System Specification language [38] for the contract specifications.

As a running example, consider a simple sensor with internal redundancy adapted from one of the example use cases from OCRA [4].

This sensor system measures some physical process (e.g., temperature, pressure, etc.) and translates the measurement to a digital value for use by another component or for display. Because a single sensor component might "glitch" and produce incorrect measurements, two identical internal sensors are used in a redundant configuration to enable error detection and correction.

A visual representation of the sensor's system decomposition can be seen in Figure 3.1.

The system decomposition defines the system interface, the components that form the implementation of the system, the interfaces of those components, and a cardinality for each component and each port.

The *interface of a component* is defined in terms of ports with different types and different directions: input, output, and bidirectional (io). The direction of a port is visually represented by its position in relation to the component block: input ports are placed entirely within the block, output ports entirely outside the block, and bidirectional ports are placed crossing the boundary, halfway in halfway out (not visible in the example). Different types are represented visually by different colors, but colors and types have no relation otherwise.

The *cardinality of a component* defines how many instances of that component (as a range) may be present in the system, as a range. It is depicted in the top-right corner of the block. If the upper and lower bound are the same, only a single number is depicted. If the upper and lower bound are both 1, then it is the default and is not depicted.

The *cardinality of a port* also defines how many instances of a port may be present in the component, but the graphical representation is different from component cadinalities. Each possible instance of the port is depicted, with optional instances represented with a dotted outline. The number of optional instances is equal to the difference between the upper bound and the lower bound of the port's cardinality.

Cardinalities are necessary to keep the problem bounded and ensure termination. Setting a cardinality for a component or a port is different from separately modelling multiple components or ports with the same interface, because instances produced from the same component or port are considered to be *interchangeable*. Architectures that differ only in how interchangeable components are connected can be safely discarded (more on this in Chapter 4).

The system is decomposed as follows:

- **Sensor:** the internal sensor component, which may fail. Exactly two instances of this component are to be present in the system, to ensure redundancy.
- **Variance Monitor:** the variance monitor checks if its sensor input differs by a large amount from the last output of the entire sensor system. Up to two such monitors may be present in the system, one for each internal sensor.
- **Difference Monitor:** the difference monitor checks if its two inputs have diverged. At most one difference monitor may be present in the system to check if the two internal sensors are giving different results.

- **Selector:** the selector picks one of its inputs, or averages them, depending on the results it obtains from the sensors, as error correction. If a monitor port in the selector is not connected, its value is assumed to be false.

As you can see in the model, all monitors are optional (lower cardinality bound of 0). This is because we expect the synthesis process to discover how many monitors, and of which type, are necessary to ensure the system requirements.

The red ports of the selector have a dotted line (lower cardinality bound of 0) to indicate they are optional. That is because those ports are meant to connect to the monitors, which may not be present.

This model, together with the formal requirements of each component, can be specified in SCALe. SCALe is designed with architecture generation in mind, so components and ports allow for variability specifications and connections are not explicitly defined (though they can be constrained). The grammar for SCALe is shown in Figure 3.7.

Contracts in SCALe specify the interface a component must have (which ports and attributes the contract expects), plus its assumptions and guarantees. A component may then be defined to satisfy a contract. If a component satisfies a contract, then it inherits the contract's interface. Multiple contracts would require the component to define its own interface and explicitly map its own ports to the ports defined in the contracts, so for simplicity we assume a single contract [1].

The contract and decomposition of the sensor system can be seen in Figure 3.2. The system has three ports:

- **reading:** this input port represents the physical reading of the environment. The value transmitted through this port is an integer value between -15 and 15, but the port is given the distinct type "analogue" to distinguish the kind of value it transmits from the digital output of the system. Only ports of the same type may connect to one another, so this introduces a structural constraint.
- **fail:** this port (which has 2 instances) is a virtual port used for fault injection. Each instance transmits a boolean signal which triggers an internal failure on the internal component it connects to.

[1] Extending SCALe to multiple contracts per component would be relatively easy to do, it would just make the examples larger and harder to explain. Multiple contracts can be merged into a single contract using the contract union operator ($\wedge$), leaving the implementation of the synthesis algorithm unchanged

```
1  let max_variance = 1
2  let max_sensor_error = 2
3  let max_sys_error = 4
4
5  type analogue = int[-15:15]
6  type digital = int[-15:15]
7
8  contract SystemReq:
9    ports:
10     in::analogue reading
11     in::bool fail [2]
12     out::digital output
13
14   assumptions:
15     !any(fail)
16     always (abs(reading - next(reading)) <= max_variance)
17     always (!all(fail))
18     always (any(fail) implies next(!any(fail)))
19
20   guarantees:
21     always (abs(reading - next(output)) <= max_sys_error)
22
23 component System satisfies SystemReq:
24   owns:
25     Sensor sensor [2]
26     VarMonitor vm [0:2]
27     DiffMonitor dm [0:1]
28     Selector selector [1]
29
30   connections:
31     each sensor.output -- * [1:3]
32     each selector.output -- * [1:3]
33     each vm.valid -- * [1:1]
34     each gm.valid -- * [1:1]
```

Figure 3.2: The SCALe specification of the redundant sensor system.

- **output:** is the final output of the system after processing by the internal sensors. While the reading is analogue the output is digital.

Only one contract is defined for the system, *SystemReq*, which has four assumptions and one guarantee.

- The first assumption, line 15, enforces that no failure happens in the system right at the moment it begins to operate. This is necessary for internal monitors to know what a "good" state of the system looks like. If no temporal constraint (like "always") is given, the assertion only has to hold in the first instant, which is the case here.
- The second assumption, line 16, asserts that the variance of the physical reading changes slowly over time. This assumption is necessary to ensure the variance monitors work as intended.

- The third assumption, line 17, asserts that at most one failure occurs in the system at any point in time. If both internal sensors fail the system cannot operate correctly.
- The fourth assumption, line 18, asserts that failures are ephemeral. If in one instant a failure occurs, it is assumed to clear up on the next instant, i.e., the only failures for which the system is resilient against are temporary glitches.
- Finally, the single guarantee, line 21, asserts that the sensor output is within `max_sys_error` of the physical reading. The internal sensors introduce small measurement errors even when functioning normally, and the error further increases if one of them fails.

The component specification decomposes the system into its constituents, with variability constraints on the number of times each of those sub-components can be instantiated. Two internal sensors and a selector component must always be present, but the number of internal monitors is variable.

The goal is to find how many internal monitors are necessary to ensure the correct operation of this sensor system and how to connect them to ensure that the system requirements are fulfilled.

We also add some connections constraints that enforces a limit of 3 connections on the output ports of the internal sensors and of the selector, and limit of 1 connection for the monitors. These structural constraints reduce the search space of possible architectures. If no constraint is given, output ports are allowed to connect any number of times (minimum 1).

The contract for the internal sensor is shown in Figure 3.3. The guarantee of the sensor simply states that if there is no failure, the output of the sensor is the same as the reading but with a slight error introduced. If there is a failure, it cannot guarantee anything.

The contract for the variance monitor can be seen in Figure 3.4. Remember that assumption 2 of the system asserts that the external readings must vary slowly over time, which is why a variance monitor works for this particular system. Note that the output of the monitor is whether the monitored component is in a valid state, not that it has failed.

The contract for the difference monitor is shown in Figure 3.5. It should be self-explanatory, it simply checks if the two internal sensors don't diverge too much from one another.

Finally, the contract for the selector is shown in Figure 3.6. The selector is expected to receive the outputs from the two internal sensors and either: select

```
contract SensorReq:
  ports:
    in::analogue input
    in::bool fail
    out::digital output

  guarantees:
    always (!fail implies (abs(input - output) <= max_sensor_error))

component Sensor satisfies SensorReq
```

Figure 3.3: The SCALe specification of the internal sensor.

```
contract VarMonitorReq:
  ports:
    in::digital input
    in::digital last
    out::bool valid

  guarantees:
    always (valid == (abs(input - last) <= 2*max_sensor_error + 1))

component VarMonitor satisfies VarMonitorReq
```

Figure 3.4: The SCALe specification of the variance monitor.

one of them, calculate the average of the two, or return the last value, based on the outputs of the various internal monitors.

The vm1, vm2 and dm ports of the selector are meant to connect to the two variance monitors and the difference monitor, respectively. They are marked as optional (a cardinality of 0 to 1), because the monitors are not required to be present in the system, so the ports may end up unconnected.

The guarantees of the selector are somewhat complex, so I will go over them in detail:

```
contract DiffMonitorReq:
  ports:
    in::digital in1
    in::digital in2
    out::bool valid

  guarantees:
    always (valid == (abs(in1 - in2) <= 2*max_sensor_error))

component DiffMonitor satisfies DiffMonitorReq
```

Figure 3.5: The SCALe specification of the difference monitor.

```
1  contract SelectorReq:
2    ports:
3      in::digital sensor1
4      in::digital sensor2
5      in::bool vm1 [0:1]
6      in::bool vm2 [0:1]
7      in::bool dm [0:1]
8      out::digital output
9
10   guarantees:
11     always (dm implies next(output) = (sensor1 + sensor2) / 2)
12     always ((!dm and !vm1 and vm2) implies next(output) == sensor2)
13     always ((!dm and !vm2 and vm1) implies next(output) == sensor1)
14     always (((!dm and !vm1 and !vm2) or (!dm and vm1 and vm2))
15       implies next(output) == output)
16
17 component Selector satisfies SelectorReq
```

Figure 3.6: The SCALe specification of the selector.

- The first guarantee, line 11, asserts that if the difference monitor says the two monitors are giving similar results, the output of the selector should be the average of the two.
- The second guarantee, line 12, asserts that if the two sensors have diverged, and the output of sensor 1 has diverged greatly from the last measurement, then the output should follow sensor 2.
- The third guarantee, line 13, similarly asserts that if the two sensors have diverged, and the output of sensor 2 has diverged greatly from the last measurement, then the output should follow sensor 1.
- The fourth and last guarantee, line 14, asserts that if every monitor detects an error, the output of the sensors cannot be trusted, and the safest option is to simply return the last measurement, since the readings vary slowly over time.

3.1.1 SCALe Syntax

The grammar for SCALe is shown in Figure 3.7. The purpose of each grammar production is defined below:

- **Model**: The top level rule which encompasses the whole SCALe model. A SCALe model contains zero or more bindings, type definitions, contracts, components, and connection constraints.

⟨*model*⟩ ::= (⟨*binding*⟩ | ⟨*typedef*⟩ | ⟨*contract*⟩ | ⟨*component*⟩)*

⟨*binding*⟩ ::= `let` ⟨*id*⟩ `=` ⟨*expression*⟩

⟨*typedef*⟩ ::= `type` ⟨*id*⟩ `=` ⟨*id*⟩ ⟨*range*⟩?

⟨*interface*⟩ ::= ⟨*attributes*⟩? ⟨*ports*⟩?

⟨*contract*⟩ ::= `contract` ⟨*id*⟩ (`:` ⟨*interface*⟩ ⟨*assumptions*⟩? ⟨*guarantees*⟩?)?

⟨*attributes*⟩ ::= `attributes :` ⟨*attribute*⟩+

⟨*attribute*⟩ ::= ⟨*typename*⟩ ⟨*range*⟩? ⟨*id*⟩

⟨*ports*⟩ ::= `ports :` ⟨*port*⟩+

⟨*port*⟩ ::= (`in` | `out` | `io`) `::` ⟨*typename*⟩ ⟨*range*⟩? ⟨*id*⟩ ⟨*range*⟩?

⟨*assumptions*⟩ ::= `assumptions :` ⟨*expression*⟩*

⟨*guarantees*⟩ ::= `guarantees :` ⟨*expression*⟩*

⟨*component*⟩ ::= `component` ⟨*id*⟩ (satisfies ⟨*id*⟩) (`:` ⟨*interface*⟩ ⟨*connections*⟩?)?

⟨*owns*⟩ ::= `owns :` (⟨*id*⟩ ⟨*id*⟩ ⟨*range*⟩?)*

⟨*connections*⟩ ::= `connections :` ⟨*connection*⟩*

⟨*connection*⟩ ::= (`each` | `all`) ⟨*endpoint*⟩ `-` ⟨*endpoint*⟩ ⟨*range*⟩

⟨*endpoint*⟩ ::= ⟨*id*⟩(`.` ⟨*id*⟩)? | `*`

Figure 3.7: The SCALe grammar.

- **Binding**: A binding is just a name given to a constant value, used in the redundant sensors example to define the maximum acceptable error for the sensors. The expression being bound is evaluated when the model is compiled and may not contain any variables. See line 1 of Figure 3.2 for an example.
- **Interface**: The attributes and ports of a contract or component.
- **Typedef**: Type definitions create distinct types, which are used to restrict allowed connections between ports. Each type definition is backed by a base type: bool, int, or real. Only the base type matters in the contracts. Type definitions may also define a range of allowed values. This range may be overriden when assigning a type to a port. See line 5 of Figure 3.2 for an example.
- **Contract**: Contracts are the main concept in SCALe, they specify the expected interface of a component and their assumptions and guarantees. See line 8 of Figure 3.2 for an example.
- **Attributes**: An optional block defining the set of internal attributes of an interface. Not used in the examples. It's the same as the ports block, but for internal attributes instead of external ports.
- **Attribute**: Attributes are internal properties. They have a type, a range of allowed values and a name. If not specified the range of values is infinite. There is currently no way in SCALe to define parameters (constant attributes) or to relate the attributes of a system with those of its sub-components. How the latter can be achieved is discussed in the future work (Chapter 7).
- **Ports**: An optional block defining the set of ports of an interface. See line 9 of Figure 3.2 for an example.
- **Port**: Ports are like attributes but they also include a direction (input, output, bidirectional). See line 10 of Figure 3.2 for an example.
- **Assumptions**: Contracts can have zero or more assumptions about the environment in which a component that satisfies this contract meant to function. If a contract has more than one assumption this is equivalent to a conjunction of those assumptions. If no assumptions are defined, the assumption of a contract is just the constant `true`. See line 14 of Figure 3.2 for an example.
- **Guarantees**: Contract can have zero or more guarantees specifying what a component that satisfies the contract is supposed to do. Like assumptions, multiple guarantees are just a conjunction, and no guarantees are the constant `true`. See line 20 of Figure 3.2 for an example.

- **Component**: Components may satisfy a contract, may specifiy attributes and ports, and may specify a hierarchical decomposition into other components. If attributes and ports are not given they are inherited from the contract. See line 23 of Figure 3.2 for an example.
- **Owns**: The owns section of a component specification dictates which sub-components, and how many instances of each sub-component, appear within it. If the range is not specified it is assumed to be exactly one instance. See line 24 of Figure 3.2 for an example.
- **Connections**: The connections section of a component specification defines connection constraints for its sub-components. See line 30 of Figure 3.2 for an example.
- **Connection**: A connection constraint limits the number of connections between two endpoints. The constrain can apply to *each* instance of a component individually, or to *all* instances of a component put together. If a port is given, only connections involving that port are considered. The right-hand side is always treated as *all*. See line 31 of Figure 3.2 for an example.
- **Endpoint**: An endpoint is either a component, a port of a component, or a wildcard ($*$). The wildcard refers to components, not ports. An endpoint of the form $c.*$ would be redundant since it would mean the same as just c.

Certain parts of the SCALe grammar – namely types, identifiers and expressions – are shared with another a language, SCL (SCALe Constraint Language), which is used as an intermediate compilation target to encode the synthesis problem, before passing it to a backend solver. Because of this the expression grammar production is defined in the SCL gammar. SCL is introduced below in section 3.2.

3.1.2 SCALe Semantics

The most important element of a SCALe model is a component that owns other components. Such component specifications relate a higher-level contract with the contracts of subcomponents, forming a contract *refinement*. The input to the synthesis process (Chapter 4) is not an entire SCALe model but a specific refinement.

Figure 3.8 shows the internal structure of a SCALe refinement. The contents of that structure are described below.

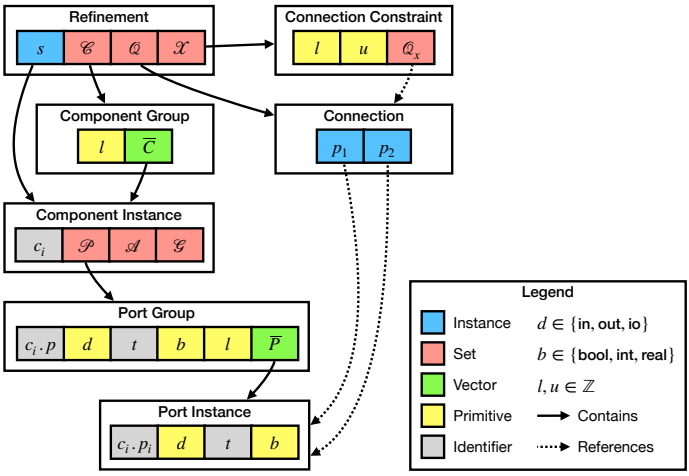


Figure 3.8: The internal structure of a SCALe refinement. Refinements contain component groups, connections and connection constraints. Component groups contain component instances. Component instances contain port groups. Port groups contain port instances. Connections contain pairs of port instances. Connection constraints contain connections.

Semantically, a refinement is a tuple $\langle s, \mathcal{C}, \mathcal{Q} \rangle$ where s is an instantiated top-level component, $\mathcal{C}$ is a set of component groups and $\mathcal{Q}$ is a set of connections.

A component group is a tuple $\langle l, \overline{C} \rangle$ where l is an integer specifying the lower bound on the number of instances of that component, and $\overline{C}$ is a vector of component instances. The lower bound l and the length of the vector $\overline{C}$ are determined by the lower and upper bounds of the range specified for that component in the refinement specification.

A component instance is a tuple $\langle c_i, \mathcal{P}, \mathcal{A}, \mathcal{G} \rangle$ where c_i is the identifier for this component instance, c referring to the component name and i referring to its position in vector $\overline{C}$. $\mathcal{P}$ is a set of port groups, $\mathcal{A}$ is a set of instantiated assumptions and $\mathcal{G}$ is a set of instantiated guarantees. Attributes, while possible to define in the grammar, are not implemented.

A port group is a tuple $\langle c_i.p, d, t, b, l, \overline{P} \rangle$, where $c_i.p$ the identifier for this port group (the component instance identifier followed by a dot and the port name), $d \in \{\textbf{in}, \textbf{out}, \textbf{io}\}$ is the port's direction, t is an identifier corresponding to the declared SCALe type name, $b \in \{\textbf{bool}, \textbf{int}, \textbf{real}\}$ is the base type, l is an integer specifying the lower bound on the number of instances of this port, and $\overline{P}$ is a

vector of port instances.

A port instance is a tuple $\langle c_i.p_j, d, t, b\rangle$, where $c_i.p_j$ is the identifier for this port instance (the component instance identifier followed by a dot, the port name, and j referring to its position in vector $\overline{P}$), and d, t and b are the same as they are in the port group.

In the case of the system component instance, when its ports are instantiated, the direction of input and output ports are reversed (**in** $\rightarrow$ **out**, **out** $\rightarrow$ **in**). This is to allow for all connections to be treated as if happening between components. At the system to subcomponent level, ports are not actually connected but rather mapped to ports with the same direction. Flipping input and output ports allows to treat these mappings as any other connections.

Instantiated assumptions (A) are conjunctions of the assumptions in the original contract, but with every reference to a port p replaced by the corresponding instantiated port $c_i.p$, which will in turn be unrolled when the instantiated expression is compiled to SCL. The unrolling depends on the specific operator applied to p. Ports with a cardinality upper bound greater than 1 are given a special "array" type in SCALe and are only allowed inside operators like *sum* that have a well defined unrolling, more on that in Chapter 4. SCL has no notion of array types.

Instantiated guarantees are the same, but with the formulas in saturated form [24], meaning they are rewritten as $\langle A \rightarrow G\rangle$, where G is the conjunction of the guarantees with the port references updated, A is the conjunction of the assumptions with the port references updated, and $\rightarrow$ is an implication. This transformation is required by the contract metatheory.

Each connection is a pair of port instances $\langle p_1, p_2\rangle$. Connections are stored in the set $\mathcal{Q}$, which is the union of all sets $\mathcal{Q}_t$, where $t \in \mathcal{T}$, and $\mathcal{T}$ is the set of all port type names present in the ports of components in the refinement.

Each $\mathcal{Q}_t$ set is obtained by first collecting all port instances into separate buckets $\mathcal{P}_{t,d}$ based on their type name t and direction d. For each type name t, the set of connections $\mathcal{Q}_t$ for that type name is computed as follows: $\mathcal{Q}_t = \{\langle, p_1, p_2\rangle \mid p_1 \in \mathcal{P}_{t,\mathbf{out}}, p_2 \in \mathcal{P}_{t,\mathbf{in}}\} \cup \{p_1 \diamond p_2 \mid p_1, p_2 \in \mathcal{P}_{t,\mathbf{io}}, p_1 \neq p_2\}$.

Each connection constraint is a tuple $\langle l, u, \mathcal{Q}_x\rangle$ where l and u are the lower and upper bound on the number of connections enforced by this constraint, while $\mathcal{Q}_x$ is the subset of connections that are *relevant* for the constraint.

A connection is considered *relevant* for the constraint, if the two port instances it references belong to either of the components or ports referenced in the SCALe model constraint (a wildcard accepts all connections). If the constraint is defined as `each`, then separate connection constraints are created for each instance

of the component, while if the constraint is defined as `all`, then connections containing ports from all instances of the component are bundled together in the same connection constraint structure.

These structures are then compiled by the algorithm in Chapter 4 into SCL constraints. SCL is introduced below.

3.2 SCALe Constraint Language

The SCALe Constraint Language (SCL) is a constraint modeling language inspired by MiniZinc [99], but with support for temporal operators based on Linear Temporal Logic (LTL) [115]. LTL is a modal logic with modalities referring to time, particularly future points in time. Past Linear Temporal Logic (PLTL) extends LTL with modalities referring to past points in time. In this thesis we focus on LTL but PLTL support is a simple extension [84].

An LTL formula consists of a set of propositional variables, logical operators and temporal operators. We extend LTL to support mathematical equations in addition to propositional variables.

The semantics of LTL are defined over an infinite path π of states s: $\pi = s_0, s_1, \ldots$, where each state s_i is a set of the values of each variable at time instant i. More on the semantics of LTL and SCL below.

3.2.1 SCL Syntax

The grammar for SCL is shown in Figure 3.9. Grammar atoms like **integer**, **real**, **id** (identifier) and the two boolean constants have their usual meaning. The purpose of each grammar production is defined below:

- **Script**: An SCL problem is defined as a script, with a series of variable declarations followed by constraints on those variables.
- **Declaration**: Declarations introduce the problem variables. The keyword *exists* introduces an existentially quantified variable, while the keyword *forall* introduces a universally quantified variable (SCL only allows a single quantifier alternation). The optional keyword *constant* means "does not change over time". SCL has no equivalent to let bindings in SCALe as it is meant as a compilation target and not a user facing language. Like with SCALe attributes and ports, SCL declarations can have an optional

⟨*script*⟩ ::= ⟨*declaration*⟩* ⟨*constraint*⟩*

⟨*declaration*⟩ ::= (`exists` | `forall`) `constant`? ⟨*type*⟩ ⟨*range*⟩? ⟨*id*⟩

⟨*constraint*⟩ ::= (`constrain` | `prove`) ⟨*expression*⟩

⟨*type*⟩ ::= `bool` | `int` | `real`

⟨*range*⟩ ::= `[` ⟨*real*⟩ (`:` ⟨*real*⟩)? `]`

⟨*expression*⟩ ::= ⟨*atom*⟩
| (`not` | `-`) ⟨*atom*⟩
| `abs (` ⟨*expression*⟩ `)`
| (`all` | `any` | `min` | `max` | `sum` | `product`) `(` (⟨*id*⟩ | ⟨*array*⟩) `)`
| ⟨*expression*⟩ (`and` | `or` | `xor` | `xnor` | `implies`) ⟨*expression*⟩
| ⟨*expression*⟩ (`+` | `-` | `*` | `/` | `%`) ⟨*expression*⟩
| ⟨*expression*⟩ (`==` | `!=` | `<` | `>` | `<=` | `>=`) ⟨*expression*⟩
| ⟨*lexicographic*⟩
| ⟨*temporal*⟩

⟨*array*⟩ ::= `[` ⟨*expression*⟩ (`,` ⟨*expression*⟩)* `]`

⟨*lexicographic*⟩ ::= `lex` (`<` | `>` | `<=` | `>=`) `(` ⟨*array*⟩, ⟨*array*⟩ `)`

⟨*temporal*⟩ ::= `next` ⟨*atom*⟩
| (`always` | `eventually`) ⟨*range*⟩? ⟨*atom*⟩
| ⟨*atom*⟩ (`until` | `release`) ⟨*range*⟩? ⟨*atom*⟩

⟨*atom*⟩ ::= `true` | `false` | ⟨*integer*⟩ | ⟨*real*⟩ | ⟨*id*⟩ | `(` ⟨*expression*⟩ `)`

Figure 3.9: The SCL grammar.

range defining their domain. For example you could define a variable as follows: `exists constant real system_weight`.

