Rigorous Simulation: Its Theory and Applications

by

Adam Duracz

Main supervisor: Walid Taha
Co-supervisor: Veronica Gaspes

A thesis submitted for the degree of
Doctor of Philosophy
at
Halmstad University
Dedicated to my parents Andrzej and Anna, and to my brother Jan.
Abstract

Designing Cyber-Physical Systems is hard. Physical testing can be slow, expensive and dangerous. Furthermore computational components make testing all possible behavior unfeasible. Model-based design mitigates these issues by making it possible to iterate over a design much faster. Traditional simulation tools can produce useful results, but their results are traditionally approximations that make it impossible to distinguish a useful simulation from one dominated by numerical error. Verification tools require skills in formal specification and a priori understanding of the particular dynamical system being studied.

This thesis presents rigorous simulation, an approach to simulation that uses validated numerics to produce results that quantify and bound all approximation errors accumulated during simulation. This makes it possible for the user to objectively and reliably distinguish accurate simulations from ones that do not provide enough information to be useful. Explicitly quantifying the error in the output has the side-effect of leading to a tool for dealing with inputs that come with quantified uncertainty.

We formalize the approach as an operational semantics for a core subset of the domain-specific language Acumen. The operational semantics is extended to a larger subset through a translation. Preliminary results toward proving the soundness of the operational semantics with respect to a denotational semantics are presented. A modeling environment with a rigorous simulator based on the operational semantics is described. The implementation is portable, and its source code is freely available. The accuracy of the simulator on different kinds of systems is explored through a set of benchmark models that exercise different aspects of a rigorous simulator. A case study from the automotive domain is used to evaluate the applicability of the simulator and its modeling language. In the case study, the simulator is used to compute rigorous bounds on the output of a model.
Acknowledgments

Many people have contributed to Acumen, its theory and the applications described in this thesis. The design of the language and its implementation was led by Walid Taha. Initial work on the user interface and traditional interpreters was done by Paul Brauner. Initial work on the enclosure interpreter was done by Jan Duracz. Initial work on the optimized traditional interpreter was done by Kevin Atkinson. The 3D visualization functionality was developed by Fei Xu and Yingfu Zeng. The current traditional and enclosure interpreters, differential equation solvers and underlying libraries were developed by myself and Ferenc A. Bartha.

The syntax and semantics (Chapters 4 and 5) are joint work with Eugenio Moggi, Ferenc A. Bartha, and Walid Taha. Eugenio Moggi contributed the denotational semantics and helped in proving the soundness of the operational semantics with respect to the denotational semantics. The case study on automotive safety (Chapter 7) is joint work with Ferenc A. Bartha, Ayman Aljarbouh, Jawad Masood, Roland Philippson, Henrik Eriksson, Jan Duracz, Fei Xu, Yingfu Zeng, Walid Taha, and Christian Grante. The case study on accuracy (Chapter 8) is joint work with Ferenc A. Bartha and Walid Taha.

This work was supported by the Center for Research on Embedded Systems (CERES) at Halmstad University, the Swedish Knowledge Foundation (KK), Vinnova, and US National Science Foundation award CPS-1136099.

On a more personal note, there are several people I want to thank, without whom I could not have finished this dissertation. My main supervisor Walid Taha for countless discussions and patiently reminding me to lift my gaze to see what was in front of me. My co-supervisor Veronica Gaspes for always finding the time to talk when I most needed it. Bertil Svensson, Roland Philippson and Stefan Byttner for their support and guidance. My brother Jan for introducing me to mathematics, convincing me to pursue a PhD, and for letting me begin this journey together with him. Ferenc A. Bartha, Amin Far-
judian and Michal Konečný for giving me courage to dig deeper into computable analysis. Eugenio Moggi for many early mornings and late nights exploring the formal semantics of Acumen. Yingfu Zeng, Fei Xu, Kevin Atkinson, Paul Brauner and other members of the EMG for being on my team, letting me contribute to their work, and putting up with my many demands. Kevin, a special thanks for creating aspell $\text{[A+2]}$. Saeed Gholami Shahbandi, Anita Sant’Anna and Kevin LeBlanc for always being there. All my colleagues at Halmstad University and climbers at HKK, for providing the fuel that kept me going over the past five years.

Finally, I would like to thank Kazunori Ueda for agreeing to serve as opponent, and Luc Jaulin, Ian Mitchell, David Broman and Josef Bigun for agreeing to serve as examiners on my defense.
## Contents

**Abstract**  
3

**Acknowledgments**  
4

**Notation**  
12

I Background  
19

1 Introduction  
21
  1.1 Motivation  
  1.2 Thesis  
  1.3 Contributions  
  1.4 History and Related Publications  
29

2 State of the Art  
29
  2.1 Modeling Formalisms  
  2.2 Analysis and Tools  
33

3 Traditional Simulation in Acumen  
39
  3.1 Past Work on Acumen  
  3.2 Equations  
  3.3 Sequences of Equations  
  3.4 If Statements  
  3.5 Match Statements  
  3.6 Model Definitions and Instantiation  
42


CONTENTS

3.7 Runtime Errors ................................................. 46
  3.7.1 Over-Constrained Models ......................... 46
  3.7.2 Under-Constrained Models ................. 46

II Theory .................................................. 49

4 Syntax and Static Semantics .................. 51
  4.1 MiniAcumen .................................................. 51
  4.2 Translating MiniAcumen to MicroAcumen ......... 53
    4.2.1 Translating MiniAcumen to FlatGuardAcumen 56
    4.2.2 Translating FlatGuardAcumen to MicroAcumen 57
    4.2.3 Complexity of Translation from MiniAcumen to FlatGuardAcumen 58
  4.3 A Type System for Mini- (and Micro-) Acumen 64

5 Mathematical Semantics ............... 69
  5.1 Introduction .............................................. 69
  5.2 Syntax of MicroAcumen .................................. 70
  5.3 Denotational Semantics of Types and Expressions . 70
    5.3.1 Topological Spaces ................................ 70
    5.3.2 Categorical Notions ................................ 72
    5.3.3 Interpretation of Types ............................ 73
    5.3.4 Interpretation of Terms ............................ 73
  5.4 Hybrid Systems and their Denotational Semantics . 74
    5.4.1 Hybrid Systems ...................................... 75
    5.4.2 Models as Hybrid Systems ..................... 76
    5.4.3 Valid Transitions .................................. 76
    5.4.4 Evolutions ......................................... 78
  5.5 Value Enclosures ........................................ 78
  5.6 Timed Enclosures ........................................ 79
  5.7 Soundness Requirements for Validated Numerical Prim- itives ........................................... 80
    5.7.1 Valid Transitions for a Model ................ 81
    5.7.2 Soundness Requirements ......................... 83
  5.8 Operational Semantics .............................. 85
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8.1 Processing of Modes</td>
<td>86</td>
</tr>
<tr>
<td>5.8.2 Processing of Models</td>
<td>90</td>
</tr>
<tr>
<td>5.8.3 Example Executions</td>
<td>92</td>
</tr>
<tr>
<td>5.9 On the Soundness of Operational Semantics</td>
<td>95</td>
</tr>
<tr>
<td>III Applications</td>
<td>97</td>
</tr>
<tr>
<td>6 Implementation</td>
<td>99</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>99</td>
</tr>
<tr>
<td>6.2 Traditional and Rigorous Simulation Semantics</td>
<td>100</td>
</tr>
<tr>
<td>6.3 Command-Line Interface (CLI)</td>
<td>101</td>
</tr>
<tr>
<td>6.4 Graphical User Interface (GUI)</td>
<td>102</td>
</tr>
<tr>
<td>6.5 Organization of the Code Base</td>
<td>103</td>
</tr>
<tr>
<td>6.6 Interpreters</td>
<td>104</td>
</tr>
<tr>
<td>6.6.1 Traditional Interpreters</td>
<td>106</td>
</tr>
<tr>
<td>6.6.2 Enclosure Interpreter</td>
<td>108</td>
</tr>
<tr>
<td>6.6.3 Active Statements</td>
<td>109</td>
</tr>
<tr>
<td>6.7 Arithmetics</td>
<td>110</td>
</tr>
<tr>
<td>6.7.1 Double</td>
<td>110</td>
</tr>
<tr>
<td>6.7.2 Interval</td>
<td>110</td>
</tr>
<tr>
<td>6.7.3 TDif</td>
<td>110</td>
</tr>
<tr>
<td>6.7.4 FDif</td>
<td>112</td>
</tr>
<tr>
<td>6.8 Solvers</td>
<td>112</td>
</tr>
<tr>
<td>6.8.1 Runge-Kutta</td>
<td>112</td>
</tr>
<tr>
<td>6.8.2 Taylor</td>
<td>113</td>
</tr>
<tr>
<td>6.8.3 Picard</td>
<td>113</td>
</tr>
<tr>
<td>6.8.4 Lohner</td>
<td>113</td>
</tr>
<tr>
<td>6.8.5 Contractor</td>
<td>114</td>
</tr>
<tr>
<td>6.8.6 Plotter and Table</td>
<td>114</td>
</tr>
<tr>
<td>6.8.7 3D View</td>
<td>115</td>
</tr>
<tr>
<td>6.9 Testing</td>
<td>115</td>
</tr>
<tr>
<td>7 An Automotive Collision Case Study</td>
<td>117</td>
</tr>
<tr>
<td>7.1 Hazard Analysis And Risk Assessment and The ISO 26262 Standard</td>
<td>118</td>
</tr>
</tbody>
</table>
Table 1 gives a brief explanation for several notations introduced in the thesis.

<table>
<thead>
<tr>
<th>Notion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>Defined</td>
</tr>
<tr>
<td>↑</td>
<td>Undefined</td>
</tr>
<tr>
<td>→</td>
<td>Total function type</td>
</tr>
<tr>
<td>↦</td>
<td>Partial function type</td>
</tr>
<tr>
<td>( \langle f_i \rangle_{i \in n} )</td>
<td>Sequence indexed by 0 to ( n - 1 )</td>
</tr>
<tr>
<td>⊕</td>
<td>Sequence concatenation</td>
</tr>
<tr>
<td>( \pi_k )</td>
<td>The ( k )th projection in a sequence, that is, ( \pi_k \langle x_i \rangle_{i \in n} = x_k )</td>
</tr>
<tr>
<td>( P_{\text{closed}}(X) )</td>
<td>The set of topologically closed subsets of ( X )</td>
</tr>
<tr>
<td>( \text{closure}(X) )</td>
<td>The smallest topologically closed set that contains ( X )</td>
</tr>
<tr>
<td>([a..b])</td>
<td>A closed real interval ( { x \in \mathbb{R} \mid a \leq x \leq b } )</td>
</tr>
<tr>
<td>((a..b))</td>
<td>An open real interval ( { x \in \mathbb{R} \mid a &lt; x &lt; b } )</td>
</tr>
<tr>
<td>([a..b])</td>
<td>A left-open real interval ( { x \in \mathbb{R} \mid a &lt; x \leq b } )</td>
</tr>
<tr>
<td>((a..b])</td>
<td>A right-open real interval ( { x \in \mathbb{R} \mid a \leq x &lt; b } )</td>
</tr>
<tr>
<td>( T, \bar{T} )</td>
<td>The lower bound ( T ) and upper bound ( \bar{T} ) of interval ( T )</td>
</tr>
<tr>
<td>∪</td>
<td>The supremum of a poset</td>
</tr>
<tr>
<td>∩</td>
<td>The infimum of a poset</td>
</tr>
</tbody>
</table>

Table 1: Overview of Notation
Table 2 explains symbols that are used throughout the thesis. A capital roman letter denotes a set of the corresponding notion.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ∈ A</td>
<td>Atomic constraint (MiniAcumen)</td>
<td>4.1.2</td>
</tr>
<tr>
<td>b ∈ B</td>
<td>Boolean value (MiniAcumen)</td>
<td>4.1.2</td>
</tr>
<tr>
<td>c ∈ C</td>
<td>Dynamics (MicroAcumen)</td>
<td>4.2.1</td>
</tr>
<tr>
<td>d ∈ D</td>
<td>Value enclosure</td>
<td>5.5.1</td>
</tr>
<tr>
<td>e ∈ E</td>
<td>Expression</td>
<td>4.1.2</td>
</tr>
<tr>
<td>f ∈ F</td>
<td>Function name</td>
<td>4.1.2</td>
</tr>
<tr>
<td>g ∈ G</td>
<td>Guarded constraint (MiniAcumen)</td>
<td>4.1.2</td>
</tr>
<tr>
<td>h ∈ R</td>
<td>Time step</td>
<td>5.6.1</td>
</tr>
<tr>
<td>i ∈ N</td>
<td>Natural number</td>
<td>4.1.2</td>
</tr>
<tr>
<td>j ∈ N</td>
<td>Natural number</td>
<td>4.1.2</td>
</tr>
<tr>
<td>k ∈ N</td>
<td>Natural number</td>
<td>4.2.9</td>
</tr>
<tr>
<td>m ∈ M</td>
<td>Model (MiniAcumen)</td>
<td>4.1.2</td>
</tr>
<tr>
<td>m ∈ Mµ</td>
<td>Model (MicroAcumen)</td>
<td>4.2.1</td>
</tr>
<tr>
<td>n ∈ N</td>
<td>Size of a set, sequence or model</td>
<td>4.1.2</td>
</tr>
<tr>
<td>N ∈ N</td>
<td>Dimension (of a model)</td>
<td>4.1.2</td>
</tr>
<tr>
<td>p ∈ P</td>
<td>Guarded constraints (MiniAcumen)</td>
<td>4.1.2</td>
</tr>
<tr>
<td>q ∈ Q</td>
<td>Mode (MicroAcumen)</td>
<td>4.2.1</td>
</tr>
<tr>
<td>r ∈ R</td>
<td>Real number</td>
<td>4.1.2</td>
</tr>
<tr>
<td>s ∈ S</td>
<td>State</td>
<td>5.3.4</td>
</tr>
<tr>
<td>t ∈ T</td>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>t ∈ T</td>
<td>Time enclosure</td>
<td>5.6.2</td>
</tr>
<tr>
<td>u ∈ U</td>
<td>Flat guard constraints (FlatGuardAcumen)</td>
<td>4.2.1</td>
</tr>
<tr>
<td>v ∈ V</td>
<td>Value</td>
<td>4.1.2</td>
</tr>
<tr>
<td>x ∈ X</td>
<td>Variable name</td>
<td>4.1.2</td>
</tr>
<tr>
<td>y ∈ X</td>
<td>Variable name</td>
<td>4.1.2</td>
</tr>
<tr>
<td>z ∈ Z</td>
<td>Timed enclosure (N.B. Not an integer)</td>
<td>5.6.3</td>
</tr>
<tr>
<td>Ω ∈ Γ</td>
<td>Value type environment</td>
<td>4.3.1</td>
</tr>
<tr>
<td>Ψ</td>
<td>Power Set of State Space</td>
<td>B.0.3</td>
</tr>
<tr>
<td>Ψclosed</td>
<td>Closed-Power Set of State Space</td>
<td>B.0.4</td>
</tr>
<tr>
<td>Σ ∈ Σ</td>
<td>Function type environment</td>
<td>4.3.1</td>
</tr>
<tr>
<td>τ ∈ Type</td>
<td>Type</td>
<td>4.3.1</td>
</tr>
</tbody>
</table>

Table 2: Symbols and Their Uses
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>A Trajectory of the Sawtooth Wave Model</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>A Type System for MiniAcumen</td>
<td>66</td>
</tr>
<tr>
<td>5.1</td>
<td>Eliminating a Blocked Mode (Blocked)</td>
<td>87</td>
</tr>
<tr>
<td>5.2</td>
<td>Stepping Discretely (Discrete)</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Stepping Continuously From the Start Time $t = 0$ (Continuous)</td>
<td>89</td>
</tr>
<tr>
<td>5.4</td>
<td>Stepping Continuously From Unknown Time $t = (0..h)$ (Continuous)</td>
<td>89</td>
</tr>
<tr>
<td>5.5</td>
<td>An Operational Semantics for MicroAcumen</td>
<td>92</td>
</tr>
<tr>
<td>5.6</td>
<td>Execution of Sawtooth Wave Model With Step $\frac{1}{2}$</td>
<td>93</td>
</tr>
<tr>
<td>5.7</td>
<td>Execution of Sawtooth Wave Model With Step 2</td>
<td>94</td>
</tr>
<tr>
<td>6.1</td>
<td>Acumen’s Graphical User Interface</td>
<td>102</td>
</tr>
<tr>
<td>6.2</td>
<td>Lines of Code in Repository</td>
<td>103</td>
</tr>
<tr>
<td>6.3</td>
<td>Implementation Components and Relationships</td>
<td>105</td>
</tr>
<tr>
<td>6.4</td>
<td>The Simulation Loop of Acumen’s Traditional Interpreters</td>
<td>106</td>
</tr>
<tr>
<td>6.5</td>
<td>An Exact Solution and a Traditional Approximation</td>
<td>107</td>
</tr>
<tr>
<td>6.6</td>
<td>A Mode Change Missed by a Traditional Approximation</td>
<td>107</td>
</tr>
<tr>
<td>6.7</td>
<td>A Mode Change Captured by an Enclosure</td>
<td>109</td>
</tr>
<tr>
<td>7.1</td>
<td>An Overview of the Test Scenario</td>
<td>120</td>
</tr>
<tr>
<td>7.2</td>
<td>A Hierarchical Hybrid Automaton Representation of the Intersection Collision Avoidance System (ICAS)</td>
<td>121</td>
</tr>
</tbody>
</table>
7.3  Enclosures for Vehicle Positions and Change in Velocity Due to Collision. .......................... 129
7.4  Enclosures Computed With Two Different Step Sizes. .............................. 132
# List of Tables

1. Overview of Notation ........................................... 12
2. Symbols and Their Uses ........................................ 13
   2.1 Common Classes of Equations ............................. 30
   2.2 Hybrid Systems Analysis Tools ........................... 34
6.1 Simulator Parameters Available in Traditional Interpreters .................. 108
6.2 Simulator Parameters Available in Enclosure Interpreter ................... 109
6.3 Operations of the Real Type .................................... 111
7.1 Summary of Parameters for Collision Scenarios ........................... 126
7.2 A Summary of Results for Simulation Scenarios ........................... 128
8.1 Enclosure Convergence .......................................... 154
Part I

Background
Chapter 1

Introduction

“He who seeks for methods without having a definite problem in mind seeks in the most part in vain.”

— David Hilbert

This thesis explores a simulation approach to model-based design based on numerical techniques that are traditionally used in verification. This chapter motivates the work, states its thesis and describes its contributions and origins.

1.1 Motivation

The miniaturization of computers has made it possible to embed one into practically every kind of product. These combinations, called Cyber-Physical Systems (CPS), often have completely new capabilities, perhaps best illustrated by the autonomy features of modern cars, or by industrial and domestic robots. As useful as these systems are, designing them is an increasingly complex process. An important reason for this is that the computational components of a CPS have very large numbers of states. This makes it difficult to understand

\footnote{The present version of the dissertation includes minor corrections suggested by the examiners and opponent.}
and test the behavior of the overall system. For example, it may be too expensive or even unfeasible to perform physical tests of the system in each of its possible states.

Model-based design is one way to manage this complexity. By performing much of the design work on the level of a model, it becomes easier to stage the design process to discover issues early on, when the cost of making mistakes is low. A model is a formal description of the system. If we imagine that there is an ideal or exact model, most models are approximations that help the user reason about certain aspects of the system, while ignoring others. For example, a model can be a diagram that captures the structure of decisions made by a system or an equation that represents the positions of the physical parts of the system.

Both diagrams and equations are examples of mathematical models. Such models have the potential to be processed by computers, making simulation (observing the behavior of a model under different conditions) easy and cheap, compared to building physical prototypes. However, even though computers have experienced a rapid gain in performance over the past decades, many kinds of questions about models are fundamentally intractable. For example, exact solutions to most equations cannot be computed, as doing so typically involves infinite processes. The alternative is computing approximations, which typically involves truncating infinite series and using finite representations of real numbers. A highly successful example is the widespread use of a certain class of rational numbers called “floating-point” numbers, in place of reals. Using such numbers, many numerical methods originally developed for use with reals, can be used as is, with useful results. Simulation tools \cite{MAT2, Fri} based on such algorithms have evolved to support different kinds of equational and graphical models that suit the needs of different domains, and are a standard part of an engineer’s toolkit. However, simulation tools built on conventional numerical methods suffer from serious and fundamental issues. These tools produce results that do not reflect the approximation error incurred by the underlying numerical operations and representations. As a result, traditional simulation tools do not consistently track this
error, and may altogether miss critical features of the model’s behavior. Also, the many features supported by these tools makes the task of formally specifying their modeling languages much more difficult.

Verification tools build on algorithms that are either error-free or produce results with known error bounds, and are guaranteed to take into account all behaviors specified by the model. They can analyze models with uncertain parameters, making it possible to verify sets of scenarios simultaneously. Generally, they are based on languages that were designed with formal semantics in mind, and are simpler than those of simulation tools. However, there are numerous challenges en route to making verification tools as widely used as simulation tools. These include the ability to deal with sufficiently detailed models (scalability), to conveniently write down the model in the given language (expressivity) and to produce informative results (accuracy). Another challenge is the ease of use of these tools. Verification tools require the user to specify properties that the tool attempts to prove. It can be particularly difficult to do so when modeling is typically done in the early stages of design, when the behavior of the system is not yet well understood.

1.2 Thesis

The thesis of this dissertation is that a domain-specific language for simulation using validated numerics can combine the ease of use of simulation with the rigor of verification.

1.3 Contributions

The work focused on the implementation of a rigorous numerical simulator for the Acumen modeling language, and makes the following contributions:

1. A formal specification of a flexible hybrid systems modeling language (Chapter 4), which includes an operational semantics
(Chapter 5) that preserves error information provided by the underlying validated numerical methods.

2. A prototype implementation of rigorous simulation based on this semantics, that simulates models using validated numerics. The simulator is integrated into the freely available Acumen testbed (Chapter 6).

3. A case study of computing bounds on the output of a model from the automotive domain (Chapter 7).

4. A collection of benchmarks for evaluating the accuracy of rigorous simulators on models of stable systems (Chapter 8).

1.4 History and Related Publications

This dissertation summarizes both published and unpublished work. This section describes the origins of the chapters that are based on published work, and explains my contributions to the chapters that have not been published.

My main contributions to the related publications are as follows:


I developed the model (initially by Walid Taha and Henrik Eriksson) into the form described in this dissertation (Chapter 7) with help from Henrik Eriksson, Ferenc A. Bartha, Ayman Aljarbouh and Yingfu Zeng. The necessary changes to both the rigorous and traditional simulators that resulted from this case study are my work.

I implemented the rigorous simulator needed to accurately simulate the discrete and hybrid models included in the paper. The differential equation solver needed to accurately simulate the continuous models described in the paper is joint work with Ferenc A. Bartha.


I introduced my co-authors to related work on reachability analysis, which resulted in adopting the passed-waiting-list approach to computing reachability. Through my work on the numerics library that underlies the prototype implementation in Acumen, the team was able to reach confidence in the correct and general realization of the approach.


I developed the traditional simulator that underlies the results presented in this paper together with Ferenc A. Bartha.

I identified the mapping of concepts necessary to implement the approach to simulation described in the paper.


I was involved in most aspects of the design of Acumen throughout the course of my PhD studies and led its implementation between 2014 and 2016.


I supported several instances of the CPS course and was responsible for the integration of feedback from students into Acumen tool.

My contributions to the unpublished parts of this dissertation are as follows. I designed and developed the rigorous simulator, as well as the reference implementation of the traditional simulator (Chapter 6). The rigorous simulator builds on numerical libraries that I designed.
1.4. HISTORY AND RELATED PUBLICATIONS

and developed together with Jan Duracz and Ferenc A. Bartha. I defined the core languages described in this dissertation, the translations between them, as well as the type system (Chapter 4). I defined the operational semantics together with Walid Taha and Ferenc A. Bartha (Chapter 5). All proofs included in this dissertation are my work (Chapter 4 and Appendix A), though advice was provided by Walid Taha, Eugenio Moggi, Ferenc A. Bartha, Jan Duracz and Amin Farjudian.

This thesis was conducted in the context of the Acumen project (Paper 6). The development of a rigorous simulator for the Acumen language grew out of work on simulating Zeno hybrid automata using validated numerical methods (Paper 3). The theory presented in this thesis is based on that work. However, the present formalization (Chapters 4 and 5, and Appendix A) has not been published. The proofs (Chapter 4 and Appendix A) have been circulated, but have not been formally peer-reviewed for the purpose of publication. The automotive case study (Chapter 7) was the result of participation in a project that, among other activities, evaluated the use of rigorous simulation for risk assessment (Paper 1). The work on benchmarking accuracy of enclosures (Chapter 8) was inspired by a principle suggested by Walid Taha, which motivated the development of a rigorous differential equation solver capable of simulating non-linear models accurately (Paper 2).

This dissertation was also influenced by other work conducted in the context of the Acumen project. This includes work on the modeling of electromechanical systems (Paper 4), that served as examples of models that the rigorous simulator should be able to process. A general theme throughout has been the treatment of systems that cannot be simulated with traditional methods (Papers 3 and 5).
Chapter 2

State of the Art in Modeling, Simulation, and Verification of Hybrid Systems

Many tools are available to designers of CPS as academic or commercial software, ranging from simulation tools to verification tools. This chapter reviews modeling formalisms and analyses commonly supported by such tools.

2.1 Modeling Formalisms

A hybrid system is a system that exhibits both discrete and continuous behavior. Models of such systems describe the behavior of variables, and the continuous variables of a hybrid system model are typically functions of time. Such a function can be expressed using a differential equation. A fundamental choice in the design of a hybrid systems modeling tool is the type of differential equations that it should support, as this class dictates what kind of analysis that can be supported. Table 2.1 lists common classes of differential equations.
supported by modeling tools, including notions that extend the concept of an ordinary differential equation to make it possible to model more systems.

Linear Differential: The right-hand side (field) of such an equation is a matrix multiplied by a state variable (vector), plus a constant (vector). Linear differential equations are important because they can be used to faithfully model many physical systems, including population growth and certain types of electrical circuits. They are also important because they have closed form solutions.

Non-Linear Differential: Differential equations whose fields can be arbitrary expressions in terms of the state variables (such as \( x^2 \)) are called ‘non-linear’. This is a much more general class of equations, and most of them do not have closed-form solutions. Still, such equations arise naturally in many domains, including physics, biology, and economics. Non-linear differential equations are important for modeling Cyber-Physical Systems. For example, they are used to model multidimensional mechanics, where common operations such as coordinate transformations and distance measurement rely on non-linear functions.

Differential-Algebraic: Some systems may not be modeled by explicit differential equations, that is, differential equations where the left-hand side is a single derivative term.
2.1. MODELING FORMALISMS

Differential Algebraic Equations (DAE) are a more general class of equations, where variables and derivatives may be combined arbitrarily. A notable sub-class of such equations are semi-explicit DAEs, whose solutions are simultaneously constrained by a differential and an algebraic equation:

\[
\begin{align*}
\frac{dx(t)}{dt} &= F(x, y) \\
0 &= G(x, y)
\end{align*}
\]

DAEs provide a more modular approach to modeling. However, they can be difficult to solve, and numerical solutions to such equations are an active area of research [NPT].

Delay Differential: Yet another class of systems that cannot be modeled by ordinary differential equations are those whose behavior at a given point in time depends not only on the current state, but also on past states. Examples of systems that can be modeled using a Delay Differential Equation (DDE) are mechanical systems that exhibit vibrations and economic exchanges with information lag.

For many systems, both the discrete and continuous state of the system is of interest. In these situations, a model that combines both a continuous dynamical model (for example, a control system’s physical environment) and a discrete model (for the control system itself) is needed. Many different formalisms for hybrid systems have been proposed, both in the scientific literature and by companies, as part of modeling environments.

Hybrid Automata: The seminal formalism for modeling hybrid systems is hybrid automata [Hen], in which the behavior of continuous variables over time is specified by constraints that include differential equations. In a hybrid automaton, the nodes of the graph (also called discrete states, modes or locations) correspond to the possible continuous dynamics, and the criteria for transitioning between them, called guards, determine when the system is allowed to make a transition. Each node is also associated with an invariant. When the invariant is falsified, the system will either take a transition whose guard is
satisfied, or it will block. Variants of hybrid automata have been explored in the literature. Hybrid I/O automata \cite{LSV} add support for parallel composition and hiding; hierarchical hybrid automata \cite{MS} make it possible to nest automata, further facilitating the modeling of larger systems; and stochastic hybrid automata \cite{HHHK} allow assigning probability distributions to model parameters.

Much as with the extensions of automata described above, hybrid systems formalisms have also been developed as extensions of logic.

**Hybrid Programs:** Hybrid programs are textual encodings of Differential Dynamic Logic (dL) formulas \cite{Pla1}, that is, formulas of a first-order dynamic logic extended with differential and algebraic equations/inequalities, and discrete jumps, among other things. Existential quantifiers in a variant \cite{Pla2} of dL can be used to model the dynamic creation of state variables, which is useful for modeling systems where the number of variables is unknown. For example, in a model of a distributed highway collision avoidance system \cite{LPN} that controls the vehicles on a certain part of a highway, the arrival of vehicles into the system’s field of view corresponds to the creation of state variables.

**HydLa:** HydLa \cite{UMT} is a language based on temporal logic extended with constructs for derivatives, referring to the value of a variable just before a discontinuity, and for expressing priorities between constraints. The latter can be used to encode hierarchies of constraints. Similar to how method overriding works in class-based object oriented languages, this feature can be used to avoid code duplication.

There are also formalisms that combine graphical and textual models.

**Simulink:** The most widely adopted industrial formalism for modeling of hybrid systems are the *block diagrams* and *stateflow charts* of Simulink /Stateflow (henceforth just Simulink), both toolboxes for
the Matlab \[\text{MAT2}\] scientific computing environment. First introduced over 20 years ago, Simulink has a large user base and there are many libraries available for different application domains. Block diagrams are a graphical language in which a system is described using graphs, whose nodes (blocks) transform \textit{signals} that can be passed to other blocks through edges. Graph notation is convenient in domains such as electrical circuits, where the layout of the block diagram is close to that of the modeled system. However, systems with many connected blocks can result in models that are difficult to read, though this issue is compensated by the ability to nest diagrams. This language was not developed with a formally defined semantics \[\text{CPPSV}\], and despite extensive work towards such a definition \[\text{SRKC; Tiw; ASK; MF}\], this work has been performed by independent researchers, and no standard formal semantics has been adopted.

\textit{Modelica:} Another major hybrid systems formalism that has seen wide adoption is Modelica \[\text{Fri}\]. This is a textual language that combines mathematical, programmatic and graphical notations. Its key features include \textit{acausal} \textit{DAEs} and inheritance-based object orientation. In acausal equations, the left-hand side is not required to be a single variable. Thus, models can consist of equations such as \[m \cdot a = F\], and the tool transforms the equation into the form (such as \[a = F/m\]) required to solve the overall system of equations. This feature helps to keep models general, as the same equation can be used in multiple contexts. Such code reuse, along with that made possible by inheritance, helps to make it feasible to model systems in great detail in Modelica \[\text{SBN}^{+}\].

2.2 Analysis and Tools

Table 2.2 lists common, actively developed hybrid systems tools. The tools can be classified along several dimensions, including the features supported by their modeling formalism, as well as the types of analysis that the tools are capable of. We may classify these analyses as
Tool Models Equations Semantics

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>P</th>
<th>L</th>
<th>N</th>
<th>A</th>
<th>D</th>
<th>Num</th>
<th>Exa</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acumen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulink</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenModelica</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zélus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CIF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STaliro</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpaceEx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ariadne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flow*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HySIA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dReach</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hyrose</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KeyMaera</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


examples of simulation or verification. In some cases, the same tool offers more than one kind of analysis for the same model, though certain model features (such as intervals or probabilistic parameters) may only be supported by a specific analysis.