- **Constraint**: Constraints define the problem to be solved in terms of the declared variables. If the keyword *prove* is used then this is a universal constraint and universally quantified variables can appear in the expression. A constraint must be a boolean expression. If no temporal operator is given, the constraint is assumed to hold at the first time instant. A constraint could be for example: `constraint system_weight < 400.0`
- **Type**: SCL supports only 3 base types for boolean, integer and real domains respectively. There is no way to define more complex types in SCL. They are defined as: `bool`, `int`, and `real`, respectively.
- **Range**: Ranges in SCL must have a start value and may have an optional end value. If the end value is not given, it is assumed to be positive infinity ($+\infty$). If a construct that expects an optional range is not given one, this is the same as passing it the unbounded range $[-\infty, +\infty]$. An example of a range would be `[0:100]`.
- **Array**: Arrays bundle a sequence of expressions together, they are used by n-ary operators.
- **Lexicographic**: Special operators used for symmetry breaking. Symmetry breaking is explained in Chapter 4.
- **Expression**: Expressions in SCL are a mix of boolean, arithmetic and temporal operations. Most operators should be self-explanatory, except % which is the modulo operator and ! = which is the not-equal operator. The *all* and *any* operators model n-ary conjunctions and disjunctions, respectively. The temporal operators and are explained below. Note that operators like *sum* that accept an *id* or an array only actually accept arrays when typechecked. The *id* case is due to SCALe reusing SCL expressions, and in SCALe ports with a cardinality greater than 1 have a special array type.
- **Temporal**: SCL supports all LTL operators [115]:
  - **next** x: Formula x must hold on the next time instant. Same as the **X** (neXt) operator in LTL. The next operator may also be applied to variables (of any type), to refer to the value of that variable on the next time instant.
  - **always** x: Formula x must hold now and on all future time instants. Same as the **G** (Globally) operator in LTL.

- **eventually** x: Formula x must hold now or on at least one future time instant. Same as the **F** (Future) operator in LTL.
- x **until** y: Formula y must hold at some point in the future and x must hold from now until that point. Same as the **U** (Until) operator in LTL.
- x **release** y: Shorthand for $\neg(\neg y\ U\ \neg x)$. Same as the **R** (Release) operator in LTL. Essentially, y must hold now and on all future time instants unless x occurs at some point, in which case y is *released* (is not required to occur from that point onwards).

Temporal operators in SCL also support an optional range, constraining the time bounds to which the operator applies. In SCL these ranges are unrolled into conjunctions or disjunctions of stacked **next** formulas referring to all relevant time instants. This only works if the range has integer bounds, otherwise this translation is not possible. For example `always [0:2] x` becomes `x and (next x) and (next (next x))`.

Ranges of temporal operators are defined in the grammar with real bounds to allow for more powerful temporal logics in the future (see Chapter 7), but non-integer values are rejected in the current implementation.

3.2.2 SCL Semantics

An SCL script defines a CSP [113], which is a tuple $\langle X, D, C\rangle$ where X is a set of variables, D is a set of domains and C is a set of constraints. Unlike most CSPs, constraints in SCL have LTL semantics, so X is split into pure constants X^σ and time-varying variables X^π. An *evaluation* e of an SCL CSP is a pair σ, π where σ is a set of values for the constants in X^σ, and π is an infinite path $\pi = s_0, s_1, \ldots$, where each state s_i in that path is a set of values for the time-varying variables in X^π at time instant i.

An SCL CSP is *satisfiable* if there is an evaluation that satisfies every constraint in C. Due to the finite model property of LTL [115], we can represent an infinite path as a finite witness of length k, by splitting π into a prefix $\pi_p = s_0, \ldots, s_{l-1}$ and a looping sequence $\pi_l = s_l, \ldots, s_k$ where $s_k = s_{l-1}$.

The synthesis algorithm (Chapter 4) takes advantage of this property to unroll the temporal CSP to a non-temporal CSP.

Constraints in SCL can also be universally quantified ($\forall$, *forall*). Universal quantification means the constraint must hold for all possible values of the universally quantified variables, while existential quantification means the

constraint must hold for at least one possible value of the existentially quantified variables. Quantifiers cannot be arbitrarily nested, a SCL script always has a single quantifier alternation ($\exists\forall$).

An SCL CSP is not solved directly, rather it is first translated to a format understood by a backend solver. In this thesis this is done by calling the native API of the backend solver directly, to allow for the efficient incremental introduction of additional constraints.

Backends are not required to support the entirety of SCL (temporal operators in particular are always unrolled first). A feature mismatch may present itself as an error during the translation, or at runtime if the backend can encode the problem but not solve it effectively.

Chapter 4

Requirement Augmented Generative Engineering

> *"Creativity is just connecting things. When you ask creative people how they did something, they feel a little guilty because they didn't really do it, they just saw something."*
>
> —*Steve Jobs*

We saw in the previous chapter how to specify, using SCALe, the *hierarchical decomposition* of a system and its requirements as formal contracts. In this chapter we will develop the algorithm that takes such a decomposition and associated contracts in SCALe as input, and produces SCL constraint problems where the solutions to those constraints represent *architectures* that fulfill the system requirements.

At a very high level, we're trying to solve the following constraint problem:

$$\exists \overline{a}.\, \mathbb{G}(\overline{a}) \wedge \forall \overline{s}.\, \mathbb{V}(\overline{a}, \overline{s}) \tag{4.1}$$

To put it into words, we are trying to find if there exists an assignment to a vector of variables $\overline{a}$, representing the search space of possible architectures of a system, constrained by architecture generation constraints ($\mathbb{G}$), such that contract refinement verification ($\mathbb{V}$) holds for all possible states of that architecture, represented by state variables $\overline{s}$.

This constraint problem is then sent to a constraint solver (or a combination of constraint solvers), to find all possible assignments of $\overline{a}$ that satisfy the constraints, each of which represents an architecture.

The rest of this chapter explains how we solve this synthesis problem, and this chapter is split into the following three sections:

1. The first section 4.1, Generation, explains the contents of $\overline{a}$ and how the architecture generation problem ($\mathbb{G}$) is encoded and solved.
2. The second section 4.2, Verification, explains the contents of $\overline{s}$ and how the contract verification problem ($\mathbb{V}$) is encoded and solved.
3. The third section 4.3, Synthesis, explains how to efficiently solve the two problems together, where contract verification is used guide the architecture generation process towards valid solutions.

We will use the redundant sensor example to illustrate some of the constraints generated throughout the process.

4.1 Generation

There are two main questions we are trying to answer in solving the generation problem: 1) what component instances are present in the architecture, and 2) how those instances are connected. We call the set of component instances in an architecture an *ensemble*. In effect, we split $\mathbb{G}$ into two separate problems: $\mathbb{EG}$ for ensemble generation, and $\mathbb{AG}$ for architecture generation, that we then solve together.

4.1.1 Ensemble Generation

In chapter3 we explained how refinement specifications are transformed into a tuple $\langle s, \mathcal{C}, \mathcal{Q}, \mathcal{X} \rangle$ where s is an instantiated system component, $\mathcal{C}$ is a set of component groups, $\mathcal{Q}$ is a set of connections and $\mathcal{X}$ is a set of connections constraints.

Each component group c in $\mathcal{C}$ is a tuple containing a lower bound l and a set of component instances $\overline{C}$, where each component instance is named c_i, i referring to the instance number.

To figure out which instances c_i are present in an architecture, we can simply model the presence of each instance as a boolean variable, and then enforce that

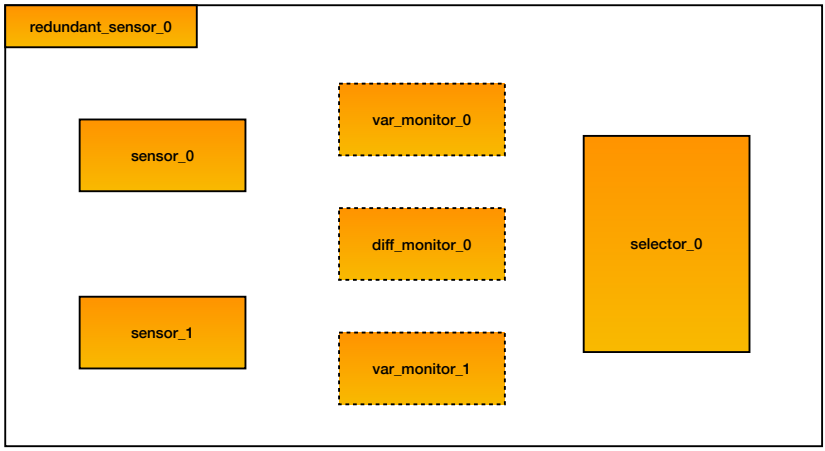


Figure 4.1: The possible components instances for the redundant sensor use case. Each block depicted in the figure maps to a boolean variable representing the presence (or lack thereof) of that component instance in the system. Components with a dotted outline may or may not appear in the architecture, while the others are mandatory.

the sum of the instance variables for a particular component is greater than or equal to that component's lower bound as defined in the model. Additionally, we enforce that higher numbered instances are only present if lower numbered instances are also present. Since instances of the same component are equivalent, there's no reason to have instance 2 but not instance 1.

Figure 4.1 shows a visual representation of ensemble generation variables for the redundant sensor use case, with one variable per component instance.

The algorithm to encode these ensemble generation constraints ($\mathbb{EG}$) is shown in Algorithm 5.

The `exists` function returns an existentialy quantified SCL variable of a given type, associated with some object (like a component instance), adding it to the model if it is not declared already. The **assert** statement adds the given expression as a constraint to the SCL model.

Algorithm 5 The algorithm for Ensemble Generation ($\mathbb{EG}$)

function EnsembleGeneration($\mathcal{C}$)
  for $\langle l, \overline{C} \rangle \in \mathcal{C}$ **do**
    $\overline{e_c} = [\text{exists}(c : \textbf{bool}) \mid c \in \overline{C}]$
    assert $\sum \overline{e_c} \geq l$
    for $c_i \in \overline{C}$ **do**
      if $i \geq 1$ **then**
        assert $\overline{e_c}[i] \rightarrow \overline{e_c}[i-1]$
      end if
    end for
  end for
end function

4.1.2 Architecture Generation

To figure out which connections exist in an architecture, we use a similar idea to Ensemble Generation. Each connection is modeled as a boolean variable indicating its presence in the architecture.

Figure 4.2a shows a visual representation of architecture generation variables for the redundant sensor use case, as a web of connections, with one variable per connection.

Each input port accepts exactly one connection when present, while output ports may connect any number of times by default, but must connect at least once if present.

In the redundant sensor example some output ports are only meant to connect once, while others connect multiple times because there are multiple monitoring devices attached to the same port.

SCALe supports user-defined connection constraints to limit the number of connections of a specific component or port. It is also possible to tell the encoder to enforce one connection per output port as the default. These constraints are handled as bounded sums over subsets of the connection variables.

Unlike with component cardinalities, leaving the number of connections unbounded does not make the problem to unbounded. Because the maximum number of component instances and port instances must be set by the user and is therefore finite, that information can be used to infer the maximum number of connections per port: it is equal to the maximum number of input ports of the same type.

One easy way to reduce the search space of a problem is to use more specific port types. Figure 4.2b exemplifies this, with additional port types greatly reducing the possible connections of the system.

The architecture generation constraints enforce the following:

- If a component instance is not present, then no port instances belonging to that component instance may be present.
- As with the ensemble generation problem, port instances are ordered, and their sum must be equal to or greater than the cardinality lower bound.
- If a port instance is not present, then no connections may be present involving that port instance. The sum of the connection variables must be 0.
- If an input port instance is present, then exactly one connection must be present involving that port instance.
- If an output port instance is present, then at least one connection must be present involving that port instance.

Port groups are stored within the component instances in the set $\mathcal{P}$, each of which has a vector $\overline{P}$ of port instances, while the web of possible connections is stored in the set $\mathcal{Q}$ (see Chapter 3 for more details).

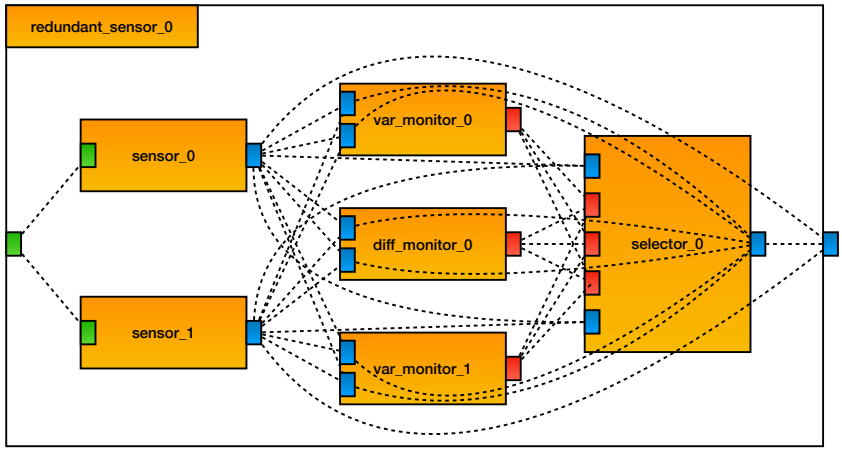
The algorithm to encode these architecture generation constraints ($\mathbb{AG}$) is shown in Algorithm 6. The last loop deals with connections constraints.

Figure 4.3 shows some of the SCL constraints output by Algorithm 6 for the redundant sensor example, in particular the connection constraints for each of the input ports of the sensor.

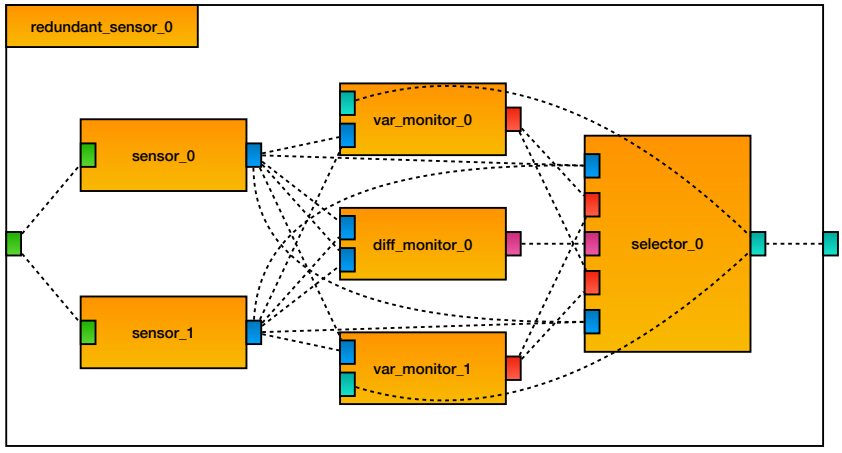
With the Ensemble Generation ($\mathbb{EG}$) and Architecture Generation ($\mathbb{AG}$) constraints combined, we can solve the general Generation problem ($\mathbb{G}$). Each solution $\overline{a}$ of $\mathbb{G}$ contains an assignment to both the component instance variables and to the connection variables, representing the structure of the system.

Using only these constraints, however, the solver will produce many architectures that do not fulfill the requirements, as we're not using them as part of the search in any capacity.

This can still be a valid and even necessary approach, specially when the requirements need detailed simulation or even physical prototypes to verify, but it does not solve the combinatorial explosion problem in the number of generated architectures.



(a) The web of connections for the redundant sensor example. Each line connecting two ports in the figure maps to a boolean variable representing the presence (or lack thereof) of that connection in the system.



(b) The web of connections for the redundant sensor example with more specific port types. The teal and purple port types remove a substantial number of possible connections from the problem.

Figure 4.2: The possible connections of the redundant sensor example. Using more specific port types removes alternatives from the problem.

Algorithm 6 The algorithm for Architecture Generation ($\mathbb{AG}$)

function ArchitectureGeneration($\mathcal{C}, \mathcal{Q}, \mathcal{X}$)
  for $\langle l, \overline{C} \rangle \in \mathcal{C}$ **do**
    for $\langle c_i, \mathcal{P}, \ldots \rangle \in \overline{C}$ **do**
      $c_v = \text{exists}(c_i : \textbf{bool})$
      for $\langle \ldots, l, \overline{P} \rangle \in \mathcal{P}$ **do**
        $\overline{p_v} = [\text{exists}(p : \textbf{bool}) \mid p \in \overline{P}]$
        assert $c_v \rightarrow \sum \overline{p_v} \geq l$
        assert $\neg c_v \rightarrow \sum \overline{p_v} = 0$
        for $\langle c_i.p_j, d, \ldots \rangle \in \overline{P}$ **do**
          if $j \geq 1$ **then**
            assert $\overline{p_v}[j] \rightarrow \overline{p_v}[j-1]$
          end if
          $p_v = \overline{p_v}[j]$
          $\mathcal{Q}_p = \{\text{exists}(q : \textbf{bool}) \mid q \in \mathcal{Q}, c_i.p_j \in q\}$
          assert $\neg p_v \rightarrow \sum \mathcal{Q}_p = 0$
          if $d =$ `out` **then**
            assert $p_v \rightarrow \sum \mathcal{Q}_p \geq 1$
          else
            assert $p_v \rightarrow \sum \mathcal{Q}_p = 1$
          end if
        end for
      end for
    end for
  end for
  for $\langle l, u, \mathcal{Q}_x \rangle \in \mathcal{X}$ **do**
    assert $l \leq \sum \{\text{exists}(q : \textbf{bool}) \mid q \in \mathcal{Q}_x\} \leq u$
  end for
end function

4.1.3 Symmetry Breaking

A problem we have yet to tackle is the problem of *symmetry*, i.e., isomorpic solutions. If we view the architectures we generate as graphs, with component and port instances as nodes and connections as edges, then many of the generated graphs will be isomorphic to another generated graph.

This is due to the equivalence between component and port instances. If we take an architecture, and swap any two component or port instances with the same source, we end up with an architecture that behaves exactly the same. Figure 4.4 shows two architectures where the two variability monitor instances

```
constrain sensor_0.out_0--selector_0.sensor1_0 +
          sensor_1.out_0--selector_0.sensor1_0 +
          selector_0.out_0--selector_0.sensor1_0 == selector_0.sensor1_0
constrain sensor_0.out_0--selector_0.sensor2_0 +
          sensor_1.out_0--selector_0.sensor2_0 +
          selector_0.out_0--selector_0.sensor2_0 == selector_0.sensor2_0
constrain vm_0.valid_0--selector_0.vm1_0 +
          vm_1.valid_0--selector_0.vm1_0 +
          dm_0.valid_0--selector_0.vm1_0 == selector_0.vm1_0
constrain vm_0.valid_0--selector_0.vm2_0 +
          vm_1.valid_0--selector_0.vm2_0 +
          dm_0.valid_0--selector_0.vm2_0 == selector_0.vm2_0
constrain vm_0.valid_0--selector_0.dm_0 +
          vm_1.valid_0--selector_0.dm_0 +
          dm_0.valid_0--selector_0.dm_0 == selector_0.dm_0
```

Figure 4.3: Subset of the constraints generated by Algorithm 6, in this case those related to the input ports of the sensor. Each constraint enforces that, if present, the port must have one and only one connection.

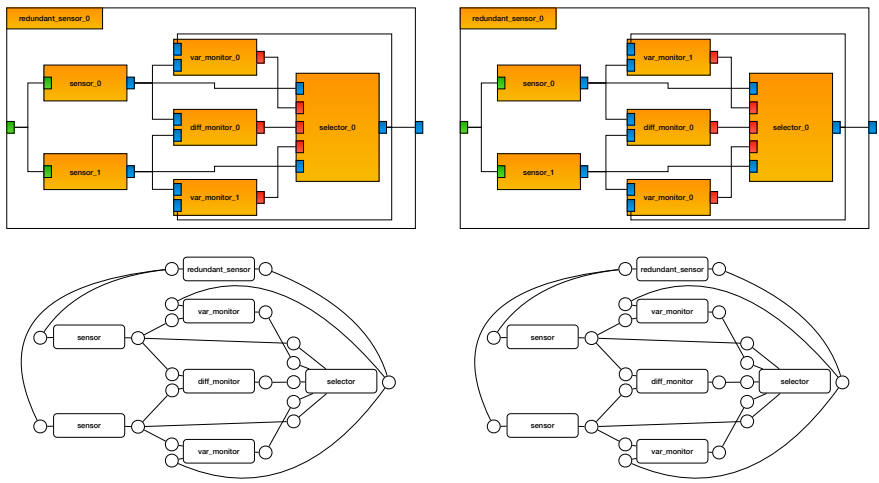


Figure 4.4: Two symmetric architectures and their respective graph representations. The variability monitor instances have been swapped, with VarMonitor 1 now connecting to Sensor 0 and vice versa. The two graph representations are identical.

have been swapped. When represented as graphs, the two architectures are identical, despite the monitors connecting to different sensor instances and to different selector ports. The two architectures are symmetric, they are the same solution for all intents and purposes.

Because we are trying to find all possible solutions these symmetries are something we need to avoid as much as possible. Any symmetry we do not eliminate results in one or more generated architecture that have no value, since an equivalent architecture would have been generated already.

To deal with symmetries we can filter out symmetric architectures as a post-processing step, by translating them to graphs and checking for isomorphism using a tool like nauty or Bliss [93].

However, post-processing to avoid symmetric solutions has the same problem as verifying the generated architectures through post-processing: the combinatorial explosion in the number of generated architectures as each symmetry doubles the number of solutions.

A better approach is to add additional constraints to the problem to prevent the generation of symmetric architectures in the first place. In the two generation algorithms we already break some symmetries by enforcing an order on the instances: lower ID component and port instances must be instantiated before higher ID instances of the same component or port may appear in a solution.

This breaks some symmetries, but connections introduce far more symmetries and do not have a simple ordering we can take advantage of.

Perfect symmetry breaking for graph problems requires, in theory, an exponential number of constraints, though there are techniques to reduce that number for small graphs when encoding into SAT [61].

For this thesis we use standard partial symmetry breaking predicates like DoubleLex [64] for connections between input and output ports and the predicate of Codish et al. [42] for connections between bi-directional (io) ports.

All of the predicates above work similarly: they apply permutations to the adjacency matrix of the graph and assert a lexicographic ordering between the original matrix and the permuted form. The more permutations considered, the stronger the symmetry breaking, albeit with a corresponding increase in the number of constraints the solver needs to handle.

As an optimization we treat each port type as a separate adjacency matrix since ports with different types cannot connect.

4.2 Verification

To deal with the requirements we will approach the problem much like we did Ensemble Generation and Architecture Generation: we develop an algorithm

for verification as a separate step, but in such a way that the constraints it introduces can be combined with the Generation constraints into one big problem.