Simulation

Simulation is used to perform virtual experiments, where a model is used to observe the behavior of the modeled system under different circumstances. Given an initial state, the model is executed to obtain
simulation trace (trajectory). For a hybrid system model this trace consists of piecewise solutions to differential equations, interrupted by discontinuities (events). A simulator for a hybrid systems modeling formalism must therefore perform two tasks. First, it must solve initial-value problems to simulate the model’s continuous behavior. Second, it must monitor these solutions to identify when an event should occur (event detection), and what the behavior should be at the discontinuity (event handling).

Equation solving generally begins with model transformation, where the raw model is translated into an executable form, typically consisting of explicit Ordinary Differential Equation (ODE) systems. These equations are then solved using an integrator (solver).

For a limited class of differential equations, analytic solutions can be obtained computationally. Assuming that the model contains only equations whose analytic solutions can be computed, simulators based on symbolic integrators \cite{MU, NRS} can produce error-free trajectories. Because expressions for the solutions are available, these simulators can be used to analyze systems with uncertain parameters, as considering another initial state simply amounts to evaluating these expressions.

In general, though, solutions to differential equations must be approximated. Traditional numerical integrators are rules that, given the state at one or more points in time, produce an estimate at the next point in time. The progress in time made by the integrator is called the time step. The integrator approximates the solution using a computationally convenient function, typically a polynomial (though there are alternatives \cite{HT}). Modern numerical integration libraries \cite{HBG+} are able to solve difficult equations such as some examples of stiff ODEs, whose solutions are particularly sensitive to the time step, as well as some types of DAEs. An important feature of these integrators is that they will adapt their behavior to limit the estimated approximation error, by decreasing the step or otherwise increasing their precision when the solution moves rapidly.

Validated numerical integrators \cite{NJC, MB1, CAP, BM, Che, KDFT} go further and produce conservative bounds on the error as part of
the solution. To guarantee that the bounds are conservative, these integrators rely on validated numerics [Tuc], numerical methods recast atop of interval arithmetic [War; Sun; MB2]. By propagating the error through the simulation, validated integrators can be used to build rigorous (interval) simulators. A side effect of being based on validated numerics is that validated numerical solvers operate on sets (Section 5.6) rather than on points. In principle this means that, like symbolic simulators, they can analyze models with uncertain parameters (Chapter 8). Recent advances in rigorous simulation techniques for hybrid systems include: set representations that balance computational complexity against precision [FLGD+; CÁS], methods for reducing approximation error during integration [Che] and event handling [GI], and parallel simulation algorithms [RG].

Verification

Verification is typically used to guarantee the correct behavior of safety-critical systems. To achieve this, verification tools take both the model and a property (that is, a formally stated hypothesis about the model) as input, and attempt to prove that the property holds. The two main approaches to verification of hybrid systems are model checking and theorem proving.

Model Checking and Reachability

Model checking of hybrid systems proceeds by building up an approximation of the reachable states of the system, a process called reachability. In its simplest form, the model checker observes the reachable state approximations and reports an error if any state violates the property. If the model checker manages to exhaust all

---

1The meaning of the term “verification” varies across different communities. In engineering domains, it is frequently used as a synonym for validation. In the programming languages and modeling communities, it typically means something stronger – that the tool has generated a result with formal guarantees about its correctness. Exceptions to this usage exist, but for the purposes of this thesis we will take the term to mean formal verification, and call verification approaches that do not come with correctness guarantees testing.
the reachable states without violating the property, it considers the property proven. For hybrid systems, both time and the domains of variables are typically modeled by real numbers, so the state space is generally infinite. Exhausting such a state space must therefore involve observing that a fixed point is reached in the reachability computation.

Due to the accumulation of approximation error the reachability computation may not reach a fixed point, even for stable systems (Chapter 8). To remain general, bounded model checking relaxes the requirement of exhausting the state space by restricting the analysis to a finite subset of the state/solution spaces. For example, the analysis may only consider a bounded time interval [Gao], or observe a finite number of discontinuities [CAS]. The reachability computation of time-bounded model checking corresponds to rigorous simulation, though model checkers may choose to traverse the state space differently. A model checker may analyze its output to guide the reachability computation in an attempt to establish the property. For example, in CounterExample-Guided Abstraction and Refinement (CEGAR), one alternates proof and falsification using increasingly more precise approximations of the sought property [RS]. Model checking is generally concerned with large uncertainties, such as proving that the property holds for a broad range of situations.

**Theorem Proving**

An alternative approach to proving properties about hybrid systems is to view the model as a logic formula, and attempt to rewrite this formula into one that implies the property. For example, a formula that contains a differential equation constraint could be rewritten by replacing the differential equation with its solution, obtained from a computer algebra system such as Mathematica [Mat1]. Theorem provers are typically *interactive*, as the available rules (*tactics*) by which intermediate formulas can be re-written automatically may not suffice to complete the derivation of the sought property. Because of this, using a theorem prover can be daunting as, instead of plots or other visualizations, any refinements to the model must be based on
intermediate logical formulas.
Chapter 3

Traditional Simulation in Acumen

Acumen [TDZ⁺] is a Domain-Specific Language (DSL) for modeling and simulation of hybrid systems. Such systems exhibit both continuous and discrete behavior, and the conditions for switching between behaviors that depend on the state of the system. Acumen is a small but expressive language, with statements that can be composed and nested to facilitate modeling of large systems. Through Acumen’s notion of objects it is possible to model systems with a dynamically changing number of variables.

The target of this thesis has been to develop a rigorous simulator for the full Acumen language. This chapter reviews past work on Acumen and describes the language to provide a sense of what it would be like once the target is reached.

3.1 Past Work on Acumen

Acumen was originally motivated by the need for a coherent toolchain for developing CPS [TDZ⁺]. The first instance of the language built on the idea of Functional Reactive Programming (FRP) where the behavior of a system is specified by composing time-indexed functions [EH, WH]. The basic FRP system, used to specify discrete
CHAPTER 3. TRADITIONAL SIMULATION IN ACUMEN

changes, was extended with constructs for specifying the continuous dynamics of a system [ZWI+].

The second instance of the language, which is described in this thesis (Chapter 6), was a ground-up redesign that, among other things, introduced the object system [BT], modeling environment [ZRT+] and rigorous simulator [DEB+; KTB+]. Throughout its development, Acumen has been used in case studies [DEB+; DBT; ZRT+] as well as for teaching courses on CPS [TCPZ2; TCPZ1; TCPZ3; THX+].

3.2 Equations

In the physics and engineering domains, continuous evolution over time is commonly modeled using differential equations. A simple example of a continuous system is a clock. We can model this system as a single variable whose derivative with respect to time is one. In Acumen such a differential equation is expressed as follows:

\[ x' = 1 \]

To model changes to the state of a system that happen instantaneously we use a discrete equation as follows:

\[ x^+ = x + 1 \]

This model says that the next value of \( x \) is equal to the current value of \( x \) plus 1.

3.3 Sequences of Equations

Systems with dimension greater than one can be modeled using systems of equations expressed using the composition operator \( \cdot \). For example, we can model a sinusoid as a system of two coupled equations:

\[ p' = v, \quad v' = -p \]

Equivalently, this system can be expressed as a higher-order differential equation \( p'' = -p \) or vector equation \( (p', v') = (v, -p) \).
3.4 If Statements

We can combine discrete and continuous equations in the same model to form a hybrid system. For example, a sawtooth wave can be modeled as a one-dimensional system whose value grows linearly over time and is periodically reset back to its initial value. Figure 3.1 illustrates the behavior of this system up to time 2. The sawtooth wave system can be modeled in Acumen as follows:

\[
\text{if } x < 1 \text{ then } x' = 1 \text{ else } x^+ = 0
\]

This model says that when the value of \( x \) is less than 1, then \( x \) grows linearly with time. Otherwise \( x \) is instantaneously reset to 0. Hybrid systems, where multiple different behaviors can occur under different circumstances, can be expressed by nesting if statements. For example, a rocket with two booster stages can be modeled as follows:

\[
\begin{align*}
\text{if } x < 1 \text{ then } x'' &= 5 \text{ else } // \text{ first stage} \\
\text{if } x < 5 \text{ then } x'' &= 1 \text{ else } // \text{ second stage} \\
&\quad \quad \quad x'' = 0 \quad // \text{ boosters off}
\end{align*}
\]

Here, the value of \( x \) accelerates quickly until it reaches 1, then accelerates slowly until it reaches 5, when it stops accelerating and continues to grow at its current, constant speed.
Discrete equations can be used to encode hybrid automata, that is, hybrid systems whose behavior is determined by the value of a variable with a discrete set of values. A simple example of a hybrid automaton is a controller for a heating system:

```plaintext
if heating == 1
  then if x < 23 then x' = 10 else heating+ = 0
  else if x > 18 then x' = -x else heating+ = 1
```

Here the behavior of the system is determined by the value of the `heating` variable.

### 3.5 Match Statements

Hybrid automata can also be compactly expressed using the `match` statement:

```plaintext
match heating with
  [ 1 -> if x < 23 then x' = 10 else heating+ = 0
  | 0 -> if x > 18 then x' = -x else heating+ = 1 ]
```

The different values that `heating` obtains during a simulation correspond to cases of the `match`, separated by “|”. We can further improve the readability of this model by making `heating` a string-valued variable:

```plaintext
match heating with
  [ "on" -> if x < 23 then x' = 10 else heating+ = "off"
  | "off" -> if x > 18 then x' = -x else heating+ = "on" ]
```

### 3.6 Model Definitions and Instantiation

The examples we have seen so far are fragments of Acumen models. To be a model, such fragments must reside inside a `model`:

```plaintext
model M() = initially x = 0, x' = 1 always x' = 1
```
A model definition consists of a name ($M$ above), zero or more parameters (zero in the model above) and two sections. The **initially** section is where the initial state of the model is declared: each variable that occurs in the model (that is not already declared as a parameter) must be specified here along with its initial value. The **always** section is where the behavior of the model is specified. The fragments that we have seen so far are examples of such behavior.

To be executable, an Acumen model must contain a sub-model called **Main** with a single parameter called **simulator**. The **simulator** parameter is used to configure the interpreter. For example, the end-time of the simulation can be changed as follows:

```plaintext
model Main(simulator) =
  initially x = 0, x' = 1
  always x' = 1, simulator.endTime+ = 5
```

Simulator parameters are modified using discrete equations. Aside from specifying the length of the simulation as in the model above, they can also be used to adjust the precision of the simulator, for example by using a smaller time step. Tables 6.1 (on page 108) and 6.2 (on page 109) list the simulator parameters that are available for each type of interpreter.

Model definitions are also useful to modularize large models and to re-use code. For example, a model with two sub-models whose continuous behavior is described by the same differential equation can be defined as follows:

```plaintext
model Trig(x,x',x'') = always x'' = -x

model Main(simulator) =
  initially sine = create Trig(0,1,0),
    cosine = create Trig(1,0,-1)
```

Here, two instances (objects) of the **Trig** model are created by the **Main** model. The **Main** model is thus called the **parent** of the two **Trig** objects, which in turn are called the **children** of the **Main** object. In
the \texttt{create}, the parent gives initial values to the parameters of the child, thus possibly affecting its behavior.

The \texttt{create}s provide initial values for the state variables $x$ and $x'$, as well as for $x''$. The latter can be considered as redundant, given the equation $x'' = -x$. However, Acumen currently does not recognize this, and requires all variables of a model to be declared and given values at initialization.

Objects created in the \texttt{initially} section are created at the same time as their parent. Objects can also be created dynamically by using \texttt{create} in the \texttt{always} section, in one of two kinds of statements. First, \texttt{create} can occur by itself:

\begin{verbatim}
model Timer(p) =
    initially t = 0, t' = 1, flag = true
    always t' = 1,
    if t >= 1 && flag then
        create Timer(0), flag+ = false
    noelse

When the $t$ variable of a \texttt{Timer} object reaches 1, another \texttt{Timer} is created. The \texttt{flag} variable is used to ensure that each timer only creates a single child object. This approach can be used to create dynamic collections of objects whose size depends on the state of the simulation.

The other way to create an object in the \texttt{always} section is as part of a discrete assignment. This way, a variable can be updated to refer to another object during the course of a simulation:

\begin{verbatim}
model Timer(t,t') = always t' = 1

model Main(simulator) =
    initially c = create Timer(0,1), flag = true
    always if c.t >= 1 && flag then
        c+ = create Timer(0,2), flag+ = false
    noelse
\end{verbatim}
Acumen's object model allows the parent to access and modify the state of its child objects:

\[
\text{model } C(p) = \text{initially } t = 0, t' = 1 \text{ always } t' = 1 \\
\text{model } \text{Main}(\text{simulator}) = \\
\quad \text{initially } c = \text{create } C(0) \\
\quad \text{always if } c.t \geq 1 \text{ then } c.p+ = c.p + 1, c.t+ = 0 \text{ noelse}
\]

Here the \( C \) model has one parameter \( p \) and a variable \( t \) that tracks time. The \( \text{Main} \) object can access the values of the parameters and variables of the \( C \) object through the name \( c \) that it was given when it was instantiated. The \( \text{Main} \) model observes the value of \( c.t \) and, when that reaches 1, it increments the counter \( c.p \) and resets the value of \( c.t \) to 0.

When an object is created, it is added to the \textit{children} list of its parent. This provides another way to access the child objects, by iterating through the \textit{children} list as follows:

\[
\text{model } C(p, p') = \text{always } p' = p \\
\text{model } \text{Main}(\text{simulator}) = \\
\quad \text{initially } t = 0, t' = 1, \\
\quad \text{c1 = create } C(1,1), \text{ c2 = create } C(2,2) \\
\quad \text{always } t' = 1, \\
\quad \text{if } t \geq 1 \text{ then} \\
\quad \text{t+ = 0, } \\
\quad \quad \text{foreach } c \text{ in } \text{children} \text{ do} \\
\quad \quad \quad \quad ( \text{if } c.\text{className} == C \text{ then } c.p+ = -1/c.p \text{ noelse} ) \\
\text{noelse}
\]

Here, the \textit{foreach} statement is used to generate a set of Acumen model fragments, one for each child object \( c \) in the \textit{children} list. This approach is useful when operating on an unbounded number of child objects, or when the object is not accessible through a variable name.
CHAPTER 3. TRADITIONAL SIMULATION IN ACUMEN

3.7 Runtime Errors

Several types of errors are possible in an Acumen model. In addition to syntax errors (such as mistyped keywords), or type errors (such as adding a real number to a string), there are several kinds of semantic errors:

3.7.1 Over-Constrained Models

Whenever more than one equation with a given left-hand side is active at any step of the simulation, then the model is considered over-constrained:

```acumen
model Main(simulator) =
  initially x = 0, x' = 1
  always x' = 1, if x > 1 then x' = 2 noelse
```

In the model above, once \( x > 1 \) is satisfied, then both \( x' = 1 \) and \( x' = 2 \) are active and an error is produced.

3.7.2 Under-Constrained Models

When a discrete assignment is active, the variables that are not defined by any discrete assignment remain unchanged. Otherwise, each variable in the model must be defined by exactly one continuous equation, or the model is considered under-constrained. For example, the following model is under-constrained when \( x \geq 1 \):

```acumen
model Main(simulator) =
  initially x = 0, x' = 1
  always if x < 1 then x' = 1 noelse
```

When \( x \) is greater than or equal to 1, then no equation is active for \( x \) and an error is generated. Thus, whenever continuous behavior is specified in an Acumen model, it must be given in full. Discrete behavior is treated differently. If at least one discrete equation is active, then the remaining variables are left unchanged. For example, the following models are equivalent:
model Main(simulator) =
initially x = 0, x’ = 1, c = 0
always if x > 1 then x+ = 0, c+ = c else x’ = 1

model Main(simulator) =
initially x = 0, x’ = 1, c = 0
always if x > 1 then x+ = 0 else x’ = 1

When $x > 1$ in the latter model, the simulator implicitly applies the identity discrete equation $c+ = c$. 
Part II

Theory
Chapter 4

Syntax and Static Semantics

This chapter formalizes the syntax for three core subsets of the Acumen language. First, we review the syntax of MiniAcumen, a subset of Acumen that is supported by the rigorous simulator. The semantics of MiniAcumen is defined by translation\(^1\) into a subset called MicroAcumen, a small subset of MiniAcumen. The translation consists of two steps that together convert the model into a form more directly amenable to solving by using standard numerical methods. The first step translates MiniAcumen into a subset called FlatGuardAcumen. The bounds on the increase in the size of the term due to this translation is established. Secondly, this chapter defines a simple type system to identify basic issues in models before simulation.

4.1 MiniAcumen

This section describes the syntax of MiniAcumen and syntactic notions that will be used throughout the rest of the thesis.

\(^1\) Throughout the dissertation, a translation refers to a mapping between the terms of two languages.
Definition 4.1.1 (Syntactic Notions). The syntax of MiniAcumen is defined in terms of the following notions:

- \( r \in \mathbb{R} \) Real numbers
- \( b \in \mathbb{B} \) Booleans
- \( N \in \mathbb{N} \) Constant natural number
- \( x, y \in \mathbb{X} \) Variable names, \(|\mathbb{X}| = N\)
- \( \langle x_i \rangle_{i \in \mathbb{N}} \) Sequence of distinct variable names
- \( f \in \mathbb{F} \) Function names

For convenience, we identify a natural number \( n \) with the set of its predecessors. For example, \( \langle x_i \rangle_{i \in n} \) becomes a proper short form for \( \langle x_i \rangle_{i \in \{0...n-1\}} \). The number \( N \in \mathbb{N} \) is a constant that represents the dimension of the state space for the model being considered.

Definition 4.1.2 (Syntax of MiniAcumen). The Backus-Naur Form (BNF) for MiniAcumen is as follows:

\[
\begin{align*}
\text{Value} & : v \in \mathbb{V} ::= r \mid b \\
\text{Expression} & : e \in \mathbb{E} ::= v \mid x \mid f\langle e_i \rangle_{i \in \mathbb{N}} \\
\text{Atomic Constraint} & : a \in \mathbb{A} ::= x' = e \mid x^+ = e \mid \text{error} \\
\text{Guarded Constraint} & : g \in \mathbb{G} ::= a \mid \text{if } e \text{ then } p_1 \text{ else } p_2 \\
\text{Guarded Constraints} & : p \in \mathbb{P} ::= [\langle \text{claim } e \rangle] \langle g_i \rangle_{i \in \mathbb{N}} \\
\text{Model} & : m \in \mathbb{M} ::= \text{model } \text{Main}() = \\
& \quad \text{initially } \langle x_j = e_j \rangle_{j \in \mathbb{N}} \text{ always } p
\end{align*}
\]

The above BNF defines the sets indicated as follows: values \( \mathbb{V} \) in MiniAcumen are real numbers or Booleans. Models are parameterized by the dimension \( N \), which will be dropped when it is obvious from the context. Expressions \( \mathbb{E} \) consist of values, variable names, and functions \( \mathbb{F} \). Atomic constraints \( \mathbb{A} \) include differential equations \( x' = e \), discrete equations \( x^+ = e \) and the error construct \text{error}. A guarded constraint \( \mathbb{G} \) is either an atomic constraint or a conditional if / then / else. Sequences of constraints \( \mathbb{P} \) may have a claim constraint attached. Claims express additional conditions under which a system operates. The conditional construct if / then / else enables guarded
dynamics. The BNF supports nesting of conditionals, making terse hierarchical models possible. A model in MiniAcumen consists of an initially section, where the initial state is specified, and a set of constraints. We use $A$ to denote a sequence of atoms $\langle a_i \rangle_{i \in n}$.

**Notation 4.1.3 (if with empty else).** We use the following short-hand for if statements with an empty else branch:

$$\text{if } / \text{then } \text{if } e \text{ then } \langle a_i \rangle_{i \in n} = \text{if } e \text{ then } \langle a_i \rangle_{i \in n} \text{ else } \langle \rangle$$

**Example 4.1.4 (Sawtooth Wave in MiniAcumen).** The sawtooth wave can be modeled in MiniAcumen as:

$$\text{if } x < 1 \text{ then } x' = 1 \text{ else } x^+ = 0$$

Its behavior over $T = [0..2]$ is illustrated in Figure 3.1.

### 4.2 Translating MiniAcumen to MicroAcumen

In the next chapter (Section 5.8) we will use MicroAcumen to define an operational semantics for a core subset of Acumen. This section defines a translation between these languages to obtain a semantics for MiniAcumen. The translation consists of two stages.

The first stage, given in Definition 4.2.5, transforms a model into a flat set of if statements with no else branch by turning every atomic constraint in the input model into an if. The guard of this if is the *path condition*: the conjunction of conditions that must hold for the atomic constraint to become active in the original model. In the process, claims are eliminated by including their conditions in the path conditions of adjacent constraints. We call this intermediate syntax FlatGuardAcumen. The second stage, given in Definition 4.2.6, combines the single-constraint if statements generated by the first stage into if statements with mutually exclusive guards. We call the resulting syntax MicroAcumen.
CHAPTER 4. SYNTAX AND STATIC SEMANTICS

To get an idea of how the two stages operate on a model let us consider translating a model of a clock in parallel with a sawtooth wave:

\[
\begin{align*}
t' &= 1, \text{if } x < 1 \text{ then } x' = 1 \text{ else } x^+ = 0 \\
\end{align*}
\]

The first stage produces a sequence of if’s without else branches, with one if for each atomic constraint in the original model:

\[
\begin{align*}
&\langle \text{if true then } t' = 1, \\
&\quad \text{if true } \land x < 1 \text{ then } x' = 1, \\
&\quad \text{if true } \land \neg (x < 1) \text{ then } x^+ = 0 \rangle
\end{align*}
\]

The second stage turns the above into a sequence of mutually exclusive if’s:

\[
\begin{align*}
&\langle \text{if true } \land x < 1 \land \neg (x < 1) \text{ then } \langle t' = 1, x' = 1, x^+ = 0 \rangle, \\
&\quad \text{if } \neg \text{true } \land x < 1 \land \neg (x < 1) \text{ then } \langle x' = 1, x^+ = 0 \rangle, \\
&\quad \text{if true } \land \neg (x < 1) \land \neg (x < 1) \text{ then } \langle t' = 1, x^+ = 0 \rangle, \\
&\quad \text{if } \neg \text{true } \land \neg (x < 1) \land \neg (x < 1) \text{ then } \langle x' = 1 \rangle, \\
&\quad \text{if true } \land \neg (x < 1) \land \neg (x < 1) \text{ then } \langle t' = 1 \rangle, \\
&\quad \text{if } \neg \text{true } \land \neg (x < 1) \land \neg (x < 1) \text{ then } \langle \rangle \rangle
\end{align*}
\]

Two of these if’s have one equation per variable, that is, a complete dynamics for the system:

\[
\begin{align*}
&\text{if true } \land \neg (x < 1) \land \neg (x < 1) \text{ then } \langle t' = 1, x^+ = 0 \rangle \\
&\text{if true } \land x < 1 \land \neg \neg (x < 1) \text{ then } \langle t' = 1, x' = 1 \rangle
\end{align*}
\]

The remaining if’s should be considered dead code, as their guards would always be false if they were evaluated with real numbers. However, because these guards will be evaluated using set extensions of the normal arithmetic and Boolean operators, contradictions such as \( x < 1 \land \neg (x < 1) \) may evaluate to both true and false. For example, this happens when \( x = [-1..1] \). To avoid this issue, such code should be eliminated during translation, or be processed by an enhanced evaluation routine that can detect such contradictions on the level of
4.2. TRANSLATION

The syntaxes of FlatGuardAcumen and MicroAcumen use some of the sets given in the definition of MiniAcumen (Definition 4.1.2): variable names \( x \in X \), expressions \( e \in E \) and atomic constraints \( a \in A \).

**Definition 4.2.1 (Syntax of FlatGuardAcumen and MicroAcumen).**

<table>
<thead>
<tr>
<th>Syntax Type</th>
<th>BNF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlatGuardAcumen (( U ))</td>
<td>( u \in U ::= \langle \text{if } e_i \text{ then } a_i \rangle^{i \in n} )</td>
<td></td>
</tr>
<tr>
<td>MicroAcumen (( M^\mu ))</td>
<td>( m \in M^\mu ::= \langle q_i \rangle^{i \in n} )</td>
<td></td>
</tr>
</tbody>
</table>

Dynamics \( C \) are sequences of differential equations, sequences of discrete equations, or a singleton sequence containing the error construct \( \text{error} \). Modes \( Q \) are \( \text{if} \) statements whose \( \text{then} \) branch is a dynamics, and whose \( \text{else} \) branch is empty. MicroAcumen models \( M^\mu \) are sequences of modes.

MicroAcumen is intended as the smallest possible subset of MiniAcumen that has the same expressivity. By defining the semantics for a minimal core language, the definition of the semantics becomes small, and thus easier to analyze.

**Definition 4.2.2 (Guards, Fields and Order).** We define the functions \( \text{guard} : Q \to E \) and \( \text{field} : Q \to E^n \) that extract the guard and field of a mode, and the function \( \text{order} : M \to N \) that extracts the dimension of a model are defined as follows:

\[
\begin{align*}
\text{guard}(\text{if } e \text{ then } A) &= e \\
\text{field}(\text{if } e \text{ then } \langle x'_i = e_i \rangle^{i \in N}) &= \langle e_i \rangle^{i \in N} \\
\text{field}(\text{if } e \text{ then } \langle x^+_i = e_i \rangle^{i \in N}) &= \langle e_i \rangle^{i \in N} \\
\text{order}(\text{model Main() = initially } \langle x_j = e_j \rangle^{j \in N} \text{ always } p) &= N
\end{align*}
\]
Modes are the basic building blocks of MicroAcumen. They are used to model the dynamics $\langle a_i \rangle_{i \in n}$ of a system whenever the guard condition $e$ holds. We distinguish two kinds of modes: those where $a_i \equiv x'_i = e_i$ are called continuous modes; those where $a_i \equiv x'^+_i = e_i$ are called discrete modes.

**Example 4.2.3** (Sawtooth Wave in MicroAcumen). The sawtooth wave can be modeled in MicroAcumen as:

$$\langle \text{if } x < 1 \text{ then } \langle x' = 1 \rangle, \text{ if } x \geq 1 \text{ then } \langle x^+ = 0 \rangle \rangle$$

**Example 4.2.4** (Bouncing Ball in MicroAcumen). A bouncing ball with a coefficient of restitution of $\frac{1}{2}$ and affected by gravity with $g = 10$ can be modeled in MicroAcumen as:

$$\langle \text{ if } x \leq 0 \land y < 0 \text{ then } \langle y^+ = -\frac{y}{2} \rangle, \text{ if } \neg (x \leq 0 \land y < 0) \text{ then } \langle x' = y, y' = -10 \rangle \rangle$$

where $x$ and $y$ are the vertical position and velocity of the ball respectively.

### 4.2.1 Translating MiniAcumen to FlatGuardAcumen

This section describes the first step of the translation, that is, a translation from MiniAcumen to FlatGuardAcumen. In this step (Definition 4.2.5), flat guard constraints are combined into mutually exclusive MicroAcumen modes, each of which are a complete dynamics for the system.

The translation is based on the notion of a path condition. This is a Boolean expression, a conjunction of all the guards (Definition 4.2.2) that must be satisfied to reach the currently translated term. The translation is initialized with a true path condition.

**Definition 4.2.5** (Flattening Normalization). The flattening normalization of a constraint or set of constraints $x \in \mathcal{G} \cup \mathcal{P}$ at a path
condition $e \in \mathbb{E}$ is a function $? : \mathbb{E} \times (\mathbb{G} \cup \mathbb{P}) \rightarrow \mathbb{U}$ defined as follows:

$$
e ? a \triangleq \langle \text{if } e \text{ then } \langle a \rangle \rangle$$
$$e ? \text{if } e_1 \text{ then } p_1 \text{ else } p_2 \triangleq e \land e_1 ? p_1 \oplus e \land \neg e_1 ? p_2$$
$$e ? \langle \text{claim } e_c \rangle \oplus \langle g_i \rangle_{i \in n} \triangleq \langle e \land e_c ? g_i \rangle_{i \in n} \oplus \langle \text{if } e \land \neg e_c \text{ then } \langle \text{error} \rangle \rangle$$

The flat normal form of the following MiniAcumen model (from $\mathbb{M}$)

$$\text{model Main}(\cdot) = \text{initially } i \text{ always } p$$

is

$$\text{model Main}(\cdot) = \text{initially } i \text{ always true ? } p$$

The second stage of the translation from MiniAcumen to MicroAcumen transforms the set of modes produced by the flattening normalization into a set of independent modes. The modes are independent both in the sense that their conditions are mutually exclusive, and in that the dynamics of each mode are fully and uniquely defined.

### 4.2.2 Translating FlatGuardAcumen to MicroAcumen

This section describes the second step of the translation, that is, a translation from FlatGuardAcumen to MicroAcumen. In this step (Definition 4.2.6), flat guard constraints are combined into mutually exclusive MicroAcumen modes, each of which are a complete dynamics for the system.

**Definition 4.2.6 (Partitioning).** The partitioning of a flat guard constraints $\langle \text{if } e_i \text{ then } \langle a_i \rangle \rangle_{i \in n}$ is the function $\text{partition} : \mathbb{U} \rightarrow \mathbb{M}^\mu$ defined as follows:

$$\text{partition}(m) = \begin{cases} 
\langle \rangle & \text{if } m = \langle \rangle \\
\langle \text{if } e \text{ then } A, \text{ if } \neg e \text{ then } \langle \rangle \rangle & \text{if } m = \langle \text{if } e \text{ then } A \rangle \\
\bigoplus_{g \in \text{partition}(m_1)} \text{inject}(g_1, g) & \text{if } m = \langle g_1 \rangle \oplus m_1
\end{cases}$$
where the function \( \text{inject} : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{M}^\mu \) is defined as follows:

\[
\text{inject}(\text{if } e_1 \text{ then } A_1, \text{if } e_2 \text{ then } A_2) = \\
\langle \text{if } e_1 \land e_2 \text{ then } A_1 \oplus A_2, \text{if } \neg e_1 \lor e_2 \text{ then } A_2 \rangle
\]

\[\quad \square\]

**Example 4.2.7** (Parallel Sub-Systems). A system with two parallel, continuous sub-systems can be modeled in FlatGuardAcumen as:

\[
\langle \text{if } e_1 \text{ then } \langle x' = e_2 \rangle, \text{if } e_3 \text{ then } \langle y' = e_4 \rangle \rangle
\]

Its partitioned form is:

\[
\langle \text{if } e_1 \land e_3 \text{ then } \langle x' = e_2, y' = e_4 \rangle, \text{if } \neg e_1 \land e_3 \text{ then } \langle y' = e_4 \rangle \rangle
\]

\[\quad \text{, if } e_1 \land \neg e_3 \text{ then } \langle x' = e_2 \rangle \quad \text{, if } \neg e_1 \land \neg e_3 \text{ then } \langle \rangle \rangle\]

### 4.2.3 Complexity of Translation from MiniAcumen to FlatGuardAcumen

The flattening normalization does not duplicate any constraints in a model, and thus yields only a moderate inflation in the size of the input. The partitioning must duplicate constraints to make each mode in the resulting model independent, and thus produces much larger models. We will now make these intuitive statements more concrete by formalizing the notion of size for a MiniAcumen model and by proving bounds for the increase in size incurred by the flattening normalization.