The goal of verification is to filter out ensembles or architectures that do not follow the system requirements. In SCALe we model requirements as formal, modular contracts.

The modular part is critical as it allows us to use the requirements of internal components as a specification of their behavior, and select the relevant specifications based on the presence of the components.

More concretely, we want to show that the combined specifications of the components that are present in the system are a refinement of the system specification. In our case this corresponds to the contract refinement check ($\preceq$) of the contract meta-theory of Benveniste et al. [24].

The composition ($\otimes$) of two contracts results in another contract (see Chapter 2), and this operation can be generalized to accept n Assume/Guarantee (AG) contracts as input rather than a pair. We follow the approach used in OCRA [38] and use the resulting simplified and flattened refinement check directly.

Given a system contract (A_s, G_s) and a generalized composition of n sub-component contracts $\bigotimes_{i=1}^{n}(A_i, G_i)$, we can encode the refinement check as:

$$\begin{aligned} (A_s \wedge \bigwedge_{i=1}^{n} G_i) &\rightarrow G_s \\ \bigwedge_{i=1}^{n}((A_s \wedge \bigwedge_{j\neq i}^{n} G_j) &\rightarrow A_i) \end{aligned} \tag{4.2}$$

The first equation says that the assumptions of the system together with the guarantees of every sub-component must imply the guarantees of the system.

The second equation is a conjunction of implications for each sub-component. Each implication says that for a component i the assumptions of the system alongside the guarantees of every other component must imply the assumptions of i.

The second equation is important because the guarantees of each sub-component are only valid if its assumptions are assured. The assumptions of a contract refer to the *environment* in which its implementation will be used. For a sub-component, its environment is made up of: 1) the environment of the system of which it is part, and 2) the components that surround it.

The surrounding components have an effect on the environment that must be taken into account. For example, a component might require a specific temperature range to function but another component might produce heat while it functions. Such interactions must be taken into account.

The use of the generalized refinement operation is critical because we want to generate the verification problem only once, the $\mathbb{V}$ in $\exists \overline{a}.\, \mathbb{G}(\overline{a}) \wedge \forall \overline{s}.\, \mathbb{V}(\overline{a}, \overline{s})$. We could generate a separate problem for each generated architecture but then we have no way of combining generation and verification into a single problem.

The first thing we need to do regarding verification is to turn the SCALe expressions in the Assumptions and Guarantees of a component into SCL expressions.

As a reminder, in SCALe each component instance includes projected Assumptions and Guarantees: $\langle _, _, \mathcal{A}, \mathcal{G} \rangle$. They are the same as the assumptions and guarantees specified for the component in the SCALe model, but with the ports in the formulas changed to the corresponding port groups in the component instance.

We need to replace the port groups in those formulas with SCL variables that encode the *value* of each port instance in that group. To do that we use the `compile` function: it takes as input a SCALe expression (like an assumption or guarantee) and a set of port groups, and does the following:

- If the expression is a constant, return it.
- If the expression is a regular operator (like addition, multiplication, etc.), then apply `compile` recursively to the operands.
- If the expression is a port group, look at its cardinality and do one of the following:
  - If cardinality is 1 to 1, return forall($p_0 : b$) where p_0 is the single port instance in the group and b its base type.
  - If cardinality is 0 to 1, return the following expression: **if** exists(p_0 : **bool**) **then** forall($p_0 : b$) **else** d, where d is the default value of type b.
  - Else, return an error.
- If the expression is an operator that accepts an array (like sum, product, etc.), and its argument is a port group, do the following:
  - If the cardinality upper bound is 1, just ignore the operator and apply `compile` to the port group directly.

  - Else, unroll the operator, i.e., sum becomes addition. Each operand becomes forall($p_i : b$) if $i < l$, where l is the cardinality lower bound, otherwise it becomes **if** exists(p_i : **bool**) **then** forall($p_i : b$) **else** u where u (unit) is the identity value for the unrolled operator, e.g., 0 for addition, 1 for multiplication, **true** for conjunction, etc.

- Else, return an error.

Algorithm 7 shows how the verification problem is encoded. The `exists` function was introduced previously in Algorithm 5. The `forall` function works the same way but introduces a universally quantified variable instead. The **prove** statement works like **assert** but introduces a universal constraint, i.e., one that is under a universal quantifier (the f in $\exists e.\forall f$).

Note that the system itself is encoded as a component instance s, with the port directions flipped. We will use the contract of that system instance in the refinement check.

The first loop collects tuples of component instance variables and the corresponding compiled assumptions and guarantees.

Variable V and the second loop accumulate a conjunction of the implications from Equation 4.2, with one key difference: The guarantees of a component are only present if that component is present (i.e., its component instance variable is true).

Variable Q and the last loop accumulate connection implication constraints. If a particular connection is present (i.e., its variable is true) then the *values* of the two connected ports must be equal. These port value variables make up the state of the system ($\overline{s}$).

The last line adds a universal constraint: if the connection implication constraints hold, then the refinement constraints must hold.

Each port value is modeled as a universally quantified variable because verification needs to hold for all possible states. The verification constraints can be merged with the generation constraints and solved together using an $\exists\forall$ capable solver.

To deal with temporal constraints, we use techniques from Bounded Model Checking (BMC). Specifically, we use the encoding from Timo et al. [83], where each temporal operator is unrolled up to a user-provided bound k.

Because we want to solve the problem in one go (merged with generation) we cannot employ complete verification techniques, as they all require multi-shot

Algorithm 7 The algorithm for Verification ($\mathbb{V}$)

function VERIFICATION($s, \mathcal{C}, \mathcal{Q}$)
  $\langle _, \mathcal{P}_s, \mathcal{A}_s, \mathcal{G}_s \rangle = s$
  $A_s = \text{compile}(\mathcal{A}_s, \mathcal{P}_s)$
  $G_s = \text{compile}(\mathcal{G}_s, \mathcal{P}_s)$
  $\overline{R} = []$
  for $\langle c_i, \mathcal{P}, \mathcal{A}, \mathcal{G} \rangle \in \mathcal{C}$ **do**
    $A = \text{compile}(\mathcal{A}, \mathcal{P})$
    $G = \text{compile}(\mathcal{G}, \mathcal{P})$
    $\overline{R} = \overline{R} \frown [\langle \text{exists}(c_i : \textbf{bool}), A, G \rangle]$
  end for
  $V = (A_s \wedge \bigwedge \{v \rightarrow G \mid \langle c_v, _, G \rangle \in \overline{R}\}) \rightarrow G_s$
  for $r \in \overline{R}$ **do**
    $\langle c_v, A, G \rangle = r$
    $V = V \wedge (c_v \rightarrow ((A_s \wedge \bigwedge \{c_o \rightarrow G_o \mid \langle c_o, _, G_o \rangle \in \overline{R} \backslash r\}) \rightarrow A))$
  end for
  $Q = \textbf{true}$
  for $q \in \mathcal{Q}$ **do**
    $q_v = \text{exists}(q : \textbf{bool})$
    $\langle p_1, p_2 \rangle = q$
    $\langle _, _, _, b_1 \rangle = p_1$
    $\langle _, _, _, b_2 \rangle = p_2$
    $v_1 = \text{forall}(p_1 : b_1)$
    $v_2 = \text{forall}(p_2 : b_2)$
    $Q = Q \wedge (q_v \rightarrow v_1 = v_2)$
  end for
  prove $Q \rightarrow V$
end function

solving with additional constraints being introduced after each solver call until some condition is met.

That means our approach is *incomplete*: some architectures may be generated that do not actually fulfill the requirements, which can happen if invalid states only occur beyond k time steps.

This is not a problem in practice since the generated architectures will be translated afterwards to simulation models and undergo further analysis, where invalid states (if any) should be caught.

4.3 Synthesis

Our synthesis algorithm, which we call Requirement Augmented Generative Engineering (RAGE), has multiple modes of operation. The mode that solves the synthesis formula directly is called **RAGE-EF** and is depicted in Algorithm 8. The functions used in the algorithm are as follows:

- Functions EnsembleGeneration, ArchitectureGeneration and Verification introduced previously add their corresponding constraints to an SCL script, which is implicit in the algorithm.
- Function SymmetryBreak adds partial symmetry breaking constraints to the problem (e.g., DoubleLex [64]).
- Function UnrollBMC is responsible for unrolling temporal operators up to a the given bound k.
- Function TranslateToBackend translates the unrolled SCL script to a given backend solver S.
- Solutions (architectures) are stored in array $\overline{A}$.
- The while loop calls Solve on the backend solver until there are no more solutions: the solver returns **UNSAT** or **UNKNOWN**.
- The ToArchitecture function is used to convert the solution produced by the solver into an architecture.
- The Isomorphic function checks if the generated architecture is isomorphic to any of the previously found architectures using a tool like nauty [93]. This is necessary as we only have partial symmetry breaking constraints. Only non-isomorphic architectures are kept.

Algorithm 8 is an *ideal* version of the synthesis algorithm as it requires a backend solver capable of both universal quantification ($\forall$) and enumerating all solutions to the constraints.

None of the backend solvers we support in this thesis are capable of doing both. A simple solution is to use blocking clauses: once the solver produces a solution, we add an additional constraint forbidding that solution.

For example, if a solver produces a solution like $\langle x = 1, y = 0, z = 1 \rangle$, we can add the constraint $x \neq 1 \cup y \neq 0 \cup z \neq 1$ to prevent the solver from producing that solution again.

Algorithm 8 RAGE-EF: The Exists/Forall-based synthesis algorithm

```
function RAGE-EF(s, C, Q, X, k, S)
    EnsembleGeneration(C)
    ArchitectureGeneration(C, Q, X)
    Verification(s, C, Q)
    SymmetryBreak(Q)
    UnrollBMC(k)
    TranslateToBackend(S)
    Ā = []
    while Solve(S) = SAT do
        m = GetModel(S)
        ā = ToArchitecture(m)
        if ¬Isomorphic(Ā, ā) then
            Ā = Ā ⌢ [ā]
        end if
    end while
end function
```

Blocking clauses are very efficient but they add additional constraints with every solution, costing increasingly more memory and eventually bogging down the solver.

There are other ways to implement non-native solution enumeration that do not require extra constraints (and hence no additional memory), but they are typically slower than blocking clauses until a large number of solutions have been produced. We discuss them in Chapter 7, the future work.

As we will show in Chapter 5, using the $\exists\forall$ encoding is actually not very efficient, despite theoretically providing the solver with perfect information.

Because of this, we provide two other synthesis algorithms that use multiple solvers working in tandem. Alongside a mode that simply generates architectures without any verification (which we call **RAGE-G**), this gives a total of four modes of operation. Figure 4.5 shows a visual representation of the modes.

Above we introduced the $\exists\forall$ solving mode, with pure architecture generation as a special case (simply remove the Verification call). Below we introduce the other two.

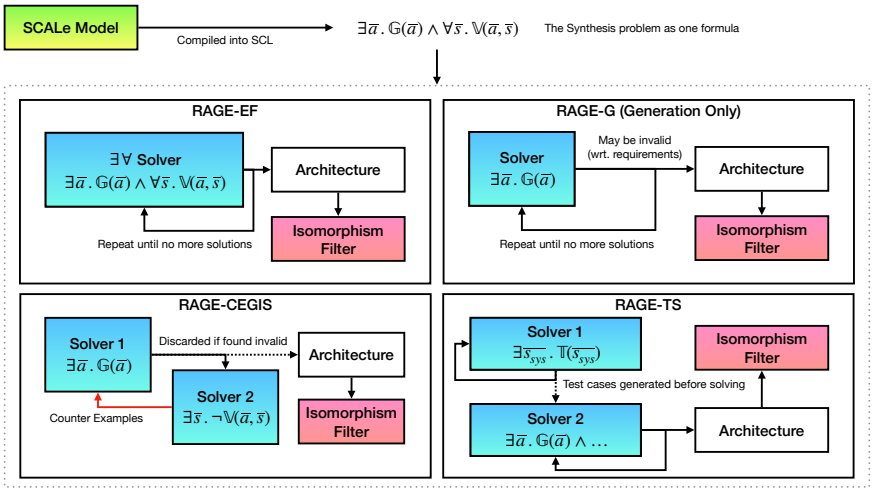


Figure 4.5: The four modes of operation supported by RAGE. Each mode, with the exception of the generation mode that simply disables verification, is explained in its own subsection. The $\mathbb{T}$ in the bottom-right mode refers to the requirements/contract of just the system itself.

4.3.1 Counter-Example Guided Inductive Synthesis

The second synthesis mode of RAGE uses a technique called Counter-Example Guided Inductive Synthesis (CEGIS) [71]. The idea of CEGIS is as follows:

- One solver, the generation solver, produces candidate solutions. In our case this solver would be given just the generation constraints to start.
- Another solver, the verification solver, takes a candidate solution produced by the generation solver and tries to find counter-examples. In our case this solver would be given the negation of the verification constraints to start, and on each call the generated architecture would be used to fix the values of the instance and connection variables.
- If the verification solver finds a solution, meaning an assignment to the state variables where the requirements *do not hold*, then this solution serves as evidence that the candidate architecture is not a valid implementation of the requirements.
- We then add the negation of the verification constraints to the generation solver, with every universally quantified variable replaced by the value it

was assigned in the counter-example. This prevents the generation solver from producing solutions that would lead to the same counter-example.

Effectively, we've transformed the $\exists \overline{a}.\, \mathbb{G}(\overline{a}) \wedge \forall \overline{s}.\, \mathbb{V}(\overline{a}, \overline{s})$ formula into $\exists \overline{a}.\, \mathbb{G}(\overline{a}) \wedge \neg \exists \overline{s}.\, \neg \mathbb{V}(\overline{a}, \overline{s})$, and then to get rid of the negation behind the second existential (which would simply turn into $\forall$ again), we split the formula into two separate existential problems: $\exists \overline{a}.\, \mathbb{G}(\overline{a})$ and $\exists \overline{s}.\, \neg \mathbb{V}(\overline{a}, \overline{s})$, with solutions of the latter problem being interpreted as *negative results*.

Through this transformation we can solve $\exists\forall$ problems using two solvers that only support existential constraints.

For certain backend solvers, the values of the instance and connection variables can be passed to the Solve call as *assumptions*. Assumptions allow the solver to retain information between calls, resulting in better efficiency over setting up a new verification problem each time. If a backend solver does not support assumptions, a new problem is set up with the relevant variables replaced by their values.

For the counter-examples, it would be better to first *generalize* the counter-example, i.e., given a counter-example we want to compute a predicate $G(\overline{a})$, such that $G(\overline{a}) \rightarrow \exists y. \neg \mathbb{V}(\overline{a}, \overline{s})$. The Yices 2 SMT solver [48] supports model generalization [49] to compute $G(x)$, and we take advantage of this capability when using Yices as the backend solver. For other backends the generalization function does a simple variable substitution.

The CEGIS-based synthesis algorithm is shown in Algorithm 9.

Figure 4.6 shows some of the verification constraints produced by Algorithm 9 for the redundant sensor example.

One nice side effect of this algorithm is that each counter-example we discover removes all solutions that would lead to that same counter-example, which includes all symmetric solutions.

4.3.2 Test-Set Guided Synthesis

The two previous synthesis modes have one major downside: neither works with backend solvers that natively support solution enumeration.

The $\exists\forall$ mode is not supported by any of our backends that supports solution enumeration, while the CEGIS mode requires adding the counter-examples to the generation solver, which in turn would cause the solution enumeration to reset.

```
constrain loop_at_t1 + loop_at_t2 <= 1
constrain loop_at_t1 -> system.fail_0_v_t0 == system.fail_0_v_t2
constrain loop_at_t1 -> system.fail_1_v_t0 == system.fail_1_v_t2
...
constrain loop_at_t2 -> system.fail_0_v_t1 == system.fail_0_v_t2
constrain loop_at_t2 -> system.fail_1_v_t1 == system.fail_1_v_t2
...
constrain in_loop_t1 == loop_at_t1
constrain in_loop_t2 == in_loop_t1 or loop_at_t2
constrain (
    (loop_at_t1 and (system.reading_0_v_in_loop == system.reading_0_v_t1))
     or
    (loop_at_t2 and (system.reading_0_v_in_loop == system.reading_0_v_t2)))
...
constrain system.fail_0--sensor_0.fail_0 -> system.fail_0_v_t0 == sensor_0.
    fail_0_v_t0
constrain system.fail_0--sensor_0.fail_0 -> system.fail_0_v_t1 == sensor_0.
    fail_0_v_t1
constrain system.fail_0--sensor_0.fail_0 -> system.fail_0_v_t2 == sensor_0.
    fail_0_v_t2
...
constrain !system.fail_0_v_t0 and !system.fail_1_v_t0
constrain abs(system.reading_0_v_t0 - system.reading_0_v_t1) <= 1
constrain !system.fail_0_v_t0 or !system.fail_1_v_t0
constrain (system.fail_0_v_t0 or system.fail_1_v_t0) -> (!system.fail_0_v_t1
     and !system.fail_1_v_t1)
# note the missing !system.fail_0_v_t1 and !system.fail_1_v_t1
constrain abs(system.reading_0_v_t1 - system.reading_0_v_t2) <= 1
constrain !system.fail_0_v_t1 or !system.fail_1_v_t1
constrain (system.fail_0_v_t1 or system.fail_1_v_t1) -> (!system.fail_0_v_t2
     and !system.fail_1_v_t2)
constrain abs(system.reading_0_v_t2 - system.reading_0_v_in_loop) <= 1
...
constrain not (
    (abs(system.reading_0_v_t0 - system.out_0_v_t1) <= 4) and
    (abs(system.reading_0_v_t1 - system.out_0_v_t2) <= 4) and
    (abs(system.reading_0_v_t2 - system.out_0_v_in_loop) <= 4) and
    in_loop_t2 and
    (in_loop_t1 -> abs(system.reading_0_v_t1 - system.out_0_v_t2) <= 4) and
    (in_loop_t2 -> abs(system.reading_0_v_t2 - system.out_0_v_in_loop) <= 4)
)
```

Figure 4.6: Subset of constraints produced by the CEGIS mode for the verifier. The various loop variables and related constraints are standard from BMC. The presence of each connection implies that the connection endpoints (the port instances) share the same value at every point in time. Then we require that the assumptions of the system hold alongside the guarantees of every subcomponent (hidden under the ellipsis), and that the guarantees of the system do *not* hold. This is because in CEGIS mode we are searching for counter-examples.

Algorithm 9 RAGE-CEGIS: The CEGIS-based synthesis algorithm

```
function RAGE-CEGIS(s, C, Q, X, k, S1, S2)
    EnsembleGeneration(C)
    ArchitectureGeneration(C, Q, X)
    Verification(s, C, Q)
    SymmetryBreak(Q)
    UnrollBMC(k)
    ⟨E, F⟩ = SplitProblem()
    TranslateToBackend(S1, E)
    TranslateToBackend(S2, F)
    Ā = []
    while Solve(S1) = SAT do
        m = GetModel(S1)
        if SolveWithAssumptions(S2, m) = UNSAT then
            ā = ToArchitecture(m)
            if ¬Isomorphic(Ā, ā) then
                Ā = Ā ⌢ [ā]
            end if
        else
            e = GetModel(S2)
            g = Generalize(S1, e)
            Assert(S1, ¬g)
        end if
    end while
end function
```

To take advantage of solvers capable of solution enumeration, we need an approach that does not require $\forall$ support nor the incremental addition of new constraints.

One way to achieve this is to unroll $\forall \overline{s}.\mathbb{V}(\overline{s})$ into every combination of values of the universally quantified variables $\overline{s}$. If the problem is small enough this could be a viable option, but the combinatorial explosion makes this approach unworkable for even moderately sized problems.

That said, if we unroll only a subset of the possible combinations we still obtain a valuable reduction in the search space. This partial approach can be combined with the CEGIS mode for more efficient generation.

It is possible to do better than selecting random combinations of the $\forall$, however, to the point where this approach can work on its own for certain use cases.

First, we need to consider the concrete problem we are solving: finding

architectures that fit the system requirements. In the real-world, this is most often done by developing simulation models and then testing them against many input/output pairs.

With this in mind, we can search for assignments only to the variables in the $\forall$ that are valid input/output pairs of the *system*, and then emulate the running of a "simulation" on each of those input/output pairs.

First, the synthesis formula is adapted to the following:

$$\exists \overline{a}.\, \mathbb{G}(\overline{a}) \wedge \forall \overline{s_{sys}}.\, \mathbb{T}(\overline{s_{sys}}) \exists \overline{s_{comp}}.\, \mathbb{V}(\overline{a}, \overline{s_s y s} \frown \overline{s_c o m p}) \tag{4.3}$$

The $\forall$ now refers solely to the system's state variables, while the component state variables are under an $\exists$. This is a weakening of the original formula, but it encodes the desired behavior: we are looking for an architecture that, when "simulated", gives the expected output for every input.

How closely this weakening matches the original formula depends on the specifications of the components. The more specific they are, the better. If the component specifications *fully specify* the behavior of the component, meaning they are of the form output $= f$(input), and all valid input/output pairs of the system are considered, then this approach will generate only valid architectures.

We have not implemented test generation for LTL, but as we will show in Chapter 5, this approach is extremely efficient for problems with massive search spaces but small input domains.

The algorithm for test-set-guided solving is shown in Algorithm 10. This version of the algorithm is similar to the original $\exists\forall$ version except we do not send the $\forall$ part directly to a backend solver. Like with the CEGIS version of the algorithm we split the problem in two, but the split verification constraints are not used directly.

First, we translate just the system requirements into constraints with the TranslateSystemRequirements function. This corresponds to $\mathbb{T}(\overline{s_{sys}})$ in the adapted formula.

These constraints are sent to a solver to find assignments to the input/output port values of the system where its specification holds true. Each test is then post processed by the GenerateTest function.

GenerateTest takes the split off verification constraints, negates them back to a positive form (though still existential), removes the implication towards the system guarantees (leaving only the component guarantees), and then performs variable substitution: the port value variables of the system are replaced by

Algorithm 10 RAGE-TS: The Test-Set Guided synthesis algorithm

```
function RAGE-TG(s, C, Q, X, k, S1, S2)
    EnsembleGeneration(C)
    ArchitectureGeneration(C, Q, X)
    Verification(s, C, Q)
    SymmetryBreak(Q)
    ⟨E, F⟩ = SplitProblem()
    R = TranslateSystemRequirement(s)
    TranslateToBackend(S1, R)
    TranslateToBackend(S2, E)
    while Solve(S1) = SAT do
        m = GetModel(S1)
        t = GenerateTest(F, m)
        Assert(S2, t)
    end while
    Ā = []
    while Solve(S2) = SAT do
        m = GetModel(S2)
        ā = ToArchitecture(m)
        if ¬Isomorphic(Ā, ā) then
            Ā = Ā ⌢ [ā]
        end if
    end while
end function
```

the test values, while the internal component port values are replaced by fresh variables.

Each substitution of the verification constraints is one test: an instance of $\exists \overline{s_{comp}}.\, \mathbb{V}(\overline{a}, \overline{s_s ys} \frown \overline{s_c omp})$ for a particular unrolling of the parent $\forall$.

The connection constraints ensure that the values of the system port values propagate through the internal ports. The constraints enforced by the internal component guarantees ensure that the values of the internal ports follow the behavior of the implementation. So the propagation of the inputs to outputs and the behavior must match.

Each of these tests is added as additional constraints to the generation solver, which is now restricted to finding instance and connection variables that pass every generated test.

4.4 Summary

In summary, RAGE supports four modes of operation, three of which take advantage of the verification-side of the problem to inform the search for valid solutions:

- **RAGE-G** only generates architectures, it does no verification.
- **RAGE-EF** relies on solving both the generation and verification problems together as a single $\exists\forall$ problem, but this is only possible with $\exists\forall$-capable solvers.
- **RAGE-CEGIS** splits the problem into two, each given to a different solver, and then iteratively uses counter-examples produced by the verification solver to assert additional constraints to the generation solver that prevent the generation of architectures that would lead to the same problem.
- **RAGE-TS** also splits the problem into two, but uses the verification problem to generate test cases. These tests are turned into constraints and asserted to the generation solver before synthesis begins, such that it only outputs architectures that pass all of the given tests.

In the following chapter we will evaluate four all modes, plus a modified version of **RAGE-CEGIS** with the learning disabled so it becomes pure generate & test (we call this pseudo-mode **RAGE-G&T**), each with multiple different backend solvers, and on three use cases.

Chapter 5

Experimental Evaluation

RAGE supports multiple synthesis algorithms (that we call configurations) and multiple backend solvers. But we do not know *a priori* which configuration and backend solver is the most efficient at finding all valid architectures for a given use case.

Our starting assumption was that **RAGE-EF** (see Chapter 4) alongside an $\exists\forall$ capable backend solver would give the best results, since it allows the solver to tackle the $\forall$ quantifier internally using state-of-the-art algorithms.

The **RAGE-CEGIS** mode, in contrast, requires splitting the problem in two, sending each subproblem to a separate solver, and incrementally asserting new constraints to reduce the search space.

Surprisingly, however, the results below show that **RAGE-CEGIS** often performs better than **RAGE-EF**.

We will evaluate each of the following RAGE configurations (see Chapter 4 for a detailed explanation of each):

- **RAGE-G**: pure generation with no verification.
- **RAGE-G&T**: generation + verification without any learning.
- **RAGE-EF**: direct solving of the $\exists\forall$ synthesis formula.
- **RAGE-CEGIS**: a generator solver sends candidates to a verification solver and receives counter-examples from which it learns.

- **RAGE-TS**: a solver produces test cases which are added as additional constraints to a generator before solving.

Each configuration (with a few exceptions explained below) will be evaluated on three use cases:

- **Redundant Sensors**: the running example used throughout the thesis.
- **Half-Adder**: a half-adder circuit made out of at most 5 nand-gates.
- **Full-Adder**: a full-adder circuit made out of at most 9 nand-gates.

For the Redundant Sensor example (see Figure 3.1 in Chapter 3) we want to know if we can satisfy the system requirements with fewer sensors than the original implementation.

For the Half-Adder and Full-Adder we want to know if we can implement them with fewer nand-gates than the typical implementation (see Figure 1.5 in Chapter 1).

The Redundant Sensor and the Adders exercise different aspects of RAGE. The Redundant Sensor example has complex LTL specifications, while the Full-Adder has simple specifications but has a huge search space with many symmetries.

We do not apply **RAGE-TS** to the Redundant Sensor because we have not implemented test generation for LTL, so this mode cannot be used.

Due to the size of the search space of the Full-Adder use case we do not bother running **RAGE-G&T**. The Half-Adder can be seen as a smaller version of the Full-Adder for which it is feasible to generate and verify (without learning from counter-examples or tests) all architectures within the time limit.

We evaluate each pair of mode plus use case on each of the following solvers, if the solver can support the mode:

- **Z3** [45]: an SMT solver with native support for $\exists\forall$ problems and that can perform efficient all-solution search through the use of a custom propagator. Unfortunately the two capabilities do not work correctly together.
- **cvc5** [21]: an SMT solver with native support for $\exists\forall$ problems and for Syntax-Guided Synthesis (SyGuS) [18]. We translate our synthesis problem to a SyGuS problem as well to see if the solver behaves differently.

- **Yices 2** [48]: an SMT solver with support for model generalization. Its $\exists\forall$ solving capabilities are only available in the executable (not via the API), but they work the same as our **RAGE-CEGIS** mode, and we use the model generalization API when running in this mode with Yices.
- **Bitwuzla** [101]: an SMT solver focused on bitvector problems, including quantified $\exists\forall$ problems. We translate our integer variables to bitvectors (8-bits wide) before passing the problem to Bitwuzla.
- **Gecode** [122]: A MAC-based CP solver. Has native all-solution search support and also natively supports a large number of global constraints, including the lexicographic ordering constraints we use for symmetry breaking. We use Gecode through MiniZinc [99] which adds some overhead.
- **Chuffed** [11]: An LCG-based CP solver. Has native all-solution search support. We use Chuffed through MiniZinc [99] which adds some overhead.
- **CP-SAT** [106]: A CP solver that combines techniques from LCG and MIP. It has native all-solution search support but cannot restrict the search to a subset of variables.
- **Choco** [111]: A MAC-based CP solver with recent versions adding some LCG capabilities. It can do all-solution search but it cannot restrict the search to a subset of variables without programming a custom search strategy. Has native support for graph problems which we unfortunately do not make use of.

We only support **RAGE-EF** on solvers with native $\exists\forall$ support. That means Z3, cvc5 and Bitwuzla. We additionally run cvc5 on a SyGuS translation of the problem.

We only support **RAGE-G&T** on solvers that natively support assumptions. In this case that consists of all the SMT solvers.

We only support **RAGE-CEGIS** on solvers that support assumptions and incrementally adding new constraints without starting from scratch, so only the SMT solvers in this case.

We only support **RAGE-TS** on solvers that can efficiently search on a subset of variables (normally). So every solver except CP-SAT and Choco. But for the Adder use cases it makes no difference to restrict the search to a subset, so we run this mode on every solver.

Table 5.1 provides an overview of the solver capabilities discussed above.

For solvers without native all-solution search we use blocking clauses [108], with the exception of Z3 where we use a custom propagator that treats every

Table 5.1: Solver Capabilities

	∃∀ Support	All-Solution Search	Subset Search
Z3	Yes	Custom [1]	Yes
cvc5	Yes	No	–
Yices 2	No [2]	No	–
Bitwuzla	Yes	No	–
Gecode	No	Yes	Yes
Chuffed	No	Yes	Yes
CP-SAT	No	Yes	No
Choco	No	Yes	No

[1] Z3 by itself does not support all-solution search but the documentation states how to implement it using a custom propagator.

[2] The Yices executable can deal with ∃∀ problems using the same CEGIS algorithm used in RAGE. This capability is not exposed in the API.

solution as invalid (forcing the solver to keep searching), unless the problem is quantified (∃∀) where we fallback to blocking clauses. Our custom propagator is more efficient than using blocking clauses, but it does not appear to work in the presence of quantifiers, so we could not use it in that case.

We ran all tests except for the comparisons with Simcenter Studio on an M1 MacBook Pro with 16 GB of RAM. Every evalution (use case + mode + solver) is given a time limit of 10 minutes. All timings are in seconds.

The comparisons with Simcenter Studio were ran within Simcenter Studio itself, running under WSL2 on a Dell laptop with an Intel Core Ultra 9 185H CPU running at 2.5 GHz and with 32GB of RAM. The Simcenter Studio comparisons were given a time limit of 1 hour.

5.1 Redundant Sensors

For the redundant sensor example we are interested in knowing if the system specification can be satisfied without using all three sensors: two variability monitors and one difference monitor.

Additionally we want to know the following:

Table 5.2: Redundant Sensors use case using RAGE-G.

	A.	F.	S.T.	F.T.	T.T.	O.T.
CP-SAT	3,217	6,637	0.36	0.43	1.77	1.41
Chuffed	3,217	6,637	1.08	0.47	2.19	1.11
Gecode	3,217	6,637	1.05	0.47	2.10	1.05
Yices 2	3,217	6,637	3.34	0.47	5.54	2.20
Z3	3,217	6,637	1.12	0.44	4.49	3.38
cvc5	3,217	6,637	18.53	0.52	27.00	8.47
Bitwuzla	3,217	6,637	8.51	0.51	13.52	5.01
Choco	3,217	6,637	0.36	0.43	**1.34**	0.98

- Raw architecture generation performance: since generating all architectures first in a short amount of time would allow us to verify all of them completely in parallel. We do not check the performance of parallel evaluations, however.
- Raw verification performance: this is how much it costs to verify all generated architectures. The modes capable of learning should give much faster results than this.
- $\exists\forall$ performance: how well can solvers with quantifier support solve the synthesis problem directly. If they can do this efficiently there is no reason to use any other mode.
- CEGIS performance: how much is the search space reduced with the addition of learning from counter-examples. Does the reduction compensate for the cost of the learning process itself?
- Filtering: how much time is spent on the post-processing with isomorphism filtering to avoid symmetric solutions.
- Overhead: how much time is spent doing something other than running a solver.

Raw generation performance was evaluated using **RAGE-G** and the results are shown in Table 5.2. All times are in seconds. Each column in the table represents the following:

- **A.**: the total number of non-symmetric architectures.

Table 5.3: Redundant Sensors use case using RAGE-G&T.

	A.	F.	G.T.	V.T.	F.T.	T.T.	O.T.
Yices 2	8	24	3.78	12.80	0.00	**18.11**	1.53
Z3	8	24	1.20	57.75	0.00	61.73	2.78
cvc5	7	4	0.97	51.66	0.00	–	–
Bitwuzla	8	24	9.01	6.76	0.00	20.05	4.28

- **F.**: the total number of filtered architectures by the post-generation isomorphism check.
- **S.T.**: the total solving time (in seconds), i.e., the total amount of time spent within "solve" calls to a backend solver.
- **F.T.**: the total filtering time (in seconds), this is basically the total amount of time spent running isomorphism checks.
- **T.T.**: the total solving time (in seconds), counting overhead. Does not include the filtering time.
- **O.T.**: the total amount of time spent outside the "solve" call (in seconds). Includes things like translating the solutions produced by the solver and adding blocking clauses, but also the filtering time.

All of the CP solvers did very well, with Choco taking a small lead, finishing in just 1.34 seconds. The complete search space of architectures is trivial to explore using any of the backend solvers.

There is a very high amount of overhead for most solvers, sometimes close to the solving time itself. While the overhead includes the filtering time, the rest can be attributed to our implementation being written in Python and the cost of translating the solver output. For some solvers there was higher overhead than the time it took them to find all architectures which was very surprising.

The SMT solvers were noticeably slower, with cvc5 in particular taking over 18 seconds to solve the problem. Z3 did very well but our Python-based custom propagator for Z3 unfortunately had very high overhead.

Raw verification performance was evaluated using **RAGE-G&T** and the results are shown in Table 5.3. Solving time was split in two columns: Generation Time (**G.T.**) and Verification Time (**V.T.**).

Table 5.4: Redundant Sensors use case using RAGE-EF.

	A.	F.	S.T.	F.T.	1.T.	T.T.	O.T.
Z3	3	1	11.44	0.00	5.90	–	–
cvc5	8	24	61.96	0.00	61.55	**62.09**	0.13
cvc5 (SyGuS)	8	24	512.51	0.00	268.00	512.52	0.01
Bitwuzla	8	24	63.78	0.00	18.81	63.79	0.01

Verification is noticeably more expensive than generation, though Yices and Bitwuzla did very well. It took Yices 2 18.11 seconds to naively generate and verify all possible architectures. One solver, cvc5, was unable to generate and verify all architectures within the time limit.

Out of the $3,217$ potential architectures, only 8 are valid. There is technically only 1 valid architecture but we are unable to break the following symmetries in our model:

- the lack of an absolute difference operator that can be applied to a port group meant that the equivalent input ports of the difference monitor had to be treated as unique ports. That introduces 1 symmetry.
- The sensor-related and validator-related input ports of the selector are equivalent when treated as a pair. If the same sensor and validator are connected to one pair, they could be connected to the other pair instead and it would not make a difference. Since we cannot model this, they introduce the remaining symmetries.

All of the generated architectures have all of the sensors present, though with a small change to the model (discussed later below) this is no longer the case.

∃∀ **performance** was evaluated using **RAGE-EF** and the results are shown in Table 5.4. A new column, **1.T.**, shows how long it took the solver to find the first valid architecture (in seconds).

In the ∃∀ case this is important for two reasons: most ∃∀ solvers use more efficient, non-incremental algorithms when searching for a single solution, and since the first solution is guaranteed to be valid, searching for others may not be necessary.

We expected one solver, cvc5, to be disadvanged in the single solution generation case because the use of incremental mode has to be selected before solving. Z3

Table 5.5: Redundant Sensors use case using RAGE-CEGIS.

	A.	C.	F.	G.T.	V.T.	F.T.	T.T.	O.T.
Yices 2	8	2,614	24	0.31	2.83	0.00	**3.49**	0.34
Z3	8	3,856	24	3.23	26.05	0.00	37.03	7.75
cvc5	8	892	24	1.49	113.47	0.00	115.85	0.89
Bitwuzla	8	3,435	24	4.89	3.28	0.00	12.50	4.32

(and maybe also Bitwuzla) uses a fast non-incremental mode for the first solve call, and then switches to incremental mode. However, there was no difference in the time to first architecture when disabling the incremental mode of cvc5.

Z3 was the standout winner at finding the first architecture, producing the first solution in just 5.9 seconds, but after finding a few more solutions it becomes stuck and unable to make progress. The reason for this is unknown.

Interestingly, cvc5, after finding the first solution, produces every other solution extremely quickly, though finding that first solution takes a very long time. The SyGuS mode performs substantially worse.

The most important result is that it took longer for the best solver, cvc5, to find all solutions using this mode than it took Yices to naively generate and verify every possible architecture. This shows that at least for all-solution search, using an $\exists\forall$ capable solver is the wrong approach.

CEGIS performance was evaluated using **RAGE-CEGIS** and the results are shown in Table 5.5. A new column, **C.**, shows the number of counter examples produced by the verifier.

The number of counter-examples generated varies per solver, which is an expected result. There are multiple reasons for this:

- We do not enforce that smaller counter-examples (which have stronger pruning power) must be found first.
- Even for counter-examples of the same size, different counter-examples have different pruning power, since each "problem" can affect a varying number of possible architectures.
- Lastly, the counter-examples produced depend on the architecture being verified, and which architectures are found first is random.

The lowest number of counter-examples produced was 892 by cvc5, which when added to the valid and filtered architectures amounts to 1024 architectures, substantially less than the amount of *unique* architectures that exist $(3,217)$. Compared to the total number of architectures generated naively $(9,854)$, we had a reduction of nearly an order of magnitude, which is a good result. Enforcing minimal counter-examples should lead to an even greater reduction in the search space.

Despite its advantage in counter-example production, cvc5 was the slowest solver due to its weak incremental solving. Yices 2, on the other hand, found all valid architectures in just 3.49 seconds, the best result so far.

This result was very surprising, as we expected $\exists\forall$ solvers to use CEGIS-like techniques internally for this sort of problem (while sharing more information and avoiding translation costs), and hence have better performance than programming effectively the same algorithm using two separate solvers with external coordination. The reason for the discrepancy is unknown.

5.1.1 Summary

All valid architectures contained all 3 sensors. With the problem as defined they are all required to satisfy the specification. The 8 variations are all due to symmetries we don't break, like the two input ports of the difference monitor, or the pairs of sensor plus variability monitor ports in the selector.

For pure generation the CP solvers are definitely the better choice performance-wise. When searching for all solutions, CEGIS mode using Yices 2 gives the best results.

However, if there were some connection constraints in the model to limit the number of times certain components were allowed to connect. In particular, the monitors were only allowed to connect to the selector once each. Below we run the same experiment with those constraints removed.

5.1.2 Redundant Sensors (Broken)

We modeled the Redundant Sensor problem with some restrictions on the connections of certain components. We only allow the output of the sensors and selector to connect a maximum of 3 times, and each monitor can only connect once. With those restrictions in place the specification is only met with every sensor present. If we remove those constraints, that is no longer the case.

Table 5.6: Redundant Sensors (Broken) use case using RAGE-G.

	A.	F.	S.T.	F.T.	T.T.	O.T.
CP-SAT	19,446	48,084	0.90	2.98	10.53	9.63
Chuffed	19,446	48,084	5.63	3.31	12.10	6.47
Gecode	19,446	48,084	5.28	3.27	11.71	6.42
Yices 2	19,446	48,084	394.77	3.67	411.96	17.19
Z3	19,446	49,439	6.54	3.07	29.70	23.16
cvc5	17,179	37,053	374.08	3.21	–	–
Bitwuzla	19,446	48,084	117.41	3.60	210.44	93.02
Choco	19,446	48,084	0.79	2.98	**7.43**	6.63

Table 5.7: Redundant Sensors (Broken) use case using RAGE-G&T.

	A.	F.	G.T.	V.T.	F.T.	T.T.	O.T.
Yices 2	94	242	392.37	74.10	0.02	478.00	11.52
Z3	94	261	7.86	552.53	0.03	580.07	19.68
cvc5	17	16	1.20	123.50	0.00	–	–
Bitwuzla	94	242	119.30	40.86	0.02	**248.43**	88.27

Table 5.6 shows the impact of removing those constraints when it comes to architecture generation. The number of generated and explored architectures has increased by over an order of magnitude, with $19,446$ unique possible architectures and $67,530$ generated in total.

The best solver was once again Choco, finding all architectures in just 7.43 seconds. Notably our overhead is even more noticeable here, with Choco finding all architectures in just 0.79 seconds, but it costing us 6.63 seconds to translate and filter the results.

Of note, Z3 actually generated more architectures than the other solvers. The extra architectures are the same as other previously generated architectures. Our guess is that Z3 is exploring the same area of the search space multiple times due to restarts and garbage collection of its "nogood" [29] database. This has little impact on the final result, however.

Table 5.7 shows the impact of removing those constraints when it comes to verification. We now have many more "valid" architectures, 94, and cvc5 was unable to find all of them within the 10 minute time limit.

Bitwuzla was the winner in this case, generating and verifying all architectures in 248.43 seconds. Yices 2 seems to slowdown substantially as the number of

Table 5.8: Redundant Sensors (Broken) use case using RAGE-EF.

	A.	F.	S.T.	F.T.	1.T.	T.T.	O.T.
Z3	50	82	179.58	0.01	19.15	–	–
cvc5 (SyGuS)	26	15	101.86	0.00	22.08	–	–
Bitwuzla	94	242	532.06	0.03	8.15	**532.20**	0.14
cvc5	0	0	–	–	–	–	–

Table 5.9: Redundant Sensors (Broken) use case using RAGE-CEGIS.

	A.	C.	F.	G.T.	V.T.	F.T.	T.T.	O.T.
Yices 2	94	13,442	242	3.59	14.27	0.02	**19.70**	1.85
Z3	94	15,481	242	49.57	126.33	0.02	208.41	32.50
cvc5	63	4,698	18	5.42	537.73	0.01	–	–
Bitwuzla	94	14,850	242	52.83	11.56	0.02	89.24	24.85

blocking clauses grows. Once again we see massive overhead, specially with Bitwuzla, costing over 88 seconds.

Table 5.8 shows the impact of removing those constraints when it comes to $\exists\forall$ solving. In this case only Bitwuzla was able to find all architectures in time, but took nearly twice as long to do so than with *naive* generate and test. The naive approach could have been made even faster by using a different solver for generation: using Choco would have saved more than 110 seconds, not counting the savings in overhead.

Finally, Table 5.9 shows the impact of removing those constraints when using CEGIS. Here we see Yices 2 as the standout winner, taking only 19.70 seconds to find all architectures. Z3 was now able to completely solve the problem, and cvc5 found many more valid architectures (63), though it still failed to find all of them within 10 minutes.

The main outcome of the experiment, however, is the change in the set of valid architectures. Allowing the monitors to connect more than once actually results in architectures that only need one or two monitors to satisfy the system specification. Figure 5.1 shows one such configuration, using only the difference monitor.

It is normal in formal specifications to only consider systems that are configured *sensibly*, meaning that the specification expects a certain structure for the system, and does not account for architectures configured in ways that do not fit the original intent (such as connecting the difference monitor to all monitor

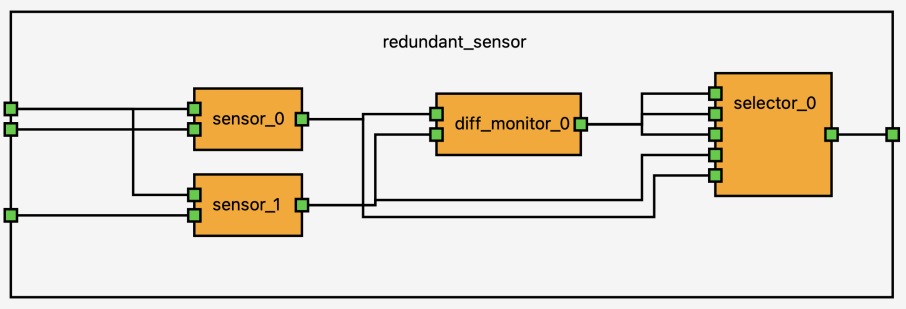


Figure 5.1: A valid solution to the redundant sensor specification, using a single difference monitor.

ports of the selector).

It is highly unlikely this would work in practice, but the specification is not strong enough to detect this. We would need to run simulations to evaluate the impact of an implementation using only one monitor. These results show that architecture generation may serve as a way to evaluate the strength of a specification.

5.1.3 Summary

As the number of architectures grows Yices 2 begins to lose out to other solvers, it seems to struggle with large amounts of blocking clauses. The original specification is not strong enough to prevent the generation of architectures that use less than three sensors, a single difference monitor allowed to connect multiple times is enough to satisfy the specification.

5.2 Half-Adder

We ran the same experiments as in the redundant sensor example with the addition of the **RAGE-TS** mode, since the specifications do not require temporal logic. Like the redundant sensor example only 5 components are necessary, but in this case they are all instances of the same component: a nand-gate.

We break many symmetries by enforcing an ordering on the connections of the nand-gates: higher “id” instances cannot connect their output ports to to

Table 5.10: Half-Adder use case using RAGE-G.

	A.	F.	S.T.	F.T.	T.T.	O.T.
CP-SAT	11,432	24,694	5.87	1.56	12.40	6.53
Chuffed	11,432	24,694	3.78	1.68	**7.80**	4.02
Gecode	11,432	24,694	4.73	1.69	8.72	3.99
Yices 2	11,432	24,694	111.87	1.85	123.43	11.56
Z3	11,432	26,626	5.07	1.68	23.02	17.94
cvc5	11,432	24,694	188.94	1.99	261.79	72.85
Bitwuzla	11,432	24,694	57.48	1.75	95.78	38.30
Choco	11,432	24,694	3.85	1.56	8.06	4.20

Table 5.11: Half-Adder use case using RAGE-G&T.

	A.	F.	G.T.	V.T.	F.T.	T.T.	O.T.
Yices 2	1	3	122.55	3.66	0.00	134.51	8.30
Z3	1	4	5.21	60.28	0.00	**80.72**	15.23
cvc5	1	3	197.41	35.64	0.00	305.03	71.98
Bitwuzla	1	3	57.67	21.47	0.00	114.46	35.32

lower "id" instances, e.g., instance 4 cannot connect back to instance 2. The implementation of a half-adder does not require loopback connections so this constraint is safe to add.

Raw generation performance was evaluated using **RAGE-G** and the results are shown in Table 5.10. This time Chuffed was the best solver, finding all solutions in just 7.80 seconds. There are $11,432$ possible unique architectures and $36,126$ in total. Once again see a massive amount of overhead.

Raw verification performance was evaluated using **RAGE-G&T** and the results are shown in Table 5.11. Out of $11,432$ possible architectures there is only 1 valid (non-isomorphic) implementation of a half-adder using nand-gates, and it requires all 5. Z3 was the best solver in this case, needing 80.72 seconds to generate and verify all architectures.

$\exists\forall$ **performance** was evaluated using **RAGE-EF** and the results are shown in Table 5.12. Every solver did incredibly well, with Z3 once again the winner, finding the one valid architecture in just 0.37 seconds, and all isomorphic

Table 5.12: Half-Adder use case using RAGE-EF.