**Definition 4.2.8** (Size). Size of expressions is a function \( |\cdot| : \mathbb{E} \rightarrow \mathbb{N} \) defined as follows:

\[
|x| \triangleq 1 \\
|v| \triangleq 1 \\
|f\langle e_i \rangle_{i \in n}| \triangleq 1 + \sum_{i \in n} |e_i|
\]

Size of atoms is a function \( |\cdot| : \mathbb{A} \rightarrow \mathbb{N} \) defined as follows:

\[
|x' = e| \triangleq 1 + |e| \\
|x^+ = e| \triangleq 1 + |e| \\
|\text{error}| \triangleq 1
\]
Size of constraints is a function $|\cdot| : \mathbb{G} \to \mathbb{N}$ defined as follows:

$$
|\text{if } e \text{ then } g_1 \text{ else } g_2| \triangleq 1 + |e| + |g_1| + |g_2|
$$

$$
|\langle g_i \rangle_{i \in n}| \triangleq 1 + \sum_{i \in n} |g_i|
$$

$$
|\langle \text{claim } e \rangle \oplus \langle g_i \rangle_{i \in n}| \triangleq 2 + |e| + \sum_{i \in n} |g_i|
$$

\[\square\]

**Proposition 4.2.9** (Complexity of Flattening Normalization). For all $g$ in $\mathbb{P} \cup \mathbb{G}$ and $e \in \mathbb{E}$ it holds that $|e ? g| \leq 4 + |e||g| + 2|g|^2$.

**Proof.** Capital letters $G$, $E$ denote the sizes $|g|, |e|$ of terms $g$, $e$. Our goal is to find a function $F : \mathbb{E} \times \mathbb{P} \cup \mathbb{G} \to \mathbb{N}$ such that:

$$
\forall g \in \mathbb{G}, \forall e \in \mathbb{E}. |e ? g| \leq F(E, G) \quad (4.1)
$$

We begin by inferring sufficient conditions on $F$ for each case of $g$. In the cases where $g$ contains sub-terms, these conditions are sufficient, assuming that (4.1) holds for these sub-terms.

**Case** $g \equiv a$:

\[
|e ? g| = |e ? a| \quad \text{(by substitution)}
\]

\[
= |\langle \text{if } e \text{ then } a \rangle| \quad \text{(by Def 4.2.5)}
\]

\[
= |\langle \text{if } e \text{ then } a \rangle \text{ else } \rangle| \quad \text{(by Def 4.1.3)}
\]

\[
= 1 + |\text{if } e \text{ then } a \rangle \text{ else } \rangle| \quad \text{(by Def 4.2.8)}
\]

\[
= 1 + 1 + E + (1 + A) + |\rangle| \quad \text{(by Def 4.2.8)}
\]

\[
= 1 + 1 + E + 1 + A + 1 \quad \text{(by Def 4.2.8)}
\]

\[
= 4 + E + A \quad \text{(by addition)}
\]

\[
= 4 + E + G \quad \text{(by substitution)}
\]

\[
\leq F(E, G) \quad \text{(sought property of F)}
\]
CHAPTER 4. SYNTAX AND STATIC SEMANTICS

Case \( g \equiv \langle g_i \rangle_{i \in n} \):

\[
\begin{align*}
|e \cdot g| & = |e \cdot \langle g_i \rangle_{i \in n}| \quad \text{(by substitution)} \\
& = |\langle e \cdot g_i \rangle_{i \in n}| \quad \text{(by Def 4.2.5)} \\
& = 1 + \sum_{i \in n} |e \cdot g_i| \quad \text{(by Def 4.2.8)} \\
& \leq 1 + \sum_{i \in n} F(E, G_i) \quad \text{(by Eq 4.1)} \\
& = F(E, 1 + \sum_{i \in n} G_i) \quad \text{(by sought property of F)}
\end{align*}
\]

Case \( g \equiv \langle \text{claim} e_c \rangle \oplus \langle g_i \rangle_{i \in n} \):

\[
\begin{align*}
|e \cdot g| & = |e \cdot \langle \text{claim} e_c \rangle \oplus \langle g_i \rangle_{i \in n}| \quad \text{(by substitution)} \\
& = |\langle e \land e_c \cdot g_i \rangle_{i \in n} \oplus \langle \text{if } e \land \neg e_c \text{ then (error) else } () \rangle| \quad \text{(by Def 4.2.5)} \\
& = |\langle e \land e_c \cdot g_i \rangle_{i \in n} \oplus \langle \text{if } e \land \neg e_c \text{ then (error) else } () \rangle| \quad \text{(by Def 4.1.3)} \\
& = 1 + \sum_{i \in n} |e \land e_c \cdot g_i| \quad \text{(by Def 4.2.8)} \\
& \quad + (1 + |e \land \neg e_c| + |\langle \text{error} \rangle| + 1) \\
& = 1 + \sum_{i \in n} |e \land e_c \cdot g_i| + (6 + E + E_c) \quad \text{(by Def 4.2.8)} \\
& = 7 + E + E_c + \sum_{i \in n} |e \land e_c \cdot g_i| \quad \text{(sought property of F)} \\
& \leq 7 + E + E_c + \sum_{i \in n} F(|e \land e_c|, G_i) \quad \text{(by Eq 4.1)} \\
& = 7 + E + E_c + \sum_{i \in n} F(1 + E + E_c, G_i) \quad \text{(by Def 4.2.8)} \\
& \leq F(E, 2 + E_c + \sum_{i \in n} G_i) \quad \text{(sought property of F)}
\end{align*}
\]

Case \( g \equiv \text{if } e_1 \text{ then } p_1 \text{ else } p_2 \):

\[
\begin{align*}
|e \cdot g| & = |e \cdot \text{if } e_1 \text{ then } p_1 \text{ else } p_2| \quad \text{(by substitution)} \\
& = |e \land e_1 ? p_1 | \cup |e \land \neg e_1 ? p_2| \quad \text{(by Def 4.2.5)} \\
& = 1 + |e \land e_1 ? p_1 | + |e \land \neg e_1 ? p_2| \quad \text{(by Def 4.2.8)} \\
& \leq 1 + F(|e \land e_1|, P_1) + F(|e \land \neg e_1|, P_2) \quad \text{(by Eq 4.1)} \\
& = 1 + F(1 + E + E_1, P_1) + F(2 + E + E_1, P_2) \quad \text{(by Def 4.2.8)} \\
& \leq F(E, G) \quad \text{(sought property of F)} \\
& = F(E, 1 + E_1 + P_1 + P_2) \quad \text{(by Def 4.2.8)}
\end{align*}
\]
4.2. TRANSLATION

In summary, we have the following four conditions on $F$:

$$3 + E + G \leq F(E, G) \quad (4.2)$$

$$1 + \sum_{i \in n} F(E, G_i) \leq F(E, 1 + \sum_{i \in n} G_i) \quad (4.3)$$

$$7 + E + E_c + \sum_{i \in n} F(1 + E + E_c, G_i) \leq F(E, 2 + E_c + \sum_{i \in n} G_i) \quad (4.4)$$

$$1 + F(1 + E + E_1, P_1) + F(2 + E + E_1, P_2) \leq F(E, 1 + E_1 + P_1 + P_2) \quad (4.5)$$

We further simplify the task of finding a suitable $F$ by introducing additional assumptions about $F$. We are free to do so, as long as we can find an $F$ that satisfies them, along with the above inequalities.

$$F(A, B) \leq F(A + 1, B) \quad (4.6)$$

$$\forall k \in \mathbb{N}^+. F(2 + k + A, B) \leq F(A, k + B) \quad (4.7)$$

We proceed by showing that (4.5) follows from (4.3), (4.6), and (4.7):

$$1 + F(1 + E + E_1, P_1) + F(2 + E + E_1, P_2) \leq F(2 + E + E_1, P_1) + F(2 + E + E_1, P_2) \quad (\text{by Eq 4.6})$$

$$\leq 1 + F(2 + E + E_1, 1 + P_1 + P_2) \quad (\text{by Eq 4.3})$$

$$\leq 1 + F(E, 1 + E_1 + P_1 + P_2) \quad (\text{by Eq 4.7})$$

Let the following be the candidate upper complexity bound:

$$F(E, G) = 4 + EG + 2G^2 \quad (4.8)$$

To show that (4.1) holds for all $g \in G$, we must first show it for all cases where $g$ contains no sub-terms, i.e. for $g \equiv a$. For the remaining cases of $g$ we must show that (4.1) holds, assuming it does so for sub-terms of $g$. As we have seen above, it thus remains to prove that (4.8) satisfies (4.2), (4.3), (4.4), (4.6), and (4.7).
Case (4.2):

\[
4 + E + G \\
\leq 4 + EG + 2G^2 \quad \text{(as } G \geq 1) \]

Case (4.3):

\[
1 + \sum_{i \in n} F(E, G_i) \\
= 1 + \sum_{i \in n} (3 + EG_i + 2G_i^2) \quad \text{(by Eq 4.8)} \\
= 1 + \sum_{i \in n} 3 + E \sum_{i \in n} G_i + 2 \sum_{i \in n} G_i^2 \quad \text{(by def. of } \sum) \\
\leq 5 + \sum_{i \in n} 4 + E (1 + \sum_{i \in n} G_i) + 2 \sum_{i \in n} G_i^2 \quad \text{(as } E \geq 1) \\
= 5 + E (1 + \sum_{i \in n} G_i) + 4 \sum_{i \in n} 1 + 2 \sum_{i \in n} G_i^2 \quad \text{(by def. of } \sum) \\
\leq 5 + E (1 + \sum_{i \in n} G_i) + 4 \sum_{i \in n} G_i + 2(\sum_{i \in n} G_i^2 + \sum_{i \neq j} G_i G_j) \quad \text{(as } G_i \geq 1) \\
= 5 + E (1 + \sum_{i \in n} G_i) + 4 \sum_{i \in n} G_i + 2(\sum_{i \in n} G_i)^2 \quad \text{(by def. of } \sum) \\
= 3 + E (1 + \sum_{i \in n} G_i) + 2(1 + \sum_{i \in n} G_i) \quad \text{(by fact.)} \\
= 3 + E (1 + \sum_{i \in n} G_i) + 2(1 + \sum_{i \in n} G_i)^2 \quad \text{(by fact.)} \\
= F(E, 1 + \sum_{i \in n} G_i) \quad \text{(by Eq 4.8)}
\]
4.2. TRANSLATION

Case (4.4):

\[ 7 + E + E_c + \sum_{i \in n} F(1 + E + E_c, G_i) \]
\[ = 7 + E + E_c + \sum_{i \in n} (4 + (1 + E + E_c)G_i + 2G_i^2) \]
\[ \leq 7 + E + E_c + \sum_{i \in n} ((4 + E + E_c)G_i + 2G_i^2) \]
\[ = 7 + E + E_c + (4 + E + E_c) \sum_{i \in n} G_i + 2 \sum_{i \in n} G_i^2 \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ = 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
\[ \leq 12 + 2E + 8E_c + (4 + E + E_c) \sum_{i \in n} G_i \]
CHAPTER 4. SYNTAX AND STATIC SEMANTICS

4.3 A Type System for Mini- (and Micro-) Acumen

This section describes a simple type system for MiniAcumen. It is used to prove that the flattening normalization (Definition 4.2.5) maps well-typed terms to well-typed terms. This type system can also be used in two ways in an implementation. First, it can ensure that simulations do not fail due to typing errors, which is useful given that these computations can be long-running. Second, the type system can be used in a compiler, to determine the correct types for terms during code generation.

Definition 4.3.1 (Types and Typing Environments).

Types \( \tau \in \text{Type} := \text{real} \mid \text{bool} \)

Environments \( \Gamma \in \Gamma := [ ] \mid \Gamma, x \mapsto \tau \)

Function type environments \( \Sigma \in \Sigma := [ ] \mid \Sigma, f \mapsto \langle \tau_i \rangle _{i \in I} \rightarrow \tau \)

The type system for MiniAcumen is given in Figure 4.1. □

The above BNF defines the sets indicated as follows: types \( \text{Type} \) are \text{real} or \text{bool}. Variable type environments \( \Gamma \) are empty or a variable type environment extended with a mapping from variable name to type. Function type environments \( \Sigma \) are empty or a function type environment extended with a mapping from function name to a function from a list of parameter types to a result type.

The rules in Figure 4.1 define the typing relation. The Real, Bool, Var and Fun rules are used to check expressions. The Real and Bool rules establish that values are well-typed. By the Var rule a variable is well-typed if the variable typing environment \( \Gamma \) contains a corresponding mapping. By the Fun rule a function application is well-typed if the function typing environment contains a corresponding mapping, and if the parameter types of that mapping match the types of the expressions used in the application.

The Error, Cont and Disc rules are used to check atomic constraints. The Error rule establishes that the error constraint is well-typed. By the Cont and Disc rules respectively, continuous and discrete equations are well-typed if the variable typing environment maps
the left-hand side to the *real*, and the type of the right-hand side expression is *real*.

By the If rule an if / then / else statement is well-typed if its condition is of type *bool*, and if both its branches are well-typed. The Seq and SeqC rules are used to check sequences of statements. Both rules require that each of the guarded constraints in the sequence is well-typed. The SeqC rule additionally requires that the *claim* expression is of *bool* type.

By the Model rule a model is well-typed if, for each of its initializations, both the right and the left-hand side are of type *real*, and if the guarded constraint *p* is well-typed.

**Proposition 4.3.2** (Flattening Preserves Typability). Whenever \( \Sigma \in \Sigma, \Gamma \in \Gamma, e \in \mathcal{E}, g \in \mathcal{G} \), and we have that \( \Sigma, \Gamma \vdash e : \text{bool} \wedge \Sigma, \Gamma \vdash g, p \) then \( \Sigma, \Gamma \vdash e \, ? \, g, e \, ? \, p \).

**Proof.** We proceed by induction on the structure of *g*. For all cases, assume that \( \Sigma, \Gamma \vdash e : \text{bool} \). In what follows we consider all possible cases:

Case \( g \equiv x' = e_1 \): We need to prove that if \( \Sigma, \Gamma \vdash x' = e_1 \) then \( \Sigma, \Gamma \vdash e \, ? \, x' = e_1 \).

\[
\begin{align*}
e \, ? \, g & \equiv e \, ? \, x' = e_1 \\
& \equiv \langle \text{if } e \text{ then } x' = e_1 \rangle \\
& \equiv \langle \text{if } e \text{ then } x' = e_1 \text{ else } \langle \rangle \rangle
\end{align*}
\]

It thus remains to prove that \( \Sigma, \Gamma \vdash \langle \text{if } e \text{ then } x' = e_1 \text{ else } \langle \rangle \rangle \).

\[
\begin{array}{c}
\Sigma, \Gamma \vdash e : \text{bool} \\
\Sigma, \Gamma \vdash x' = e_1 \\
\Sigma, \Gamma \vdash \langle \rangle
\end{array}
\]

\[
\frac{\text{Seq}}{\Sigma, \Gamma \vdash \langle \text{if } e \text{ then } x' = e_1 \text{ else } \langle \rangle \rangle}
\]

\[
\begin{array}{c}
\Sigma, \Gamma \vdash e \in \text{bool} \\
\Sigma, \Gamma \vdash \langle \rangle
\end{array}
\]

\[
\frac{\text{SeqC}}{\Sigma, \Gamma \vdash \langle \text{if } e \text{ then } x' = e_1 \text{ else } \langle \rangle \rangle}
\]

As both \( \Sigma, \Gamma \vdash e : \text{bool} \) and \( \Sigma, \Gamma \vdash x' = e_1 \) hold by assumption, we are done proving this case.
\[\Sigma, \Gamma \vdash e\]
\[\Sigma, \Gamma \vdash r : \text{real} \quad (\text{Real}) \quad \Sigma, \Gamma \vdash b : \text{bool} \quad (\text{Bool})\]
\[\Gamma(x) = \tau \quad (\text{Var}) \quad \Sigma, \Gamma \vdash \langle \tau_i \rangle_{i \in I}^{\tau} \quad (\text{Fun})\]
\[\Sigma, \Gamma \vdash a \quad (\text{Error})\]
\[\Sigma, \Gamma \vdash e : \text{real} \quad \Sigma, \Gamma \vdash e : \text{real} \quad \Sigma, \Gamma \vdash \text{error} \quad (\text{Cont}) \quad \Gamma(x) = \text{real} \quad \Gamma(x) = \text{real} \quad (\text{Disc})\]
\[\Sigma, \Gamma \vdash g \quad \Sigma, \Gamma \vdash e : \text{bool} \quad \Sigma, \Gamma \vdash g_1 \quad \Sigma, \Gamma \vdash g_2 \quad (\text{If})\]
\[\Sigma, \Gamma \vdash p \quad \{\Sigma, \Gamma \vdash g_i \}_{i \in n}^{\text{Seq}} \quad \Sigma, \Gamma \vdash e : \text{bool} \quad \{\Sigma, \Gamma \vdash g_i \}_{i \in n}^{\text{SeqC}} \quad (\text{Seq}) \quad (\text{SeqC})\]
\[\Sigma, \Gamma \vdash m \quad \{\Sigma, \Gamma \vdash x : \text{real}, e_j : \text{real} \}_{j \in n}^{\text{Model}} \quad \Sigma, \Gamma \vdash p \quad \Sigma, \Gamma \vdash \text{model Main()} = \text{initially} \langle x_j = e_j \rangle_{j \in n}^{\text{always} p} \quad (\text{Model})\]

Figure 4.1: A Type System for MiniAcumen
4.3. A TYPE SYSTEM FOR MINI- (AND MICRO-) ACUMEN

Case \( g \equiv \text{if } e_1 \text{ then } g_1 \text{ else } g_2 \): We need to prove that if \( \Sigma, \Gamma \vdash \text{if } e_1 \text{ then } g_1 \text{ else } g_2 \) then \( \Sigma, \Gamma \vdash e? \text{ if } e_1 \text{ then } g_1 \text{ else } g_2 \).

\[
e? g = e? \text{ if } e_1 \text{ then } g_1 \text{ else } g_2
\]

(by substitution) \quad \equiv \quad (by Def 4.2.5) \quad = \quad e \land e_1? g_1 \cup e \land \neg e_1? g_2

It thus remains to prove that \( \Sigma, \Gamma \vdash e \land e_1? g_1 \cup e \land \neg e_1? g_2 \). We note that by the (If) rule of Figure 4.1 we have that \( \Sigma, \Gamma \vdash e_1, g_1, g_2 \).

\[
\begin{align*}
\Sigma, \Gamma &\vdash e_1 & \text{Fun} \\
\Sigma, \Gamma &\vdash \neg e_1 & \text{Fun} \\
\Sigma, \Gamma &\vdash e_1 \land \neg e_1? g_2 & \text{I.H.}
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Gamma &\vdash e_1 & \text{Fun} \\
\Sigma, \Gamma &\vdash e_1 & \text{Fun} \\
\Sigma, \Gamma &\vdash e_1 \land e_1? g_1 & \text{I.H.}
\end{align*}
\]

\[
\Sigma, \Gamma \vdash e_1 \land e_1? g_1 \cup e_1 \land \neg e_1? g_2 & \text{Seq}
\]

Case \( g \equiv \langle g_i \rangle_{i \in n} \): We need to prove that if \( \Sigma, \Gamma \vdash \langle g_i \rangle_{i \in n} \) then \( \Sigma, \Gamma \vdash e? \langle g_i \rangle_{i \in n} \).

\[
\begin{align*}
\Sigma, \Gamma &\vdash e? g & \text{(by substitution)} \\
\Sigma, \Gamma &\vdash e? \langle g_i \rangle_{i \in n} & \text{(by Def 4.2.5)} \\
\Sigma, \Gamma &\vdash \langle e? g_i \rangle_{i \in n} & \text{(Seq)} \\
\Sigma, \Gamma &\vdash \langle g_i \rangle_{i \in n} & \text{(I.H.)}
\end{align*}
\]

\[
\begin{align*}
\Sigma, \Gamma &\vdash e? g & \text{(by substitution)} \\
\Sigma, \Gamma &\vdash e? \langle g_i \rangle_{i \in n} & \text{(by Def 4.2.5)} \\
\Sigma, \Gamma &\vdash \langle e? g_i \rangle_{i \in n} \\
\Sigma, \Gamma &\vdash \langle g_i \rangle_{i \in n} \cup \langle \text{error} \rangle & \text{(I.H.)}
\end{align*}
\]

This holds by the induction hypothesis, as \( \forall i. g_i \) is a sub-term of \( g \).

Case \( g \equiv \langle \text{claim } e_c \rangle \oplus \langle g_i \rangle_{i \in I} \): We need to prove that if \( \Sigma, \Gamma \vdash e_c \) and \( \bigwedge_{i \in I} \Sigma, \Gamma \vdash g_i \) then \( \Sigma, \Gamma \vdash e? \langle \text{claim } e_c \rangle \oplus \langle g_i \rangle_{i \in I} \).

\[
\begin{align*}
\Sigma, \Gamma &\vdash e? g & \text{(by substitution)} \\
\Sigma, \Gamma &\vdash e? \langle \text{claim } e_c \rangle \oplus \langle g_i \rangle_{i \in n} & \text{(by Def 4.2.5)} \\
\Sigma, \Gamma &\vdash \langle e \land e_c? g_i \rangle_{i \in n} \cup \langle \text{if } e \land \neg e_c \text{ then } \text{error} \rangle
\end{align*}
\]
It remains to prove that $\Sigma, \Gamma \vdash \langle e \land e_c ? g_i \rangle_{i \in n} \oplus \langle \text{if } e \land \neg e_c \text{ then } \langle \text{error} \rangle \rangle$.

The Seq rule says that we need to show that $\Sigma, \Gamma \vdash \langle e \land e_c ? g_i \rangle_{i \in n}$ and $\Sigma, \Gamma \vdash \{\text{if } e \land \neg e_c \text{ then } \langle \text{error} \rangle \}$ independently. The former follows from the Seq rule and the latter from the If rule.
Chapter 5

Mathematical Semantics

This chapter defines the denotational semantics (Subsection 5.4.4) and operational semantics (Section 5.8) of MicroAcumen. The denotational semantics gives a meaning to MicroAcumen terms as mathematical objects. Its purpose is to serve as an independent, unambiguous definition against which another definition can be compared. The operational semantics is a model of the way in which the rigorous simulator constructs enclosures. We also present a conjecture about the soundness of the operational semantics with respect to the denotational semantics (Section 5.9). Results that may help in proving this conjecture are given in Appendix A.

5.1 Introduction

A mathematical semantics gives meaning to the terms of a language in a way that makes it possible to analyze the interpretation.

The following diagram illustrates the semantic functions ($\mathcal{H}(\cdot)$, $\cdot \hookrightarrow \cdot$ and $[\cdot]$) presented in this chapter, as well as the soundness property conjectured in Section 5.9.
5.2 Syntax of MicroAcumen

To recall, the syntax of MicroAcumen (Definition 4.2.1) is as follows:

\[
\begin{align*}
\text{MicroAcumen} : & M^\mu \\
\text{Value} & : v \in V ::= r \mid b \\
\text{Expression} & : e \in E ::= v \mid x \mid f(e_i)^{i \in n} \\
\text{Dynamics} & : c \in C ::= \langle x'_i = e_i \rangle^{i \in N} \mid \langle x^+_i = e_i \rangle^{i \in N} \mid \langle \text{error} \rangle \\
\text{Mode} & : q \in Q ::= \text{if } e \text{ then } c \\
\text{Model} & : m \in M^\mu ::= \langle q_i \rangle^{i \in n}
\end{align*}
\]

To define the semantics of MicroAcumen we will use the concept of a field expression, which represents the right-hand sides of a dynamics \( \langle x'_i = e_i \rangle^{i \in N} \) or \( \langle x^+_i = e_i \rangle^{i \in N} \).

**Definition 5.2.1** (Field Expression). A sequence of expressions \( \langle e_i \rangle^{i \in N} : \mathbb{E}^N \) is called a field expression. \( \square \)

5.3 Denotational Semantics of Types and Expressions

We can now define a denotation for types as reals or Booleans, and for terms as partial functions between topological spaces.

5.3.1 Topological Spaces

We will use three topological spaces to define the semantics of Boolean expressions (Definition 5.3.3), of expressions that occur in fields (Definition 5.3.2), and of models (Definition 5.3.4).
**Definition 5.3.1** (Topologies and Topological Notions). A *topology* on a set $X$ is a set $\mathcal{T}$ of subsets of $X$ that satisfies the following conditions:

1. $\emptyset, X \in \mathcal{T}$
2. $x, y \in \mathcal{T} \implies x \cap y \in \mathcal{T}$
3. $Y \subseteq \mathcal{T} \implies \bigcup Y \in \mathcal{T}$

That is, the empty set and all of $X$ are in the topology, and $\mathcal{T}$ is closed under finite intersections and arbitrary unions. The elements of $\mathcal{T}$ are called the *open sets* of $X$.

Let $\mathcal{T}$ be a topology on $X$, then:

1. The structure $(X, \mathcal{T})$ is called a *topological space*.
2. The *complement* of a set $Y \subseteq X$, written as $\overline{Y}$, is defined as:
   
   $\overline{Y} \triangleq \{x \in X \mid x \not\in Y\}$

3. A subset $Y \subseteq X$ is *closed* if and only if $Y = \overline{Z}$ for some $Z \in \mathcal{T}$
4. The *topological closure* of $Y \subseteq X$, written $\mathrm{closure}(Y)$, is the smallest closed subset of $X$ that contains $Y$

We equip the reals with the topology used in classical real analysis:

**Definition 5.3.2** (Real Numbers with the Standard Topology). The *real numbers with the standard topology* is the topological space $(\mathbb{R}, \mathcal{T})$ where $\mathcal{T}$ is a set that satisfies the conditions in Definition 5.3.1 and contains all *open intervals*, that is:

$$\forall a, b \in \mathbb{R}. \ a < b. \ \{x \in \mathbb{R} \mid a < x < b\} \subseteq \mathcal{T}$$
We equip the Booleans with the trivial topology, that is, the simplest possible definition that permits us to view the Booleans as a topological space:

**Definition 5.3.3** (Booleans with the Trivial Topology). The *Booleans with the trivial topology* is the topological space $(\mathbb{B}, \mathcal{T})$ where $\mathcal{T}$ contains only $\emptyset$ and the whole space $\mathbb{B}$. □

The above definitions will let us interpret expressions as partial continuous functions (Definition 5.3.8).

**Definition 5.3.4** (State Space). The state space is the topological space $(\mathbb{R}^n, \mathcal{T})$, where $\mathbb{R}^n$ is the set of $n$-dimensional vectors of real numbers, that is:

$$\mathbb{R}^n = \langle x_i \mid x_i \in \mathbb{R} \rangle_{i \in \mathbb{N}}$$

and $\mathcal{T}$ is the subset of $\mathcal{P}(\mathbb{R}^n)$, such that when $X \in \mathcal{T}$, then $\forall i \in \mathbb{N}. \pi_i(X)$ is an open subset of $\mathbb{R}$ with respect to the standard topology. □

### 5.3.2 Categorical Notions

Category theory is a general framework for describing maps and the objects they operate on. The semantics interprets models as partial maps on the state space, seen as a topological space. In other words, the maps are morphisms in the following category:

**Definition 5.3.5** (Category of Topological Spaces and Partial Continuous Maps). The category $\textbf{Top}_p$ is the category whose objects are topological spaces and whose morphisms are partial continuous maps. □

**Notation 5.3.6** (Hom-Sets). Given a category $\mathbf{K}$ with objects $X$ and $Y$, we write $\mathbf{K}(X,Y)$ to denote the *hom-set* (set of morphisms) from $X$ to $Y$. □
5.3.3 Interpretation of Types

**Definition 5.3.7** (Interpretation of Value Types). Value types are interpreted as follows:

\[
\begin{align*}
\llbracket \tau \rrbracket &= \mathbb{R} \\
\llbracket \text{real} \rrbracket &= \mathbb{R} \\
\llbracket \text{bool} \rrbracket &= \mathbb{B}
\end{align*}
\]

where \( \mathbb{R} \) is the topological space of reals with the standard topology (Definition 5.3.2) and \( \mathbb{B} \) is the space of Booleans with the trivial topology (Definition 5.3.3).

5.3.4 Interpretation of Terms

Having defined the mathematical framework we want to map a MicroAcumen model into, we can now describe the interpretations of its terms. Each mode is interpreted as a relation on the state space. Basically, the relation associates a point that satisfies the guard expression with the value of the field expression in that point. In the process, we must pay attention to cases when the interpretation may fail, or when the resulting map may be undefined. Thus, well-typed expressions are interpreted as partial maps between topological spaces.

Depending on where these expressions occur in a model, their type will be either Boolean (guard expressions) or real vector (field expressions). Specifically:

**Definition 5.3.8** (Admissible Interpretations of Guard and Field Expressions). Guard expressions \( e \in \mathcal{E} \) may be interpreted as follows:

\[
\llbracket \vdash e : \text{bool} \rrbracket \\
\llbracket \cdot \rrbracket : \mathcal{E} \rightarrow \mathbf{Top}_p(\mathcal{S}, \mathcal{B})
\]

Field expressions \( (e_i)_{i \in \mathbb{N}} \in \mathcal{E}^\mathbb{N} \) may be interpreted as follows:

\[
\llbracket (\vdash e_i : \text{real})_{i \in \mathbb{N}} \rrbracket \\
\llbracket \cdot \rrbracket : \mathcal{E}^\mathbb{N} \rightarrow \mathbf{Top}_p(\mathcal{S}, \mathcal{S})
\]

Such interpretations are undefined on ill-typed expressions, such as \((1 > 2) > 3\). Note that this category allows us to interpret partial functions, for example, the expression \(1/x\).
Definition 5.3.9 (Interpretation of Guarded Field Expressions). A field expression \(<e_i>_{i \in \mathbb{N}}\) guarded by a condition \(e\) is interpreted by the function \(\llbracket \text{if } \cdot \text{ then } \cdot = \cdot \rrbracket : \mathbb{E} \times \mathbb{E}^\mathbb{N} \rightarrow \mathcal{P}(\mathbb{S}^2)\), defined as follows:

\[
\llbracket \text{if } e \text{ then } \cdot = \cdot \rrbracket \triangleq \{(s_1, s_2) \in \mathbb{S} \times \mathbb{S} | \llbracket e \rrbracket s_1 = \text{true} \land s_2 = \llbracket \langle e_i \rangle_{i \in \mathbb{N}} \rrbracket s_1\}
\]

where the set \(\llbracket \text{if } e \text{ then } \cdot = \cdot \rrbracket\) is empty if \(\llbracket e \rrbracket\) or \(\llbracket \langle e_i \rangle_{i \in \mathbb{N}} \rrbracket\) are undefined.

The “\(\cdot\)” in the definition above indicates that the interpretation of guarded fields is the same for discrete and continuous equations.

Subsection 5.7.1 defines the function uncondition that replaces the guard expression of a mode with the expression true. The following proposition formalizes the idea that such a replacement does not make the interpretation of a mode smaller.

Proposition 5.3.10. Let \(s \in \mathbb{S}, e_1, e_2 \in \mathbb{E} \text{ and } f \in \mathbb{E}^n\) then

\[
(\llbracket e_1 \rrbracket s = \text{true} \implies \llbracket e_2 \rrbracket s = \text{true}) \implies \llbracket \text{if } e_1 \text{ then } \cdot = f \rrbracket \subseteq \llbracket \text{if } e_2 \text{ then } \cdot = f \rrbracket \quad (5.1)
\]

Proof. Let \((s_1, s_2) \in \llbracket \text{if } e_1 \text{ then } \cdot = f \rrbracket\) and assume that \(\llbracket e_1 \rrbracket s_1 = \text{true}\).

\[
(s_1, s_2) = (s_1, \llbracket f \rrbracket s_1) \quad \text{(by Definition 5.3.9)}
\]

\[
\in \{(s_3, s_4) \in \mathbb{S} \times \mathbb{S} \mid \llbracket e_2 \rrbracket s_3 = \text{true} \land s_4 = \llbracket f \rrbracket s_3\} \quad \text{(by Equation 5.1)}
\]

\[
= \llbracket \text{if } e_2 \text{ then } \cdot = f \rrbracket \quad \text{(by Definition 5.3.9)}
\]

\[\square\]

5.4 Hybrid Systems and their Denotational Semantics

This section defines a denotational semantics for MicroAcumen by translation to hybrid systems. The translation turns dynamics into
relations on the set state space $S$. These relations are used to define the interpretation of hybrid systems as evolutions, which are closed sets of time-states. Evolutions will be used to state a conjecture about the soundness of the operational semantics of MicroAcumen.