	A.	F.	S.T.	F.T.	1.T.	T.T.	O.T.
Z3	1	3	0.66	0.00	0.37	**0.68**	0.02
cvc5	1	3	1.18	0.00	1.17	1.18	0.00
cvc5 (SyGuS)	1	3	9.99	0.00	6.40	9.99	0.00
Bitwuzla	1	3	1.28	0.00	1.03	1.28	0.00

Table 5.13: Half-Adder use case using RAGE-CEGIS.

	A.	C.	F.	G.T.	V.T.	F.T.	T.T.	O.T.
Yices 2	1	442	3	0.09	0.04	0.00	**0.18**	0.06
Z3	1	538	3	0.50	0.85	0.00	2.00	0.65
cvc5	1	603	3	0.70	0.56	0.00	1.60	0.34
Bitwuzla	1	452	3	0.60	0.27	0.00	1.12	0.25

architectures to it in just 0.68 seconds, a two orders of magnitude improvement over naive generate and test.

CEGIS performance was evaluated using **RAGE-CEGIS** and the results are shown in Table 5.13. Due to the lack of a temporal aspect, all counter-examples are minimal. The stronger counter-examples (compared to the redundant sensor use case) meant Yices 2 only needed 442 counter examples to find all valid architectures. Yices 2 was also the winner, taking just 0.18 seconds to find all valid architectures. This was a better result than the best native $\exists\forall$ solver (Z3).

In constrast to the results of the Redundant Sensor example, our CEGIS implementation was slower than the native $\exists\forall$ solving for Z3 and cvc5. Bitwuzla, however, still slightly benefited from an external CEGIS loop, a result we cannot explain.

Test-Set-augmented generation performance was evaluated using **RAGE-TS** and the results are shown in Table 5.14. As there are only 4 possible combinations of inputs to a half-adder, it only takes 4 tests to completely cover every possible behavior of the system. Additionally, the components are fully specified, i.e., in their specifications the outputs are fully determined from the inputs. This is the optimal type of problem for this mode.

Using **RAGE-TS**, Yices finds all architectures so quickly that the total time shows as 0.00 seconds, the problem is solved instantly. Every solver did very

Table 5.14: Half-Adder use case using RAGE-TS.

	A.	F.	S.T.	F.T.	T.T.	O.T.
CP-SAT	1	3	0.05	0.00	0.06	0.00
Chuffed	1	3	0.39	0.00	0.54	0.15
Gecode	1	3	0.37	0.00	0.51	0.14
Yices 2	1	3	0.00	0.00	**0.00**	0.00
Z3	1	3	0.02	0.00	0.04	0.01
cvc5	1	3	0.03	0.00	0.03	0.00
Bitwuzla	1	3	0.03	0.00	0.03	0.00
Choco	1	3	0.09	0.00	0.29	0.20

well, with Chuffed and Gecode taking the longest, but most of the time they took can be attributed to how long it takes to start MiniZinc and compile the problem to FlatZinc.

5.2.1 Summary

The strength of the counter-examples has a massive impact on their pruning power, in this case resulting in two orders of magnitude less architectures being explored than enumerating every single possible one.

For problems with only boolean variables, like this half-adder use case, the native $\exists\forall$ capabilities of the solvers do very well, though still overall worse than CEGIS with a faster incremental solver like Yices.

RAGE-TS is so efficient for this use case that one of the solvers appeared to take “0” seconds (this was due to displaying only 2 decimal places). The actual number is still 2 orders of magnitude faster than CEGIS and over 4 to 5 orders of magnitude faster than naive generate and test.

5.3 Full-Adder

But the half-adder use case is just meant as a starting point for the real problem, the full-adder use case. The two problems are nearly identical, the only differences are: We need 9 nand-gates instead of 5, and there is one additional input port, equivalent to the other two.

Table 5.15: Full-Adder use case using RAGE-G.

	A.	F.	S.T.	F.T.	T.T.	O.T.
CP-SAT	261,465	21,235	463.76	29.55	–	–
Chuffed	177,068	678,127	381.13	97.73	–	–
Gecode	83,451	189,068	397.25	30.69	–	–
Yices 2	48,558	3,182	552.14	6.30	–	–
Z3	334,700	68,348	114.55	42.88	–	–
cvc5	20,304	22,334	426.86	5.46	–	–
Bitwuzla	67,220	28,787	268.22	10.70	–	–
Choco	410,745	87,067	599.15	51.51	–	–

Table 5.16: Full-Adder use case using RAGE-EF.

	A.	F.	S.T.	F.T.	1.T.	T.T.	O.T.
Z3	0	0	–	–	–	–	–
cvc5	0	0	–	–	–	–	–
cvc5 (SyGuS)	0	0	–	–	–	–	–
Bitwuzla	0	0	–	–	–	–	–

Raw generation performance was evaluated using **RAGE-G** and the results are shown in Table 5.15. None of the solvers was able to find all architectures within 10 minutes, though Choco found the most: $410,745$. Chuffed got stuck producing massive amounts of isomorphic architectures, showing the importance of symmetry breaking.

We skipped **RAGE-G&T** since the likelihood of finding the one valid (+ 7 isomorphic) solutions without being able to explore the entire search space is very small. Even if a solver managed it would be by pure chance.

$\exists\forall$ performance was evaluated using **RAGE-EF** and the results are shown in Table 5.16. None of the solvers was able to find a solution within the 10 minute time limit.

CEGIS performance was evaluated using **RAGE-CEGIS** and the results are shown in Table 5.17. None of the solvers was able to find a solution within the 10 minute time limit. Sadly due to a limitation of our codebase we were unable to get the number of counter-examples considered (they are only reported once a solution is found).

Table 5.17: Full-Adder use case using RAGE-CEGIS.

	A.	C.	F.	G.T.	V.T.	F.T.	T.T.	O.T
Yices 2	0	–	0	–	–	–	–	–
Z3	0	–	0	–	–	–	–	–
cvc5	0	–	0	–	–	–	–	–
Bitwuzla	0	–	0	–	–	–	–	–

Table 5.18: Full-Adder use case using RAGE-TS.

	A.	F.	S.T.	F.T.	T.T.	O.T.
CP-SAT	1	7	0.85	0.00	1.30	0.45
Chuffed	1	7	15.01	0.00	16.69	1.69
Yices 2	1	7	0.72	0.00	0.73	0.01
Z3	1	7	3.50	0.00	3.60	0.10
cvc5	1	7	27.33	0.00	27.35	0.02
Bitwuzla	1	7	0.43	0.00	**0.45**	0.02
Gecode	0	0	–	–	–	–
Choco	0	0	–	–	–	–

Test-Set-augmented generation performance was evaluated using **RAGE-TS** and the results are shown in Table 5.18. As there are only 8 possible combinations of inputs to a half-adder, it only takes 8 tests to completely cover every possible behavior of the system. Additionally, the components are fully specified, i.e., in their specifications the outputs are fully determined from the inputs. This is the optimal type of problem for this mode.

In this case the problem becomes trivial to solve for all the "SAT-powered" solvers: all SMT solvers plus the LCG-based CP solvers. Gecode and Choco were unable to solve the problem, but it is important to note we have left them in their default configurations, which do not employ nogood learning. It is possible that turning on a learning mechanism would also allow them to solve the problem.

Bitwuzla was the winner in this case, finding all solutions to a problem where not a single solution could be found by the other modes within the 10 minute time limit, and doing so in just 0.45 seconds.

All other solvers did very well, with the exception of cvc5 which took 2 orders of magnitude longer than Bitwuzla for some reason. It would be interesting to run cvc5 as a bitvector solver as well to contrast the performance in that case.

5.3.1 Summary

This problem cannot be solved by any mode except **RAGE-TS** which, in turn, makes the problem trivial to solve. However, it is only trivial for the SAT-based solvers. The learning power of CDCL appears to be required to take advantage of the extra constraints derived from the test-set.

These adder circuits are particularly fitting for this mode, which makes the result not representative of the mode's performance in general. Most problems (including the redundant sensor example) have much larger input domains, and less strict specifications, that would make this mode unable to solve the problem completely on its own.

However, combining test-set-derived constraints with CEGIS should work well, and the tests could be generated lazily as each architecture is explored instead of all being generated before solving.

While not shown, we ran some tests where Yices was able to find some solutions, within the time limit, in CEGIS mode, if given 4 tests (half the input space). The main reason it could not find all solutions was that it started slowing down substantially after around $20,000$ counter-examples were produced. So even incomplete test-sets lead to a substantial reduction in the search space.

5.4 Comparison with Simcenter Studio

The semantics of SCALe are similar but not entirely the same as those of Simcenter Studio. The examples above cannot be modeled *directly* in Simcenter Studio as output ports can only connect once, while SCALe and RAGE allow any number of connections.

Having said that, the examples can be adapted slightly to allow us to compare the two tools in terms of generation performance. All use cases require at most 3 connections from the output ports to produce valid architectures, so we can replace each output port with a port group with 1 to 3 instances. For the inputs of the full-adder, similarly, we use a port group of 3 to 6 instances (3 inputs, we only need 2 connections from each).

We modified both the redundant sensor and the full-adder to use port groups instead of n-ary output ports for use in Simcenter Studio. In addition, we weakened the port types in the redundant sensor use case to just integer and boolean, increasing the number of possible architectures to $388,125$.

Table 5.19: RAGE-G vs. Simcenter Studio on the Redundant Sensor case

	A.	F.	S.T.	F.T.	T.T.	O.T.
RAGE-G	388,125	954,098	128.84	109.55	321.55	192.71
Studio	380,767	1,142,097	3,263.84	254.70	–	336.36

For this evaluation we increased the time limit to 1h. For RAGE only the best performing solvers are given. The results for the adapted redundant sensor use case are are shown in Table 5.19 while the results for the full-adder use case are shown in Table 5.20. For the Redundant Sensor use case the best performing solver was Chuffed, and that is the result shown on the Table.

We see a massive performance difference between RAGE-G and Simcenter Studio in the redundant sensor use case. Studio is unable to generate all possible architectures within the 1h time limit, whereas RAGE-G is able to do so in around 5 minutes.

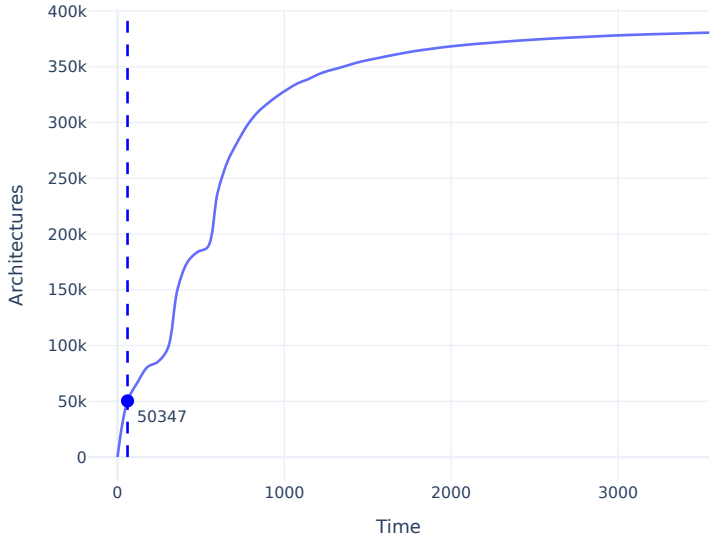
Figure 5.2 shows how the generation performance for both tools evolves over time. We see that the performance of RAGE-G (using Chuffed as the backend solver) stays relatively consistent, with a noticeable slowdown near the end. Simcenter Studio, on the other hand, slows down substantially as the number of generated architectures increases, with a nearly logarithmic shape to the plot. The dashed line shows the number of generated architectures at the 1 minute mark: after 1 minute RAGE-G generated nearly 3 times as many architectures.

The reason, we posit, comes down to the backend solver. The algorithms and encodings used by RAGE-G and Simcenter Studio are very similar, and while the native support for n-ary output ports makes a difference in overall performance, it cannot explain the slowdown over time seen in Figure 5.2.

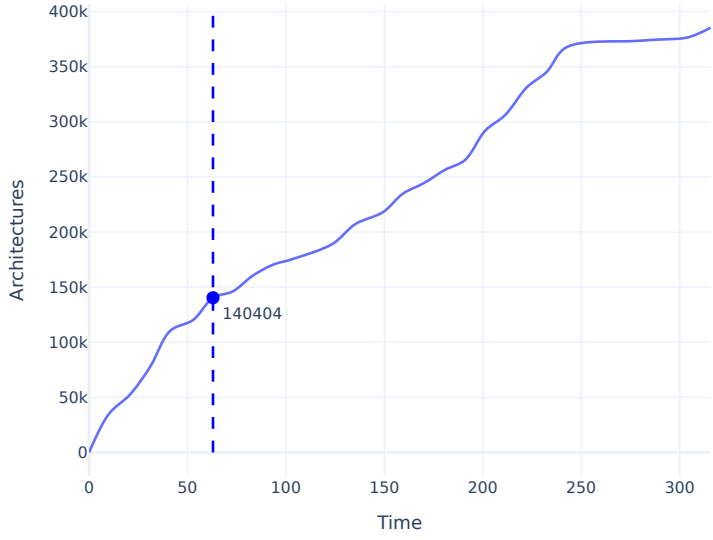
However, RAGE-G is using an CP [113] solver as a backend (Chuffed), while Studio is using a SAT solver as a backend (CryptoMiniSAT [121] as the default, at the time of writting).

The key difference is the native support for *all-solution search* that most CP solvers have. Simcenter Studio uses blocking clauses [108] to implement all-solution search, and those clauses eventually bog down the SAT solver. We saw a very similar issue when using RAGE with SMT solvers, where we also rely on blocking clauses. Z3 [45] was the only SMT solver that behaved differently, due to the custom propagator we used to implement all-solution search.

For the full-adder use case we could not use Chuffed due to a memory issue, so we contrast CP-SAT (another LCG-based CP solver) and Choco (a MAC-based



(a) Simcenter Studio generation performance over time.



(b) RAGE-G generation performance over time.

Figure 5.2: Comparison between Simcenter Studio (a) and RAGE-G (b) on their architecture generation performance over time. The dashed line shows the number of generated architectures at the 1 minute mark.

Table 5.20: RAGE-G vs. Simcenter Studio on the Full-Adder case

	A.	F.	S.T.	F.T.
RAGE-G (CP-SAT)	1,281,337	3,357,759	1,836.03	416.94
RAGE-G (Choco)	1,808,704	656,792	3,233.09	366.91
Studio	83,846	234,225	3,251.42	319.64

CP solver). We encoded a pattern in ACEL (Simcenter Studio's modelling language) that forbids all backwards connections between nand-gates to emulate the forward-only symmetry breaking predicate used in RAGE for this use case.

Neither RAGE-G nor Simcenter Studio were able to generate all architectures of the adapted full-adder use case, despite the reduction in search space (from the restricted connection counts) and despite being given 1 hour to do so. However, RAGE-G performed substantially better, finding over 1.8 million unique architectures using Choco and nearly 1.3 million using CP-SAT, while Simcenter Studio only managed around 84 thousand.

CP-SAT actually generated many more architectures than Choco, nearly 4.2 million compared to nearly 2.5 from Choco, but the search order Choco used avoided many symmetries, so the number of unique architectures (which is what we want to obtain) was greater.

We did not program a specific variable order, we just used the default ordering, but this result does imply choosing a good ordering for the variables could lead to substantial speedups in larger problems.

It is worth noting that Simcenter Studio supports parallel solving where the sythesis problem gets subdivided based on component ensembles or using cube-and-conquer [62], and then each subproblem is solved independently. This minimizes the performance issues caused by blocking clauses since the number of blocking clauses handled by each subsolver is smaller. But the problem remains and even with parallel solving Simcenter Studio is slower on both use cases.

5.5 Summary

We evaluated the four modes supported by RAGE (G, EF, CEGIS, TS) on 3 use cases: redundant sensors, half-adder and full-adder.

In all use cases, the original architecture was the only architecture that satisfied

the requirements. However, removing the connection constraints from the redundant sensor use case resulted in architectures that should not work in practice, revealing a weakness in the specification.

For the modes that do not require incremental solving nor $\exists\forall$ capabilities – such as the pure architecture generation mode RAGE-G and the test-set augmented mode RAGE-TS – solvers with native all-solution search performed substantially better than solvers that required the use of blocking clauses. This result was inline with our expectations.

We saw a similar result when comparing RAGE-G with Simcenter Studio in pure architecture generation performance: its use of blocking clauses resulted in a gradual performance decrease as the number of generated architectures increased.

On the other hand, the native $\exists\forall$ mode of some SMT solvers showed disappointing results, losing to our CEGIS implementation using the same solver, which was a very surprising result.

Chapter 6

Related Work

> *"I'm always open to the possibility that somebody's got a better idea than I have. It happens with some frequency."*
>
> —*William Shatner*

There are few generative engineering tools publicly available, and the ones that are work quite differently from one another, making direct comparisons difficult. However, there are quite a few contract-based verification tools whose specification capabilities and verification approach we can contrast with.

We will split the rest of the related work into three main categories: contract-based synthesis, structural synthesis, and contract-based verification.

Lastly we will also briefly discuss reactive synthesis, which is an entirely different but complementary synthesis problem to the one we tackled in this thesis.

6.1 Contract-Based Synthesis

In this section we will cover some tools and research papers that are similar in scope to RAGE, in that they synthesize architectures using requirement specifications (in the form of modular contracts) to guide the search.

6.1.1 OCRA

OCRA [38] is the tool that served as the inspiration for the contract specifications used in this thesis and how the contract verification problem is encoded.

Starting with version 2.0 [5], OCRA includes support for synthesis problems, but in a very general form: the user has to explicitly model every variability point and constraints on those variability points. That includes all connection variability and related symmetry breaking constraints when modelling that type of structural problem. Those constraints are complex so this can only be done by experts. In contrast, all variability constraints are generated internally by RAGE based on the given decomposition of the system.

OCRA's modelling language, like SCALe, defines the system decomposition in terms of components, and relates their contracts through a refinement relation. The redundant sensor example modeled in OCRA can be seen in Figure 6.1.

Like SCALe, OCRA supports LTL, but it also supports a more powerful logic called Hybrid Linear Temporal Logic with Regular Expressions (HRELTL) [40]. HRELTL is interpreted over hybrid traces, a mixture of continuous time evolution and discrete state transitions, and supports expressing requirements involving the derivative of continuous variables.

That said, to make the problem tractable, the expressiveness of HRELTL is restricted in some ways. For example, it only allows for linear constraints on the derivates and only polynomial constraints otherwise.

HRELTL is translated to LTL with Regular Expressions (RELTL) plus real arithmetic. RELTL plus real arithmetic is undecidable in general, but incomplete methods like bounded model checking [83] can still be used to find counter-examples to the refinement specification. I discuss similar ideas in Chapter 7 to improve the expressiveness of SCALe.

6.1.2 PyCo

PyCo [66] is a Python library to synthesize architectures from a library of components. It uses meta-programming features of Python to implement an internal DSL for contract modelling. PyCo's synthesis algorithm is very similar to the CEGIS approach presented in this thesis, though with different encodings.

The semantics of its modeling approach are different from SCALe, however. In SCALe we model the problem as a hierarchical decomposition of components, whereas PyCo models the problem as a search for components from a library.

```
#include "macros.h"

COMPONENT RedundantSensors system
  INTERFACE
    INPUT PORT reading: -15..15;
    INPUT PORT fail1: boolean;
    INPUT PORT fail2: boolean;
    OUTPUT PORT output: -15..15;

  CONTRACT behavior
    assume:
      (not fail1 and not fail2) and
      always ((abs_diff(reading, next(reading)) <= max_variance)
      and not (fail1 and  fail2)
      and ((fail1 or fail2) implies then (not fail1 and not fail2)));
    guarantee:
      always (abs_diff(reading, next(output)) <= max_sys_error);

  REFINEMENT
    SUB sensor1: Sensor;
    SUB sensor2: Sensor;
    SUB dm: DiffMonitor;
    SUB vm1: VarMonitor;
    SUB vm2: VarMonitor;
    SUB selector: Selector;

    CONNECTION selector.vm1 := vm1.valid;
    CONNECTION output := selector.output;
    ...

  CONTRACT behavior REFINEDBY
    sensor1.behavior, sensor2.behavior, selector.behavior,
    vm1.behavior, vm2.behavior, dm.behavior;

COMPONENT Selector
  INTERFACE
    INPUT PORT sensor1 : -15..15;
    INPUT PORT sensor2 : -15..15;
    INPUT PORT vm1 : boolean;
    INPUT PORT vm2 : boolean;
    INPUT PORT dm : boolean;
    OUTPUT PORT output : -15..15;

  CONTRACT selector
    assume: TRUE;
    guarantee:
      always ((dm implies next(output) = (sensor1+sensor2)/2) and
          ((not dm and not vm1 and vm2) implies  next(output) = sensor2) and
          ((not dm and not vm2 and vm1) implies next(output) = sensor1) and
          ((not dm and vm1 and  vm2) or (not dm and not vm1 and not vm2)
            implies next(output) = output));
```

Figure 6.1: An OCRA version of the redundant sensor system and the selector. Connections must be explicitly defined, for space only 2 are shown.

While finding an implementation of a contract from a library of components might seem similar to finding an implementation of a contract based on a system model decomposition, there are some differences.

The main (and most important) difference is how the bounds are specified. In PyCo, the component library is designed in such a way that it can be used to find solutions to various use cases, whereas the system model specifies exactly which components are to be considered and how many for a specific use case.

Because of this, rather than explicit component and connection bounds, PyCo uses a somewhat vague *redundancy metric* to restrict the amount of components and connections to consider in the model. However, most use cases end up using restricted subsets of libraries anyway to be tractable, which in practice means the library approach is the same as a system decomposition.

On the other hand, PyCo supports connectivity constraints that are useful for the redundant sensor example but are not supported by SCALe: distinct component constraints and same component constraints.

These constraints enforce that a set of ports cannot be connected, or must be connected, to the same component instance. For the selector component, as an example, we could say that the ports relating to the variability monitors and the sensors must be connected to different instances.

Another difference is that PyCo is mostly focused on finding a single optimal architecture rather than exploring all possible architectures. It can only optimize the architecture in terms of static properties (e.g., the total number of components). Like SCALe, PyCo is based on LTL, but unlike SCALe it has no support for arithmetic in the LTL formulas themselves, so the redundant sensors example cannot be modeled in PyCo.

PyCo uses a complete LTL solver as a verifier, it is not limited by a time bound. In RAGE we could use a complete LTL solver as a post-processing step (similar to how we do filtering based on graph isomorphism), to ensure every generated architecture is valid.

6.1.3 Nuzzo et al.

Nuzzo et al. [102] applied an approach similar to ours to the synthesis of an Electric Power System (EPS). In their work the synthesis algorithm and constraints are use-case specific and encoded manually, but the same concepts of architecture generation and verification are present.

The architecture generation problem in their work is modeled as a graph

search problem, where edges represent the contactors of the EPS (equivalent to connections in our setting), and where the presence of each possible edge is encoded as a boolean variable, and where component selection is done from a library of components (as in PyCo)

The requirements of the system are in regards to: power flow, reliability, and performance. The objective is to minimize the number of components and cost. The goal is to find the optimal architecture, rather than finding all valid architectures as in this thesis. Because of this they encode the problem into MIP, which has native optimization support.

Power flow is a static property of the system and those constraints can be included in the architecture generation side of the problem. Reliability, however, cannot be dealt with easily in that manner, and is instead dealt with similarly to RAGE's CEGIS mode: the generator solver produces candidate a candidate solution, that solution is sent to a verifier, and the verifier produces additional constraints that are added to the generator to reduce the search space.

The contracts of the components and the respective LTL specifications are not used to find valid architectures but rather to synthesize a controller that implements the required power-switching behavior. Controller synthesis is discussed at the end of this chapter 6.5.

Lastly, the real-time performance requirements are modeled in Signal Temporal Logic(STL) [89] and verified by first translating the architecture to a simulation model (as done in Simcenter Studio), and then evaluating it via Breach [47], which checks if the simulation model satisfies the STL specification through monitoring and test generation.

6.2 Structural Synthesis

In this section we will cover some tools and research papers that synthesize architectures, but where architecture verification is dealt with as a separate step and the results of which are not used to guide the search.

6.2.1 Simcenter Studio

Simcenter Studio [9] is the generative engineering tool that served as the starting point for this thesis. While this thesis does not directly build on the synthesis algorithm of Simcenter Studio, the synthesis algorithm presented here can be

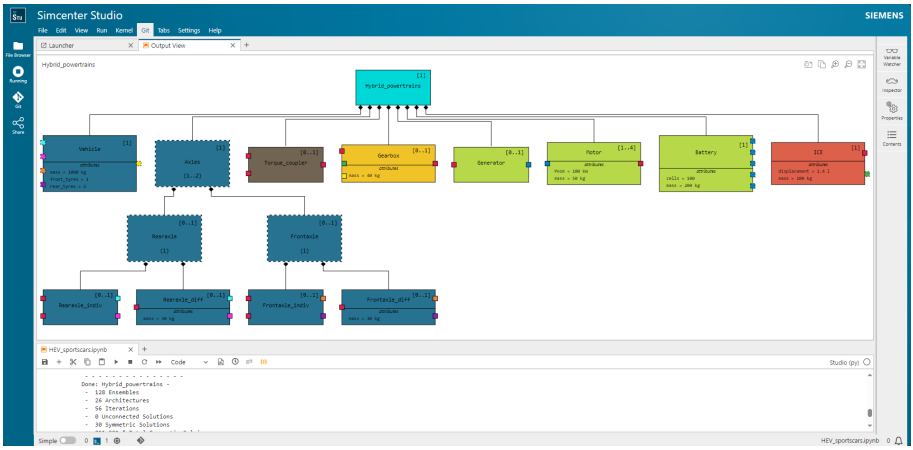


Figure 6.2: The Simcenter Studio UI, showing a hierarchical decomposition of a hybrid vehicle system.

seen as a simplified version of it with different trade-offs. Figure 6.2 shows Simcenter Studio's user interface.

Simcenter Studio has much stronger symmetry breaking, and splits the generation problem into separate ensemble generation and architecture generation problems, where multiple architectures are produced from each ensemble. The architecture generation problem can be further split using a cube-and-conquer approach [62].

This separation means each ensemble can be explored separately and in parallel, but it also means some of the generated ensembles may not lead to any useful architectures. In RAGE, because the generation problem is merged, those "ensembles" are never explored.

Simcenter Studio includes a language called ACEL to specify system models. The syntax and semantics of the structural parts of SCALe are inspired by ACEL, though certain aspects like port-variability and connection constraints are modeled differently.

Unlike SCALe, ACEL does not have any built-in support for requirement specifications. Rather, it has a mechanism to link components to simulation models and a mechanism to define simulation scenarios. See Figure 6.3 for the ACEL version of the redundant sensor system decomposition and the selector.

In ACEL the decomposition of a system is part of that system's specification. Components can also own other components allowing for multilevel systems.

```
system System:
  owns:
    Sensor
    VarMonitor
    DiffMonitor
    Selector

  ports::group 1:
    in::int reading1
    in::int reading2

  ports::individual:
    in::bool fail1
    in::bool fail2
    out::int output

component Selector:
  properties:
    cardinality = 1..1
  ports::individual:
    in::int sensor1
    in::int sensor2
    (in)::bool vm1
    (in)::bool vm2
    (in)::bool dm

  ports::group 1:
    out::int output1
    (out)::int output2
    (out)::int output3
```

Figure 6.3: An ACEL version of the redundant sensor system and the selector. Unlike SCALe, output ports are only allowed to connect once, so to achieve a similar result the output port of the selector and the input port of the system have to be replicated multiple times. The use of the keyword group makes the ports symmetric. Requirement specifications are not supported.

ACEL does not handle sub-systems separately from the larger system, synthesis in ACEL always starts from a top level system specification.

A major difference is how cardinalities are handled. For components, cardinalities are specified at the component level using the *multiplicity* property, rather than as part of a decomposition.

For ports there are no cardinalities and ports always connect 1-to-1. Instead, a port can be marked as optional by wrapping its direction in parenthesis – the port itself may be missing in the system, equivalent to a cardinality of 0-to-1 – and/or a set of ports may be placed in a group, where they are treated as being equivalent to one another. Groups are used to add symmetry breaking constraints to the synthesis problem, to avoid generating equivalent architectures, same as how port cardinalities are used in this thesis.

If groups are used to model the ability of a port to connect more than once, this adds many extra symmetries to the problem.

Requirement verification in ACEL is typically handled by automatically translating the architectures to simulation models based on component mappings, and then running user-defined Python scripts to rank and evaluate the results of the simulations. While this approach is very general, it does not allow for the requirements to guide the search for architectures. I discuss a potential avenue for synthesis to learn from simulation results in Chapter 7.

6.2.2 Dessia

Dessia is a Generative Engineering application that can be used to explore various design alternatives through the application of *bots*, which are pieces of software that mix *know-that* with *know-how*.

The *know-that* is a description of the problem and related engineering knowledge. In Dessia this is done through an API that allows to model the relevant engineering concepts as Python classes.

The *know-how* is a description of the sequence of steps necessary to solve the problem. This is usually done by adding a method to a class that is responsible for performing the search.

For example, for an architecture generation problem, the design space would be modeled as a method of a Generator class that enumerates all possible configurations. This can be done in a brute-force manner, but Dessia also has built-in support for decision trees [79] which can be used to perform more intelligent search.

Constraints can be specified as Python code, which checks if assignments produced by the decision tree are valid or not. If not, the decision tree is informed of this and learns from it, avoiding similar solutions in future assignments. It was applied successfully in the generation and evaluation of gearbox architectures [92].

Much like Simcenter Studio, Dessia tries to be as general as possible. Dessia is even more general than Simcenter Studio as it does not enforce a particular structure for the problem, whereas Simcenter Studio only deals with hierarchical decompositions of components with ports and connections, same as our work.

The downside is that because it is so general it cannot automatically handle common challenges like symmetry breaking, and its decision-tree based search is not as efficient as solving a CSP.

6.3 Contract-Based Verification

In this section we discuss tools that use contracts (or at least modular requirement specifications) to verify if an architecture is a valid implementation of the system specification. These tools focus solely on verification and do not synthesize architectures.

6.3.1 Stimulus

Stimulus [10] is a formal requirement verification tool where requirements are specified in STL. The requirements are formal, as in our approach, but they are not in the form of contracts as defined in the meta-theory [24].

More concretely, Stimulus only models guarantees. Assumptions can only be modeled as implications, following the weak assumption semantics of the meta-theory.

The key selling point of Stimulus is a large library of requirement templates, displayed in a natural language-like form, that can be visually combined via drag-and-drop to create larger sentences that closely resemble the original natural-language requirement, but now fully formally specified. A screenshot of the Stimulus user interface can be seen in Figure 6.4.

Stimulus relies on test generation and monitoring of simulations to verify the requirements. While this approach cannot prove that a particular design is correct, it can handle real-world problems, and shows that the required expressiveness of real-world industrial requirements is quite high.

In a way it is the mirror of Simcenter Studio. While Simcenter Studio focuses on architecture generation and translation to simulation models, Stimulus focuses on checking requirements against simulation models.

We discuss ways to support more expressive requirements, without sacrificing the guided synthesis capabilities presented in this thesis, in Chapter 7.

6.3.2 AGREE

AGREE (Assume Guarantee Reasoning Environment) [43] is an open-source, contract-based verification framework that enables compositional reasoning over Architecture Analysis and Design Language (AADL) [46] models. It is integrated into OSATE, the Open Source AADL Tool Environment [12]. A screenshot of OSATE and AGREE can be seen in Figure 6.5.

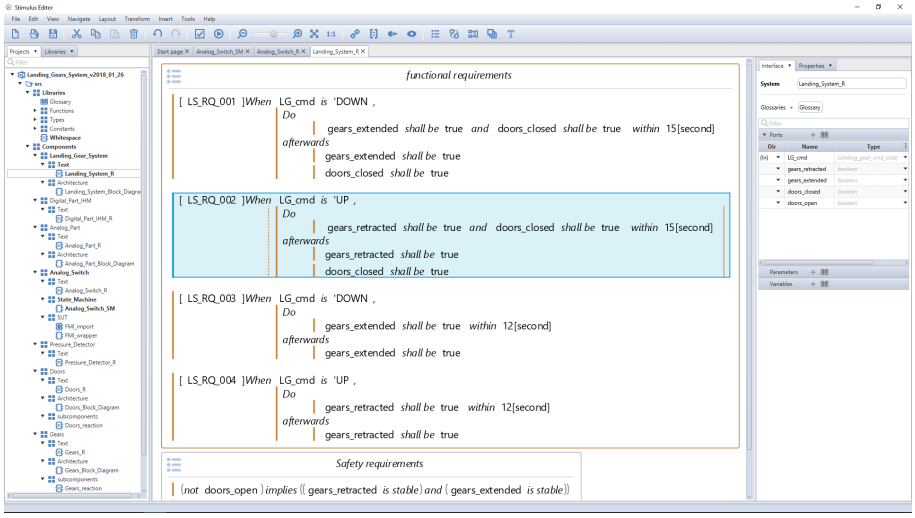


Figure 6.4: The Stimulus UI, showing a set of requirements. While the requirements look like natural language, they are actually formal STL specifications.

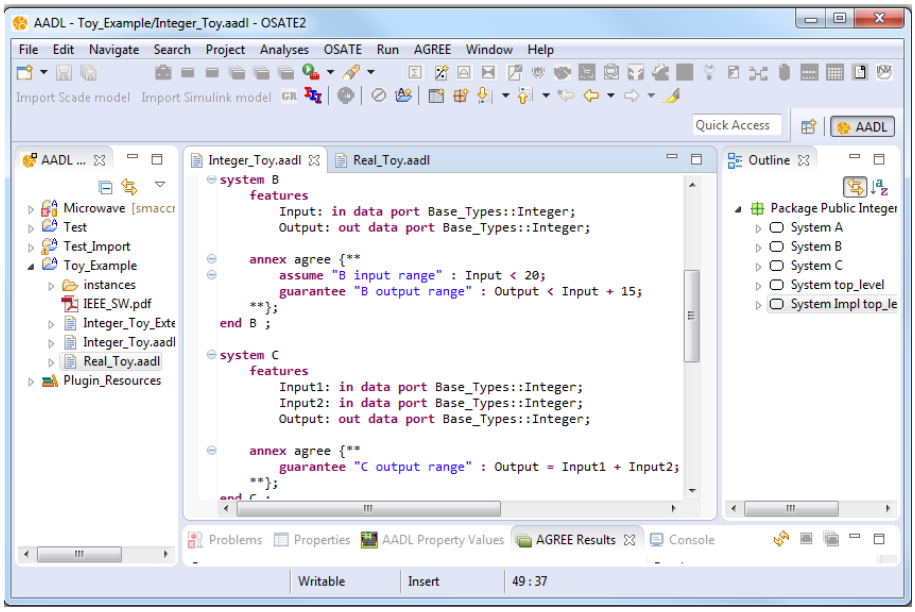


Figure 6.5: The OSATE UI, showing an example with the AGREE annex. Taken from the AGREE documentation.

In AGREE components in an AADL model are annotated with assume/guarantee contracts. The annotations are done via an AADL annex, which is AADL's way of supporting extensions. AADL itself does not have native assume/guarantee contract support.

AGREE follows the same assume-guarantee paradigm used in this thesis, meaning each component can be verified independently based on its contract. Unlike SCALe, AGREE uses *past* LTL, meaning, instead of referring to future points in time using a *next* operator, you refer to previous points in time using a *pre* operator.

AGREE also always treats every assumption and guarantee as if under the *always* operator by default, though it is possible to state that an assumption or guarantee only needs to hold in the first time step for example.

Lastly, AGREE uses the weak notion of assumption from the contract meta-theory [24], meaning the assumption does not need to hold, it is equivalent to placing the guarantee under an implication in SCALe.

Under the hood, AGREE translates the contracts and system model into a formal specification and dispatches the resulting model to an SMT-based model checker (e.g., JKind [53]). The model checker then exhaustively explores the state space to verify whether the properties and contracts hold.

The redundant sensor example modeled in AADL with AGREE annex contract specifications can be seen in Figure 6.6.

6.3.3 OCRA

While usable as a synthesis tool as of version 2.0 [5], OCRA was conceived as a verification tool and was only usable as a verification tool in its original version [38]. Because it now has synthesis capabilities, we covered OCRA in the the contract-based synthesis section 6.1.1.

6.4 Tool Overview

Table 6.1 shows an overview of the capabilities of the synthesis tools above. The work of Nuzzo et. al. is not included for space reasons and due to being tailor-made for a specific use case. The pure verification tools like Stimulus and AGREE are also not included. We contrast the following 6 properties:

```
data Range15
  properties
    Data_Representation => Integer;
    Range => -15 .. 15;
end Range15;

system RedundantSensors
  features
    reading : in data port Range15;
    fail1 : in data port Base_Types::Boolean;
    fail2 : in data port Base_Types::Boolean;
    output : out data port Range15;
  annex agree {**
    assume "A1" : not fail1 and not fail2 -> true;
    assume "A2" : abs(reading - pre(reading)) <= max_variance
    assume "A3" : not (fail1 and fail2);
    assume "A4" : (pre(fail1) or pre(fail2)) => (not fail1 and not fail2);
    guarantee "G1" : abs(reading - pre(output)) <= max_sys_error;
  **};
end RedundantSensors;

system implementation RedundantSensors.impl
  subcomponents
    sensor1 : system Sensor.impl;
    sensor2 : system Sensor.impl;
    dm : system DiffMonitor.impl;
    vm1 : system VarMonitor.impl;
    vm2 : system VarMonitor.impl;
    selector : system Selector.impl;
  connections
    c_vm1_to_selector: port vm1.valid -> selector.vm1;
    c_selector_output: port selector.output -> output;
    ...
end RedundantSensors.impl;

system Selector
  features
    s1 : in data port Range15;
    s2 : in data port Range15;
    vm1 : in data port Base_Types::Boolean;
    vm2 : in data port Base_Types::Boolean;
    dm : in data port Base_Types::Boolean;
    output : out data port Range15;
  annex agree {**
    guarantee "G1" : pre(dm) => output = (pre(s1) + pre(s2)) / 2;
    guarantee "G2"
      : (not pre(dm) and not pre(vm1) and pre(vm2)) => output = pre(s2);
    guarantee "G3"
      : (not pre(dm) and not pre(vm2) and pre(vm1)) => output = pre(s1);
    guarantee "G4"
      : ((not pre(dm) and pre(vm1) and pre(vm2)) or
         (not pre(dm) and not pre(vm1) and not pre(vm2)))
        => output = pre(output);
  **};
end Selector;
```

Figure 6.6: An AADL + AGREE version of the redundant sensor system and the selector. The arrow operator (`->`) defines a transition from the first step to the rest. Not to be confused with the implication operator (`=>`).

Table 6.1: Overview of Synthesis Tools

	RAGE	Studio	OCRA	PyCo	Dessia
Structure	Hierarchy	Hierarchy	Hierarchy	Library	Manual
Generation	CP/SMT	SAT	MC	MC	Dec. Trees
Symmetry	Auto	Auto	Manual	Auto	Manual
Requirements	LTL(A)	User Def.	HRELTL	LTL	User Def.
Verification	BMC	Simulation	MC	MC	User Def.
Learning	Yes	No	Unknown	Yes	Some

- Structure: How the search space is defined. Hierachy means that it is defined as a hierarchical decomposition of components with user-defined cardinalities. Library means that it is given as a set of components that may be instantiated freely. Manual means that the search space needs to be specified by the user in code.
- Generation: What technology is used to explore the search space. MC means a model checker is used.
- Symmetry: Does the tool filter symmetric solutions automatically.
- Requirements: How the requirements are specified. User Def. means that the requirements are encoded as a user-defined function in some scripting language.
- Verification: What technology is used to verify if a generated architecture follows the requirements. Simulation means the architecture is first translated to a simulation model that can then be evaluated. (B)MC means (bounded) model checking is used. User Def. means that a user-defined script is necessary.
- Learning: If the tool is cabable of adapting its search based on the results of verification. Some in the case of Dessia refers to the possibility of removing branches of the decision tree based on the verification results, but this must still be partially handled by the user. For OCRA its learning capabilities are unknown.

None of the tools supports an equivalent to RAGE's test-generation mode. For the pure-verification tools Stimulus and AGREE (not included in the table) only the Requirements and Verification rows would be relevant. Stimulus uses STL and Simulation and AGREE uses (Past-only) LTL and Model Checking.

6.5 Reactive Synthesis

Reactive synthesis [109] (also called controller synthesis) is the automated construction of a reactive system (i.e., controller) from a formal specification. In this context, a reactive system is one that maintains ongoing interactions with its environment (e.g., embedded controllers, software for protocols, robotics, etc.).

Most modern industrial systems have a physical side (usually called the plant model) made up of various interconnected physical components from different domains like mechanical and electrical, plus a software side (usually called the control model), which *controls* the physical components based on sensor output.

Our redundant sensor example is of this form: the sensors are physical parts, the monitors are sensors, and the selector is the controller.

Reactive synthesis can be used to generate concrete code for the selector to execute, i.e., to synthesize a concrete implementation of its contract that could then be used in a simulation model or even as part of the final system implementation.

The basic approach is to treat the environment and controller as adversarial players in a game (played on a graph) with a winning condition derived from the temporal logic specification: the environment tries to break the specification, while the controller tries to prevent it from doing so [109]. The outcome is a state machine where the controller "wins", i.e., where the enviroment (i.e., the system) is unable to enter an invalid state.

There are a variety of reactive synthesis tools available, with the standard format being the Temporal Logic Synthesis Format (TLSF) [69].

A TLSF file specifies the interface of the controller (the inputs and the outputs) plus its specification in the form of an assume/guarantee contract, with assumptions and guarantees given in LTL.

Reactive synthesis is highly complementary to our synthesis capabilities because we generate plant models (architectures), while reactive synthesis generates control models (software).

Chapter 7

Conclusions and Future Work

In this thesis we presented a novel generative engineering algorithm called RAGE and its associated modelling language called SCALe.

Given a hierarchical decomposition of a system written in SCALe, RAGE can synthesize all possible architectures of that system with state-of-the-art performance.

If formal specifications of the system and its components are available, RAGE can make use of them to efficiently guide the search towards valid architectures, avoiding the scalability issues present in the more "blind" architecture generation approaches used in industry.

Formal specifications enable generative engineering to tackle much larger problems, but without industrial tools that require them there is little incentive to develop them the same way there are simulation models for many common components. But since formal specifications are scarce, industrial generative engineering tools are not developed to make use of them.

By operating with or without formal specifications, RAGE can be adopted in industrial tools with little downside, while providing a path where formal specifications are incrementally developed and adopted, allowing generative engineering to scale to much larger problems and achieving our goal.

RAGE supports four modes of operation: one that performs architecture generation without formal specifications (**RAGE-G**), and three different modes using different approaches to guide the search when formal specifications are available (**RAGE-EF**, **RAGE-CEGIS**, and **RAGE-TS**).

We had three assumptions at the start of this work, that we evaluated on multiple use cases and using multiple backend solvers.

1. Guided architecture generation is substantially more efficient than pure architecture generation and evaluation.
2. Passing the complete synthesis problem to an $\exists\forall$-capable solver as in **RAGE-EF** is more efficient than using two interacting solvers as in **RAGE-CEGIS**.
3. **RAGE-TS** should be able to find the single valid solution to the full-adder use case faster than any other mode (but we did not know how much of a difference it would have).

Our first assumption was easily confirmed, the total synthesis time is greatly reduced when formal specifications are available and used to guide the search.

Our second assumption turned out to be false, in almost all cases **RAGE-EF** had worse performance than **RAGE-CEGIS** for the same backend solver. This shows that there is a lot of room for improvement to the native quantifier handling of many solvers, since they could simply use the same CEGIS algorithm but avoiding all the communication overhead.

Of note, the Yices 2 [48] SMT solver uses the exact same CEGIS algorithm we use to deal with quantifiers, but it does not make this capability available through its API, only in the executable, which prevented us from comparing the performance of its internal implementation versus our external one.

Our third assumption turned out to be true, but the difference was much greater than what we originally anticipated. The other modes cannot find the single valid architecture (ignoring symmetries) of the full-adder use case within 10 minutes, nor explore the entire design space within 1 hour, but **RAGE-TS** can find it within milliseconds with the right backend solver.

For pure architecture generation our backend CP solvers using native all-solution search showed substantially better performance than our backend SMT solvers using the "blocking clause" method [108], although Z3 [45] with a custom propagator to implement all-solution search was competitive. This result shows the importance of native all-solution search in generative engineering.

We witnessed similar results when comparing the pure generation performance of RAGE with the most similar generative engineering tool available on the market, Simcenter Studio [9].

For our use cases, RAGE using a CP solver was orders of magnitude faster than Simcenter Studio. While there are other factors weakening Simcenter

Studio's performance on these particular use cases, its use of blocking clauses for all-solution search appears to be the main culprit. We witnessed clear performance degradation over time as the number of generated architectures increased, similar to what we witnessed in RAGE using the same blocking clause approach.

7.1 Future Work

There are six main areas of future work for this project: 1) additional structural constraints, 2) facilitating requirement modelling, 3) logics with higher expressiveness, 4) improved symmetry breaking, 5) optimization, and 6) improved solving performance.

7.1.1 Additional Structural Constraints

SCALe allows three types of structural constraints: bounds on the number of instances of a component, bounds on the number of instances of a port, and bounds on the number of connections between components and/or ports.

PyCo's same/distinct component constraints (a set of ports must all connect to the same component instance or to distinct component instances), would be a relatively simple addition.

Simcenter Studio supports many other forms of structural constraints: more advanced forms of connection constraints, collections (constraints on the total number of instances from a multi-component set), patterns (subgraphs that must or must not be present in an architecture), and implication constraints (i.e. if a certain component is present, another component must also be present).

All of the above would greatly increase the modelling expressiveness of SCALe.

Beyond these kinds of structural constraints, constraints based on static attributes of components, for example limiting the total weight or cost of instantiated components, could also be supported. Note that this is not a verification problem, these constraints are part of the generation problem.

Finding architectures with the *lowest* weight or cost requires optimization, which is discussed in subsection Optimization 7.3.

7.1.2 Facilitating Requirement Modelling

In this thesis we introduced an approach to generate all valid architectures of a given system model annotated with formal contracts. A valid architecture, in the context of this thesis, means "the composition of the component contracts for a given architecture is a refinement of the system contract".

The advantage of using formal contract specifications is that, as shown in previous chapters, we can use them to guide the search for architectures, making the process orders of magnitude more efficient.

The disadvantage is that producing quality contract specifications for every component in the system can be prohibitively expensive. Companies rarely have formal methods experts on hand to write such specifications, and translating a complex behavioral model into a linear temporal logic specification can be challenging even for experts.

This is one of the reasons why tools like Simcenter Studio focus on translating the generated architectures into simulation models, as simulation models of common components are readily available (for example the Amesim libraries [8] or the Modelica standard library [2]), even if this means a more brute-force search process compared to the contract-based approach presented in this thesis.

There are two paths to ease the creation of the contract specifications:

- Translate informal (in a computer-science sense) requirement specifications into formal contract specifications.
- Compute approximate contract specification from detailed simulation models.

We will explore both alternatives below, though the most effective option is likely to be the latter. The main reason being that the (informal) requirement specifications of the system are often not modular, they refer only to the behavior of the system as a whole. That means a manual mapping step from general requirements to component specific requirements is necessary. On the other hand component-level simulation models tend to be readily available.