### 5.4.1 Hybrid Systems

A hybrid system is an abstract representation of how a state changes over time, according to either discrete or continuous dynamics. This definition of a hybrid system formalizes the informal notion used in Section 2.1. The definition is similar to the languages described in Section 2.1, but simpler. For example, unlike hybrid automata, hybrid systems (as per Definition 5.4.1) do not explicitly identify locations (parts of the flow map that correspond to different continuous dynamics), and do not explicitly specify transitions between locations. This makes hybrid systems close to the Acumen language, where locations do not need to be explicitly specified in the syntax.

This subsection defines a translation from MicroAcumen to hybrid systems.

**Definition 5.4.1 (Hybrid System).** A hybrid system $(F, J) \in \mathbb{H}$ on a state space $S = \mathbb{R}^N$ is a pair of binary relations $F, J \subseteq S \times S$. □

The *jump relation* $J$ represents the discrete dynamics, and the *flow relation* $F$ represents the continuous dynamics. These two relations are interpreted differently: $J$ is a simple mapping between states, $F$ specifies a vector field by associating states with their derivatives. For example:

**Example 5.4.2 (Sawtooth Wave Hybrid System).** The sawtooth wave system can be modeled as a hybrid system as follows:

$$(F, J) = (\{(x, 1) \mid x < 1\}, \{(x, 0) \mid x \geq 1\})$$

The flow and jump relations are sets of pairs of states. Next, we will see how to obtain these sets given a MicroAcumen model.
5.4.2 Models as Hybrid Systems

This section gives a semantics to a MicroAcumen model by translation into a hybrid system. It is obtained by translating each of the model’s modes, and combining the resulting hybrid systems. In the translation, discrete modes contribute to the jump relation and continuous modes to the flow relation.

Definition 5.4.3 (Interpretation of Models and Modes). A model \( m \in \mathbb{M}^\mu \) is interpreted as a hybrid system \( \mathcal{H}(m) \) using a function \( \mathcal{H}() : \mathbb{M}^\mu \rightarrow \mathbb{H} \) defined as follows:

\[
\mathcal{H}(m) \triangleq \mathcal{H}(\cdot) : \mathbb{M}^\mu \rightarrow \mathbb{H} \quad \mathcal{H}((q_i)_{i \in \mathbb{N}}) \triangleq (\bigcup_{i \in \mathbb{N}} F_i, \bigcup_{i \in \mathbb{N}} J_i) \text{ where } (F_i, J_i) = \mathcal{H}(q_i)
\]

A mode \( q \in \mathbb{Q} \) is interpreted as a hybrid system using a function \( \mathcal{H}(\cdot) : \mathbb{Q} \rightarrow \mathbb{H} \) defined as follows:

\[
\mathcal{H}(q) \triangleq \mathcal{H}(\cdot) : \mathbb{Q} \rightarrow \mathbb{H} \quad \mathcal{H}(\text{if } e \text{ then } \langle x_i' = e_i \rangle_{i \in \mathbb{N}}) \triangleq (\big\lceil \text{if } e \text{ then } = \langle e_i \rangle_{i \in \mathbb{N}} \big\rceil, \emptyset)
\]

Thus, the hybrid system of a model is the component-wise union of the relations obtained by translating its modes.

5.4.3 Valid Transitions

The key notion needed to define the denotational semantics is the transition relation that relates states of a system as it evolves over time. States may be related by flow (continuous) or jump (discrete) transitions. A flow transition relates two states through a differentiable map:

Definition 5.4.4 (Derivatives and Continuously Differentiable Maps). A map \( x : \mathbb{R} \rightarrow \mathbb{R}^n \) is continuously differentiable if the derivative
\[
\frac{dx(t)}{dt} \triangleq \lim_{h \to 0^+} \frac{x(t + h) - x(t)}{h}
\]
is defined and is continuous over \(\text{dom}(x)\). We write \(C^1(A, B)\) to denote the set of once continuously differentiable maps from \(A\) to \(B\), that is, for \(x \in C^1(A, B)\), the derivative \(\frac{dx(t)}{dt}\) is defined on \(A\). \(\square\)

Using this notion we can now define the valid transitions for a hybrid system.

**Definition 5.4.5** (Valid Transitions for a Hybrid System). The transition relation \(\cdot \xrightarrow{\delta} \cdot : S \times S \to B\) for the hybrid system \((F, J)\) and real \(\delta \geq 0\) is defined as follows

\[
(F, J), s_1 \xrightarrow{\delta} s_2
\]

\[
\delta > 0 \\
x \in C^1([0, \delta], S) \\
x(0) = s_1, \ x(\delta) = s_2 \\
\forall t \in [0, \delta], (x(t), x'(t)) \in F \\
\lim_{t \to 0^+} x'(t) \downarrow \land \lim_{t \to \delta^-} x'(t) \downarrow
\]

\[
(F, J), s_1 \xrightarrow{0} s_2 \\
(F, J), s_1 \xrightarrow{\delta} s_2
\]

\(\square\)

The two rules for valid transitions of a hybrid system correspond to its flows \(F\) and jumps \(J\). The rule for jumps says that a transition in zero time between two states is possible when these states are in the jump relation. The rule for flow says that a transition between two states in time \(\delta > 0\) is possible when these states are connected by a continuously differentiable function \(x(t)\), such that this function and its derivative \(x'(t)\) are in the flow relation, and the derivative is defined for all \(t \in [0 \ldots \delta]\). We call \(x(t)\) a solution of an Initial Value Problem (IVP) defined by the triple \((x'(t), x(0), \delta)\), where \(x'(t)\) is called the field, \(x(0)\) the initial condition and \(\delta\) the step.

**Notation 5.4.6.** For convenience, we omit the \(\delta\) when its value is not relevant. That is,
\((F, J), s_1 \rightarrow s_2\)

\[(F, J), s_1 \rightarrow s_2 \triangleq \exists \delta \geq 0. (F, J), s_1 \stackrel{\delta}{\rightarrow} s_2\]

5.4.4 Evolutions

With the definition of the valid transitions for a hybrid system in hand, we can now define the behavior of a hybrid system over time, called its evolution:

**Definition 5.4.7 (Evolution of a Hybrid System).** The evolution of a hybrid system \((F, J)\) is a map \([[(F, J)]] : \mathcal{P}_{\text{closed}}(S) \rightarrow \mathcal{P}_{\text{closed}}(T \times S)\) defined as the least fixpoint of a map \(F\). For an initial set of states \(S \subseteq S\), the map \(\mathcal{F} : \mathcal{P}_{\text{closed}}(T \times S) \rightarrow \mathcal{P}_{\text{closed}}(T \times S)\) is defined as follows:

\[
\mathcal{F}(I) = \text{closure}((\{0\} \times S) \cup \{(t + \delta, s_2) \mid (t, s_1) \in I \land (F, J), s_1 \stackrel{\delta}{\rightarrow} s_2\})
\]

An evolution for a hybrid system consists of sets of states that are reachable through the hybrid system at a given time. Any state in the evolution is reachable from the initial state through a chain of transitions (Definition 5.4.5).

5.5 Value Enclosures

Enclosures are computationally convenient subsets of the state space for a particular time step.

**Definition 5.5.1 (Value Enclosure).**

A *value enclosure* \(d \in \mathcal{D} \subset \mathcal{P}_{\text{closed}}(S)\) is a representation of a non-empty closed subset of \(S\) and comes with a decidable inclusion predicate \(d_1 \subseteq d_2\).
Value enclosures represent uncertainty about the state of the system. This uncertainty can come from measurement error, approximation error, or from considering sets of configurations of a system at the same time. Examples of concrete classes of enclosures are interval boxes $[\text{War}; \text{Sun}; \text{MB2}]$, parallelepipeds $[\text{Loh}; \text{Ned}; \text{Bar}]$, and affine forms $[\text{DFS}]$.

5.6 Timed Enclosures and their Denotational Semantics

Timed Enclosures and their Denotational Semantics

We now turn to the notion of timed enclosures, which will be used to define the operational semantics. The definition relies on:

**Notation 5.6.1.** For clarity, we point out that we will use the following convention:

$h$\hspace{1cm} A time step $h > 0$

Intuitively, a timed enclosure is a value enclosure associated with a subset of the time interval $[0..h]$. This interval is an over-approximation of a single time point between 0 and $h$.

**Definition 5.6.2 (Time Enclosure).** Time enclosures are represented by the following BNF:

$t \in \hat{T} \quad t \in \hat{T} ::= [0] \mid (0..h)$

Here $[0]$ represents the start time 0 and $(0..h)$ represents an unknown time between 0 and $h$.

**Definition 5.6.3 (Timed Enclosure).** Timed enclosures are represented by the following BNF:

$z \in \mathbb{Z} \quad z \in \mathbb{Z} ::= d@t \mid d_1/d_2@(0..h)$

The first case $d@t$ is an enclosure at a time represented by $t$ and the second case $d_1/d_2@(0..h)$ is a pair of enclosures: $d_1$ at time $(0..h]$ and $d_2$ at time $[h]$. 

□
The output of the operational semantics (Definition 5.8.2) will be a set of timed enclosures. A proof of soundness must establish that the time-states produced by the denotational semantics (Definition 5.4.7) are included in the elements of this set. For this purpose, we define the following interpretation of timed enclosures as sets of time-states.

**Definition 5.6.4 (Interpretation of Timed Enclosures).** A timed enclosure $z$ is interpreted by the function $\llbracket \cdot \rrbracket : Z \rightarrow \mathcal{P}(T \times S)$, defined as follows:

\[
\llbracket d@[0] \rrbracket = [0] \times d \\
\llbracket d@(0..h) \rrbracket = (0..h) \times d \\
\llbracket d_1/d_2 @(0..h) \rrbracket = ((0..h) \times d_1) \uplus ([h] \times d_2)
\]

A set of timed enclosures $Z$ is interpreted by the function $\llbracket \cdot \rrbracket : \mathcal{P}(Z) \rightarrow \mathcal{P}(T \times S)$, defined as follows:

\[
\llbracket Z \rrbracket = \bigcup \{\llbracket z \rrbracket \mid z \in Z\}
\]

5.7 Soundness Requirements for Validated Numerical Primitives

The operational semantics builds on three numerical primitives (auxiliary functions): active, reset and solveIVP. The primitives reset and solveIVP compute enclosures for the solutions of discrete and continuous equations respectively. The primitive active determines the set of modes whose guards may be satisfied by an enclosure.

This section specifies the conditions that these numerical primitives must meet to allow us to build a sound operational semantics. For each operation, we state a set of assumptions that are necessary to give a complete definition of the operational semantics and to prove its soundness with respect to the denotational semantics. The requirements are intended to be minimal. Their purpose is to avoid giving full specifications of the implementations of these primitives,
which can be complex, and which would still need to be consistent with the requirements given in this section.

5.7.1 Valid Transitions for a Model

This section defines the valid transitions for a model as a variant of the valid transitions for the associated hybrid system. The variant only requires the guard of a continuous mode to be satisfied by the initial state. This relaxation echoes the way that an implementation of solveIVP always solves initial-value problems over a given time interval.

We start by defining a shorthand. The transition relation for a model uses the function \textit{uncondition} to bypass the guard condition \( e \) of each continuous mode in the model.

\textbf{Definition 5.7.1 (Uncondition Mode).} The function \textit{uncondition} : \( \mathbb{Q} \rightarrow \mathbb{Q} \) is defined as follows:

\[
\text{uncondition}(q) \quad \text{uncondition}(\text{if } e \text{ then } c) \triangleq \text{if } \text{true} \text{ then } c
\]

The transition relation of a model mirrors the transition relation of a hybrid system (Definition 5.4.5), but is defined directly in terms of syntax (models and modes). This way, the dynamics \( c \) of every mode \( \text{if } e \text{ then } c \) in the model is available to \textit{uncondition}.

\textbf{Definition 5.7.2 (Valid Transitions for a Model).} The transition relation \( m, \cdot \xrightarrow{\delta} \cdot : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{B} \) for the model \( m \) and real \( \delta \geq 0 \) is defined as follows:

\[
m, s_1 \xrightarrow{\delta} s_2 \quad m, s_1 \xrightarrow{\delta} s_2 \iff \exists q \in m. q, s_1 \xrightarrow{\delta} s_2
\]

where the transition relation \( q, \cdot \xrightarrow{\delta} \cdot : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{B} \) for a mode \( q \) and a time step \( \delta \geq 0 \) is defined as follows:
One rule deals with continuous modes, the other with discrete ones. When the mode is discrete, the hybrid system is simply $\mathcal{H}(q)$. When the mode is continuous, the hybrid system is that of the unconditional version of $q$, that is, $\mathcal{H}(\text{uncondition}(q))$. Thus, the key difference between the transition relation of the mode $q$ and the transition relation of its hybrid system $\mathcal{H}(q)$, is that the former continues to flow beyond the guards of continuous modes.

**Notation 5.7.3.** For convenience, we omit the $\delta$ in Definition 5.7.2 when its value is not relevant. That is,

\[
\begin{array}{c}
q, s_1 \xrightarrow{\delta} s_2 \\
\mathcal{H}(q), s_1 \xrightarrow{0} s_2 \\
q, s_1 \xrightarrow{0} s_2
\end{array}
\]

The soundness requirements in Definition 5.7.6 include the property that each numerical primitive is inclusion isotone. This property formalizes the idea that these functions grow together with their inputs. Such functions, more generally called monotone (Definition B.0.2(1) on page 183), are common in definitions that involve sound approximations.

**Definition 5.7.4** (Inclusion Isotonicity for Partial Maps). A partial map $g : \mathcal{P}(A) \rightarrow \mathcal{P}(B)$ is inclusion *isotone* if it satisfies:
Corollary 5.7.5 (Inclusion Isotonicity for Total Maps). A total map $f : \mathcal{P}(A) \to \mathcal{P}(B)$ is inclusion isotone if it satisfies:

$$\text{iso}(f) \triangleq \forall a_1, a_2 \in \mathcal{P}(A). \quad a_1 \subseteq a_2 \land g(a_2) \downarrow \implies g(a_1) \downarrow \land g(a_1) \subseteq g(a_2)$$

5.7.2 Soundness Requirements

Having defined the valid transitions for a model and the notion of inclusion isotonicity for a function, we are now ready to state the soundness requirements on the validated numerical primitives.

The function active is used to determine the dynamics according to which the system should evolve, given an enclosure. It takes a model $m = \langle q_i \rangle_{i \in \mathbb{N}}$ and an enclosure $d$, and returns a set containing those modes $q_i = \text{if } e \text{ then } c_i$ for which $[e_i]s$ is true for some $s \in d$. We say that the modes in this set are activated by $d$.

The function reset is used to apply discrete assignments to an enclosure. It takes mode $q = \text{if } e \text{ then } c$ and an enclosure $d$, and returns an enclosure of all states that can be obtained by applying $[c]$ to a state $s \in d$.

The function solveIVP applies continuous assignments to an enclosure. It takes a mode $q$, an enclosure $d$ and a time step $h$ and returns a pair of enclosures for the corresponding IVP. For example, when $q \equiv \text{if } e \text{ then } \langle x_i' = e_i \rangle_{i \in \mathbb{N}}$, then we can view $(\langle e_i \rangle_{i \in \mathbb{N}}, d, [0..h])$ as an ODE IVP. The first enclosure returned by solveIVP must enclose the solution to the IVP over $[0..h)$, and we call this the range enclosure. It is used to guarantee that events that happen between time steps are not missed by the rigorous simulator. The second enclosure must enclose the solution at $[h]$, and we call this the end-time enclosure. This enclosure is used as the initial condition for IVPs solved in subsequent time steps.
enclosure as tight as possible. This is important for simulation, where error introduced at each step accumulates quickly.

**Definition 5.7.6 (Sound Numerical Primitives).** The sound numerical primitives are a triple (active, reset, solveIVP), such that:

1. **active**: $\mathcal{M}^\mu \times \mathbb{D} \rightarrow \mathcal{P}(m)$
   
   reset : $\mathbb{Q} \times \mathbb{D} \rightarrow \mathbb{D}$
   
   solveIVP : $\mathbb{Q} \times \mathbb{D} \times T \rightarrow \mathbb{D} \times \mathbb{D}$

2. **active.** Whenever the set of active modes $\text{active}(m,d)$ is defined for a given model $m$ and enclosure $d$, then for all $q \in m$:
   
   (a) $\text{iso}(\text{active}(m, \cdot))$
   
   (b) $\forall s \in d. \ [[\text{guard}(q)]s \downarrow$
   
   (c) $\exists s \in d. \ [[\text{guard}(q)]s = \text{true} \implies q \in \text{active}(m,d)$

3. **reset.** Whenever the reset function for guarded fields, $\text{reset}(q,d)$, is defined for a given mode $q$ and enclosure $d$, then:
   
   (a) $\text{iso}(\text{reset}(q, \cdot))$
   
   (b) $q \in \text{active}(m,d)$
   
   (c) $\forall s \in d. \ [[\text{guard}(q)]s = \text{true} \implies [[\text{field}(q)]s \in \text{reset}(q,d)

4. **solveIVP.** Whenever $\text{solveIVP}(q,d,h) = (d_{(0..h)},d_h)$ for a given mode $q$, enclosure $d$ and time step $h$, then
   
   (a) $\text{iso}(\text{solveIVP}(q, \cdot, h))$
   
   (b) $q \in \text{active}(m,d)$
   
   (c) i. $\forall s_0 \in d, \delta \in [0..h)$.
      
      $\{(\delta, s_1) \mid q, s_0 \xrightarrow{\delta} s_1 \} \subseteq [0..h) \times d_{(0..h)}$
      
      ii. $\forall s_1 \in d, \delta = h. \ q, s_1 \xrightarrow{h} s_2 \implies s_2 \in d_h$

Definition 5.7.6 formalizes the following properties of the safe numerical primitives:
Each numerical primitive is inclusion isotone with respect to the enclosure parameter.

The active function will be used to evaluate the guards of all modes in its model parameter. This is why this condition requires that if \( \text{active}(\langle \text{if } e_i \text{ then } c_i \rangle_{i \in \mathbb{N}}, d) \) is defined then \([e_i]s\) must also be defined for all \( s \in d \).

When a state in \( d \) satisfies the guard of a mode \( q \in m \) then that mode must be in the set \( \text{active}(m, d) \).

When \( \text{reset}(q, d) \) is defined then \( d \) must activate \( q \), that is, some state in \( d \) must satisfy the guard of \( q \).

When a state \( s \in d \) satisfies the guard of \( q \) then \( \text{reset}(q, d) \) encloses the image of \( s \) under the map \([\text{field}(q)]\).

When \( \text{solveIVP}(q, d, h) \) is defined then \( d \) must activate \( q \).

Any state reachable from \( d \) by the transition relation of the mode \( q \) (Definition 5.7.2) within time \( h \) is included in the enclosure \( d_{(0..h)} \).

Any state reachable from \( d \) by the transition relation of the mode \( q \) (Definition 5.7.2) in exactly time \( h \) is included in the enclosure \( d_h \).

5.8 Operational Semantics

This section presents an operational semantics for MicroAcumen, specified by rules that follow the syntax. Notable efforts that use this approach to specify a hybrid systems language include operational semantics for Simulink [BC], the Compositional Interchange Format [AvBR] and the Modeling Kernel Language [Bro]. Other related efforts focus on specific issues, such as choosing appropriate domains for time and the state space in an operational semantics [BF].
Unlike these examples, the goal of our operational semantics is to produce enclosures, which requires propagating error bounds provided by auxiliary functions, and combining these with additional error incurred by the operational semantics itself. A comprehensive treatment of such aspects of semantics has been given in the context of domain theory \cite{EP}, but without a clear connection to an existing implementation. Specifications of algorithms that take approximation into account, and for which there are implementations, can be found in literature on model checking \cite{ADI; Gao}, though these typically specify algorithms in pseudocode rather than by rules. The semantics presented in this chapter is in many ways similar to the operational semantics for TinyHCC developed in the thesis of Daisuke Ishii \cite{Ish}. A key difference is that our work is explicit about the way in which the discretization of time affects the precision of enclosures (the case $t = (0..h)$ in the Continuous rule of Figure 5.5). Further study of the relationship between these languages and their semantics is part of our future work.

The operational semantics is defined in terms of two functions: a main function that processes modes and a wrapper function that processes models. The definitions of these functions use validated numerical primitives that are specified in terms of their key properties (Definition 5.7.6), as well as the following notation:

**Notation 5.8.1 (Mode in the Context of a Model).** We will use the following notation to state that a mode $q$ belongs to a particular model $m$.

$$m \vdash q$$

$m \vdash q \triangleq (m, q)$ such that $q \in m$

## 5.8.1 Processing of Modes

The main function is defined by three rules: Blocked, Discrete, and Continuous. Each rule processes a mode $q_1$, a time enclosure $t$ and a value enclosure $d_1$, and produces a set of timed enclosures.
5.8. OPERATIONAL SEMANTICS

5.8.1.1 Blocked

Figure 5.1 illustrates the Blocked rule. This rule applies when the mode parameter \( q \) is not activated by the enclosure parameter \( d \), that is, when \( \text{active}(m, d) \) does not contain \( q \).

![Diagram](image)

Figure 5.1: Eliminating a Blocked Mode (Blocked)

The rule is as follows:

\[
\begin{align*}
\text{active}(m, d_1) & \downarrow \\
q_1 & \not\in \text{active}(m, d_1) \\
m & \vdash q_1, (\mathcal{E}, d_1) & \xrightarrow{h} & \emptyset \\
& \quad \text{(Blocked)}
\end{align*}
\]

It produces an empty set, and thus formalizes the idea that nothing is reachable from an enclosure through a mode that is not activated by that enclosure.

5.8.1.2 Discrete

Figure 5.2 illustrates the Discrete rule, which applies when the mode \( q_1 \) is activated and discrete. It uses the numerical primitive reset to compute a value enclosure \( d_2 \) (colored in red) that encloses the image of the input enclosure \( d_1 \).
CHAPTER 5. MATHEMATICAL SEMANTICS

Figure 5.2: Stepping Discretely (Discrete)

The rule is as follows:

\[
\begin{align*}
q_1 \equiv & \text{ if } e \text{ then } (x_i^+ = e_i)^i \in \mathbb{N} \\
& \text{active}(m, d_1) \downarrow \\
& q_1 \in \text{active}(m, d_1) \\
d_2 = & \text{reset}(q_1, d_1) \\
\{m \vdash q,(\ell, d_2) \xrightarrow{h} Z_q\} q \in m & \\
\{m \vdash q_1,(\ell, d_1) \xrightarrow{h} \{d_2@\ell\} \cup \bigcup_{q \in m} Z_q\} & \text{(Discrete)}
\end{align*}
\]

It produces a set of timed enclosures that consists of two parts. The first part is a timed enclosure with value enclosure \(d_2\) and time enclosure \(\ell\), which encloses the time-states reachable from \(d_1\) by a discrete transition. The time enclosure \(\ell\) is left unchanged because discrete transitions occur in zero time. The second part is a set of timed enclosures obtained by applying the rules once for each mode in \(m\) with \(d_2\) as the enclosure parameter, and with the input time \(\ell\) as the time parameter. This set encloses the states that are reachable from the timed enclosure in the first part.

5.8.1.3 Continuous

Figures 5.3 and 5.4 illustrate the Continuous rule, which applies when the mode is activated and continuous. It uses solveIVP to compute two value enclosures \(d_{(0..h)}\) and \(d_2\) (colored in red). They enclose the solution of the ODE IVP given by the field of the mode, the enclosure, and the time interval \([0..h]\). The enclosure \(d_{(0..h)}\) encloses
this solution over (0..h] and $d_2$ encloses the solution at the end-time $h$.

To produce its output, this rule selects the enclosure $d_3$ to be either $d_2$ (Figure 5.3) or $d_{(0..h)}$ (Figure 5.4), depending on whether the time enclosure is [0] or $\xi$ respectively. When the input time enclosure is (0..h), this means that the initial condition of the ODE IVP processed by solveIVP is a state at an unknown time in (0..h). Selecting $d_3$ to be $d_{(0..h)}$ corresponds to enclosing the union of all solutions, for each possible time in (0..h).

Figure 5.3: Stepping Continuously From the Start Time $\xi = [0]$ (Continuous)

Figure 5.4: Stepping Continuously From Unknown Time $\xi = (0..h)$ (Continuous)

The rule is as follows:
\[ q_1 \equiv \text{if } e \text{ then } \langle x'_i = e_i \rangle_{i \in \mathbb{N}} \]
\[ \text{active}(m, d_1) \downarrow \]
\[ q_1 \in \text{active}(m, d_1) \]
\[ (d_{(0..h)}, d_2) = \text{solveIVP}(q_1, d_1, h) \]
\[ m_1 = m \setminus \{q_1\} \]
\[ \{m \vdash q, ((0..h), d_{(0..h)}) \xrightarrow{h} Z_q\}_{q \in m_1} \]
\[ (t = [0] \land d_3 = d_2) \lor (t = (0..h) \land d_3 = d_{(0..h)}) \]
\[ m \vdash q_1, (t, d_1) \xrightarrow{h} \{d_{(0..h)}/d_3@([0..h]) \} \cup \bigcup_{q \in m_1} Z_q \]

The rule produces a set of timed enclosures that consists of two parts. The first part is a timed enclosure of \(d_{(0..h)}/d_3\) at the time interval \((0..h]\). The second part is a set of timed enclosures obtained by processing each mode in \(m \setminus \{q_1\}\), with \(d_{(0..h)}\) as the enclosure parameter, and with \((0..h)\) as the time parameter. In other words, this set encloses the states reachable from \(d_{(0..h)}\) at any time in \((0..h]\), through a mode different from \(q_1\).

A key difference between the Continuous and Discrete rules is that, in the Continuous case, the time when a change from \(q_1\) to another mode occurs is not known. The set \(Z_q\) must enclose the time-states that are reachable from those at which the system switches to the other mode \(q\). So, because the time when the switch happens is not known precisely (we can only bound it to be between 0 and \(h\)), the state is also not known precisely (we can only bound it to be in the range of the solution over \((0..h)\)). Because the value enclosure \(d_{(0..h)}\) encloses the range of the solution over \((0..h)\), this enclosure is used as a safe approximation of the true state at which the system switched to the other mode.

5.8.2 Processing of Models

The function \(m, d \xrightarrow{h} Z\) is defined by a rule that takes a model \(m\), an enclosure \(d\) and a time step \(h\), and returns a set of timed states \(Z\):

\[ \{m \vdash q, ([0], d) \xrightarrow{h} Z_q\}_{q \in m} \]
\[ m, d \xrightarrow{h} \{d@([0]) \} \cup \bigcup_{q \in m} Z_q \]
This set consists of the timed enclosure $d@[0]$ (the initial state of the system) and a union of sets of timed enclosures produced by the main function $m \vdash q, (\cdot, \cdot) \hookrightarrow \cdot$ (the states that are reachable from the initial state by each mode $q \in m$).

The execution of a model using enclosures can be seen as an abstract interpretation [CC] that makes it possible to answer questions about a model, such as whether a particular state is reachable, even though the exact problem is undecidable [HKPV].

We combine the definitions of the main function (that processes modes) and wrapper function (that processes models) into an operational semantics for MicroAcumen:

**Definition 5.8.2** (Operational Semantics). The operational semantics for a model is given by the function $\cdot \hookrightarrow \cdot : M^\mu \times D \times T \rightarrow \mathbb{Z}$ and the function $\cdot \hookrightarrow \cdot : Q \times \hat{T} \times D \times T \rightarrow \mathcal{P}_f(\mathbb{Z})$ defined in Figure 5.5. □
\[ m \vdash q, (\ell, d) \xrightarrow{h} Z \]

\[
\begin{array}{l}
\text{active}(m, d_1) \downarrow \\
q_1 \notin \text{active}(m, d_1) \\
\hline
m \vdash q_1, (\ell, d_1) \xrightarrow{h} \emptyset \\
\end{array}
\] (Blocked)

\[
q_1 \equiv \text{if } e \text{ then } \langle x_i^+ = e_i \rangle_{i \in \mathbb{N}} \\
\text{active}(m, d_1) \downarrow \\
q_1 \in \text{active}(m, d_1) \\
d_2 = \text{reset}(q_1, d_1) \\
\{ m \vdash q_1, (\ell, d_2) \xrightarrow{h} Z_q \}_{q \in m} \\
\]

\[
\begin{array}{l}
\rightarrow \\
\text{Discrete}
\end{array}
\]

\[
q_1 \equiv \text{if } e \text{ then } \langle x'_i = e_i \rangle_{i \in \mathbb{N}} \\
\text{active}(m, d_1) \downarrow \\
q_1 \in \text{active}(m, d_1) \\
(d_{(0..h)}, d_2) = \text{solveIVP}(q_1, d_1, h) \\
m_1 = m \setminus \{ q_1 \} \\
\{ m \vdash q, ((0..h), d_{(0..h)}) \xrightarrow{h} Z_q \}_{q \in m_1} \\
(\ell = [0] \land d_3 = d_2) \lor (\ell = (0..h) \land d_3 = d_{(0..h)}) \\
\hline
m \vdash q_1, (\ell, d_1) \xrightarrow{h} \{ d_{(0..h)}@0 \} \cup \bigcup_{q \in m_1} Z_q \\
\]

\[
\rightarrow \\
\text{Continuous}
\]

\[
\begin{array}{l}
m, d \xrightarrow{h} Z \\
\text{m, d} \xrightarrow{h} \{ [0], d \} \xrightarrow{h} Z_q \}_{q \in m} \\
\hline
m, d \xrightarrow{h} \{ d@[0] \} \cup \bigcup_{q \in m} Z_q \\
\end{array}
\]

Figure 5.5: An Operational Semantics for MicroAcumen

5.8.3 Example Executions

Figure 5.6 illustrates an example execution of the sawtooth wave model (Example 4.2.3) using the operational semantics, with model \( m = \langle q_1, q_2 \rangle = \langle \text{if } x < 1 \text{ then } (x' = 1), \text{if } x \geq 1 \text{ then } (x^+ = 0) \rangle \), initial state \( x = [\mathbb{1}/4 .. \mathbb{3}/4] \) and step \( h = \mathbb{1}/2 \). The resulting set of timed enclosures is as follows:

\[
\{ [\mathbb{1}/4 .. \mathbb{3}/4]@[0], [\mathbb{1}/2 .. \mathbb{5}/4]/[1 .. \mathbb{5}/4]@[0 .. \mathbb{1}/2], [0]@[0 .. \mathbb{1}/2], [0 .. \mathbb{1}/2]/[0 .. \mathbb{1}/2]@[0 .. \mathbb{1}/2] \}
\]
Each step of the execution corresponds to the application of a rule, which produces a set of enclosures. This set is either empty (the Blocked rule) or consists of a singleton combined with sets obtained by further applications of rules. Execution begins with the wrapper function (Section 5.8.2), and continues with applications of the main function (Section 5.8.1).