Translating Informal Requirements

The design of mechatronic system always starts with the gathering of its requirements. This results in a detailed document written in natural language,

describing the various functions the system needs to perform and the conditions in which they must be performed.

That's very similar to the formal contracts we've looked at, except that the *assumptions* and *guarantees* in such a document are often not clearly labeled as such, and they are not necessarily system level specifications, often referring to the behavior of internal components in an implicit manner. Certain requirements may also be vague or ambiguous, which is a common problem [24].

We will leave the problem of mapping requirements to the appropriate component as out of scope, and focus on translating the informal requirements into a formal form.

The new SysMLv2 [97] standard for systems engineering, however, has formal semantics and considers modular requirement specifications explicitly, including a separation between assumptions and guarantees. That means the approach introduced in this thesis could work well in industry as more companies adopt SysMLv2. SysMLv2 supports both formal and informal requirements, but they're modular in the same way our contracts are.

The requirement modeling tool Stimulus [10], mentioned previously in the related work (Chapter 6), tackles the issue of formalizing requirements by providing a graphical user interface that allows for natural language-like sentences to be created by dragging and dropping various operators with associated formal semantics. The idea being that a user will "write" something resembling the original informal requirement with these operators, except now the same sentence has well defined formal semantics.

Improving the usability of the formal requirement writing process is one way to tackle this issue, but for this thesis we're interested in higher degrees of automation where the amount of necessary expert intervention is minimal.

Matthew B. Dwyer et. al. have collected a set of patterns that appear often in the specification of finite state systems [50]. The patterns, in turn, have mappings to precise statements in LTL and other formalisms. Others have extended these properties to include, for example, real-time specifications [78].

Patterns are written in structured english and are much closer to what a human might write as a requirement than raw temporal logic. For example, the pattern:

> Globally, it is always the case that if (x = 0) holds, then (y = 1) eventually holds.

Has the same meaning as the LTL formula $\square(x = 0 \rightarrow \Diamond(y = 1))$, which in SCALe would be written as: **always**(x = 0 **implies eventually**(y = 1)).

The idea, then, is to detect which pattern a natural language requirement corresponds to, and then translate the pattern to the corresponding statements in an appropriate logic. The pattern above has a simple LTL translation, but more complex patterns have substantially more complex translations:

> After Q until R, it is always the case that if P holds, then S eventually holds (S responds to P).

Becomes in LTL:

$$\square(Q \wedge \neg R \rightarrow ((P \rightarrow (\neg R \,\mathcal{U}\, (S \wedge \neg R))) \,\mathcal{W}\, R))$$

Translating the patterns to LTL is trivial as it consists simply of mapping the pattern to its formal definition. Detecting the pattern that corresponds to some natural language requirement requires some natural language processing (NLP). This has become substantially easier to accomplish with the advent of Large Language Models (LLMs) [35].

However, simply mapping a natural language requirement to a pattern is insufficient, we must also map the entities referred to in the text to variables (like ports or attributes) in the system or its components.

This mapping will likely require user intervention, but it at least does not have to be done by a formal methods expert; any engineer familiar with the system will suffice. The quality of the translated patterns, i.e., how well they capture the intended meaning of the natural language requirement, also requires human intervention.

Improvements to specification quality can be facilitated with a mutation testing approach as used by STLInspector [112].

Approximating Simulation Models

The alternative to translating natural language requirements to formal specifications is to start from detailed behavioral specifications and translate them into a form usable by the search procedure.

It might seem strange at first glance to go from *more* detailed specifications to *less* detailed specifications when creating the less detailed specifications was considered too expensive.

The difference is that modern engineering processes almost always involve the use of simulation software, and simulation tools come with extensive libraries

full of behavioral models for commonly used components. The Modelica 4.0 standard library, for example, includes 1417 component models and 512 example models at the time of writing [2].

The problem with these simulation models is that they are too detailed. The only way to evaluate them is to run a simulation to see how they behave. In the test case generation mode we could use a black-box optimization tool [20], but in a CEGIS configuration we need to be able to learn from counter-examples.

There are a couple of techniques to explore: surrogate modeling and custom propagators.

Surrogate modelling is a technique heavily used in black-box optimization [20]. A surrogate model is a function that approximates a behavioral model, but is much faster to evaluate. Surrogates are computed by sampling the behavioral model a few times and then trying to fit a "curve" of some kind to the points. There are various kinds of surrogate models, but preferably we would like surrogates that can be encoded efficiently into CP or SMT.

One example of a generally applicable surrogate model that can be translated to CP constraints is a neural-network based on rectified linear units (ReLUs) [55], that are then encoded as piece-wise linear functions. ReLU-based neural networks have already been used to solve constrained black-box optimization problems as mixed-integer problems [104].

Of course, such surrogate models are not necessarily accurate. Unlike the approach presented in the thesis, which is complete (it finds all valid solutions, but may also return invalid ones), using these surrogate models naively may lead to valid architectures failing verification, as the surrogate may not accurately replicate behaviors that are necessary to satisfy all the system-level requirements.

Instead of running simulations solely on valid architectures, we can also run them on invalid architectures, and compare the generated traces. If both the surrogate trace and the real trace fail verification, then the surrogate trace is safe to learn from as a counter-example. If the surrogate trace fails but the real trace succeeds, then the counter-example is invalid. It would be good to update the surrogates in this latter case but it might be too expensive, so instead we forbid the verifier from returning that particular counter-example again.

Custom propagators are a capability offered by some constraint solvers that allow a programmer to introduce their own global constraints. We would introduce a new global constraint f_c per black-box function f of the form:

$$f_c(x_1, x_2, x_3, \dots, y_1, y_2, y_3, \dots)$$

Where each x_i is an input variable to the function and each y_j is an output variable to the function.

Given a partial assignment where all x are set, we run the black-box function. If any y is already set and does not match, we record extra constraints forbidding a partial assignment $\neg(\bigwedge_i (x_i = v_{xi}) \wedge y_j = v_{yj})$ for each y_j that was set in the partial assignment. If all y that are set already match, set the domain of the rest to their values as a returned by the black-box function.

If we replace f with a sufficiently accurate surrogate, we can do more efficient prunning, such as reducing the bounds of input variables based on the values of other variables. As before, this might lead to incorrect results.

These techniques only help with the specifications of the lower-level components. The system specification must be handled some other way, for example, via the natural language translation approach described previously.

The more accurate and detailed the system specification, the fewer invalid architectures slip through, but the harder it is to make that specification in the first place. There is a middle ground between the fully guided approach in this thesis and the more brute-force approach of a tool like Simcenter Studio.

7.2 More Expressive Logics

RAGE supports requirement specifications in propositional logic and LTL with arithmetic. Specifications of real-world systems, however, often require more powerful temporal constraints.

More concretely, hybrid system specifications may include both discrete transition constraints (e.g., p holds at the next time instant) and continuous time constraints (e.g., p holds 3 to 5 seconds from now) [117].

LTL cannot model continuous time constraints, such constraints require a more powerful logic like Metric Temporal Logic (MTL), but MTL satisfiability checking is undecidable in general [110].

However, if the MTL specification is restricted to discrete time steps, then the specification is just a more succint form of LTL with the same expressiveness: a bounded always operator can be unrolled into a conjunction of increasingly nested next operators (e.g., $\Box_{\leq 3}\phi \equiv \phi \wedge \circ\phi \wedge \circ(\circ\phi) \wedge \circ(\circ(\circ\phi))$, where $\Box$ is the always operator and $\circ$ is the next operator). Because of this equivalence discrete

MTL is sometimes refered to as Metric LTL. This results in an exponential blowup of the formula but the resulting formula can be solved by a standard LTL solver.

If we focus solely on bounded satisfiability checking, then the translation of the bounded temporal operators *always* and *eventually* in particular is about as expensive as in LTL, since the temporal unrolling is similar [110].

Discrete MTL as described above can be used to verify continuous-time problems if the system under consideration cannot change state faster than a given minimal time step (e.g., 0.01 seconds). It suffices to scale every operator time bound and the bounded-model checking time bound by a number that would make the minimal time-step an integer (e.g., multiplying every bound by 100).

A scaled formula like the above is very expensive to check. Thankfully, for certain subsets of MTL, there are more efficient approaches.

Metric Interval Temporal Logic (MITL) is a subset of MTL that disallows singleton time intervals (e.g, $\Box_{=t}$). Unlike MTL, MITL is decidable [26]. Satisfiability checking of MITL can be performed via a translation to Constrained LTL over clocks (CLTLoc) [26].

CLTL is the same as the LTL we support in RAGE, that is, LTL with arithmetic constraints. CLTLoc, in particular, introduces extra variables and constraints modelling various "clocks" for the temporal operators in the original MITL formula.

7.3 Optimization

In Systems Engineering it is not usually relevant to know all possible architectures (unless the goal is patenting). Rather, the objective is to find one or more architectures that are *pareto efficient* solutions for a given set of objective functions (i.e., solutions on the *pareto front*, a "border" outside which there are no better solutions). For example, finding vehicle architectures that maximize acceleration while minimizing fuel consumption.

There are also situations where we might have *soft* requirements, that is, possibly conflicting requirements we would like to fulfil but the system remains fit for purpose even if they are not satisfied. In that case we are searching for architectures that maximize the number of satisfied requirements.

Both kinds of optimization problems above can be integrated into our existing infrastructure, though not necessarily in the most efficient manner. Depending

on what is being optimized: static properties, dynamic properties or soft requirements, the problem has to be dealt with differently.

7.3.1 Optimizing Static Properties

If we're optimizing static properties of the system (e.g., weight and cost), then we can add an optimization goal to the architecture generation problem directly.

If the optimal architecture fails verification, we learn from the counter-example and repeat until we find a solution that satisfies the requirements. The problem with this naive approach is that the generator is searching for optimal solutions "blind". Mixing in the test generation mode, when applicable, should help performance considerably.

However, there is more we can do. Optimization software typically works better given a starting point: a feasible but suboptimal solution. We can first search for a valid architecture as usual to find such a feasible solution, and only then switch to optimization mode.

If the optimization result is negative, then we now have an upper and a lower bound on the problem. There are no worse solutions than the result of the valid solution (the upper bound) and no better solutions than the result of the failed solution (the lower bound). Given the two bounds we can now perform a binary search.

First, we find the middle point of the two bounds. Assuming a minimization problem, we ask the solver for a feasible solution in the bottom half. If we find one, then that's our new upper bound. If not, then we know the objective value is somewhere in the upper half.

We ask the solver for an optimal architecture in the upper half. If it is valid, then we have the best possible architecture. If the architecture is invalid, we now have a new lower bound. Repeat until we find an optimal architecture.

7.3.2 Optimizing Dynamic Properties

Optimizing dynamic properties, meaning properties that depend on the behavior of the system, is a lot trickier, since we're trying to find a solution that is optimal for all possible inputs [1]. Enumerating all input combinations may be prohibitively expensive, but we can work around it.

[1] if we are only concerned with optimal results for a single set of inputs this is much simpler

Additionally, there are two situations to consider: the behavior is fully specified in the requirements, or the exact behavior is only known through simulation.

The behavior is fully specified: In this case the requirements for the components each describe an outputs $= f(\text{inputs})$ relation. Given some inputs we know the exact output for that particular component (note: if we have such a relation for the system, then optimizing behavior makes no sense).

First we find a valid architecture as normal, let us call it a. Once we have a valid architecture, we can compute its outputs for any given inputs.

Choose some random input assignment x. Add the following constraint to the generator: there exists a different input assignment y and architecture b where b gives more pareto efficient results for x, y than a.

We then search for valid architectures as normal. If no new architectures are found, then the one we found previously was already optimal.

If we find another valid architecture c, then repeat the process: add the following constraint to the generator: there exists an input assignment z and architecture c where c gives more pareto efficient results for x, y, z than b.

The process above builds an increasingly accurate approximation of the pareto front. We repeat until no more valid architectures exist.

The behavior is only known through simulation: In this case we do not have the component specifications necessary to use the approach described above. "Less-specific" requirements are insufficient as that allows the solver to pick an arbitrary output within a range which might not actually occur in practice.

If we use surrogates as specifications, then we can use them as in the fully specified case except the results are potentially innacurate, we might find architectures that appear to be better but behave differently in practice, or we might not find a better architecture even though it exists.

We deal with each case separately: If the results of the current optimal architecture are not actually better when simulated, we forbid that solution and ask for another optimal solution. If the solver is unable to find a better architecture, then the last feasible solution serves as an upper bound (assuming minimization).

There is no good way to find a globally optimal architecture without component specifications. We would need to generate every possible architecture and evaluate each one on every possible set of inputs.

If there is only one relevant set of inputs (or no inputs at all, meaning we're evaluating a closed system), then we only need to run a simulation once per architecture which makes the problem tractable.

7.3.3 Optimizing Soft Requirements

Soft requirements have a similar distinction as optimizing an objective function: some requirements refer to static properties while others refer to behavior.

Soft requirements related to static properties are easy, we proceed in the same manner as optimizing an objective function on static properties.

Each soft requirement has an associated boolean variable, which wraps the soft requirement in an implication. The soft requirement is only active if the associated variable is active. We can then find pareto optimal sets of satisfied soft requirements by maximizing the sum of these boolean variables.

For behavioral requirements we need a different approach, as they must be dealt with in the verifier. Due to how the problem is structured, adding more requirements might make the verification formula satisfiable, meaning there's a counter example due to the introduction of the additional requirements.

Because of this we solve the problem by maximizing the number of *negated* soft requirement booleans.

If all soft requirement booleans were negated and the verifier finds a counter-example, then the architecture fails the hard requirements.

If all soft requirement booleans were negated, and the verifier cannot find a counter-example, then we have found the best possible architecture, one that fulfils every soft requirement.

The last possible case is that the verifier finds a counter-example but in order to do so it had to include some soft requirements r. We add a constraint $\neg(\bigwedge r)$ and verify again. We do this until there are no more satisfiable solutions (meaning no other combo of soft requirements causes the architecture to fail). With those constraints in place maximize the sum of the positive requirement booleans, that's the best set of soft requirements the function can fulfil.

We then add a constraint to the verifier that the number of positive soft requirements must be equal to or greater than that number for every other architecture, and repeat until no more valid architectures can be found.

7.4 Improved Solver Performance

The first major improvement would be to rewrite the project in a compiled programming language with direct access to the solver APIs (e.g., C and C++). This should result in a substantial reduction in the overhead detected throughout the solver evaluation.

The second major improvement would be to improve the test-generation mode to support LTL, improve its encoding to need less auxiliary variables, and use smart sampling approaches [107] for use cases with large input domains.

Split Ensemble Generation: Another improvement relates to the generation encoding. RAGE encodes the synthesis problem as a single $\exists\forall$ formula which may then be split off into two separate formulas when solving in a CEGIS or test-generation mode.

Following Simcenter Studio [9], we can further split generation in two, by solving ensemble generation and architecture generation separately. The presence of port instances can also be moved to ensemble generation, with additional constraints to ensure "enough" output ports of each type exist for the input ports. In RAGE this is slightly more complicated than in Simcenter Studio because output ports may connect any number of times.

Since the ensemble generator has less information than the combined generator, it will produce ensembles that may not have any corresponding architectures (i.e., there is no satisfiable set of connections for that ensemble), or none of the architectures for a particular ensemble are valid.

But the separation provides two advantages: 1) each ensemble results in a separate architecture generation problem, which can be solved in parallel, 2) the verification problem no longer needs to consider component instance and port instance variables.

The lack of presence variables results in a major simplification of the encoding of the verification problem: functions and predicates over port groups can be encoded by a simple unrolling, no need to include checks for the presence of port instances in the encoding. Since quantified formulas are expensive to solve, it is highly desirable to simplify them as much as possible.

Parallel LTL Unrollings: Another simple trick to increase performance is to use a separate verification solver for each unrolling of the LTL formula up to the max time bound k. Smaller unrollings are faster to check and it is likely

that many of the generated architectures will fail verification at step 0. This also results in smaller counter examples.

It would also be possible to verify every bound up to k in parallel and stop once one of the verifiers returns a counter-example.

Minimal Blocking Clauses: The CEGIS mode implemented in this thesis adds the negation of a counter-example to the generation solver, generalized if possible. Only the Yices 2 backend solver exposes a model generalization API.

However, because all of the existential variables are boolean, another approach would be to compute the minimal set of assumption literals that makes the negated verification formula satisfiable.

This set of literals encodes a subgraph pattern (including missing edges) that will always lead to a non-working system implementation.

Asserting a minimal blocking clause consisting solely of those literals also blocks all assignments to other variables not part of that set, achieving the same outcome as a counter example but in a smaller formula. One possible implementation is as follows:

- Generate and verify an architecture. The generator produces an assignment to the architecture-related variables that is given as assumptions to the verifier. The verifier returns a counter-example as an assignment of the state variables.
- Substitute every state variable in the counter example by its assignment, this turns the formula into a pure boolean formula once simplified, containing only the assumption variables.
- For each assumption variable, flip its value (true to false, false to true) and check the satisfiability of the counter-example formula. If the counter example remains satisfiable after the flip, that variable is not *relevant*, meaning it is not necessary to ensure satisfiability.
- Assert a blocking clause consisting solely of the *relevant* variables.

Due to the impact it has on the number of architectures explored, improved symmetry breaking would also contribute greatly to performance.

7.5 Improved Symmetry Breaking

There are four avenues to explore in order to improve the symmetry breaking in RAGE: 1) component-level symmetry breaking, 2) alternative symmetry breaking constraints, 3) abstract connections, and 4) SAT modulo symmetries.

7.5.1 Component-Level Symmetry Breaking

The symmetry breaking constraints in RAGE deal mainly with ports. We use standard double-lex where swaps of ports (as rows and columns) are constrained to be lexicographically smaller than the unswapped adjacency matrix.

For components, there are special constraints like the ones used in the half-adder and full-adder examples where instances of the same component are only allowed to connect forward (meaning instance 2's output ports cannot connect to instance 1's input ports).

But we could also apply a lexicographic ordering to components. The issue is that swapping two components means swapping all of their connections, and that might conflict with other constraints in the system. For example, if we have the forward-only connection constraint, then we can only swap component instance 1 with instance 3 if component instance 1 is not connected to component instances 2 or 3. The constraints would be in conflict in that case.

Additionally, since we're swapping multiple connections by swapping entire instances, we need to have some sort of ordering on sets of connections. We want the complete swap to be lexicographically smaller, for some definition of smaller.

7.5.2 Alternative Symmetry Breaking Constraints

In this thesis we used standard lex-leader constraints, but those are not the only way to break symmetries in graph problems.

Ulyantsev et. al. implemented a symmetry breaking predicate based on a Breath-First Search (BFS)-inspired enumeration of the nodes of the graph in a finite automata identification problem [124]. The same approach was later adapted to arbitrary undirected graphs [96].

To use these predicates we must ensure that they don't conflict with other constraints in our encoding. Note however that despite having input and output ports, architectures where ports are explicitly modeled can be treated

as undirected graphs. An output connected to an input or an input to an output means the same thing, all that matters is that edges between outputs and outputs and inputs and inputs are not allowed.

The BFS predicate does require a fully connected graph, which considering only the ports is not the case. We need to also model the connection between nodes and ports, at least as far as the predicate is concerned. Connections between a component and its own ports don't vary so we just need to make sure that the BFS enumeration takes them correctly into account.

Another approach to explore involves emulating the algorithm used by the nauty [93] graph isomorphism tool to compute canonical representations as constraints [41]. We will not go into detail here on how this approach works but the main requirement to use it is to add support for node labels as supported by nauty, where only nodes with the same label can be swapped.

7.5.3 Abstract Connections

We currently model every possible connection between port instances in RAGE, but this is actually more information than we need in the generation stage.

For example, if we have two port groups a and b, one with two output ports and one with two input ports, then we have the following possibilities:

- there are no connections between the groups.
- one port instance of a is connected to one port instance of b.
- one port instance of a is connected to both port instances of b.
- each port instance of a is connected to a port instance of b.

Since the port instances of a group are equivalent, it doesn't really matter which particular instances are connected when looking at the groups on their own. We don't need to model every possible pairing of ports.

When looking at the complete architecture we do need to know the specific output port, however (but not the input port). The reason being is that since output ports may connect multiple times, there's a difference between an output port instance of a group connecting 2 times to one group and 1 time to another (for a total of 3 connections for that instance) or another instance being responsible for the latter connection (for a 2-1 split).

Even so, this reduces the number of alternatives to be explored for the group. The following lists show possibilities the current implementation needs to consider versus the abstracted variant:

- Concrete: $[(), (0 \to 0), (0 \to 1), (1 \to 0), (1 \to 1), (0 \to 0, 0 \to 1), (1 \to 0, 1 \to 1), (0 \to 0, 1 \to 1), (1 \to 0, 0 \to 1)]$, 9 possibilities.
- Abstract: $[(0, 0), (1, 0), (0, 1), (2, 0), (0, 2), (1, 1)]$, 6 possibilities.

The above already serves as a form of symmetry breaking, as each abstract configuration represents a set of possible concrete configurations.

But we can take the abstraction further. Instead of treating port groups as nodes with edges between them, we treat connections between port groups as *edge labels*, and we model architectures as labeled directed multigraphs, i.e., graphs with multiple edges between nodes where both nodes and edges can have labels.

Then, for symmetry breaking, we first focus on the component level. Each row and column of our adjacency matrix represents a component, and each element of the matrix stores a set of edge labels.

Each edge label is a tuple of integers representing the number of connections from each output port instance. Edge labels form a partial order: $(2, 0)$ is better than $(1, 0)$, but $(2, 0)$ and $(1, 1)$ are incomparable. All elements of the label must compare in the same direction with those of the other label. Sets of edges also form a partial order: to be comparable the labels of each edge must be comparable, and all labels must compare in the same direction.

These symmetry breaking constraints are harder to encode due to the partial order, but the trick is to flip the constraints. Rather than enforcing a lexicographic order by stating that a swap must be lexicographically smaller or equal to the original adjacency matrix, we require that a swap must *not* be greater. If the comparison returns false due to not being comparable, then the negation effectively turns off the constraint.

Then we also need to break the remaining port symmetries. If output port instances are only allowed to connect once each, then we can also abstract output port instances away entirely and just count the total number of connections between port groups.

If ports are allowed to connect more than once, then as explained previously we need to know how many connections are coming from each instance. However, we can enforce a preference towards lower instances: $(2, 0)$ is preferable to $(0, 2)$.

This can also be encoded by enforcing a lexicographic order, but in this case one between the variables representing the count of connections stemming from each output port instance.

7.5.4 SAT Modulo Symmetries

All of the previous ideas for symmetry breaking revolve around *static* symmetry breaking, meaning symmetry breaking through the addition of various constraints to the problem before it is solved.

Another approach is to break symmetries during the search. SAT Modulo Symmetries [75] achieves this by adding a special symmetry-breaking propagator to a SAT solver, in a similar way to how SMT solvers work.

As the SAT solver assigns values to the variables representing nodes and edges of the graph, the symmetry-breaking propagator runs a minimality check on the partially defined graph. A graph is minimal if its adjacency matrix is lexicographically smaller than every possible swap of that matrix, and a partial graph is minimal if there is an extension of the assignment that leads to a complete minimal graph.

If the minimality check fails, a clause is added representing the reason why the partial graph cannot be part of a minimal graph, e.g., an edge that must be or cannot be present in a minimal extension.

The advantage of the dynamic approach is that a complete static symmetry break can have a prohibitively large encoding size (the number of constraints grows exponential with the size of the problem). In SAT Modulo Symmetries constraints are added as necessary during the search, and each such constraint might imply many others that no longer need to be explicitly enumerated.

Bibliography

[1] Dessia: The Augmented Intelligence Software Framework for Concurrent Engineering. https://www.dessia.io, accessed on 2024-11-13.

[2] Modelica. https://modelica.org, accessed on 2024-10-22.