Some executions of the operational semantics do not terminate. Figure 5.7 illustrates such a case, where the sawtooth model is executed
with the same initial state but with a larger step $h = 2$. This execution contains a cycle that begins when the Discrete rule applies for the first time, which results in the enclosure $[0]@([0..2])$. The cycle arises when the enclosure produced by the Continuous rule again activates the discrete mode $q_2$.

\[
\text{Figure 5.7: Execution of Sawtooth Wave Model With Step 2}
\]
5.9 On the Soundness of Operational Semantics

In this section I state the basic connection between the operational (Definition 5.8.2) and denotational semantics (Definition 5.4.7). The denotational semantics characterizes the reachable states of a system for all times. Simulation traditionally focuses on behavior over a finite time interval. A statement of a soundness property for the operational semantics must therefore focus on the set of time-states contained in the denotational semantics, restricted to such an interval:

**Definition 5.9.1** (Time-Restric tion). The restriction of the set \( I \subseteq T \times S \) to the interval \( T \subseteq T \) is a function \( (\cdot)|_T : T \times S \rightarrow T \times S \), defined as follows:

\[
I|_T \triangleq I \cap (T \times S)
\]

Our conjecture of the soundness of the operational semantics with respect to the denotational semantics over a single time step \([0..h]\) is as follows:

**Conjecture 5.9.2** (Soundness). Given a MicroAcumen model \( m \) with state space \( S = \mathbb{R}^n \) a time \( h > 0 \) and an enclosure \( d \), whenever the operational semantics produces an enclosure \( Z \), then the denotational semantics of \( Z \), namely \( [Z] \), contains the denotational semantics of \( m \) applied to \( d \), namely \( ([\mathcal{H}(m)]d)\), on the interval \([0..h]\):

\[
\forall m \in \mathbb{M}, h > 0, d \in \mathbb{D}, m \xrightarrow{h} Z. ([\mathcal{H}(m)]d)_{[0..h]} \subseteq [Z]
\]
Part III

Applications
Chapter 6

Implementation

In this chapter we review the implementation of the Acumen language, which was developed jointly by Halmstad University and Rice University. The rest of this chapter is structured as follows. First, we review the two semantics that the implementation supports. Second, we review the user interfaces into which the modeling and simulation environment is integrated. Third, we review the organization of the code base and its key components. Last, we review two approaches to testing that are used in Acumen’s automated test suite.

6.1 Introduction

The implementation was partly developed before the theory presented in Chapters 4 and 5, and partly in parallel with it. There were three main reasons for doing so. First, the implementation allowed us to develop an understanding of the characteristics of both the traditional and validated numerics that hybrid systems simulators can build upon. Second, the implementation served as a testbed for experimenting with the syntax and user interface. This led us to a small but flexible syntax that translates smoothly into the core syntax in terms of which the formal semantics is defined, and a user interface that supports features such as 3D visualization, which are vital for productivity when simulating large systems. Third, the implementation
allowed us to develop a collection of more than 500 example models of hybrid systems\(^1\). This collection contains both small models that serve as benchmarks for specific semantic features (e.g. variants of a bouncing ball), and large models that test the expressivity of the language and performance of the semantics (e.g. the model that Chapter 7 describes). Having access to these examples helped us identify the core features of a hybrid systems formalism, and provided us with ongoing feedback on design choices.

As a result of developing the theory in parallel with the implementation, there are places where they are not in full correspondence. There are two main differences. First, the implementation does not perform the translation into the core syntax before simulation, but instead performs an analogous transformation before each simulation step. Second, the implementation supports some syntactic and semantic features that have not yet been formalized in the theoretical framework presented in the thesis. This includes support for sub-models, and support for simulating Zeno systems \([KTB]^{+}\).

The simulation results presented in this thesis can be reproduced using the 2016.11.17 release of Acumen\(^2\), which is freely available, as open source software. The models are included in the distribution under examples/05_Language_Research/24_ISO26262 and examples/05_Language_Research/26_Convergence.

### 6.2 Traditional and Rigorous Simulation Semantics

Two kinds of interpreters are included in the implementation. The first kind implements the traditional semantics, which uses floating-point numbers to represent real numbers and is based on traditional numerical ODE IVP solvers, such as 4th order Runge-Kutta and arbitrary order Taylor Series integrators. The traditional semantics is the most complete and most widely used semantics, due to its predictable

\(^1\)These models are available in the distribution \([aR]\), under examples.

\(^2\)The “2015 Enclosure” option must be selected in the Semantics menu.
performance and support for features such as 3D visualization. It has played a crucial role in allowing us to explore the language design space and to converge on an expressive, minimal syntax for modeling hybrid systems and on a formal (exact) semantics for solutions. Among other design choices, it allowed us to make an early decision to support the notion of super-dense time, first introduced in the verification literature [MMP], and later advocated by Liu, Matsikoudis and Lee [LML]. This notion facilitates modeling of discrete subsystems with multiple idealized (zero time) and externally observable internal transitions.

The second kind of semantics supported by the implementation is the enclosure semantics, which allows rigorous simulation using a subset of the full Acumen language. This semantics is intended to produce rigorous over-approximations (guaranteed upper and lower bounds) for all simulations [DEB+; KTB+]. A model of the subset of Acumen supported by the enclosure simulation semantics is formalized in Chapter 5. The operational semantics in Figure 5.5 models the behavior of this interpreter.

Acumen is both a language and an integrated modeling environment that includes a code editor, interpreters, and facilities for visualization. The implementation provides two main ways to interact with an Acumen model.

6.3 Command-Line Interface (CLI)

The Command Line Interface (CLI) is a standard textual interface for executing models in a terminal or from a script. Using the CLI it is possible to trace the states of a model during a simulation, measure the time it takes to execute a model, and record reference output files for the automated tests, among other things. To see all the available commands, the --full-help flag must be passed when starting Acumen in a terminal:

```
java -jar acumen.jar --full-help
```
The Graphical User Interface (GUI) is a ‘desktop’ application. It provides a model authoring environment with a syntax highlighting code editor [RSy]. When a model is executed, plots and 3D visualizations [Zen; Xu] are generated and displayed in real-time. Figure 6.1 shows the main features of the GUI. The code editor provides syntax highlighting, code completion, and jumps to the location of errors that occur during simulation (1). The console displays status and error messages (2). The Plot view displays the state of the system over time, as one band for each variable (3). The Table view displays the state of the system as one column for each variable and one row for each simulation step (4). The 3D view displays three-dimensional models embedded in the source code (5).
By default, Acumen produces a standardized plot (3) for a simulation. The GUI allows the user to zoom in and point to parts of the enclosure to see the value of a variable at a given time. To read the plot, the reader should be aware of the following conventions:

- Each white rectangle represents the plot for a particular variable.
- The vertical scale is normalized to fit the minimum and maximum values for each plot during the simulation time.
- By default, the simulation is for 10 seconds. The horizontal axis represents time, with time 0 at the left end and time 10 at the right end.

Further details can be found in the reference manual, which is available under the Help menu of the GUI.

6.5 Organization of the Code Base

This section sketches the evolution of the code base, and describes key components of the implementation, with detailed overviews of the enclosure interpreter and its key dependencies.
Figure 6.2 shows the evolution of the code base in terms of source code lines checked-in to the Acumen source code repository [Rep], between July 2012 and October 2016. The figure shows both the size of the implementation Scala and Java code, and Acumen code. The Acumen code comprises both example models, and models used by the automated tests (Section 6.9). The size of the code base jumped on several occasions. In the second half of 2012, the back-end of the GUI was re-written in an asynchronous style to keep it responsive during simulation. In the middle of 2013, the first prototype of the enclosure interpreter based on a Picard integrator (Section 6.8.3) was merged into the code base. An alignment of the traditional and enclosure interpreters was performed in the beginning of 2015. In the beginning of 2016, a validated differential equation solver based on Taylor series methods (Section 6.8.2) was added to the code base.

Figure 6.3 summarizes the key components of the Acumen code base. This diagram uses the following conventions: rectangles are concrete types and functions, rounded rectangles represent abstract types, arrows represent function calls or actor messages, dashed arrows represent dispatch, and lines represent other kinds of association. The following subsections walk through the different components “top-down”, that is, roughly in the order in which they depend on one another during a simulation. Descriptions of components whose behavior is mostly orthogonal to the concept of rigorous simulation have been left out.

6.6 Interpreters

Simulation is driven by a controller that repeatedly calls the step() function of the Interpreter component. Each time this happens, the controller passes a Store (an abstract type concertized in each inter-

---

3-The implementation is mostly written in Scala [Oa], a programming language that runs on the Java [AGH] virtual machine. Scala is both functional and object-oriented. This means that there are many ways to reuse code, which has caused many components to span many files, types, and functions. Figure 6.3 is thus largely a conceptual diagram.
6.6. INTERPRETERS

Figure 6.3: Implementation Components and Relationships
Three interpreters are currently supported. The Optimized and Reference interpreters both implement the traditional semantics, which uses standard numerical methods and floating-point numbers to do traditional simulation. The enclosure interpreter implements the enclosure semantics, which uses validated numerical methods and intervals for rigorous simulation.

6.6.1 Traditional Interpreters

The traditional interpreters simulate a model by taking discrete and continuous steps until the end-time is reached. Figure 6.4 shows how the steps are interleaved. First, a step is taken to initialize the store with the values from the initially section. Then discrete steps are taken until a fixpoint is reached, that is, until no discrete assignments are active. If the end-time has also been reached, the simulation finishes. Otherwise, a continuous step is taken, time is incremented, and the loop starts over by taking another discrete step. The resulting simulation trace is a sequence of stores for each time step of the simulation. Though simulation tools (including Acumen) typically visualize such traces by connecting the values by lines, these lines are not part of the output of the simulator. Figure 6.5 shows an example output of a traditional simulator for a single step of a simulation. The blue curve is the exact solution, and the red squares the approximation.

The Optimized interpreter is an imperative implementation based
6.6. INTERPRETERS

on a mutable Store type. This is the default interpreter. The reference interpreter is a purely functional implementation based on an immutable Store type. Its purpose is to make experimentation with new semantic features simpler, and to act as a reference against which the Optimized interpreter is tested.

Though traditional simulation tends to approximate solutions well, there are situations that these methods are not able to cope with. Figure 6.6 shows an example of such a situation, where the exact solution briefly changes from one mode \(q_1\) to another \(q_2\) over the course of a single step. The traditional simulator only observes the solution at the end-points of the step, and will therefore not observe the mode change.

Table 6.1 lists parameters supported by the traditional interpreters.
CHAPTER 6. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>endTime</td>
<td>End-time</td>
<td>10</td>
</tr>
<tr>
<td>timeStep</td>
<td>Fixed time step</td>
<td>$2^{-6}$</td>
</tr>
<tr>
<td>method</td>
<td>Integrator, one of:</td>
<td>&quot;RungeKutta&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;EulerForward&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;RungeKutta&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Taylor&quot;</td>
<td></td>
</tr>
<tr>
<td>orderOfIntegration</td>
<td>Taylor series truncation</td>
<td>4</td>
</tr>
<tr>
<td>order</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Simulator Parameters Available in Traditional Interpreters.

6.6.2 Enclosure Interpreter

The enclosure interpreter simulates a model by computing an enclosure of all states that are reachable from the current store. When such an enclosure has been computed for the current time step, the enclosure at the end of the step is used as the store for the next time step.

Figure 6.7 shows an example of an enclosure for the exact solution. This is different from those produced by the traditional interpreters in several ways. First, the enclosure for a given step gives bounds for the behavior of the system over the whole step, and not just at the end-points. Thus, while the traditional interpreters may miss an event if it happens in between two steps, the enclosure interpreter is guaranteed to take into account all behaviors expressed by a model. Second, because enclosures are over-approximations, the trace of the enclosure interpreter contains all traces that are possible by starting from the initial store.

Table 6.2 lists parameters supported by the traditional interpreters.
Table 6.2: Simulator Parameters Available in Enclosure Interpreter.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>endTime</td>
<td>End-time</td>
<td>10</td>
</tr>
<tr>
<td>timeStep</td>
<td>Fixed time step</td>
<td>$2^{-6}$</td>
</tr>
<tr>
<td>minTimeStep</td>
<td>Minimum adaptive time step</td>
<td>$2^{-6}$</td>
</tr>
<tr>
<td>maxTimeStep</td>
<td>Maximum adaptive time step</td>
<td>$2^{-6}$</td>
</tr>
<tr>
<td>method</td>
<td>Integrator, one of:</td>
<td>&quot;Taylor&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Picard&quot;, &quot;Taylor&quot;</td>
<td></td>
</tr>
<tr>
<td>orderOfIntegration</td>
<td>Taylor series truncation order</td>
<td>4</td>
</tr>
<tr>
<td>dynSet</td>
<td>Enclosure type, one of:</td>
<td>&quot;Cuboid&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Cuboid&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;IntervalBox&quot;</td>
<td></td>
</tr>
</tbody>
</table>

6.6.3 Active Statements

All Acumen interpreters must implement the parallel semantics of Acumen’s statement composition operator “,”. This is achieved as follows: given the current store, the Abstract Syntax Tree (AST) of the model is traversed to collect the set of active equations. These equations are then solved to obtain the next store. Compared to an interpreter for an imperative language, this type of interpretation is more similar to a program transformation. This approach is convenient in the context of equation solving because it turns the model into a set of mathematical equations, for which solvers are readily
available.

6.7 Arithmetics

Underlying the numerical code in the implementation is an abstract type Real, which collects all operations that algorithms need to perform on “numeric” values. Algorithms expressed in terms of the Real type then become generic, making it possible to run them with inputs of a given concrete instance of Real, to produce outputs of the same type. This is the basis for considerable code reuse between the different interpreters in the implementation, both among the traditional interpreters, and between the traditional and enclosure interpreters. For example, the Runge-Kutta and Taylor solvers can be run with both Double and Interval representations of state variables, and indeed any other instance of Real. Table 6.3 lists operations supported by the Real type.

6.7.1 Double

The Real operations of the Double type forward to those of the Scala double-precision floating-point number type.

6.7.2 Interval

The Real operations of the Interval type forward to those of the JInterval library [NZ], configured to use outward-rounded operations, rational end-points and IEEE 754R Binary64 [IEE] format for internal calculations.

6.7.3 TDif

The TDif type is used to represent Taylor series, and the Real operations for this type implements a Taylor series arithmetic [Tuc]. For example, for the Real operations add (+) and mul (*), given two Taylor series $x$ and $y$, this arithmetic uses rules such as the following to
### 6.7. ARTIHMETICS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0 Real</td>
</tr>
<tr>
<td>one</td>
<td>1 Real</td>
</tr>
<tr>
<td>abs</td>
<td>(</td>
</tr>
<tr>
<td>add</td>
<td>(\cdot + \cdot) Real \times Real \rightarrow Real</td>
</tr>
<tr>
<td>sub</td>
<td>(\cdot - \cdot) Real \times Real \rightarrow Real</td>
</tr>
<tr>
<td>mul</td>
<td>(\cdot \times\cdot) Real \times Real \rightarrow Real</td>
</tr>
<tr>
<td>neg</td>
<td>(-\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>div</td>
<td>(\cdot / \cdot) Real \times Real \rightarrow Real</td>
</tr>
<tr>
<td>pow</td>
<td>((\cdot)^{\cdot}) Real \times Real \rightarrow Real</td>
</tr>
<tr>
<td>sin</td>
<td>sin(\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>cos</td>
<td>cos(\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>tan</td>
<td>tan(\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>asin</td>
<td>sin^{-1}(\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>acos</td>
<td>cos^{-1}(\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>atan</td>
<td>tan^{-1}(\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>exp</td>
<td>(e^{\cdot}) Real \times Real \rightarrow Real</td>
</tr>
<tr>
<td>log</td>
<td>log(\cdot) Real \rightarrow Real</td>
</tr>
<tr>
<td>square</td>
<td>((\cdot)^{2}) Real \rightarrow Real</td>
</tr>
<tr>
<td>sqrt</td>
<td>(\sqrt{\cdot}) Real \rightarrow Real</td>
</tr>
<tr>
<td>fromInt</td>
<td>Int \rightarrow Real</td>
</tr>
<tr>
<td>fromDouble</td>
<td>Double \rightarrow Real</td>
</tr>
<tr>
<td>toInt</td>
<td>Real \rightarrow Int</td>
</tr>
<tr>
<td>toDouble</td>
<td>Real \rightarrow Double</td>
</tr>
<tr>
<td>isZero</td>
<td>Real \rightarrow Boolean</td>
</tr>
<tr>
<td>isValidInt</td>
<td>Real \rightarrow Boolean</td>
</tr>
<tr>
<td>isValidDouble</td>
<td>Real \rightarrow Boolean</td>
</tr>
</tbody>
</table>

Table 6.3: Operations of the Real Type
compute the coefficient of the \( k \)\:th term of the combined series:

\[
(x + y)_k = x_k + y_k \\
(x \ast y)_k = \sum_{i=0}^{k} x_i y_{k-i}
\]

6.7.4 FDif

The FDif type is used to represent gradients. In the context of Acumen, a gradient is the value of a variable together with its partial derivatives, with respect to all other variables in a model. The Real operations for this type implement a gradient arithmetic \([Ra1]\). For example, for the Real operations add (+) and mul (\( \ast \)), given two gradients \( x \) and \( y \), this arithmetic uses rules such as the following to compute the components of the combined gradient:

\[
(x + y)_k = x_k + y_k \\
(x \ast y)_k = \begin{cases} x_0 y_0 & \text{if } k = 0 \\ x_0 y_k + y_0 x_k & \text{otherwise} \end{cases}
\]

Here, the 0:th component contains the value of the variable itself.

6.8 Solvers

The traditional interpreters can use one of two integrators to solve differential equations: Runge-Kutta and Taylor. Both of them are implemented in terms of the Real numeric type (Section 6.7), and can thus be evaluated using any of its subtypes.

6.8.1 Runge-Kutta

The Runge-Kutta solver implements a fourth-order Runge-Kutta integrator that behaves well for many models, and is computationally efficient. It is the default solver for the traditional interpreters.
6.8.2 Taylor

The Taylor solver implements a (truncated) Taylor series integrator. Because the Taylor series can be chosen to be of any order, this integrator can be used to solve highly non-linear equations, such as those used to describe the motion of a double pendulum, whose solutions may become unstable with the default integrator.

The enclosure interpreter makes use of two rigorous ODE IVP solvers and an interval box contractor [CL].

6.8.3 Picard

The Picard solver implements a first-order polynomial function interval Picard integrator [KDFT]. It is based on a variant of the Picard iterator:

\[
    x(t) = x(0) + \int_{s=0}^{t} f(s, x(s)) \, ds
\]  

(6.1)

where the solution \(x(t)\) at each iteration is represented by a function interval [DFKT], that is, a pair of functions (the upper and lower bounds). Though the current implementation of this solver is limited by its first-order polynomial enclosure type, it can (in principle) be used to solve any ODE IVP whose field is Lipschitz continuous. Notably, this includes ODEs, whose fields are not differentiable.

6.8.4 Lohner

The Lohner solver implements a validated Taylor series integrator [Loh; Ned; Bar]. This solver is suitable for solving non-linear ODE IVPs and uses the Cuboid enclosure (DynSet) type as its input and output. A Cuboid represents the continuous state of a model as a parallelepiped, that is, as follows:

\[
    \text{midpoint} + \text{linearTransform} \times \text{width} + \text{error}
\]  

(6.2)

where \text{midpoint}, \text{width}, and \text{error} are Real vectors and \text{linearTransform} is a Real matrix. When the solver is initialized, the initial
condition is represented in centered form as \textit{midpoint} and \textit{width}. The \textit{linearTransform} is initialized to an identity matrix, and \textit{error} is initialized to a null vector. Basically, the solver integrates an ODE by updating the \textit{linearTransform} matrix and multiplying it by the Jacobian of the solution approximation at each step, to capture the effect that the ODE exerts on the initial condition.

Both the TDif and FDif types are parameterized by another Real type that represents their coefficients (TDif) and components (FDif). This ability to nest Real types is crucial for the implementation of this solver, where the computation of the Jacobian of the solution map involves a call to the Taylor solver that combines Interval, TDif and FDif arithmetics.

\subsection{6.8.5 Contractor}

The Contractor implements the HC4Revise interval box contractor. Let \( \mathbb{IR}^n \) be the set of interval boxes, that is, the set of \( n \)-dimensional real interval vectors. Given an interval box \( x \in \mathbb{IR}^n \) and a constraint \( K : \mathbb{R}^n \rightarrow \mathbb{B} \) the contractor is a function \( C : \mathbb{IR}^n \rightarrow \mathbb{IR}^n \) that satisfies the following two properties:

\begin{itemize}
  \item \( C(x) \subseteq x \),
  \item \( C(x) \cap z = x \cap z \),
\end{itemize}

where \( z \) is the set \( \{ v \in \mathbb{R}^n \mid K(v) \} \) associated with \( K \). In the implementation, the interval box is a value of the Box type and the constraint a set of Boolean expressions.

\subsection{6.8.6 Plotter and Table}

The Plotter and Table components provide direct views of the data produced by the interpreter. The Table displays data for each simulation step as a row whose columns are variable names. Value produced by the enclosure interpreter are displayed either as sets of values (for string or Boolean variables) or as intervals (for real variables). The
Plotter component displays simulation steps differently depending on the interpreter that produced them. Data for steps produced by the traditional interpreters are plotted as points connected by lines. Data for steps produced by the enclosure interpreter are plotted in one of two ways. Discrete (string and Boolean) variables are plotted as sets of points. Points at consecutive time steps that correspond to the same discrete value are connected by lines. Real variables are plotted as intervals.

6.8.7 3D View

The 3D View component [Zen] visualizes 3D models embedded in an Acumen model. By visualizing the simulation result in a form that closely matches the modeled system, it provides a way to quickly understand overall system behavior. For example, the 3D visualization proved invaluable in debugging the model of the automotive case study (Chapter 7), where a large number of variables interact to produce events such as a collision.

3D models are built from built-in geometric primitives, or from 3D meshes created in a third-party tool such as Blender [Ble]. The 3D visualization is animated by using variables from the dynamical model in the 3D model. The animation can be displayed in real time during a simulation, and the user can interact with the simulation using an input device [Xu].

6.9 Testing

Given the complexity of Acumen, and its objective to be rigorous, automated testing has been an important aspect of its development. The automated tests can be divided into two categories.

The first kind are unit tests, which check individual test cases. The basic unit tests check the correct behavior of a function for a particular input. Such tests are useful to detect regressions, that is, changes to the code base that break existing functionality. For example, when a bug has been fixed, a unit test marked with an
identifier for that bug is added to the code base to make the test suite fail if a subsequent change re-introduces the bug. Acumen’s test suite also includes regression tests that check the behavior of the interpreters on example models from the distribution. The output is compared to previously generated reference output files. These tests are useful to detect accidental, semantics-altering changes.

The second kind are generator-driven property-based tests [CH, TGP, Sca]. In such tests, the property is specified as a Boolean expression in terms of the tested function. The the test cases are then generated based on the types of the expression’s free variables. Property-based tests are used extensively in the implementation to check basic properties of the arithmetics. For example, the implementation contains tests of trigonometric identities such as $-1 \leq x \leq 1 \implies x = \sin(\sin^{-1}(x))$. Equalities do not generally hold when working with approximations, so these tests typically use an approximate equality operator. This introduces the problem of choosing how close two values must be to be considered equal. This problem can be avoided when testing operations based on validated numerics, by replacing equality with subset inclusion or intersection. For example, the above equation can be replaced by: $x \subseteq \sin(\sin^{-1}(x))$. This weakens the test – an implementation of $\sin$ that returns the constant $[-1..1]$ satisfies the new property – but combined with other tests these properties have still helped us uncover bugs and increase confidence in the implementation.

The soundness criteria given in Definition 5.7.6 are further examples of properties. Though such tests have not yet been added to the implementation, they could serve as the basis of a test suite for external implementations of the validated numerical primitives.
Chapter 7

An Automotive Collision Case Study

Formal methods can provide guaranteed answers to questions in the presence of uncertainty. This makes them interesting in the design of safety-critical systems where unaccounted error, be it from modeling or from computation, can have dire consequences. Verification of safety-critical systems typically aims to establish that a certain safety property holds. In the case of air traffic control, such a property could be that the trajectories of two airplanes remain at a safe distance from one another, that an evasive maneuver is enacted if the distance falls below some threshold, and that the maneuver successfully maintains the safe distance. Due to the controlled nature of avionics and the

---

1This chapter consolidates and extends several results presented at FISITA 2014 [MPD⁺] and ICES 2015 [DEB⁺]. The first version investigated the feasibility of using rigorous simulators in the automotive domain, using a simple one-dimensional model expressed as a single hybrid automaton. The second paper reported on a simulator capable of executing two-dimensional scenarios, with collision and sensor models based on rectangular vehicles and a braking controller based on time-to-collision. The present version adds a significantly improved model, including a combination of multiple braking criteria based on recommendations from the automotive industry, and a more precise model of the sensor range (trapezoid). Sections 7.3 to 7.6 and were added to explain the improved model in detail.
high risks involved in it, significant effort has gone into the verification of air traffic control systems [PC JM]. With the recent introduction of autonomy features in road vehicles, formal verification of these control systems is becoming increasingly important. Recent work on the verification of such systems includes a safety proof of a distributed automotive traffic control algorithm [LPN] and computing the probability of a crash [A+1]. In this chapter we will look at how rigorous simulation can be used to perform a related task – to compute bounds on a quantity estimated by a model. The result is used in a safety analysis to assess the danger of a given simulated scenario.

7.1 Hazard Analysis And Risk Assessment and The ISO 26262 Standard

To address the need for automotive system safety guidelines and regulations, the industry has developed the ISO 26262 standard [ISO]. It formalizes safety in (passenger) road vehicles, including requirements for the design of control systems and notions, based on which the safety of such systems can be evaluated. To comply with the standard, a system must undergo a Hazard Analysis and Risk Assessment (HARA).

In risk assessment terminology, a hazard is a possible source of harm and risk is the probability of this harm. Risk is further broken down as a combination of severity, exposure, and likelihood. Severity is a measure of the injury that follows from harm. Exposure is the expected frequency of the conditions under which the injury can occur. Likelihood is the probability that the harm will occur given the exposure.

Traffic hazards include sudden lane changes, slippery roads, and intersections. The harm associated with these hazards is injury due to a collision. We can break down the associated risk into the seriousness of the injury (severity), the proportion of time that drivers spend crossing intersections (exposure) and the probability that a collision will take place when crossing the intersection (likelihood).

Mirroring standards from other safety-critical domains, ISO 26262
quantifies risk in terms of a \textit{safety integrity level} [Red]. The \textbf{Automotive Safety Integrity Level (ASIL)} breaks down the risk associated with a hazard into three components: severity (S), exposure (E) and controllability (C). Compared to the likelihood used in the standard definition of risk, controllability measures the ability to affect the severity of a hazard (e.g. by steering or braking), given the exposure. Each of these components is quantized, with four levels (0 to 3) for severity and controllability, and five levels (0 to 4) for exposure. The ASIL obtained by multiplying S, E, and C can help system developers determine which hazards to focus on during design and testing.

During \textbf{HARA}, each component of the \textbf{ASIL} must be estimated. Tools have been developed to support the ISO 26262 compliance process, including the task of determining ASIL [Ris; med; SOX; ALM]. These tools help users manage the complexity of the standard by organizing requirements and data, but rely on existing estimates of S, E, and C. In this case study, we evaluate the use of rigorous simulation for estimating the severity level of a collision hazard.

\section*{7.2 A Collision Avoidance Scenario}

Every year millions of accidents happen at intersections [Cho]. This has made intersections a focus area of automotive safety over the past two decades [BTR; CB; LFBZ; MCD+]. One of the proposed approaches is that of an \textbf{ICAS}. As opposed to a single-vehicle safety feature such as an \textbf{Advanced Emergency Braking System (AEBS)} an \textbf{ICAS} integrates controllers and sensors on multiple vehicles with road-side infrastructure to coordinate a safe passage through an intersection. By fusing data from the vehicles with information about the intersection’s geometry and road conditions, the \textbf{ICAS} is better-positioned to make good decisions than each individual vehicle.

In this case study we analyze an \textbf{ICAS} by simulating a scenario where two vehicles – a truck (Vehicle 1) and a car (Vehicle 2) – approach an intersection. The truck is equipped with an AEBS that should be triggered by the \textbf{ICAS} to avoid a collision. The scenario is based on an EU regulation [EU] for road vehicle \textbf{AEBS}. In the test
The truck, situated 120 m behind the car, moves toward it at a speed of $80 \pm 2$ km/h. In the event of a collision, a coefficient of restitution of 0.5 is assumed. To pass the test procedure, the AEBS must prevent a collision from happening. A variation on this scenario served as a common milestone for the different parties involved in the Next Generation Test Methods for Active Safety Functions (NG-TEST) project [WBWB$^+$], a joint industrial-academic project that gave rise to this case study.

Figure 7.1 illustrates the NG-TEST scenario, where the car (red) goes through an intersection to enter the road that the truck (gray and blue) is on. The truck’s AEBS sensor detects the car when the trapezoid that bounds its field of view (yellow) intersects with the car’s rectangular bounding box. This may occur at any point during the simulation, including during the turn. The trapezoid is 50 m in length (shortened to 10 m in Figure 7.1 for illustration), 1 m wide near the sensor, and 2 m wide furthest away from the truck.

The system should avoid a collision by switching the control mode of the two vehicles, based on the time that is left until a collision will happen (Time-To-Collision (TTC)) and on the distances required for the driver to react (Critical Warning Distance (CWD)) and for the brakes to be effective (Critical Braking Distance (CBD)). The following sections describe the Acumen model that was developed in this case study. Simulation results are presented in Section 7.6.
7.2. A COLLISION AVOIDANCE SCENARIO

Figure 7.2: A Hierarchical Hybrid Automaton Representation of the ICAS.
7.3 Vehicle and Collision Models

Unlike sensor-based collision avoidance systems such as Automatic Emergency Braking (CEB), where a single vehicle acts autonomously to avoid a collision, ICA systems are a cooperative approach to vehicle safety. Infrastructure located in the intersection coordinates two or more vehicles based on position and sensor data from the vehicles, as well as knowledge about the geometry of the intersection. Section 7.4 describes the Acumen model of the collision detection and avoidance system, including the two vehicles. Figure 7.2 gives an overview of the model as a partial, nested hybrid automaton, where the controllers for the two vehicles are nested inside an automaton that handles the collision.

7.4 Vehicle Dynamics (Pre-Collision)

Each vehicle is modeled as a two-dimensional point mass, as well as a (possibly rotated) bounding rectangle. The controller for each vehicle is modeled as a match statement that roughly corresponds to a hybrid automaton. The modes of this automaton correspond to the cases of the match, that is, the values of two variables: state1 for the truck; state2 for the car. Figure 7.2 illustrates the continuous dynamics and key state variables that comprise the model. For space reasons, guard conditions and reset maps have been left out of the diagram. In the top-level “No Collision” mode, the dynamics of each vehicle are controlled by a separate automaton. Initial parameters passed to the Scenario sub-model determine the initial positions and velocities for the two vehicles, as well as the braking/acceleration applied when the car enters its “Accelerating” mode.