[3] MPS file format. https://lpsolve.sourceforge.net/5.5/mps-format.htm accessed on 2024-10-25.

[4] OCRA - Examples. https://ocra.fbk.eu/examples.html accessed on 2024-10-28.

[5] OCRA user manual. https://ocra.fbk.eu/download/OCRA_Language_User_Guide.pdf, accessed on 2024/10/12.

[6] OMG System Modeling Language Specification Version 2.0. https://www.omg.org/spec/SysML/2.0/Beta2/About-SysML accessed on 2024-10-22.

[7] SAT Competitions. https://satcompetition.github.io/ accessed on 2024-10-10.

[8] Simcenter amesim. https://plm.sw.siemens.com/en-US/simcenter/systems-simulation/amesim/, accessed on 2024-10-22.

[9] Simcenter Studio. https://plm.sw.siemens.com/en-US/simcenter/integration-solutions/studio/ accessed on 2024-08-17.

[10] Stimulus. https://www.3ds.com/products/catia/stimulus, accessed on 2024-10-22.

[11] The Chuffed CP Solver. https://github.com/chuffed/chuffed, accessed on 2024-9-23.

[12] The Open Source AADL Tool Environment (osate). https://osate.org/index.html, accessed on 2024/12/4.

[13] Logics and integer-programming representations, Sept. 2016.

[14] IBM ILOG CPLEX Optimization Studio, Dec. 2022. `https://www.ibm.com/docs/en/icos/22.1.1?topic=cplex-lp-file-format-algebraic-representation` accessed on 2024-10-25.

[15] Abrial, J.-R. *Modeling in Event-B: System and Software Engineering*, illustrated edition ed. Cambridge University Press, Cambridge, June 2010.

[16] Achterberg, T. SCIP: solving constraint integer programs. *Mathematical Programming Computation 1*, 1 (July 2009), 1–41.

[17] Allen, J., and Kouppas, P. Computer Aided Design: Past, Present, Future. Jan. 2012, pp. 97–111.

[18] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design* (Oct. 2013), pp. 1–8.

[19] Arnold, V. I., and Silverman, R. A. *Ordinary Differential Equations*, revised ed. edition ed. The MIT Press, July 1978.

[20] Audet, C., and Hare, W. *Derivative-Free and Blackbox Optimization*, 1st ed. 2017 edition ed. Springer, Cham, Dec. 2017.

[21] Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I* (2022), D. Fisman and G. Rosu, Eds., vol. 13243 of *Lecture Notes in Computer Science*, Springer, pp. 415–442.

[22] Barrett, C., Fontaine, P., and Tinelli, C. The SMT-LIB Standard: Version 2.6. `www.smt-lib.org` accessed on 2024-08-10.

[23] Beldiceanu, N., Carlsson, M., Demassey, S., and Petit, T. Global Constraint Catalogue: Past, Present and Future. *Constraints 12*, 1 (Mar. 2007), 21–62.

[24] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A.,

Damm, W., Henzinger, T. A., and Larsen, K. G. Contracts for System Design. *Foundations and Trends® in Electronic Design Automation 12*, 2-3 (2018), 124–400.

[25] Bernhardsson, B. Explicit solutions to the N-queens problem for all N. *SIGART Bull. 2*, 2 (Feb. 1991), 7.

[26] Bersani, M. M., Rossi, M., and San Pietro, P. An SMT-based approach to satisfiability checking of MITL. *Information and Computation 245* (Dec. 2015), 72–97.

[27] Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., and Pollitt, F. CaDiCaL 2.0. In *Computer Aided Verification* (Cham, 2024), A. Gurfinkel and V. Ganesh, Eds., Springer Nature Switzerland, pp. 133–152.

[28] Biere, A., Fazekas, K., Fleury, M., and Heisinger, M. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions* (2020), T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2020-1 of *Department of Computer Science Report Series B*, University of Helsinki, pp. 51–53.

[29] Biere, A., Heule, M., Maaren, H. V., and Walsh, T. *Handbook of Satisfiability*, 2nd edition ed. IOS Press, Mar. 2021.

[30] Blazek, J. *Computational Fluid Dynamics: Principles and Applications*, 3rd edition ed. Butterworth-Heinemann, Amsterdam, Apr. 2015.

[31] Borralleras, C., Larraz, D., Rodríguez-Carbonell, E., Oliveras, A., and Rubio, A. Incomplete SMT Techniques for Solving Non-Linear Formulas over the Integers. *ACM Trans. Comput. Logic 20*, 4 (Aug. 2019), 25:1–25:36.

[32] Borutzky, W. *Bond Graph Methodology: Development and Analysis of Multidisciplinary Dynamic System Models*, 2010th edition ed. Springer, London, Feb. 2010.

[33] Boussemart, F., Lecoutre, C., Audemard, G., and Piette, C. XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems, Aug. 2024.

[34] Bradley, A. R. SAT-Based Model Checking without Unrolling. R. Jhala and D. Schmidt, Eds., vol. 6538, Springer Berlin Heidelberg, pp. 70–87. Book Title: Verification, Model Checking, and Abstract Interpretation Series Title: Lecture Notes in Computer Science.

[35] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, Dec. 2020), NIPS '20, Curran Associates Inc., pp. 1877–1901.

[36] Cagan, J., Campbell, M., Finger, S., and Tomiyama, T. A Framework for Computational Design Synthesis: Model and Applications. *Journal of Computing and Information Science in Engineering - JCISE 5* (Sept. 2005).

[37] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)* (Copenhagen, Denmark, July 2002), vol. 2404 of *LNCS*, Springer.

[38] Cimatti, A., Dorigatti, M., and Tonetta, S. OCRA: A tool for checking the refinement of temporal contracts. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov. 2013), 702–705.

[39] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2013), N. Piterman and S. A. Smolka, Eds., Springer, pp. 93–107.

[40] Cimatti, A., Roveri, M., and Tonetta, S. HRELTL: A temporal logic for hybrid systems. *Information and Computation 245* (Dec. 2015), 54–71.

[41] Codish, M., Gange, G., Itzhakov, A., and Stuckey, P. J. Breaking Symmetries in Graphs: The Nauty Way. M. Rueher, Ed., vol. 9892, Springer International Publishing, pp. 157–172. Book Title: Principles and Practice of Constraint Programming Series Title: Lecture Notes in Computer Science.

[42] Codish, M., Miller, A., Prosser, P., and Stuckey, P. J. Breaking symmetries in graph representation. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence* (Beijing, China, Aug. 2013), IJCAI '13, AAAI Press, pp. 510–516.

[43] Cofer, D., Gacek, A., Miller, S., Whalen, M. W., LaValley, B., and Sha, L. Compositional Verification of Architectural Models. In *NASA Formal Methods* (Berlin, Heidelberg, 2012), A. E. Goodloe and S. Person, Eds., Springer, pp. 126–140.

[44] de Moura, L., and Bjørner, N. Model-based Theory Combination. *Electronic Notes in Theoretical Computer Science 198*, 2 (May 2008), 37–49.

[45] De Moura, L., and Bjørner, N. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems* (Berlin, Heidelberg, Mar. 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[46] Delange, J. *AADL In Practice: Become an expert of software architecture modeling and analysis*. Reblochon Development Company, June 2017.

[47] Donzé, A. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Computer Aided Verification* (Berlin, Heidelberg, 2010), T. Touili, B. Cook, and P. Jackson, Eds., Springer, pp. 167–170.

[48] Dutertre, B. Yices 2.2. In *Computer-Aided Verification (CAV'2014)* (July 2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744.

[49] Dutertre, B. Solving Exists/Forall Problems With Yices, 2015. `https://yices.csl.sri.com/papers/smt2015.pdf` accessed on 2024-08-15.

[50] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering* (New York, NY, USA, May 1999), ICSE '99, Association for Computing Machinery, pp. 411–420.

[51] Eén, N., Mishchenko, A., and Brayton, R. Efficient implementation of property directed reachability. *2011 Formal Methods in Computer-Aided Design (FMCAD)* (Oct. 2011).

[52] Frisch, A. M., and Giannaros, P. A. SAT Encodings of the At-Most-k Constraint Some Old , Some New , Some Fast , Some Slow.

[53] Gacek, A., Backes, J., Whalen, M., Wagner, L., and Ghassabani, E. The JKind Model Checker. In *Computer Aided Verification* (Cham, 2018), H. Chockler and G. Weissenbacher, Eds., Springer International Publishing, pp. 20–27.

[54] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, first edition ed. W. H. Freeman, New York u.a, Jan. 1979.

[55] Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011* (2011), G. J. Gordon, D. B. Dunson, and M. Dudík, Eds., vol. 15 of *JMLR Proceedings*, JMLR.org, pp. 315–323.

[56] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial networks. *Commun. ACM 63*, 11 (Oct. 2020), 139–144.

[57] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024. `https://www.gurobi.com` accessed on 2024-10-25.

[58] Hadarean, L., Bansal, K., Jovanović, D., Barrett, C., and Tinelli, C. A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors. In *Computer Aided Verification* (Cham, 2014), A. Biere and R. Bloem, Eds., Springer International Publishing, pp. 680–695.

[59] Heinen, M. The A380 Program, Nov. 2006. `https://web.archive.org/web/20061103062416/http://www.eads.com/xml/content/OF00000000400004/0/74/41485740.pdf` accessed on 2024-08-15.

[60] Henderson, R., and Clark, K. Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms. *Administrative science quarterly 35* (Mar. 1990).

[61] Heule, M. J. The Quest for Perfect and Compact Symmetry Breaking for Graph Problems. In *2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (Sept. 2016), pp. 149–156. ISSN: 2470-881X.

[62] Heule, M. J. H., Kullmann, O., Wieringa, S., and Biere, A. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In *Hardware and Software: Verification and Testing* (Berlin, Heidelberg, 2012), K. Eder, J. Lourenço, and O. Shehory, Eds., Springer, pp. 50–65.

[63] Holt, J., and Perry, S. *SysML for Systems Engineering: A Model-Based Approach*, 3rd ed. edition ed. Institution of Engineering & Technology, Jan. 2019.

[64] Huang, X., and Lee, J. Doublelex revisited and beyond. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China, Aug. 2019), IJCAI'19, AAAI Press, pp. 1101–1107.

[65] Huangfu, Q., and Hall, J. A. J. Parallelizing the dual revised simplex method. *Mathematical Programming Computation 10*, 1 (Mar. 2018), 119–142.

[66] Iannopollo, A., Tripakis, S., and Sangiovanni-Vincentelli, A. *Constrained Synthesis from Component Libraries*. Springer International Publishing, Cham, 2017, pp. 92–110.

[67] Incer Romeo, I., Sangiovanni-Vincentelli, A., Lin, C.-W., and Kang, E. Quotient for Assume-Guarantee Contracts. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)* (Oct. 2018), pp. 1–11.

[68] INCOSE, Ed. *INCOSE Systems Engineering Handbook*, 5th edition ed. Wiley, July 2023.

[69] Jacobs, S., Perez, G., and schlehuber caissier, P. *The Temporal Logic Synthesis Format TLSF v1.2*. Mar. 2023.

[70] Jansson, D., and Smith, S. Design fixation. *Design Studies 12* (Jan. 1991).

[71] Jha, S., and Seshia, S. A. A theory of formal synthesis via inductive learning. *Acta Informatica 54*, 7 (Nov. 2017), 693–726.

[72] Kaelbling, L. P., Littman, M. L., and Moore, A. W. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research 4* (May 1996), 237–285.

[73] Karhula, H. O. H., Desmet, W., Van der Auweraer, H., and Nicolai, M. Automated concept design for automatic automotive transmissions, 2020-06-02.

[74] King, T., Barrett, C., and Tinelli, C. Leveraging Linear and Mixed Integer Programming for SMT. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design* (Austin, Texas, Oct. 2014), FMCAD '14, FMCAD Inc, pp. 139–146.

[75] Kirchweger, M., and Szeider, S. SAT Modulo Symmetries for Graph Generation and Enumeration. *ACM Trans. Comput. Logic 25*, 3 (July 2024), 18:1–18:30.

[76] Kleiner, S., and Kramer, C. Model Based Design with Systems Engineering Based on RFLP Using V6. M. Abramovici and R. Stark, Eds., Springer Berlin Heidelberg, pp. 93–102. Book Title: Smart Product Engineering Series Title: Lecture Notes in Production Engineering.

[77] Kojevnikov, A., Kulikov, A. S., and Yaroslavtsev, G. Finding Efficient Circuits Using SAT-Solvers. In *Theory and Applications of Satisfiability Testing - SAT 2009* (Berlin, Heidelberg, 2009), O. Kullmann, Ed., Springer, pp. 32–44.

[78] Konrad, S., and Cheng, B. H. C. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, May 2005), ICSE '05, Association for Computing Machinery, pp. 372–381.

[79] Kotsiantis, S. B. Decision trees: a recent overview. *Artificial Intelligence Review 39*, 4 (Apr. 2013), 261–283.

[80] Kremer, G., Reynolds, A., Barrett, C., and Tinelli, C. Cooperating Techniques for Solving Nonlinear Real Arithmetic in the cvc5 SMT Solver (System Description). In *Automated Reasoning: 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8–10, 2022, Proceedings* (Berlin, Heidelberg, Aug. 2022), Springer-Verlag, pp. 95–105.

[81] Ladyman, J., Lambert, J., and Wiesner, K. What is a complex system? *European Journal for Philosophy of Science 3*, 1 (Jan. 2013), 33–67.

[82] Lanzarotto, D., Marchesoni, M., Passalacqua, M., Prato, A. P., and Repetto, M. Overview of different hybrid vehicle architectures. *IFAC-PapersOnLine 51*, 9 (2018), 218 – 222.

[83] Latvala, T., Biere, A., Heljanko, K., and Junttila, T. Simple Bounded LTL Model Checking. D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A. J. Hu, and A. K. Martin, Eds., vol. 3312, Springer Berlin Heidelberg, pp. 186–200. Book Title: Formal Methods in Computer-Aided Design Series Title: Lecture Notes in Computer Science.

[84] Latvala, T., Biere, A., Heljanko, K., and Junttila, T. Simple Is Better: Efficient Bounded Model Checking for Past LTL. In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2005), R. Cousot, Ed., Springer, pp. 380–395.

[85] Lecoutre, C., Saïs, L., Tabary, S., and Vidal, V. Recording and Minimizing Nogoods from Restarts. *Journal on Satisfiability, Boolean Modeling and Computation 1*, 3-4 (Jan. 2006), 147–167.

[86] Lee, J. H. M., Schulte, C., and Zhu, Z. Increasing nogoods in restart-based search. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona, Feb. 2016), AAAI'16, AAAI Press, pp. 3426–3433.

[87] Mackworth, A. K. Consistency in networks of relations. *Artificial Intelligence 8*, 1 (Feb. 1977), 99–118.

[88] Madni, A. M., Augustine, N., and Sievers, M. *Handbook of Model-Based Systems Engineering*, 2023rd edition ed. Springer, July 2023.

[89] Maler, O., and Nickovic, D. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems Series Title: Lecture Notes in Computer Science* (Berlin, Heidelberg, 2004), D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, Y. Lakhnech, and S. Yovine, Eds., vol. 3253, Springer Berlin Heidelberg, pp. 152–166.

[90] Martin, F., and Rebecca, P. *Domain-Specific Languages*, 1st edition ed. Addison Wesley, Upper Saddle River, NJ, Sept. 2010.

[91] Marwedel, P. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*, 4th ed. 2021 edition ed. Springer, Jan. 2021.

[92] Masfaraud, S., Danes, F., Dumouchel, P.-E., De Vuyst, F., and Vayatis, N. Automatized gearbox architecture design exploration by exhaustive graph generation. In *WCCM XII 2016* (July 2016).

[93] McKay, B. D., and Piperno, A. Practical graph isomorphism, II. *Journal of Symbolic Computation 60* (Jan. 2014), 94–112.

[94] Menu, J., Nicolai, M., and Zeller, M. Designing Fail-Safe Architectures for Aircraft Electrical Power Systems. In *2018 AIAA/IEEE Electric Aircraft Technologies Symposium (EATS)* (July 2018), pp. 1–14.

[95] Mitchell, M. *An Introduction to Genetic Algorithms*, reprint edition ed. MIT Press, Cambridge, Mass., Feb. 1998.

[96] Moklev, V., and Ulyantsev, V. BFS Enumeration for Breaking Symmetries in Graphs. *ArXiv* (Apr. 2018).

[97] Molnár, V., Graics, B., Vörös, A., Tonetta, S., Cristoforetti, L., Kimberly, G., Dyer, P., Giammarco, K., Koethe, M., Hester,

J., Smith, J., and Grimm, C. Towards the Formal Verification of SysML v2 Models. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems* (New York, NY, USA, Oct. 2024), MODELS Companion '24, Association for Computing Machinery, pp. 1086–1095.

[98] Morgado, A., Heras, F., Liffiton, M., Planes, J., and Marques-Silva, J. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints 18*, 4 (Oct. 2013), 478–534.

[99] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. MiniZinc: towards a standard CP modelling language. In *Proceedings of the 13th international conference on Principles and practice of constraint programming* (Berlin, Heidelberg, Sept. 2007), CP'07, Springer-Verlag, pp. 529–543.

[100] Nguyen, T. *FORM-L: A MODELICA Extension for Properties Modelling Illustrated on a Practical Example*. Mar. 2014. Pages: 1236.

[101] Niemetz, A., and Preiner, M. Bitwuzla. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II* (2023), C. Enea and A. Lal, Eds., vol. 13965 of *Lecture Notes in Computer Science*, Springer, pp. 3–17.

[102] Nuzzo, P., Xu, H., Ozay, N., Finn, J., Vincentelli, A., Murray, R., Donzé, A., and Seshia, S. A Contract-Based Methodology for Aircraft Electric Power System Design. *Access, IEEE 2* (Jan. 2014), 1–25.

[103] Ohrimenko, O., Stuckey, P. J., and Codish, M. Propagation = Lazy Clause Generation. In *Principles and Practice of Constraint Programming – CP 2007* (Berlin, Heidelberg, 2007), C. Bessière, Ed., Springer, pp. 544–558.

[104] Papalexopoulos, T. P., Tjandraatmadja, C., Anderson, R., Vielma, J. P., and Belanger, D. Constrained Discrete Black-Box Optimization using Mixed-Integer Programming. In *Proceedings of the 39th International Conference on Machine Learning* (June 2022), PMLR, pp. 17295–17322. ISSN: 2640-3498.

[105] Pardalos, P. M., and Resende, M. G. C., Eds. *Handbook of Applied Optimization*, 1st edition ed. Oxford University Press, Feb. 2002.

[106] Perron, L., and Didier, F. Cp-sat.

[107] Pesant, G., Quimper, C.-G., and Verhaeghe, H. Practically Uniform Solution Sampling in Constraint Programming. In *Integration of*

Constraint Programming, Artificial Intelligence, and Operations Research (Cham, 2022), P. Schaus, Ed., Springer International Publishing, pp. 335–344.

[108] Phan, Q.-S., and Malacaria, P. All-Solution Satisfiability Modulo Theories: Applications, Algorithms and Benchmarks. In *2015 10th International Conference on Availability, Reliability and Security* (Aug. 2015), pp. 100–109.

[109] Pnueli, A., and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, Jan. 1989), POPL '89, Association for Computing Machinery, pp. 179–190.

[110] Pradella, M., Morzenti, A., and Pietro, P. S. Bounded satisfiability checking of metric temporal logic specifications. *ACM Trans. Softw. Eng. Methodol. 22*, 3 (July 2013), 20:1–20:54.

[111] Prud'homme, C., and Fages, J.-G. Choco-solver: A java library for constraint programming. *Journal of Open Source Software 7*, 78 (2022), 4708.

[112] Roehm, H., Heinz, T., and Mayer, E. C. STLInspector: STL Validation with Guarantees. In *Computer Aided Verification* (Cham, 2017), R. Majumdar and V. Kunčak, Eds., Springer International Publishing, pp. 225–232.

[113] Rossi, F., Beek, P. v., and Walsh, T., Eds. *Handbook of Constraint Programming*, 1st edition ed. Elsevier Science, Amsterdam ; Boston, Oct. 2006.

[114] Rozenberg, G., Ed. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Pub Co Inc, Singapore u.a, Jan. 1997.

[115] Rozier, K. Y. Linear Temporal Logic Symbolic Model Checking. *Computer Science Review 5*, 2 (May 2011), 163–203.

[116] Ruijters, E., and Stoelinga, M. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review 15-16* (Feb. 2015), 29–62.

[117] Schupp, S., Ábrahám, E., Chen, X., Ben Makhlouf, I., Frehse, G., Sankaranarayanan, S., and Kowalewski, S. Current Challenges in the Verification of Hybrid Systems. In *Cyber Physical Systems. Design, Modeling, and Evaluation* (Cham, 2015), M. R. Mousavi and C. Berger, Eds., Springer International Publishing, pp. 8–24.

[118] Sheeran, M., Singh, S., and Stålmarck, G. Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design* (Berlin, Heidelberg, Nov. 2000), FMCAD '00, Springer-Verlag, pp. 108–125.

[119] Shoham, Y., and Leyton-Brown, K. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, 1st edition ed. Cambridge University Press, Cambridge ; New York, Dec. 2008.

[120] Shukla, A., Biere, A., Pulina, L., and Seidl, M. A Survey on Applications of Quantified Boolean Formulas. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)* (Nov. 2019), pp. 78–84.

[121] Soos, M., Nohl, K., and Castelluccia, C. Extending SAT Solvers to Cryptographic Problems. In *Theory and Applications of Satisfiability Testing - SAT 2009* (Berlin, Heidelberg, 2009), O. Kullmann, Ed., Springer, pp. 244–257.

[122] Tack, G. *Constraint propagation: models, techniques, implementation.* PhD thesis, Saarland University, Germany, 2009.

[123] Tseitin, G. S. On the Complexity of Derivation in Propositional Calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, J. H. Siekmann and G. Wrightson, Eds. Springer, Berlin, Heidelberg, 1983, pp. 466–483.

[124] Ulyantsev, V., Zakirzyanov, I., and Shalyto, A. BFS-Based Symmetry Breaking Predicates for DFA Identification. In *Language and Automata Theory and Applications* (Cham, 2015), A.-H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, Eds., Springer International Publishing, pp. 611–622.

[125] Van der Auweraer, H., Anthonis, J., De Bruyne, S., and Leuridan, J. Virtual Engineering at Work: the Challenges for Designing Mechatronic Products. *Engineering with Computers 29*, 3 (Sept. 2012), 389–408.

[126] Vanhuyse, J., Desmet, W., and Van Der Auweraer, H. Closed-loop Design Methodology for Hybrid Electric Vehicle Powertrains, May 2018.

[127] Wheeler, J., Ed. *Modeling and Simulation in Science, Technology and Engineering.* States Academic Press, Sept. 2022.

[128] Williams, H., and Yan, H. Representations of the all_different Predicate of Constraint Satisfaction in Integer Programming. *INFORMS Journal on Computing 13*, 2 (May 2001), 96–103.

[129] Wolsey, L. A. *Integer Programming*, 2nd edition ed. Wiley, Hoboken, NJ Chichester, West Sussex, Oct. 2020.

[130] Wouters, L., Creff, S., Effa, E., and Koudri, A. Collaborative systems engineering: Issues & challenges. pp. 486–491.

List of Publications

- C. A. R. Santos, A. H. Saleh, T. Schrijvers and M. Nicolai, "CONDEnSe: Contract Based Design Synthesis", *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Munich, Germany, 2019, pp. 250-260
- C. A. R. Santos, T. Schrijvers, A. H. Saleh, M. Nicolai, "Divide et Impera: Efficient Synthesis of Cyber-Physical System Architectures from Formal Contracts", *Formal Methods*, Beijing, China, 2021, pp. 776-787.



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
cesar.santos@student.kuleuven.be