Truck

The dynamics of the truck are defined by a hybrid controller that varies the truck’s acceleration along the main road. Depending on how close the vehicles are to a collision, the truck’s controller enters one of
five cases: 0-Cruising, 1-Sensing, 2-Pre-Braking, 3-Full-Braking and 4-Stopped, which correspond to different levels of engagement of the truck's sensor and AEBS. In the 0-Cruising and 1-Sensing cases, the truck moves forward with the initial velocity. The 1-Sensing case is entered when the car enters the trapezoid that represents the field-of-view of the truck’s sensor. In the 2-Pre-Braking and 3-Full-Braking cases, the truck decelerates with a rate determined by control input, brake efficiency, and friction parameters. For example, in the 2-Pre-Braking case, the controller of the truck is as follows:

\[
\begin{align*}
| "2-Pre-Braking" \rightarrow \\
\text{if } y_1' > 0 \text{ then} \\
y_1'' &= a_1 u_{1pb} \mu \\
\text{else} \\
\text{state1+} &= "4-Stopped", \\
\text{...} \\
\text{if } -y_{\text{diff}} < ttcfb*y_{\text{diff}} \text{ || distance < ds then} \\
\text{state1+} &= "3-Full-Braking" \\
\end{align*}
\]

Here \( a_1 \) is the brake efficiency, \( u_{1pb} \) the control input and \( \mu \) the surface friction. The moderate control input during pre-braking is intended as a haptic warning. If the driver does not react by braking sufficiently before the ICAS determines that a collision is imminent, the controller attempts to avoid the collision by entering the 3-Full-Braking case, where the control input is greater. The criteria for concluding that the system should begin avoiding a collision combine TTC and the distance between the vehicles. Detecting that we have reached the TTC threshold for activating pre-braking is done by evaluating the expression \( -y_{\text{diff}} < ttcfb*y_{\text{diff}} \). This check is basically equivalent to the condition \( (y_1-y_2)/(y_1'-y_2') \geq ttcpb \). The difference is that it avoids using division, because it is a partial function that would result in a simulation error when the two vehicles reach the same speed. As discussed in Section 7.8, taking such errors into account is important in modeling for rigorous simulation.
Car

The car automaton controls the car’s behavior through and after the turn. The two parameters $\rho$ (rotation) and $\tau$ (turning radius) determine how the car turns through the intersection. The car’s controller also consists of five cases: 0-Cruising-Before-Turn, 1-Turning, 2-Cruising-After-Turn, 3-Accelerating and 4-Stopped. In the initial case 0-Cruising-Before-Turn the car moves forward at a constant speed. The 1-Turning case executes a right turn (by $\pi/2$ radians):

\[
| "1-Turning" -> \\
| \text{if (rotation < pi/2) then} \\
| \text{rotation'} = 1, \\
| x2'' = radius - x2, \\
| y2'' = radius - y2 \\
| \text{else} \\
| \text{state2+ = "2-Cruising-After-Turn"}
\]

After the turn (in the case 2-Cruising-After-Turn) the car again moves forward at a constant speed, until the ICAS determines that a collision is imminent. When this happens, the car attempts to avoid the collision by accelerating (case 3-Accelerating). There, $y2'' = a2*u2a$, where $a2$ is the car’s brake efficiency and $u2a$ is the control input.

7.5 Sensing and Collision Detection

Detecting a collision and detecting that the car has entered the truck sensor’s field-of-view both correspond to checking for intersection between polygons: the vehicle bounding rectangles and the sensor trapezoid. The model determines if these two-dimensional, convex polygons intersect by checking if any of their sides intersect, which amounts to evaluating a large number of similar Boolean expressions. For example, the following is an excerpt of the code that detects when the vehicles collide:
collide = caseCollideA || caseCollideB || caseCollideC
        || caseCollideD || caseCollideE || caseCollideNR,
        // front of V1 collides with the left side of V2
        caseCollideA = ((fly2 - fry1) * (rly2 - fry1) <= 0) &&
        (( (frx1-rlx2)*(fly2-rly2) +
          (fry1-rly2)*(rlx2-flx2) ) *
          ( (flx1-rlx2)*(fly2-rly2) +
            (fly1-rly2)*(rlx2-flx2) ) <= 0),

This computation is based on the coordinates of the corners of the
vehicle bounding rectangles. For example, frx1 is the x coordinate
of the front-right corner of the bounding rectangle of Vehicle 1 (the
truck). Notably, computing these coordinates based on the position
and orientation of each vehicle requires using non-linear (trigonomet-
ric) functions. For example:

        flx2 = x2 - vw2/2 * sin(rotation) - vl2/2 * cos(rotation)

7.6 Computing the Severity Class

Modeling and simulation were done in the Acumen integrated model-
ing and simulation environment [Acu] described in Section 6.4. This
section describes and analyzes the results of simulating the model
presented in the previous sections.

Table 7.1 summarizes five scenarios that were used in this case
study. Associated with each scenario is a set of initial conditions
and parameters, which influence the results of the simulation. These
include whether or not the car will make a turn in the intersection
(state2, rot2), whether the car will accelerate after making the turn
(u2a), and whether there will be a collision (related to several param-
eters, including: ttcpb, ttcfb, u1pb, u1fb).

Scenarios 2-4 were constructed to exhibit collisions with three dif-
ferent severity classes. Scenario 11 was constructed to bring the ve-
hicles close to, but avoid, a collision. Scenario 11.2 is a variant of
### Table 7.1: Summary of parameters for collision scenarios. † Vehicle 1 and Vehicle 2 are abbreviated as V1 and V2 respectively.

<table>
<thead>
<tr>
<th>Scenario parameter†</th>
<th>Var. name</th>
<th>Scen. 2 Slow collision</th>
<th>Scen. 3 Fast collision Sense in-turn</th>
<th>Scen. 4 Fast collision Sense post-turn</th>
<th>Scen. 11 EU347/2012</th>
<th>Scen. 11.2 EU347/2012 Full truck velocity range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 initial position in x dim. (m)</td>
<td>x1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V1 initial position in y dim. (m)</td>
<td>y1</td>
<td>-75</td>
<td>-135</td>
<td>-170</td>
<td>-120</td>
<td>-120</td>
</tr>
<tr>
<td>V1 initial speed in x dim. (m/s)</td>
<td>x1v</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V1 initial speed in y dim. (m/s)</td>
<td>y1v</td>
<td>19.55</td>
<td>30</td>
<td>40.2</td>
<td>[22.5, 22.8]</td>
<td>[21.6, 22.8]</td>
</tr>
<tr>
<td>V1 initial position in x dim. (m)</td>
<td>x2</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V2 initial position in y dim. (m)</td>
<td>y2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V2 initial speed in x dim. (m/s)</td>
<td>x2v</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V2 initial speed in y dim. (m/s)</td>
<td>y2v</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[2.7, 2.8]</td>
<td>[2.7, 2.8]</td>
</tr>
<tr>
<td>V2 control input for acceleration</td>
<td>u2a</td>
<td>0</td>
<td>1</td>
<td>0.75</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V2 initial mode</td>
<td>state2</td>
<td>2-Cruising-Before-Turn</td>
<td>2-Cruising-Before-Turn</td>
<td>2-Cruising-Before-Turn</td>
<td>2-Cruising-After-Turn</td>
<td>2-Cruising-After-Turn</td>
</tr>
<tr>
<td>V2 rotation (ρ)</td>
<td>rot2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>π/2</td>
<td>π/2</td>
</tr>
<tr>
<td>V1 pre-brake ttc threshold (s)</td>
<td>ttcpb</td>
<td>[3.5, 4.0]</td>
<td>[2.0, 2.5]</td>
<td>[3.5, 4.0]</td>
<td>[3.4, 3.5]</td>
<td>[3.4, 3.5]</td>
</tr>
<tr>
<td>V1 full-brake ttc threshold (s)</td>
<td>ttcfb</td>
<td>[2.5, 3.0]</td>
<td>[0.7, 1.0]</td>
<td>[2.5, 3.0]</td>
<td>[2.4, 2.5]</td>
<td>[2.4, 2.5]</td>
</tr>
<tr>
<td>V1 control input for pre-brake deceleration</td>
<td>u1pb</td>
<td>[-1.35, -1.25]</td>
<td>[-1, -0.9]</td>
<td>[-0.75, -0.5]</td>
<td>[-1.4, -1.35]</td>
<td>[-1.4, -1.35]</td>
</tr>
<tr>
<td>V1 control input for full-brake deceleration</td>
<td>u1fb</td>
<td>[-2.25, -2.15]</td>
<td>[-2, -1.8]</td>
<td>[-1.6, -1.2]</td>
<td>[-2.3, -2.25]</td>
<td>[-2.3, -2.25]</td>
</tr>
</tbody>
</table>
Scenario 11 with additional uncertainty in its parameters. It is constructed to produce a conservative rigorous simulation result that includes the possibility of a collision.

Table 7.2 includes the simulated values of $\Delta yv^2$ (change in velocity of the car due to collision) at the end time for a given pair of traditional and rigorous simulations of the same scenario. The traditional simulations were executed on the extreme values (those that contribute to increasing $\Delta yv^2$) of the parameter bounds used for the rigorous simulation. In Scenarios 2, 3, 4, and 11 the results are consistent, with the caveat that the outcome of the rigorous simulator includes the lower severity classes corresponding to its over-approximation of $\Delta yv^2$. In Scenario 11.2 the rigorous simulation result is also conservative. In this case the traditional simulation (of the extreme values of the parameters used in the scenario) yields no collision, while the enclosure produced by the rigorous simulation yields the possibility of a collision with severity class 1 or 2. The results show that rigorous simulation is able to produce useful bounds on the severity class and that, in different cases, the rigorously computed severity classification can be conservative and have different levels of precision.

The enclosure for the car velocity change from the collision produced, when simulating the first three scenarios, is shown in Figure 7.3. In these examples, the full brake deceleration applied by the controller is less than it should be. The figure shows the effect of the resulting collisions on the velocity of the car, shown in Table 7.2. The collisions happen around time 5, 6, and 4 for Scenarios 2, 3, and 4, respectively. The dotted horizontal lines indicate the upper bounds for the severity classes S1 (6 m/s, lower dotted line) and S2 (11 m/s upper dotted line). The plot for Scenario 2 shows that severity does not go beyond level S1, while the plot for Scenario 3 shows that severity does not go beyond level S2.

Unlike information produced using a traditional simulator, the bounds reported in Table 7.2 (and thus also the corresponding severity classes) are guaranteed to take two important types of error into account: that which is expressed in the model as an uncertain pa-
<table>
<thead>
<tr>
<th>Simulation result/parameter</th>
<th>Sc. 2</th>
<th>Sc. 3</th>
<th>Sc. 4</th>
<th>Sc. 11</th>
<th>Sc. 11.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Traditional: 4.5</td>
<td>Traditional: 6.7</td>
<td>Traditional: 42.5</td>
<td>Traditional: 0</td>
<td>Traditional: 0</td>
</tr>
<tr>
<td>Severity class corresponding to change in velocity</td>
<td>Rigorous: {1,2}</td>
<td>Rigorous: {1,2,3}</td>
<td>Traditional: None</td>
<td>Traditional: {1,2}</td>
<td>Traditional: None</td>
</tr>
<tr>
<td>Runtime (seconds)</td>
<td>Rigorous: 2,468</td>
<td>Rigorous: 244</td>
<td>Rigorous: 1,616</td>
<td>Rigorous: 1,562</td>
<td>Rigorous: 5,527</td>
</tr>
<tr>
<td></td>
<td>Traditional: 6</td>
<td>Traditional: 10</td>
<td>Traditional: 10</td>
<td>Traditional: 10</td>
<td>Traditional: 11</td>
</tr>
<tr>
<td>Maximum time step</td>
<td>$2^{-8}$</td>
<td>$2^{-1}$</td>
<td>$2^{-1}$</td>
<td>$2^{-9}$</td>
<td>$2^{-9}$</td>
</tr>
<tr>
<td>Minimum time step</td>
<td>$2^{-8}$</td>
<td>$2^{-9}$</td>
<td>$2^{-9}$</td>
<td>$2^{-9}$</td>
<td>$2^{-9}$</td>
</tr>
<tr>
<td>Taylor approx. order</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 7.2: A Summary of Results for Simulation Scenarios.

rameter, and that which arises from numerical approximations during simulation. Thus, compared to traditional simulation, rigorous simulation lifts some of the burden in inferring the ASIL of a hazardous event away from the modeler and on to the simulation tool.

In Table 7.2 each Scenario column shows results for a family of simulations, in the sense that each Scenario was executed using uncertain model parameters, specified as intervals in the model. Interval-valued parameters can be used to reflect variability of components, such as sensors or brakes (corresponding respectively to the Time-To-Collision, Pre-Brake and Full-Brake thresholds ttcpb/ttcfb and the control inputs u1pb/u1fb).
Simulation with uncertain parameters makes it possible for a single simulation to consider a range of scenarios. Acumen’s current rigorous simulator can simulate some models with uncertain initial state (Chapter 8) but is sensitive to large uncertainties. When many uncertainties are combined it may take much longer to complete a simulation, and the result may also be inconclusive. For example, the scenario described in the EU347/2012 regulation, represented by Scenario 11 in our model, considers speeds of 80 ± 2 km/h for the initial speed of the truck ($v_1$). Acumen is currently able to simulate this scenario successfully, showing that no collision is possible. Introducing uncertainty into other variables simultaneously (Scenario
11.2) leads to a simulation result that does not rule out the possibility of a collision. This example illustrates the limits of what Acumen’s rigorous simulator is currently capable of.

7.8 Remarks About our Experience Developing the Model using Acumen

This section summarizes practical issues that arose during the process of modeling and rigorous simulation of this system.

Acumen allows the user to run the same model using traditional and rigorous simulation. The traditional simulation is faster and, in the case of an error in the model, it produces simpler error messages. Thus, we found it convenient to use non-rigorous simulation to develop the model in the first place, before switching to rigorous simulation. The latter currently takes significantly more time to complete, and the run-time properties of rigorous simulation are more sensitive to the choice of simulation step. The ability to visualize the traditional simulation results in 3D makes debugging efficient, and the integration of the rigorous simulator with the 3D visualization facility of Acumen is particularly important for enabling seamless alternation between non-rigorous and rigorous simulation, as the model is being developed. We now turn to the challenges that we encountered when switching between the traditional and rigorous simulators during our work on the model.

7.8.1 Avoiding Missed Events

The relatively complex geometry of the sensor field-of-view forced us to implement sensing using general intersection computations. As the shapes involved are convex polygons, we opted to check for intersections between the edges. This highlighted one of the benefits of using rigorous simulation. Namely, during the first edge intersection in front/rear collisions (or sensing), the respective segments become collinear. Traditional simulation easily overlooks such scenarios, and the user needs to implement additional, superfluous conditionals in
the model. Rigorous simulation never misses any intersections, making the modeling task more intuitive and the resulting simulations more robust. However, as we will see in the following section, getting the model to work with the rigorous simulator requires manual intervention of a different nature.

7.8.2 Avoiding Undefined Operations

A pervasive problem when working with rigorous numerical computation is over-approximation. The main underlying cause is that standard interval arithmetic ignores dependencies between variables. This dependence problem shows itself when evaluating an expression that contains the same variable more than once. For example, evaluating $x - x$ with $x = [0, 1]$ using interval arithmetic gives $x - x = [0, 1] - [0, 1] = [-1, 1] \neq 0$. Mitigating the dependence problem is a major concern in validated numerics. Common approaches rely on state representations that keep track of dependencies between values [MB1; DFS; DFKT] and on algorithms that avoid evaluating expressions with wide intervals [NJC].

Over-approximation can give rise to operations being evaluated outside their domain. An example we encountered with this model is division by zero. In standard interval arithmetic, division is only defined when the denominator interval does not contain zero. Occasionally, over-approximation can be avoided by increasing the precision of the simulation. This may be achieved by subdivision that is simultaneously subdividing in space and decreasing the time step used. The improved precision (small resulting interval) can eliminate the problem of getting an interval that contains zero. However, this approach may lead to longer simulation times and can simply fail to solve the problem when the result is an open interval adjacent to (but not including) zero. In such cases it is important to consider whether the model can be reformulated to avoid the use of a partial operation.

A practical example that arose in this model was a condition $a < b/c$, which was replaced with $a \cdot c < b$ to avoid the problem under the assumption that $c$ is positive, and $a \cdot c > b$ when it is non-positive. The most obvious lesson that can be drawn from this experience is
that it is better to avoid the use of partial functions when total functions would suffice. A deeper insight is that rigorous simulation tools may *nudge the user* in the direction of such better modeling practices sooner than traditional tools, since rigorous simulation tools work with sets of values, which provide more extensive testing of models in one run than do traditional simulation tools.

7.8.3 Selecting the Simulation Time Step

In traditional numerical simulation, reducing simulation time steps generally leads to longer simulations, but yields more precise results (until we get to very small time steps). The situation is considerably more involved with rigorous simulations. First, decreasing the time step might lead to a loss of precision known as the *wrapping effect* [NJC]. Second, both increasing and decreasing simulation steps can increase computational cost. Depending on the dynamics of a

---

2In the case study, this was not a significant issue due to Acumen having appropriate data types to avoid wrapping for the relatively simple shapes involved in these simulations.
model, choosing a smaller step may help us obtain a conclusive outcome from a simulation. For example, one of the scenarios supported by the model has the vehicles come close to collision but still avoid it. As shown in Figure 7.4, selecting a step that is too large ($2^{-5}$, light colors) yields enclosures that include paths corresponding to a collision, which is insufficient to rule out the possibility of a collision. Decreasing the step by a factor of three (to $2^{-8}$, dark colors) shows conclusively that the collision does not occur. However, this does increase computational cost. Increasing simulation time steps can also increase runtime. This happens because it can lead to greater uncertainty in the values of variables, which in turn can lead to more branching. Branching can happen when multiple time steps are needed to determine conclusively that an enclosure crosses an event guard. During those steps, the simulator must branch the simulation to take both cases into account (the event happening or not happening in that step). It is possible that this problem is compounded if, before the branches can be recombined, they lead to more branching. In such situations, decreasing the step can lead to a faster simulation, by reducing the time spent on crossing event boundaries.

Acumen’s rigorous simulator can balance runtime and precision by adapting the step size during the simulation. When different minimum and maximum step sizes are provided by the user, the simulator will dynamically decrease the step size around time intervals where the dynamics change (such as when the control modes of the vehicles change, or at the collision), and increase the step size otherwise. Doing so can dramatically improve the runtime of a simulation. For example, the runtime of Scenario 3 is an order of magnitude faster than that of Scenario 2, even though the minimum time step used in Scenario 2 is smaller. However, Acumen’s current adaptive stepping strategy does incur a certain overhead, which may make the default fixed stepping preferable in some cases.
Chapter 8

Benchmarks for the Accuracy of Enclosures

This chapter reviews a collection of example models that can be used as benchmarks to evaluate the accuracy of a rigorous simulator. The examples are models of stable systems, that is, models whose solutions do not diverge. Rigorous numerical computation, including rigorous simulation, is sometimes criticized as being infeasible in practice, on account of an inevitable accumulation of numerical error. The collection of example models is used to empirically evaluate a design principle that addresses this criticism. The key insight underlying the principle is twofold:

1. Most man-made systems are designed to be stable and robust to variations in inputs and physical parameters.

2. Uncertainty due to approximation may be compensated by contracting dynamics.

These observations together suggest that the following principle is both desirable and plausible:

\footnote{This chapter is based on results presented at SNR 2016 [DBT]. Section 8.1.1 has been extended with a discussion of the impact of the time step on over-approximation.}
Accurate rigorous simulation should be possible for good designs. In particular, enclosures computed by a rigorous simulation tool for the trajectories of a robust and stable hybrid system should be converging.

This principle has been an important motivation for our own work, and we hope that it could be of value to others pursuing rigorous simulation or similar methods.

In the rest of this chapter we present a suite of small, concrete benchmarks that can be used to assess the extent to which a rigorous simulator upholds this principle. The benchmarks are classified into discrete and timed systems (Section 8.1); continuous systems (Section 8.2); and hybrid systems (Section 8.3). As the benchmarks are presented, we identify those where Acumen’s current rigorous simulator [aR] already succeeds, and those where challenges still remain. Table 8.1 summarizes the results.

For the purposes of this chapter, we will view an enclosure as converging if it reaches a fixed point where it is a subset of the enclosure at a previous point in simulation time. For simplicity of implementation, we will further approximate this condition by testing set containment only in the enclosure of the simulation step that immediately precedes it.

8.1 Discrete and Timed Systems

This section presents examples of discrete systems where enclosures should converge, despite the presence of considerable uncertainty, not only from numerical calculations but also from potential variability in system parameters.

8.1.1 Iteration and Nested Loops

Iteration can be expressed naturally in Acumen. For example, the computation for factorial of 5 can be expressed as follows:

```model Disc1 () =```
initially
  n = 5, a = 1
always
  if n > 0 then n+ = n-1, a+ = a*n noelse

A more interesting nested loop can be expressed as follows:

model Disc2 () =
  initially i = 1, i_max = 3,
  j = 1, j_max = 4,
  k = 1, k_max = 5,
  a = 0
always
  if i<=i_max then
    if j<=j_max then
      if k<=k_max then
        a+ = a+1, k+ = k+1
      else (j+ = j+1, k+ = 1)
    else (i+ = i+1, j+ = 1)
  noelse

This example simply counts the number of iterations into the variable a. Because the computations are discrete (no integration, only representable numbers), Acumen computes a thin (that is, single value) enclosure for the result.

Even in models that abstract from continuous dynamics, we often want to model computations as taking some time to compute. This can be achieved using a clock variable to produce the following model:

model Disc3 () =
  initially i = 1, i_max = 3,
  j = 1, j_max = 4,
  k = 1, k_max = 5,
  a = 0, t = 0, t' = 1
always
  if t>0.1 then
    t+ = 0,
if i<=i_max then
    if j<=j_max then
        if k<=k_max then
            a+ = a+1, k+ = k+1
        else (j+ = j+1, k+ = 1)
    else (i+ = i+1, j+ = 1)
else t’= 1

Section 6.4 describes Acumen’s standard plot and how to read it. To read the plots included in this chapter, the reader should be aware of the following conventions:

- For diverging enclosures, the simulation is stopped at an earlier time than the default (10) to provide a more informative visualization. In such cases, the time at which the simulation is stopped is indicated in the text.

- All plots are generated using a fixed time step of $2^{-6}$ to make it easy to observe error caused by the discretization of time.

The plot for the model above is as follows:
Here, instead of getting the answer 60 in the variable $a$ at time 0, we get to see the evolution of the value of $a$ over time, as well as the evolution of the values of all three counter variables ($i$, $j$, and $k$). Despite the accumulation of numerical errors in the clock variable $t$, after a finite amount of time the value of $a$ becomes exactly 60. In the interactive development environment, this is confirmed by hovering the mouse over the graph so that the exact value of the interval is displayed textually. The widening of the enclosure for the variable $t$ (the increasing sawtooth) is due to an over-approximation of the state during steps when a discrete assignment is active. When the model does not contain any uncertainties, this effect can be mitigated by enabling adaptive stepping. For example, setting:

\[
\begin{align*}
\text{simulator.minTimeStep} & = 1/2^{32}, \\
\text{simulator.maxTimeStep} & = 1/2
\end{align*}
\]

in the same model produces an improved simulation result:
Zooming in on the first two events (times at which the state activates a discrete assignment) in the variable $t$ shows that decreasing the time step around events largely eliminates this over-approximation:

In models that contain uncertainties, crossing an event inherently takes a certain amount of (model) time, and this approach becomes ineffective in such models. Likewise, in models where the discrete assignments involve complex expressions, error can accumulate rapidly. Thus, in general, avoiding the generation and accumulation in simulation of models with many events requires avoiding over-approximation (preserving the enclosure structure) across events. Note that this effect has no bearing on the main point of this chapter, but is
Returning to the models, we can consider whether convergence continues to hold if some parameters of the system are not known exactly. For example, the following model specifies uncertainty in the rate of the clock and the timing of the clock reset:

```plaintext
model Disc3i2 () =
  ...
  always
  ...
  else t' = [0.90 .. 1.0]
```

The expression `[0.90 .. 1.0]` denotes an interval literal that describes that `t'` can be as little as 0.90 or as large as 1.0. The plot for the model above is as follows:

For this model and the previous three, Acumen produces enclosures that first expand until a certain time value is reached. At that time
value, all enclosures except that of $t$ contract to a single value. Such convergence of enclosures on discrete systems with uncertainty about timing can provide a useful tool for estimating worst-case execution times, given a particular computational platform for which parameters are known only with a modest precision. Technically, achieving single-step containment requires disabling branch merging for all variants of this model.

### 8.1.2 Finite Impulse Response (FIR) Filter

As a minimal example Digital Signal Processing (DSP) model, we consider a basic Finite Impulse Response (FIR) filter:

```plaintext
model Disc4 () =
    initially $x_0 = 10$, $x_1 = 0$, $t = 0$, $t’ = 1$
    always
        if $t > 0.1$ then
            $t’ = 0$,
            $x_{-1}’ = x_0$,
            $x_0’ = 0.33*x_0 + 0.33*x_1 + 0.33*5$
        else $t’ = 1$
```

This is a second order filter that responds to a signal with values 10, 0, and then 5 thereafter. The enclosure produced by Acumen seems to quickly converge towards 5 for the value of $x_0$, even though the clock variable $t$ continues to accumulate uncertainty.
To further evaluate the ability of rigorous simulation to demonstrate the robustness of this filter, we can model uncertainty about the value of multiple parameters, including initial values, clock speed, transition timing, and the precision of the calculation of the multiplications (reflected by imprecision in the representation of coefficients) as follows:

model Disc4i5 () =
initially x_0 = [0.0 .. 10.0],
    x_1 = [-10.0 .. 0],
    t = 0, t’ = 1
always
if t> [0.09 .. 0.11] then
    t+ = 0,
    x_1+ = x_0,
    x_0+ = [0.33 .. 0.34] * x_0 +
    [0.33 .. 0.34] * x_1 +
    [0.33 .. 0.34] * 5
else t’= [0.8 .. 1.2]

The plot for this model is as follows:

Acumen’s results for this model converge towards 5, although there is a notable “error margin” around 5 that the enclosures stay outside. This is encouraging in that it shows that the computed behavior is robust. However, this plot should inspire a modeler/designer to dig deeper to better understand the most significant causes for this “error margin”.
Both examples presented in this section illustrate how enclosures can grow during a simulation of a discrete system and still converge at later parts, thus providing computational proof that the particular system being considered is stable and robust to a given degree of variability, in the exact values of key parameters within the given level of uncertainty. Interestingly, in these examples, single-step containment was not achieved except in the final (most general) example. It was necessary to disable branch merging, and the earlier examples generated too many branches.

8.2 Continuous Systems

Continuous systems are modeled by differential equations in Acumen. Guaranteed solutions to differential equations are provided by two different integrators. The first is based on the function interval Picard operator (Section 6.8.3) and the second on Taylor series methods (Section 6.8.4).

Uncertainty about the initial state of the system described by the second model above can be expressed as follows:

\[
\text{model Cont0i0 () =} \\
\text{initially t} = [0.0 \ldots 1.0], \ t' = 1 \\
\text{always t'} = 1
\]

The enclosure for this model is not contracting. Instead, we get a wide band representing all possible solutions starting between 0.0 and 1.0 and increasing at a constant rate of 1. The non-convergent nature of the enclosure is to be expected, as the underlying system is not stable in the sense of, for example, converging exponentially to a given value.

8.2.1 A First Order Linear System

A prototypical example of an exponentially stable system can be expressed as follows:

\[
\text{model Cont1 () =}
\]
initially $x = 0$, $x' = 1$
always $x' = 1 - x$

Uncertainty about the initial value for such a system can be expressed as follows:

model Cont1i0 () =
initially $x = [-0.75 .. 0.75]$, $x' = 1$
always $x' = 1 - x$

and has the following plot:

The plot shows that the enclosure for $x$ converges towards 1. Even if we add uncertainty about the value of $x$ used in the equation, as expressed in this model, the enclosure still converges:

model Cont1i1 () =
initially $x = [-0.75 .. 0.75]$, $x' = 1$
always $x' = 1 - [0.9 .. 1.1] * x$

The plot for this enclosure is as follows:

Convergence is towards an “error margin” around the value 1 because the precise value depends on the coefficient of $x$. If there is explicit uncertainty about the target, as expressed in this model, the resulting enclosure has similar behavior:

model Cont1i2 () =
initially $x = [-0.75 .. 0.75]$, $x' = 1$
always $x' = [0.9 .. 1.1] - x$
The plot for this enclosure is as follows:

The following model combines all three types of uncertainty:

```plaintext
model Cont1i3 () =
  initially x = [-0.75 .. 0.75], x’ = 1
  always x’ = [0.9 .. 1.1] -
    [0.9 .. 1.1] * x
```

The following plot shows that we still have convergence:

It also seems that the error margin becomes about as big as the sum of the two previous error margins. As in the FIR example, it will be up to the modeler/designer to determine the implications of this error margin on the success of the design.

### 8.2.2 A Second Order Linear System

A second order system exposes more subtleties and some challenges to convergence. Consider the following basic model:

```plaintext
model Cont2 () =
  initially x = 0, x’ = 0, x’’ = 1
  always x’’ = (1-x)-x’
```

The plot for this model is as follows:
8.2. CONTINUOUS SYSTEMS

This enclosure appears to converge. We can add uncertainty about the initial values as follows:

model Cont2i1 () =
initially x = [-0.5 .. 0.5],
   x’ = [-0.5 .. 0.5], x” = 1
always x”’ = (1 - x) - x’

The plot for this model is as follows:

The enclosure is more interesting and still appears to converge. Next, we may consider introducing uncertainty about the target value as follows:

model Cont2i2_ () =
initially x = 0, x’ = 0, x”’ = 1
always x”’ = ([0.90 .. 1.1] - x) - x’

The plot for this model up to time 5 seconds is as follows:
Unfortunately, this model is not converging. But can a slight refor-
mulation help convergence? Consider the following model:

\[
\text{model Cont2i2} () = \\
\text{initially} \ x = 0, \ x' = 0, \ x'' = 1, \quad a = [0.9 \ldots 1.1], \ a' = 0 \\
\text{always} \ x'' = (a - x) - x', \ a' = 0
\]

It has the following plot:

With default settings, the enclosure for the model still eventually
diverges. However, adding a basic error redistribution method does
achieve single-step containment for all variants above, except the last
one (2i2). Thus, seemingly minor changes to the model can affect
the impact of redistribution.

Now let us consider introducing uncertainty in the gain of the
feedback in this equation:

\[
\text{model Cont2i3} () = \\
\text{initially} \ x = 0, \ x' = 0, \ x'' = 1, \\
\quad b = [0.9 \ldots 1.1], \ b' = 0 \\
\text{always} \ x'' = b \times (1 - x) - x', \ b' = 0
\]

With default settings, this has a divergent enclosure. Similarly, even
a named coefficient, as expressed here, yields a divergent enclosure:

\[
\text{model Cont2i4} () = \\
\text{initially} \ x = 0, \ x' = 0, \ x'' = 1,
\]
8.2. CONTINUOUS SYSTEMS

\[ c = [0.5 .. 1.5], \ c' = 0 \]
\[ \text{always } x'' = (1 - x) - c \times x', \ c' = 0 \]

Enabling error redistribution removes the apparent divergence, but is not sufficient to achieve single-step containment.

8.2.3 A Second Order Non-Linear System

A classic non-linear equation is that of a pendulum:

```plaintext
model Cont3 () =
initially x = 0, x' = 0, x'' = 1
always x'' = \sin(\pi/2-x)-x'
```

This kind of non-linearity is typical in 2- and 3-dimensional classical mechanics. The plot for this model is as follows:

The enclosure for this system appears to converge. However, consider a variant with even minor uncertainty about the initial conditions:

```plaintext
model Cont3i1_ () =
initially x =[-0.05 .. 0.05],
\quad x' = 0, x''=1
always x'' = \sin(\pi/2-x)-x'
```

The resulting enclosure diverges:
We can recover single-step containment by error redistribution in all cases where we have uncertainty in one parameter at a time. But when we have uncertainty in all parameters, this fails and we observe divergence.

### 8.3 Hybrid Systems

In this section we present observations and benchmarks relating to hybrid systems. The benchmarks illustrate both how quickly combining the continuous and discrete can compound complexities for achieving convergence, and how it can also offer some new opportunities for facilitating it.

#### 8.3.1 Discretized Sensing/Actuation

The following model represents a situation where the output of a controller is not written continually to the system being controlled but rather in a sampled manner:

```plaintext
model Cont4hd_ () =
    initially x = 0, x' = 0, i = 0,
    o = 1, t = 0, t' = 1
    always
    x' = o,
    if t>0.5
    then o+ = 1-x, t+ = 0
    else t'=1
```

The plot for this model is as follows:
In the current implementation, the enclosure for this system diverges due to over-approximations in event handling.

### 8.3.2 Zeno Systems

A final and important type of system that is useful for evaluating the extent to which an implementation realizes the proposed principle is Zeno systems \([\text{KTB}^+]\). Such systems exhibit an infinite number of discrete changes over a finite time interval. Zeno behavior is known to be a phenomenon unique to hybrid systems, naturally arising in models of mechanical systems, control systems, and others.

A classic example of a model that exhibits Zeno behavior is a bouncing ball:

```plaintext
model Hybrid1 () =
  initially
  x = 10, x' = 0, x'' = -10
  always
  claim x>=0,
  if x == 0 & x’ < 0
  then x’+ = -x’/2
  else x’’ = -10
```

The variable \(x\) represents the height of a ball that falls to the ground. Upon impact with the ground, the ball bounces (switches direction)
with half the speed. The \texttt{claim} statement is essentially syntactic sugar for a conditional that leads to blocking the system (having no solution) in the else branch. Analysis of the model would reveal that the bouncing speed becomes zero after a finite amount of time, known as the Zeno time. However, the ball would have to bounce a countably infinite number of times before reaching this point.

Previous work \cite{KTB+} shows that it is possible to simulate such models past the Zeno point, using a fixed-point computation on enclosures during simulation. However, such enclosures are not necessarily converging past the Zeno point. They generally have the following shape:

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example.png}
\end{figure}

This enclosure is already interesting in that it is going beyond the Zeno point, which occurs at time 3 for this system, around the time when the enclosures start widening. Traditional methods either loop infinitely or skip some events, sacrificing rigor.

To get convergence in such situations, it seems necessary for the user to enrich the model with additional information that a powerful theorem prover could possibly infer, but which is not self-evident. For the system described, the addition would be an explicit model of the energy $e$ of the bouncing ball, and an explicit statement of the relation between the speed of the ball and the energy at a given time. This model is expressed as follows:

\begin{verbatim}
model Hybrid2 () =
initially
  x = 10, x' = 0, x'' = -10,
  e = 0*0/2 + 10*10
always
  claim x >= 0,
\end{verbatim}
8.3. HYBRID SYSTEMS

\[ \text{claim } e = x'x'/2 + 10x, \]
\[ \text{if } x == 0 \&\& x' < 0 \]
\[ \text{then } x'^+ = -x'/2, \ e^+ = [0 .. 0.25]e \]
\[ \text{else } x'' = -10 \]

It results in the following enclosure:

The enclosure converges, and adding a degree of uncertainty to virtually all parameters in this model does not seem to interfere with this convergence.

We note that this model cannot be simulated with branch merging disabled. Zeno systems generate a large number of branches that necessitate frequent merging.
<table>
<thead>
<tr>
<th>Type</th>
<th>Feature</th>
<th>Model Uncertainty</th>
<th>Benchmark</th>
<th>Convergence (A/B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete Loop</td>
<td>None</td>
<td>Disc1, Disc2</td>
<td></td>
<td>Y / Y</td>
</tr>
<tr>
<td>Timed Loop</td>
<td>None</td>
<td>Disc3</td>
<td></td>
<td>– / Y</td>
</tr>
<tr>
<td></td>
<td>Rate, or/and reset</td>
<td>Disc3i0 ... 2</td>
<td></td>
<td>– / Y</td>
</tr>
<tr>
<td>FIR</td>
<td>Rate, reset, arithmetic, or IVs</td>
<td>Disc4i0 ... 4</td>
<td></td>
<td>– / –</td>
</tr>
<tr>
<td></td>
<td>Rate, reset, arithmetic, and IVs</td>
<td>Disc4i5</td>
<td></td>
<td>– / Y</td>
</tr>
<tr>
<td>Continuous Linear, 1st ord.</td>
<td>None</td>
<td>Cont1</td>
<td></td>
<td>Y / Y</td>
</tr>
<tr>
<td></td>
<td>IV, gain, or/and target</td>
<td>Cont1i0 ... 3</td>
<td></td>
<td>Y / Y</td>
</tr>
<tr>
<td>Linear, 2nd ord.</td>
<td>None</td>
<td>Cont2</td>
<td></td>
<td>N / Y</td>
</tr>
<tr>
<td></td>
<td>Various</td>
<td>Cont2i0,1,2,3,4</td>
<td></td>
<td>N / Y</td>
</tr>
<tr>
<td></td>
<td>Various</td>
<td>Cont2i2,3,4</td>
<td></td>
<td>N / –</td>
</tr>
<tr>
<td>Nonlinear, 2nd ord.</td>
<td>None</td>
<td>Cont3</td>
<td></td>
<td>N / Y</td>
</tr>
<tr>
<td></td>
<td>IVs, target, or gains</td>
<td>Cont3i1, ... 5</td>
<td></td>
<td>N / Y</td>
</tr>
<tr>
<td></td>
<td>IVs, target, and gains</td>
<td>Cont3i6</td>
<td></td>
<td>N / N</td>
</tr>
<tr>
<td>Hybrid Linear, 1st ord.</td>
<td>None</td>
<td>Cont4</td>
<td></td>
<td>– / –</td>
</tr>
<tr>
<td>Sample &amp; pass</td>
<td>None</td>
<td>Cont4h</td>
<td></td>
<td>– / –</td>
</tr>
<tr>
<td>Sample hold</td>
<td>None</td>
<td>Cont4hd</td>
<td></td>
<td>N / N</td>
</tr>
<tr>
<td>BB</td>
<td>None</td>
<td>Hybrid1</td>
<td></td>
<td>N / N</td>
</tr>
<tr>
<td>BB with energy</td>
<td>None</td>
<td>Hybrid2</td>
<td></td>
<td>Y / Y</td>
</tr>
<tr>
<td></td>
<td>IVs, all parameters</td>
<td>Hybrid2i1 ... 5</td>
<td></td>
<td>Y / –</td>
</tr>
</tbody>
</table>

Table 8.1: Enclosure convergence of primary dynamics in benchmarks, as demonstrated using Acumen’s most recent rigorous simulator. Shorthands: A = Without error redistribution. B = With error redistribution and branch merging disabled. Y = Single-step containment confirmed. “–” = No apparent divergence (up to model time 50). N = Divergence observed. FIR = Finite impulse response. IV = Initial value. BB = Bouncing ball.
Chapter 9

Conclusion

This thesis described rigorous simulation, an approach to analyzing hybrid systems that combines the ease of use of simulation with the rigor of verification. It presented a formal specification of a rigorous simulator integrated into the modeling and simulation tool Acumen. A case study and a collection of benchmarks were used to evaluate the approach.

9.1 Summary of Contributions

The main contributions of this thesis are as follows:

1. The syntax, denotational semantics and operational semantics of a subset of the Acumen language are formally specified (Chapters 4 and 5). The specification leaves out the details of the validated numerical primitives that the operational semantics is built on, and instead specifies these in terms of the properties required to prove soundness. The purpose is to separate concerns, making the semantics as general as possible. Preliminary results toward a proof of soundness of the operational semantics with respect to the denotational semantics are presented (Appendix A).

2. A modeling environment that includes both traditional and
rigorous simulators for the Acumen language is described (Chapter 6). The implementation and its source code are freely available under a permissive license.

3. A case study and a collection of benchmarks that illustrate the feasibility of the rigorous simulator are presented (Chapters 7 and 8).

9.2 Lessons Learned

The main lesson learned during this work is that simulation can be both practical and rigorous, but that achieving these goals requires an effort that spans multiple disciplines. Practicality is a subtle matter. It is subjective, and depends both on the user and on the kind of problem that the user wants to simulate. Being practical touches on issues as varied as supporting a flexible syntax, supporting numerical methods that produce informative results, providing several flexible views of the simulation output, providing actionable error messages, and making the simulator’s user interface convenient. The notion of rigor is more objective. However, achieving it in a tool requires knowledge of many domains, including mathematics, computer science and software engineering. From mathematics, rigorous simulation relies on numerical methods and real analysis. From computer science, formal semantics is useful to precisely specify the modeling formalism and its properties. Software engineering practices such as testing are needed to realize the theory in an implementation. Combining all of these domains becomes a challenge on the level of terminology and notation. Finding a language that is familiar without carrying the wrong connotations thus becomes a challenge.

The rest of this section describes some specific lessons learned during this work. Some of them relate to language design and specification. Others relate to the difficulty of maintaining soundness in an implementation of a rigorous simulator, in particular when it is integrated with a traditional simulator.
9.2. LESSONS LEARNED

9.2.1 The Role of Properties in Semantics

A specification of a semantics can benefit from leaving out details. For example, in this thesis, the operational semantics (Section 5.8) is given in terms of numerical primitives whose precise definitions are left out. Instead, the primitives are specified as properties (Section 5.7.2). These properties are chosen to be only as specific as is necessary to prove the soundness of the operational semantics with respect to the denotational semantics. This approach has several benefits. Leaving out details keeps the operational semantics minimal, by separating treatment of the simulation algorithm from the that of the primitives it depends on. Doing so also makes the operational semantics general, in that any definition of a primitive that satisfies the corresponding properties is compatible with the operational semantics.

9.2.2 Equations Versus Assignments

Early versions of the implementation interpreted equations as assignments, that is, statements that mutate the state as soon as they are processed by the interpreter. Such an interpretation is incompatible with how differential equations are solved in mathematics, which requires that sequences of equations (Section 3.3) are processed in parallel. Two different approaches were taken to implement this semantics. In the optimized traditional interpreter, two copies of the state are maintained: one that is updated during interpretation and another that is used to look up values of variables. In the reference traditional interpreter and enclosure interpreter, the same issue is solved by flattening the AST into a system of equations, and passing this system to a solver.

9.2.3 Translations as a Design Tool

Translations can be a useful tool when developing a language, as they can help to uncover problems. In Section 4.2 we saw an example of this which shows that some translations can affect the rigorous simulation result of a model. A rigorous simulator must evaluate
Boolean expressions in a way that takes uncertainties in state variables into account. Just like interval arithmetic causes identities such as \( x - x = 0 \) to break, a set extension of Boolean logic causes identities such as \( p \land \neg p \iff \text{false} \) to break\(^1\). To avoid a loss of precision, a translation must therefore not produce such terms, or they must be identified and eliminated during evaluation. We discovered this issue when validating the translations described in Section 4.2 using our implementation.

9.2.4 Soundness Pitfalls

Developing a rigorous simulator requires maintaining a correctness guarantee, fundamentally provided by interval arithmetic, throughout a code base that may span many thousands of lines.

9.2.4.1 Parsing

Before the simulation can start, the tool needs to parse the textual model into a data structure, typically an abstract syntax tree, that can then be compiled or interpreted. Parsing libraries\(^2\) typically come with parsers for numerical literals. Seeing as these typically parse literals into floating-point numbers, any non-representable number (such as 0.1), will be rounded. Even if the resulting floating-point number is then converted into an interval before simulation, this parsing is unsound. Because 0.1 will be converted to a thin interval (whose end-points are equal), a check such as \( 10 \times 0.1 > 1 \) may return true, depending on the rounding mode used during parsing. To solve this problem, parsing of numeric constants can be deferred until they are interpreted (e.g. as intervals), or by parsing into an arbitrary-precision numeric type, such as rational numbers.

\(^1\) When \( p \) is the set \{true, false\} then \( p \land \neg p = \{p_1 \land \neg p_2 \mid p_1 \in p, p_2 \in p\} = \{\text{true, false}\} \), even though the formula is equivalent to false for any point value of \( p \). This effect is analogous to dependency \[\text{Moo}\] in interval arithmetic.  
\(^2\) Acumen’s parser is implemented using the Scala Standard Parser Combinator library \[\text{MPO}\].
9.2.4.2 Constants

Numerical constants such as a π are useful for modeling, especially in combination with trigonometric functions. The correct way to add these to a rigorous simulator is to provide a high-precision interval that encloses the true value. It is, for example, not sufficient to convert the constant as obtained from an arbitrary-precision numerical library. This value must be truncated by one significand, using the last bit of precision to obtain the bounds for the enclosing interval.

9.2.4.3 Preprocessing

Most simulation tools perform some kind of preprocessing before simulation, be it converting the model into an executable form (e.g. index reduction of DAEs), or for performance (e.g. partial evaluation). When transformations involve numerical operations, such as those performed during constant folding, they must be performed with a sound arithmetic. For example, this could be exact arithmetic of an arbitrary-precision numeric type, or interval arithmetic.

9.2.4.4 Safe Comparisons

During rigorous simulation, different kinds of approximations must be compared. Often, operations can be safely performed on over-approximations of the operands, but this is sometimes unsound. For example, to detect a fixed point in a reachability computation, it is necessary to check if one state is covered by another state. Such a subset containment check must either be performed directly on the operand types, or with an over-approximation of the possible subset and an under-approximation of the possible superset.

9.3 Future Work

This section outlines work needed to further support the thesis of this dissertation (Section 1.2). The foundations for rigorous simulation
are there, and development in several different directions is possible to increase the practicality of the approach.

9.3.1 A Richer Syntax and Semantics

To better support practical systems modeling, the rigorous simulator needs to be extended to the full Acumen language. This includes adding support for: vectors and matrices, including notation for updating parts of a matrix; dynamic creation, termination and movement of objects \( [\text{BT}] \); and acausal modeling with DAEs \( [\text{ZRT}^+] \).

The current implementation makes two kinds of fixpoint checks during simulation. The first is used to check if a fixpoint is reached in the intermediate enclosures computed within a single step. The other checks if the enclosure computed for the current step is included in the one for the previous step. To be able to detect fixpoints of periodic systems, it is necessary to extend the horizon over which the fixpoint is detected. The check could also be made more general by checking for coverings, rather than for containment in an individual state.

Acumen’s current semantics treats over-specified models, such as \( \{x' = f(x), x' = g(x)\} \), as modeling errors. There are several conceivable alternatives that are natural in the setting of a rigorous simulator. For example, \( \{x' = f(x), x' = g(x)\} \) could be interpreted as the differential inclusion \( x' \in [\min(f,g) .. \max(f,g)] \), where \( [\min(f,g) .. \max(f,g)] \) can be approximated by taking the enclosure hull of the solutions computed for \( x' = f(x) \) and \( x' = g(x) \) individually.

In the immediate future, we would like to focus on better understanding, and on improving the performance of, the current implementation of rigorous simulation in Acumen. We plan to continue to use this implementation to explore better ways to reduce and control branching.

More precise simulations of models with uncertain initial state can be achieved by splitting the uncertainties and running a separate simulation for each combination of parts. We plan to add support for automating this kind of batch simulation into Acumen. More elaborate schemes are also possible, where the state is split during the
course of the simulation. Such schemes are powerful but more difficult to implement efficiently, because the number of cases to consider when splitting a state increases exponentially in the number of subdivisions. This process can be prohibitively expensive if it is done iteratively, as in the case of a simulation algorithm.

From discussions with industrial practitioners, it is also clear that there is user demand for stochastic modeling, such as assigning probability distributions to variables. A simple approach to simulating such models is by splitting the probability distributions and running each part as a separate simulation.

9.3.2 Visualization

Being able to execute the same model using both the rigorous and traditional simulators has been useful in great part because the latter supports 3D visualization of the simulation results. Preliminary work on extending this functionality to the rigorous simulator was developed in the thesis of Pawlik and Andersson [PA], where the simulation result is visualized by sampling and rough (box) over-approximation. This work can be taken further by tightening the over-approximation.
CHAPTER 9. CONCLUSION
Appendices
Appendix A

Preliminary Results

In this appendix I establish that the operational semantics encloses a finitary denotational semantics that is also defined in this appendix. Establishing this result requires several new definitions and lemmas. Among these are two monotonicity results (Propositions [A.0.3 and A.0.2]), as well as transition relations and functions (Definitions 5.7.2, A.0.6 and A.0.8) that reflect the imprecision that results from the discretization of time in the operational semantics. The results presented in this appendix use induction, following the structure of the definition of the finitary denotational semantics. I believe that they will be useful in proving Conjecture 5.9.2 though that proof will likely be based on the Tarski fixpoint theorem [Tar].

Definition A.0.1 (Finitary Evolution of a Hybrid System). The finitary evolution of a hybrid system \((F, J) \in H\) is a function \(\llbracket (F, J) \rrbracket : \mathcal{P}(S) \rightarrow \mathcal{P}(T \times S)\) from the set of states to the set of time-states. Given an initial closed set of states \(S\), the finitary evolution \(\llbracket (F, J) \rrbracket S\) of \((F, J)\) from \(S\) is defined as follows:

\[
\begin{align*}
\llbracket (F, J) \rrbracket S & \\
(0, s) \in \llbracket (F, J) \rrbracket S & \quad (t, s_1) \in \llbracket (F, J) \rrbracket S \quad (F, J), s_1 \xrightarrow{\delta} s_2 & \quad (t + \delta, s_2) \in \llbracket (F, J) \rrbracket S
\end{align*}
\]
Proposition A.0.2. Whenever \( q \in m \in M^\mu \) and \( d_1, d_2 \in D \) then
\[
d_2 \subseteq d_2 \land q \not\in \text{active}(m, d_1) \implies q \not\in \text{active}(m, d_2)
\]

Proof.
\[
q \not\in \text{active}(m, d_1) \\
\implies \forall \text{ if } e \text{ then } A \in m. \text{ true } \not\in \begin{Bmatrix} [e]s \mid s \in d_1 \end{Bmatrix} \quad \text{(by Def 5.7.6 2c)} \\
\implies \forall \text{ if } e \text{ then } A \in m. \text{ true } \not\in \begin{Bmatrix} [e]s \mid s \in d_2 \end{Bmatrix} \quad \text{(as } d_2 \subseteq d_1) \\
\implies q \not\in \text{active}(m, d_2)
\]

Proposition A.0.3. The operational semantics \( m \vdash q_1, (\xi, d) \xrightarrow{h} Z \) is inclusion isotonic with respect to its enclosure parameter \( d \).

\[
\forall q \in m \in M^\mu, \xi \in \hat{T}, d_1, d_2 \in D, h \in T.
\]
\[
d_2 \subseteq d_1 \land (m \vdash q, (\xi, d_1) \xrightarrow{h} Z_1) \implies
(m \vdash q, (\xi, d_2) \xrightarrow{h} Z_2) \land [Z_2] \subseteq [Z_1]
\]

Proof. By structural induction on the derivation of \( \xrightarrow{} \).

Base case: The non-recursive cases are that of the Blocked rule and the Continuous rule when \( m = \{q\} \).

Case Blocked: We know that \( q \not\in \text{active}(m, d_1) \) and \( Z_1 = \emptyset \). Then \( q \not\in \text{active}(m, d_2) \) by Proposition A.0.2, and thus \( m \vdash q, (\xi, d_2) \xrightarrow{h} Z_2 \) by the Blocked rule and \( [Z_2] = \emptyset \subseteq [Z_1] \).

Case Continuous, \( m = \{q\} \): We know that \( Z_1 = \{d_1^{1}(0..h) / d_2^{1}@0..h\} \) and \( q \in \text{active}(m, d_1) \). Then \( q \in \text{active}(m, d_2) \) by Proposition A.0.2 and \( m \vdash q, (\xi, d_2) \xrightarrow{h} Z_2 \) by the Continuous rule. Further, \( Z_2 = \{d_2^{2}(0..h) / d_2^{2}@0..h\} \) because \( m_1 = \emptyset \). Now
\[
d_2 \subseteq d_1 \\
\implies (d_2^{2}(0..h) \subseteq d_1^{1}(0..h) \land d_2^{2} \subseteq d_1^{2}) \quad \text{(by Def 5.7.6 4a)} \\
\implies [Z_2] \subseteq [Z_1] \quad \text{(by Def 5.6.4)}
which completes the base case.

Inductive case: The recursive rules are Discrete and Continuous when \( m \neq \{ q \} \).

Case Discrete: We know that
\[
\begin{align*}
    & m \vdash q, (\ell, d_1) \xrightarrow{h} Z_1 \\
    & d_2^1 = \text{reset}(q, d_1) \\
    & \{ m \vdash q', (\ell, d_2^1) \xrightarrow{h} Z_{q'}^1 \} \quad q' \in m \\
    & Z_1 = \{ d_2^1 @ \ell \} \cup \bigcup_{q' \in m} Z_{q'}^1
\end{align*}
\]
If \( q_1 \not\in \text{active}(m, d_2) \) then \( Z_2 = \emptyset \subseteq Z_1 \). Otherwise we have \( d_2^3 = \text{reset}(q_1, d_2) \) with \( d_2^3 \subseteq d_2^1 \) by Definition 5.7.6.3a and
\[
\begin{align*}
    & \{ m \vdash q', (\ell, d_2^3) \xrightarrow{h} Z_{q'}^2 \} \quad q' \in m
\end{align*}
\]
with \( \forall q \in m. \ Z_{q'}^2 \subseteq Z_{q'}^1 \) by the induction hypothesis. Therefore
\[
\begin{align*}
    & m \vdash q, (\ell, d_2) \xrightarrow{h} Z_2, \text{ with } Z_2 = \{ d_2^3 @ \ell \} \cup \bigcup_{q' \in m} Z_{q'}^2
\end{align*}
\]
Finally \( [ Z_2^2 ] \subseteq [ Z_1 ] \) by Definition 5.6.4 and thus also \([ Z_2 ] \subseteq [ Z_1 ]\), which completes the case for Discrete.

Case Continuous, \( m \neq \{ q \} \land \ell \equiv [0] \land d_3^1 = d_2^1 \): We know that
\[
\begin{align*}
    & m \vdash q, (\ell, d_1) \xrightarrow{h} Z_1 \\
    & (d_2^3 @ [0..h]), d_2^1) = \text{solveIVP}(q, d_1, h) \\
    & \{ m \vdash q', ((0..h), d_2^3 @ [0..h]) \xrightarrow{h} Z_{q'}^1 \} \quad q' \in m_1 \\
    & Z_1 = \{ d_2^3 @ [0..h] / d_2^3 @ [0..h] \} \cup \bigcup_{q' \in m_1} Z_{q'}^1
\end{align*}
\]
Now, if \( q_1 \not\in \text{active}(m, d_1) \) then \([ Z_2 ] = \emptyset \subseteq [ Z_1 ] \). Otherwise we note that \( (d_2^3 @ [0..h], d_2^3) \) = solveIVP\((q, d_2, h)\) with \( d_2^3 @ [0..h] \subseteq d_2^1 \land d_2^3 \subseteq d_2^1 \) by Definition 5.7.6.4a. Therefore
\[
\begin{align*}
    & \{ m \vdash q', ((0..h), d_2^3 @ [0..h]) \xrightarrow{h} Z_{q'}^2 \} \quad q' \in m_1, \ [ Z_{q'}^2 ] \subseteq [ Z_{q'}^1 ]
\end{align*}
\]
by the induction hypothesis. Finally,

\[
[d_2^2(0..h)/d_2^2(0..h)] \subseteq [d_1^2(0..h)/d_1^2(0..h)]
\]

by Definition 5.6.4 and thus also \([Z_2] \subseteq [Z_1]\), which completes the case for Continuous.

\[\square\]

**Lemma A.0.4.**

\[\forall q \in \mathbb{Q}. \ H(q), s_1 \xrightarrow{\delta} s_2 \implies q, s_1 \xrightarrow{\delta} s_2\]

**Proof.** Let \(q \in \mathbb{Q}\) and assume that \(H(q), s_1 \xrightarrow{\delta} s_2\). There are two cases to check: when \(\delta = 0\) and when \(\delta > 0\).

Case \(\delta = 0\): The statement follows directly from Definition 5.4.5.

Case \(\delta > 0\): Since \(\text{guard(uncondition}(q)) = \text{true}\), Proposition 5.3.10 gives us that \(\pi_1 H(\text{uncondition}(q)) \subseteq \pi_1 H(q)\).

Thus \(H(\text{uncondition}(q)), s_1 \xrightarrow{\delta} s_2\), which in turn gives us that \(q, s_1 \xrightarrow{\delta} s_2\) by Definition 5.4.5.

\[\square\]

**Lemma A.0.5.**

\[\forall m \in \mathbb{M}^\mu. H(m), s_1 \xrightarrow{\delta} s_2 \implies m, s_1 \xrightarrow{\delta} s_2\]

**Proof.** Assume that \(H(m), s_1 \xrightarrow{\delta} s_2\). There are two cases for \(\delta\) that we must check.

Case \(\delta = 0\): Then \(\exists q \in m. H(q), s_1 \xrightarrow{\delta} s_2\) by Definitions 5.4.5 and 5.7.2. Then the statement follows directly from Definition 5.7.2.

Case \(\delta > 0\): Then by Definition 5.4.5 there exists a function \(x\) defined over \([0..\delta]\) such that \(x(0) = s_1, x(\delta) = s_2\) and

\[\forall t \in [0..\delta]. (x(t), x'(t)) \in \pi_1(H(m))\]

Because \(m\) is in partitioned form (Definition 4.2.6), the guards of the modes in \(m\) partition \(x\) into (disjoint, consecutive) segments
with domains \( x_\sigma(t) \) with time domains \( T_\sigma \) such that \( \exists q_\sigma \in m. \forall t \in T. (x_\sigma(t), x'_\sigma(t)) \in \pi_1(\mathcal{H}(q_\sigma)) \). Then \( q_\sigma, x_\sigma(T) \longrightarrow x'_\sigma(T) \) by Proposition \ref{prop:5.3.10} and Lemma \ref{lem:A.0.4}. Thus, by Definition \ref{def:5.7.2} we get \( m, x_\sigma(T) \longrightarrow x'_\sigma(T) \) for each segment and, consequently, we also get \( m, x(0) \longrightarrow x(\delta) \). □

**Definition A.0.6 (Evolution of a Mode).** The evolution of a mode \( q \in m \) is a function \( \llbracket m \vdash q \rrbracket : \mathcal{P}_\text{closed}(T \times S) \rightarrow \mathcal{P}_\text{closed}(T \times S) \) from the set of closed time-states to itself. Given an initial closed set of time-states \( I \in \mathcal{P}_\text{closed}(T \times S) \), the evolution \( \llbracket m \vdash q \rrbracket I \) of \( q \) from \( I \) is defined as follows:

\[
\frac{(t, s_1) \in I \quad q, s_1 \xrightarrow{\delta} s_2}{(t + \delta, s_2) \in \llbracket m \vdash q \rrbracket I}
\]

\[
\frac{(t, s_1) \in \llbracket m \vdash q \rrbracket I \quad m, s_1 \xrightarrow{\delta} s_2}{(t + \delta, s_2) \in \llbracket m \vdash q \rrbracket I}
\]

□

**Proposition A.0.7.**

\( \text{iso}(\llbracket m \vdash q \rrbracket) \)

*Proof.* By structural induction on the derivation of \( \llbracket m \vdash q \rrbracket \). Assume that

\[
I_2 \subseteq I_1 \subseteq T \times D \tag{A.1}
\]

\[
\llbracket m \vdash q \rrbracket I_1 \downarrow \tag{A.2}
\]

There are two rules in Definition \ref{def:A.0.6} by which \((t + \delta, s_2) \in \llbracket m \vdash q \rrbracket I_2\): one that is not recursive (the base case) and one that is recursive (the inductive case).

Base case \((t, s_2) \in I_1 \land q, s_1 \xrightarrow{\delta} s_2\): Then \((t, s_1) \in I_1\) by \((A.1)\) and thus \((t + \delta, s_2) \in \llbracket m \vdash q \rrbracket I_1\) by Definition \ref{def:A.0.6}.

Inductive case \((t, s_1) \in \llbracket m \vdash q \rrbracket I_2 \land m, s_1 \xrightarrow{\delta} s_2\): Then \((t, s_1) \in \llbracket m \vdash q \rrbracket I_1\) by the induction hypothesis and \((m, s_1 \xrightarrow{\delta} s_2)\) by \((A.2)\).
Thus \((t + \delta, s_2) \in \[m \vdash q\]_2\) by Definition A.0.6.

**Definition A.0.8** (Evolution of a Model). The evolution of a model \(m\) is a function \([m] : \mathcal{P}_{\text{closed}}(S) \to \mathcal{P}_{\text{closed}}(T \times S)\) from the set of closed time-states to itself. Given an initial set of states \(S\), the evolution \([m]S\) of \(m\) from \(S\) is defined as follows:

\[
\begin{align*}
(0, s) & \in [m]S \\
(t, s) & \in [m]S, m, s_1 \xrightarrow{\delta} s_2 \\
(t + \delta, s_2) & \in [m]S
\end{align*}
\]

**Lemma A.0.9.** Let \(m \in \mathbb{M}^\mu\), \(S \in \mathcal{P}_{\text{closed}}(S)\) and \(I = \{0\} \times S\) then

\([H(m)]S \subseteq [m]S\)

**Proof.** By induction on the structure of the derivation of \([H(m)]S\) that follows the rules for \([\cdot] : \) in Definition A.0.1. There are three cases by which \((t, s) \in [H(m)]S\). Two of these cases correspond to rules in Definition A.0.1. The third corresponds to \([H(m)]S\) being closed, meaning that \((t, s)\) is on the boundary of \([H(m)]S\). In each case we need to show that \((t, s) \in [m]S\).

Case \((t, s) \in I\): Then \((t, s) \in [m]S\) directly by Definition A.0.8.

Case \((t, s) \notin I\): We know that

\((t_1, s_1) \in [F, J]S\) \quad \([H(m)]S, s_1 \xrightarrow{\delta} s_2\) \quad \((t, s) = (t_1 + \delta, s_2)\)

But then \((t_1, s_1) \in [m]S\) by the induction hypothesis and \(m, s_1 \xrightarrow{\delta} s_2\) by Lemma A.0.5. Thus \((t_1 + \delta, s_2) = (t, s) \in [m]S\) by Definition A.0.8.

Case \((t, s)\) is in the boundary of \([H(m)]S\): By the first two cases we have shown that all \((t, s)\) in the interior of \([H(m)]S\) are also in
Since by definition $\llbracket m \rrbracket S$ is a closed set, it must also contain the boundary of $\llbracket H(m) \rrbracket S$.

**Lemma A.0.10.** Let $m \in \mathbb{M}^\mu$, $S \in \mathcal{P}_{\text{closed}}(S)$ and $I = \{0\} \times S$. Then

$$\llbracket m \rrbracket S \subseteq I \cup \bigcup_{q \in m} \llbracket m \vdash q \rrbracket I$$

**Proof.** By structural induction on the derivation of $\llbracket m \rrbracket S$.

Case $s \in S$: Then $(t,s) = (0,s) \in I$ so the statement holds trivially.

Case $(t_1, s_1) \in \llbracket m \rrbracket S \land m, s_1 \xrightarrow{\delta} s_2 \land (t, s) = (t_1 + \delta, s_2)$: Then $\exists q_1 \in m.q_1$, $s_1 \xrightarrow{\delta} s_2$ by Definition 5.7.2. By the induction hypothesis we can assume that

$$(t_1, s_1) \in I \cup \bigcup_{q \in m} \llbracket m \vdash q \rrbracket I$$

so we need to consider two additional sub-cases.

Case $(t_1, s_1) \in I$: Then $(t_1 + \delta, s_2) \in \llbracket m \vdash q_1 \rrbracket I$.

Case $(t_1, s_1) \in \bigcup_{q \in m} \llbracket m \vdash q \rrbracket I$: Then $(t_1, s_1) \in \llbracket m \vdash q_2 \rrbracket I$ for some $q_2 \in m$. But then $(t_1 + \delta, s_2) \in \llbracket m \vdash q_2 \rrbracket I$ by Definition A.0.6 and Definition 5.7.2.

**Definition A.0.11** (Move). The function $\text{move} : \mathbb{Q} \times (\mathbb{T} \times \mathbb{D}) \to \mathcal{P}_{\text{closed}}(\mathbb{T} \times \mathbb{D})$ is defined as follows:

$$\text{move}(q, I) = \text{closure}(\{(t + \delta, s_2) \mid (t, s_1) \in I \land (q, s_1 \xrightarrow{\delta} s_2)\})$$
Proposition A.0.12. When \( q \equiv e \) then \( \langle x'_i = e_i \rangle_{i \in \mathbb{N}} \) and \( I \subseteq T \times S \) then

\[
\text{move}(q, \text{move}(q, I)) \subseteq \text{move}(q, I)
\]

Proof. Let \( I_1 = \text{move}(q, I) \), \( I_2 = \text{move}(q, I_1) \) and \((t, s) \in I_2 \). Then there exists \((t_1, s_1) \in I\) such that \( q, s_1 \xrightarrow{\delta} s_2 \wedge q, s_2 \xrightarrow{\delta} s \). But then by Definition 5.7.2 we also have \( q, s_1 \xrightarrow{\delta} s \) and thus \((t, s) \in I_1\). □

Lemma A.0.13. Let \( q_1 \in m, S \subseteq S, T \in \mathbb{R} \) and \( I = T \times S \). Then

\[
[m \vdash q_1] I \subseteq I_1 \cup \bigcup_{q \in m} [m \vdash q] I_1
\]

(A.3)

where \( I_1 = \text{move}(q_1, I) \).

Proof. By structural induction on the derivation of \( [m \vdash q_1] I \). Let \( S = I_1 \cup \bigcup_{q \in m} [m \vdash q] I_1 \).

Base case \((t, s_1) \in I\) and \( q_1, s_1 \xrightarrow{\delta} s_2\): Then \((t + \delta, s_2) \in \text{move}(q_1, I) \subseteq S\) by Definition A.0.11.

Inductive case \((t, s_1) \in [m \vdash q_1] I\) and \( m, s_1 \xrightarrow{\delta} s_2\): Then \( \exists q_2 \in m. q_2, s_1 \xrightarrow{\delta} s_2\) by Definition 5.7.2. We have that \((t, s_1) \in S\) by the induction hypothesis, and we need to show that \((t + \delta, s_2) \in S\). There are two cases to check: when \((t, s_1) \in I_1\) and when \((t, s_1) \notin I_1\).

Case \((t, s_1) \in I_1\): Then \((t + \delta, s_2) \in \text{move}(q_2, I_1) \subseteq [m \vdash q_2] I_1 \subseteq S\).

Case \((t, s_1) \notin I_1\): Then \( \exists q_3 \in m. (t, s_1) \in [m \vdash q_3] I_1\), and so \((t + \delta, s_2) \in [m \vdash q_3] I_1 \subseteq S\) by Definition A.0.6. □

Corollary A.0.14. When \( q_1 \equiv e \) then \( \langle x'_i = e_i \rangle_{i \in \mathbb{N}} \) then

\[
[m \vdash q_1] I \subseteq I_1 \cup \bigcup_{q \in m \setminus \{q_1\}} [m \vdash q] I_1
\]

where \( I_1 = \text{move}(q_1, I) \).
Proof. Let $S = I_1 \cup \bigcup_{q \in m \setminus \{q_1\}} [m \vdash q] I_1$. Lemma A.0.13 states that

$$[m \vdash q_1] I \subseteq \text{move}(q_1, I) \cup \bigcup_{q \in m} [m \vdash q] \text{move}(q_1, I)$$

Thus it suffices to prove that

$$\text{move}(q_1, I) \cup \bigcup_{q \in m} [m \vdash q] \text{move}(q_1, I) \subseteq \text{move}(q_1, I) \cup \bigcup_{q \in m \setminus \{q_1\}} [m \vdash q] \text{move}(q_1, I)$$

or, equivalently

$$[m \vdash q_1] \text{move}(q_1, I) \subseteq \text{move}(q_1, I) \cup \bigcup_{q \in m \setminus \{q_1\}} [m \vdash q] \text{move}(q_1, I) \quad (A.4)$$

We prove (A.4) by structural induction on the derivation of $[m \vdash q_1] \text{move}(q_1, I)$. There are two ways in which $(t + \delta, s_2) \in [m \vdash q_1] \text{move}(q_1, I)$ by Definition A.0.6.

Base case $(t, s_1) \in \text{move}(q_1, I) \land q_1, s_1 \xrightarrow{\delta} s_2$: Then $(t + \delta, s_2) \in \text{move}(q_1, I) \subseteq S$ by Definition A.0.11.

Inductive case $(t, s_1) \in [m \vdash q_1] \text{move}(q_1, I) \land m, s_1 \xrightarrow{\delta} s_2$: Then $\exists q_2 \in m. q_2, s_1 \xrightarrow{\delta} s_2$ by 5.7.2 and $(t, s_1) \in S$ by the induction hypothesis.

Case $(t, s_1) \in \text{move}(q_1, I)$: Then $(t + \delta, s_2) \in \text{move}(q_2, \text{move}(q_1, I))$. If $q_2 = q_1$ then $\text{move}(q_2, \text{move}(q_1, I)) = \text{move}(q_1, I)$ by Proposition A.0.12 so $(t + \delta, s_2) \in S$. If $q_2 \neq q_1$ then $\text{move}(q_2, \text{move}(q_1, I)) \subseteq [m \vdash q_2] \text{move}(q_1, I)$ by Lemma A.0.13, so again $(t + \delta, s_2) \in S$.

Case $(t, s_1) \not\in \text{move}(q_1, I)$: Then $(t, s_1) \in [m \vdash q_3] \text{move}(q_1, I) \subseteq S$ for some $q_3 \in m \setminus \{q_1\}$. But then we also have $(t + \delta, s_2) \in [m \vdash q_3] \text{move}(q_1, I)$ by Definition A.0.6 and thus $(t + \delta, s_2) \in S$. 

$\square$
Lemma A.0.15 (Soundness With Respect to Finitary Denotational Semantics When Starting From a Mode). Given a MicroAcumen model $m \in M^\mu$ with state space $S = \mathbb{R}^N$, a mode $q_1 \in m$, a time $h > 0$, and an enclosure $d_1$, we have that

\[(m \vdash q_1, (t, d_1) \xrightarrow{h} Z^t) \implies \forall t \in T^t. (\llbracket m \vdash q_1 \rrbracket \llbracket \{t\} \times d_1 \rrbracket)_{[0..h]} \subseteq [Z^t] \quad (A.5)\]

where $T^{[0]} = \{0\}$, $T^{(0..h)} = (0..h)$.

Proof. By structural induction on the derivation of $\hookrightarrow$.

Base case: The only rule of Definition 5.8.2 that is not recursive is Blocked. This rule works identically for $I = \{0\} \times d_1$ and $I = \{t\} \times d_1$ with $t \in (0..h)$, so we prove the cases for $t = [0]$ and $t = (0..h)$ simultaneously. So, assume that $m \vdash q_1, (t, d_1) \xrightarrow{h} Z$ by the Blocked rule. Then $[Z] = [\emptyset] = \emptyset$, active$(m, d_1)$, and $q_1 \notin$ active$(m, d_1)$. Thus, assumption 2c on the auxiliary function active, stated in Definition 5.7.6, gives us that

\[\{s \mid s \in d_1 \land \llbracket \text{guard}(q_1) \rrbracket = \text{true}\} = \emptyset\]

so by definitions 5.3.9 and 5.7.2

\[\{s_2 \mid s_1 \in d_1 \land (q_1, s_1 \rightarrow s_2)\} = \emptyset\]

and thus $(\llbracket m \vdash q_1 I \rrbracket)_{[0..h]} = \emptyset$ by Definition A.0.6

Inductive case: The recursive rules are Discrete and Continuous. Lemma A.0.13 allows us to unfold the definition of $\llbracket m \vdash q_1 \rrbracket$ for any $I \subseteq T \times S$ as follows:

\[(\llbracket m \vdash q_1 I \rrbracket)_{[0..h]} \subseteq (I_1 \cup \bigcup_{q \in m} \llbracket m \vdash q I_1 \rrbracket)_{[0..h]} \quad (A.6)\]

where

\[I_1 = \{(t + \delta, s_2) \mid (t, s_1) \in I \land (q_1, s_1 \xrightarrow{\delta} s_2)\} \quad (A.7)\]
Case Discrete: Assume that \( m \vdash q_1, (\langle t, d_1 \rangle) \xrightarrow{h} Z \) by the Discrete rule. Recall that, in this case, we know that

\[
q_1 \equiv \text{if } e \text{ then } \langle x_i^+ = c_i \rangle_{i \in \mathbb{N}}
\]

\[
q_1 \in \text{active}(m, d_1)
\]

\[
d_2 = \text{reset}(q_1, d_1)
\]

\[
\{ m \vdash q, (\langle t, d_2 \rangle) \xrightarrow{h} Z_q \}_q \in m
\]

\[
Z = \{ d_2 \oplus \mathbb{N} \} \cup \bigcup_{q \in m} Z_q
\]

By assumption 3c on the auxiliary function reset, stated in Definition 5.7.6 and by Definition 5.6.4, it follows that \( I_1 \) specializes to

\[
I_1 = \{ (t, s_2) \mid s_1 \in d_1 \land t \in T_{\mathbb{N}} \land (q_1, s_1 \xrightarrow{0} s_2) \}
\]

\[
\subseteq T_{\mathbb{N}} \times d_2
\]

\[
= \lfloor d_2 \oplus \mathbb{N} \rfloor
\]

\[
\subseteq \lfloor \{ d_2 \oplus \mathbb{N} \} \cup \bigcup_{q \in m} Z_q \rfloor
\]

\[
= \lfloor Z \rfloor
\]

Furthermore,

\[
((\lfloor m \vdash q \rfloor I_1)|_{[0..h} \subseteq ((\lfloor m \vdash q \rfloor (T_{\mathbb{N}} \times d_2))|_{[0..h} \quad \text{(by Definition 5.7.6 3c)}
\]

\[
\subseteq \lfloor Z_q \rfloor \quad \text{(by the I.H.)}
\]

This completes the proof of the case for Discrete.

Case Continuous: Recall that, in this case, we know that

\[
q_1 \equiv \text{if } e \text{ then } \langle x_i' = e_i \rangle_{i \in \mathbb{N}} \quad \text{(A.8)}
\]

\[
q_1 \in \text{active}(m, d_1) \quad \text{(A.9)}
\]

\[
(d_{(0..h)}, d_2) = \text{solveIVP}(q_1, d_1, h) \quad \text{(A.10)}
\]

\[
m_1 = m \setminus \{ q_1 \} \quad \text{(A.11)}
\]

\[
\{ m \vdash q, ((0..h), d_{(0..h)}) \xrightarrow{h} Z_q \}_q \in m_1 \quad \text{(A.12)}
\]

\[
Z = \{ d_{(0..h)} \oplus d_3 @ (0..h) \} \cup \bigcup_{q \in m_1} Z_q \quad \text{(A.13)}
\]
and Corollary A.0.14 allows us to replace Equation (A.6) with
\[
([\text{m} \vdash q_1 I]|_{0..h}) \subseteq (I_1 \cup \bigcup_{q \in m_1} [m \vdash q I]|_{0..h}) \quad (A.14)
\]

There are two cases to check based on the type of \( t \): when \( t = [0] \land d_3 = d_2 \) and \( t = (0..h) \land d_3 = d(0..h) \).

Case Continuous, \( t = [0] \land d_3 = d_2 \): Let
\[
I = T^t \times d_1 = T^0 \times (0..h) = \{0\} \times T^0 \times d_1
\]

Then \( I_1 \) becomes
\[
I_1 = \{(t + \delta, s_2) \mid s_1 \in d_1 \land t \in T^t \land (q_1, s_1 \delta s_2)\}
\]
\[
= \{(t + \delta, s_2) \mid s_1 \in d_1 \land t \in T^0 \land (q_1, s_1 \delta s_2)\}
\]
\[
= \{(t + \delta, s_2) \mid s_1 \in d_1 \land t \in \{0\} \land (q_1, s_1 \delta s_2)\}
\]
\[
= \{(\delta, s_2) \mid s_1 \in d_1 \land (q_1, s_1 \delta s_2)\}
\]

Equation (A.14) gives us a way to prove Equation (A.5) by showing that the two constituents of the union in the right-hand side of Equation (A.14) are included in \([Z]\). Thus, to prove Equation (A.5) it is sufficient, by Equation (A.13), to prove the following two inclusions
\[
(I_1)|_{0..h} \subseteq \{d(0..h)/d_3 @ (0..h)\} \quad (A.15)
\]
\[
(\bigcup_{q \in m_1} [m \vdash q I]|_{0..h}) \subseteq \bigcup_{q \in m_1} Z_q \quad (A.16)
\]

Recalling that \( d(0..h) \) is an enclosure for \( I_1 \) over \((0..h)\), we obtain Equation (A.15) as follows:
\[
(I_1)|_{0..h} = \{(\delta, s_2) \mid s_1 \in d_1 \land (q_1, s_1 \delta s_2)\}|_{0..h} \subseteq \{d(0..h)/d_3 @ (0..h)\} \quad \text{(by Def 5.7.6 4c)}
\]
\[
= \{d(0..h)/d_3 @ (0..h)\} \quad \text{(by Definition 5.6.4)}
\]
To prove (A.16) it is sufficient to show that
\[ \forall q \in m_1. (\| m \vdash q \| I_1)|_{0..h} \subseteq \lbrack Z_q \rbrack \subseteq \lbrack Z \rbrack \quad (A.17) \]

To prove (A.17) we use what we know about \( d_{(0..h)} \) based on the assumptions on solveIVP in Definition 5.7.6. So, let \( q \in m_1 \). Then, by the induction hypothesis
\[ \forall t \in T^t. (\| m \vdash q \| (\{t\} \times d_{(0..h)}))|_{0..h} \subseteq \lbrack Z_q \rbrack \]
\[ \implies \forall t \in (0, h). (\| m \vdash q \| (\{t\} \times d_{(0..h)}))|_{0..h} \subseteq \lbrack Z_q \rbrack \quad \text{(taking } t \text{ to be } (0..h)) \]
\[ \implies (\| m \vdash q \| (\cup_{t \in (0,h)} \{t\} \times d_{(0..h)}))|_{0..h} \subseteq \lbrack Z_q \rbrack \]
\[ \implies (\| m \vdash q \| I_1)|_{0..h} \subseteq \lbrack Z_q \rbrack \quad \text{(by Def 5.7.6 4(c)i)} \]

Case Continuous, \( t = (0..h) \wedge d_3 = d_{(0..h)} \): Let
\[ I = T^t \times d_1 = T^{(0..h)} \times d_1 = (0..h) \times d_1 \]

Then \( I_1 \) becomes
\[
I_1 = \{(t + \delta, s_2) \mid s_1 \in d_1 \wedge t \in T^t \wedge (q_1, s_1 \delta \rightarrow s_2) \}
\]
\[
= \{(t + \delta, s_2) \mid s_1 \in d_1 \wedge t \in T_{(0..h)} \wedge (q_1, s_1 \delta \rightarrow s_2) \}
\]
\[
= \{(t + \delta, s_2) \mid s_1 \in d_1 \wedge t \in (0..h) \wedge (q_1, s_1 \delta \rightarrow s_2) \}
\]

Equation (A.14) gives us a way to prove Equation (A.5) by showing that the two constituents of the union in the right-hand side of Equation (A.14) are included in \( \lbrack Z \rbrack \). Thus, to prove Equation (A.5) it is sufficient, by Equation (A.13), to prove the following two inclusions
\[ I_1|_{0..h} \subseteq \lbrack \{d_{(0..h)}/d_3@(0..h)\} \rbrack \quad (A.18) \]
\[ (\bigcup_{q \in m_1} \| m \vdash q \| I_1)|_{0..h} \subseteq \bigcup_{q \in m_1} \lbrack Z_q \rbrack \quad (A.19) \]

Recalling that \( d_{(0..h)} \) is an enclosure for \( I_1 \) over \( (0..h) \), we obtain
APPENDIX A. PRELIMINARY RESULTS

(A.18) as follows:

\[ I_1|_{[0..h)} = \{(t + \delta, s_2) \mid t \in (0..h) \land s_1 \in d_1 \land (q_1, s_1 \to^\delta s_2)\}|_{[0..h)} \]

\[ = \{(t + \delta_1, s_3) \mid t \in (0..h) \land (\delta_1, s_3) \in \{(\delta, s_2) \mid s_1 \in d_1 \land (q_1, s_1 \to^\delta s_2)\}\}|_{[0..h)} \]

\[ \subseteq \{(t + \delta_1, s_3) \mid t \in (0..h) \land (\delta_1, s_3) \in (0..h) \times d_{(0..h)}\}|_{[0..h)} \]

\[ = \{[0..h) \times d_{(0..h)}\} \] \hspace{1em} (by Def 5.9.1)

\[ \subseteq \{[d_{(0..h)}]_2 @(0..h)\} \] \hspace{1em} (by Definition 5.6.4)

\[ \subseteq \{[d_{(0..h)}]_3 @(0..h)\} \] \hspace{1em} (by Def 5.7.6 4(c)i)

\[ = \{d_{(0..h)}\} \] \hspace{1em} (as \( d_3 = d_{(0..h)} \))

To prove (A.19) it is sufficient to show that

\[ \forall q \in m_1. ([m \models q]I_1)|_{[0..h)} \subseteq [Z_q] \subseteq [Z] \] (A.20)

To prove (A.20) we use what we know about \( d_{(0..h)} \) based on the assumptions on solveIVP in Definition 5.7.6 So, let \( q \in m_1 \). Then, by the induction hypothesis

\[ \forall t \in T^q. ([m \models q](\{t\} \times d_{(0..h)}))|_{[0..h)} \subseteq [Z_q] \]

\[ \implies \forall t \in (0..h). [m \models q]|_{[0..h)}(\{t\} \times d_{(0..h)}) \subseteq [Z_q] \] (taking \( t \) to be \( (0..h) \))

\[ \implies ([m \models q](\bigcup_{t \in (0..h)} \{t\} \times d_{(0..h)}))|_{[0..h)} \subseteq [Z_q] \]

\[ \implies ([m \models q](0..h) \times d_{(0..h)}))|_{[0..h)} \subseteq [Z_q] \]

\[ \implies ([m \models q]I_1)|_{[0..h)} \subseteq [Z_q] \] (by Def 5.7.6 4(c)i)

This completes the proof of the case for Continuous. \( \blacksquare \)

**Theorem A.0.16** (Soundness With Respect To Finitary Denotational Semantics). Given a MicroAcumen model \( m \) with state space \( S = \mathbb{R}^n \) a time \( h > 0 \) and an enclosure \( d \), and assuming that \( m, d \vdash_h Z \), then \([Z]\) encloses the denotational semantics \([\mathcal{H}(m)]d\) of \( m \) applied to \( d \) on the half-open interval \([0..h)\):

\[ \forall m, h > 0, d. ([\mathcal{H}(m)]d)|_{[0..h)} \subseteq [Z] \] (A.21)
Proof. Let \( I = \{0\} \times d \). Then

\[
\begin{align*}
& \subseteq \text{(by Lemma A.0.9)} \\
& \subseteq \text{(by Lemma A.0.10)} \\
& \subseteq \text{(by Lemma A.0.15)} \\
& = \text{(by Definition 5.6.4)} \\
& = \text{(by Definition 5.8.2)} \\
& = \text{(by Definition 5.6.4)}
\end{align*}
\]
Appendix B

State Space Power Sets and Closed-Power Sets as Lattices

This appendix casts two variants of the power set of state space, the domain over which the denotational semantics of MicroAcumen is defined in Section 5.4.4 as a complete lattice. A complete lattice is a structure that consists of a set and an ordering relation, along with certain requirements on its elements in terms of this relation. Maps over complete lattices often possess useful properties. Crucially, when a map is monotone, Tarski’s fixpoint theorem guarantees that the map has a least and a greatest fixpoint. Section 5.4.4 exploits this fact to define the evolution of a hybrid system as the least fixpoint of a monotone map.

The appendix is organized as follows. First, the notion of a complete lattice is defined. Second, this notion is used to define two concrete, complete lattices. The first lattice is over $\mathcal{P}(\mathbb{R}^n)$, the power set of the state space, and the second is over $\mathcal{P}_{\text{closed}}(\mathbb{R}^n)$, the set of closed subsets of the state space. Basically, these are lattices where suprema are unions and infima are intersections. However, for $\mathcal{P}(\mathbb{R}^n)$, the union of an infinite set of closed sets may not be closed. So, for $\mathcal{P}_{\text{closed}}(\mathbb{R}^n)$, we must define the supremum as the topological closure.
APPENDIX B. POWER SETS AS LATTICES

of the union.

**Definition B.0.1 (Complete Lattice).** A structure \((X, \sqsubseteq, \bigcup, \bigcap, \top, \bot)\) is a complete lattice if, and only if, it satisfies the following properties:

1. It is a partially ordered set (poset), that is, \(X\) is a set and \(\sqsubseteq\) is a relation, such that:
   
   (a) \(\forall x \in X. \ x \sqsubseteq x\) (reflexivity)
   (b) \(\forall x, y, z \in X. \ (x \sqsubseteq y \land y \sqsubseteq z) \implies x \sqsubseteq z\) (transitivity)
   (c) \(\forall x, y \in X. \ (x \sqsubseteq y \land y \sqsubseteq x) \implies x = y\) (antisymmetry)

2. Given the auxiliary notions \(\text{ub}(\cdot)\) and \(\text{lb}(\cdot)\), every subset \(Y \subseteq X\) has a unique element \(s \in \text{ub}(Y)\) called the supremum of \(Y\), written \(\bigcup Y\), and a unique element \(i \in \text{lb}(Y)\) called the infimum, written \(\bigcap Y\), such that:

   (a) The set of upper bounds of \(Y \subseteq X\), written as \(\text{ub}(Y)\), is defined as:

   \[
   \text{ub}(Y) \triangleq \{ x \in X \mid \forall y \in Y. \ y \subseteq x \} 
   \]

   (b) The set of lower bounds of \(Y \subseteq X\), written as \(\text{lb}(Y)\), is defined as:

   \[
   \text{lb}(Y) \triangleq \{ x \in X \mid \forall y \in Y. \ x \subseteq y \} 
   \]

   (c) The a supremum (join or least upper bound) \(s\) of \(Y \subseteq X\), written as \(\bigcup Y\), satisfies:

   \[
   \forall y \in \text{ub}(Y). \ s \subseteq y 
   \]

   (d) The infimum (meet or greatest lower bound) \(i\) of \(Y \subseteq X\), written as \(\bigcap Y\), satisfies:

   \[
   \forall y \in \text{lb}(Y). \ y \subseteq i 
   \]

3. It has unique greatest and least elements, such that:
(a) The greatest (top) element of $X$, written as $\top$, is defined as the supremum of $X$, that is:

$$\top = \bigcup X$$

(b) The least (bottom) element of $X$, written as $\bot$, is defined as the infimum of $X$, that is:

$$\bot = \bigcap X$$

We define the following additional notions for every complete lattice:

**Definition B.0.2** (Auxiliary Notions).

1. Given complete lattices $(X, \sqsubseteq_X, \ldots)$ and $(Y, \sqsubseteq_Y, \ldots)$, a map $f : X \to Y$ is monotone order preserving (or monotone for short) if:

$$\forall x, y \in X. \; x \sqsubseteq_X y \implies f(x) \sqsubseteq_Y f(y)$$

2. Given $x, y \in X$, the pairwise supremum, written as $x \sqcup y$, is defined as:

$$x \sqcup y \triangleq \bigcup \{x, y\}$$

The pairwise infimum, written as $x \sqcap y$, is defined as:

$$x \sqcap y \triangleq \bigcap \{x, y\}$$

3. A chain is a subset $Y \subseteq X$, such that:

$$\forall x, y \in Y. \; (x \sqsubseteq y) \lor (y \sqsubseteq x)$$

4. Given complete lattices $(X, \ldots)$ and $(Y, \ldots)$, a map $f : X \to Y$ is (Scott-) continuous if, and only if:

(a) It is monotone
Definition B.0.3 (Power Set of State Space as a Structure). We define the structure $\Psi$ over the power set $\mathcal{P}(\mathbb{R}^n)$ of the state space $\mathbb{R}^n$ for a given $n \in \mathbb{N}$, ordered by set inclusion. That is, $\Psi \triangleq (\mathbb{R}^n, \subseteq, \bigcup, \bigcap, \mathbb{R}^n, \emptyset)$, where $\subseteq \triangleq \subseteq$, $\bigcup X \triangleq \bigcup X$, $\bigcap X \triangleq \bigcap X$, $\top \triangleq \mathbb{R}^n$ and $\bot \triangleq \emptyset$. □

Definition B.0.4 (Closed-Power Set of State Space as a Structure). We define the structure $\Psi_{\text{closed}}$ over the set $\mathcal{P}_{\text{closed}}(\mathbb{R}^n)$ of closed subsets of the state space $\mathbb{R}^n$ for a given $n \in \mathbb{N}$, ordered by set inclusion. That is $\Psi_{\text{closed}} \triangleq (\mathcal{P}_{\text{closed}}(\mathbb{R}^n), \subseteq, \lambda X. \text{closure}(\bigcup X), \bigcap, \mathbb{R}^n, \emptyset)$, where $\mathcal{P}_{\text{closed}}(X) \triangleq \{X \mid X = \text{closure}(X) \land X \in \mathcal{P}(X)\}$, $\subseteq \triangleq \subseteq$, $\bigcup X \triangleq \text{closure}(\bigcup X)$, $\bigcap X \triangleq \bigcap X$, $\top \triangleq \mathbb{R}^n$ and $\bot \triangleq \emptyset$. □

The above definition uses lambda notation to define an anonymous function. For example, $\lambda X. \text{closure}(\bigcup X)$ can equivalently be written as $X \mapsto \text{closure}(\bigcup X)$.

Proposition B.0.5. The two structures $\Psi$ and $\Psi_{\text{closed}}$ are complete lattices with

1. $\subseteq = \subseteq$ for both $\Psi$ and $\Psi_{\text{closed}}$

2. $\forall X \subseteq \mathbb{R}^n. \bigcup X = \bigcup X$, $\forall X \subseteq \mathbb{R}^n. \bigcap X = \text{closure}(\bigcup X)$ and $\forall X \subseteq \mathbb{R}^n. \bigcap X = \bigcap X$

3. $\top = \mathbb{R}^n$ and $\bot = \emptyset$ for both $\Psi$ and $\Psi_{\text{closed}}$

Proof. We consider each of the three requirements for being a complete lattice as follows:

1. Both $\Psi$ and $\Psi_{\text{closed}}$ are posets since $\subseteq$ is reflexive, transitive, and antisymmetric.

2. The structure $\Psi$: 

$$\forall Z \subseteq X. f(\bigcup Z) = \bigcup f(Z)$$
(a) Let $X \subseteq \mathbb{R}^n$. Then $\forall x \in X. \ x \subseteq \bigcup X$, so $\bigcup X \in \text{ub}(X)$.

Now let $y \in \text{ub}(X)$ and $a \in \bigcup X$. Then $\exists x \in X. \ a \in x$. Since $y \in \text{ub}(X)$, we must have $x \subseteq y$. Thus $a \in y$ and therefore also $\bigcup X \subseteq y$.

(b) Let $X \subseteq \mathbb{R}^n$. Then $\forall x \in X. \ \bigcap X \subseteq x$, so $\bigcap X \in \text{lb}(X)$.

We prove that $\forall y \in \text{lb}(X). \ y \subseteq \bigcap X$ by contradiction. Assume that $\exists y \in \text{lb}(X). \ y \nsubseteq \bigcap X$. Then $\exists a \in y. \ a \notin \bigcap X$. Thus $\exists x \in X. \ y \nsubseteq x$ by the definition of $\bigcap$, which contradicts $y \in \text{lb}(X)$.

The structure $\Psi_{\text{closed}}$:

(a) Let $X \subseteq \mathbb{R}^n$. Then $\forall x \in X. \ x \subseteq \text{closure}(\bigcup X)$, which means that $\text{closure}(\bigcup X) \in \text{ub}(X)$.

Now let $y \in \text{ub}(X)$ and $a \in \bigcup X$. Then $\exists x \in X. \ a \in x$. Since $y \in \text{ub}(X)$, we must have $x \subseteq y$. Thus $a \in y$ and therefore also $\bigcup X \subseteq y$. But $y \in \mathbb{R}^n$, which means that $y$ is closed, so if $\bigcup X \subseteq y$ then $\text{closure}(\bigcup X) \subseteq y$.

(b) Let $X \subseteq \mathbb{R}^n$. Then $\forall x \in X. \ \bigcap X \subseteq x$, so $\bigcap X \in \text{lb}(X)$.

We prove that $\forall y \in \text{lb}(X). \ y \subseteq \bigcap X$ by contradiction. Assume that $\exists y \in \text{lb}(X). \ y \nsubseteq \bigcap X$. Then $\exists a \in y. \ a \notin \bigcap X$. Thus $\exists x \in X. \ y \nsubseteq x$ by the definition of $\bigcap$, which contradicts $y \in \text{lb}(X)$.

3. (a) $\top = \bigcup \Psi = \mathbb{R}^n$ by Definition [B.0.1.3a]

(b) $\bot = \bigcap \Psi = \emptyset$ by Definition [B.0.1.3b]

Proposition B.0.6. Fix $S \in \mathbb{R}^n$, $(F,J) \in \mathbb{H}$. Then the map $F : \mathbb{R}^n \to \mathbb{R}^n$ defined as:

$F(I) = \text{closure} \{0\} \times S \cup \{(t+\delta, s_2) \mid (t, s_1) \in I \land (F,J), s_1 \rightarrow^\delta s_2\}$

is monotone.
APPENDIX B. POWER SETS AS LATTICES

**Proof.** Fix $S \subseteq \mathbb{R}^n$, let $x, y \in \mathbb{R}^n$, such that $x \subseteq y$, and let $a \in F(x)$. Then there are two cases for $a$:

Case $a \in \text{closure}(\{0\} \times S)$: Then $a \in F(y)$ by the definition of $F$.

Case $a = (t + \delta, s_2)$ for some $(t, s_1) \in x$ with $(F, J), s_1 \xrightarrow{\delta} s_2$. Then $(t, s_1) \in y$ since $x \subseteq y$ and thus also $(t + \delta, s_2) \in F(y)$. So $F(Y)$ contains every point of the form:

$$\{(t + \delta, s_2) \mid (t, s_1) \in I \land (F, J), s_1 \xrightarrow{\delta} s_2\}$$

But since $F(Y)$ is closed, it must also contain:

$$\text{closure}(\{(t + \delta, s_2) \mid (t, s_1) \in I \land (F, J), s_1 \xrightarrow{\delta} s_2\})$$

and therefore all of $F(X)$. 

\qed
Bibliography


[DBT] Adam Duracz, Ferenc A. Bartha, and Walid Taha. Accurate rigorous simulation should be possible for good


[LFBZ] Long Le, Andreas Festag, Roberto Baldessari, and Wen-hui Zhang. V2x communication and intersection safety.


[Ris] RiskCAT. [http://www.hitex.co.uk](http://www.hitex.co.uk) 2016.


[ZWI+] Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Marcia O’Malley, Robert Cartwright,
Glossary

**AEBS**  Advanced Emergency Braking System. [119] [120] [122]

**ASIL**  Automotive Safety Integrity Level. [118] [119]

**AST**  Abstract Syntax Tree. [108] [157]

**BNF**  Backus-Naur Form. [52] [55] [64] [79]

**CBD**  Critical Braking Distance. [120]

**CLI**  Command Line Interface. [101]

**CPS**  Cyber-Physical Systems. [21] [29] [39] [40]

**CWD**  Critical Warning Distance. [120]

**DAE**  Differential-Algebraic Equation. [30] [31] [33] [35] [159] [160]

**DDE**  Delay Differential Equation. [31]

**dL**  Differential Dynamic Logic. [32]

**DSL**  Domain-Specific Language. [39]

**DSP**  Digital Signal Processing. [142]

**FIR**  Finite Impulse Response. [142] [146]
**FRP**  Functional Reactive Programming.  39

**GUI**  Graphical User Interface.  101, 102, 103

**HARA**  Hazard Analysis and Risk Assessment.  118, 119

**ICAS**  Intersection Collision Avoidance System.  15, 119, 120, 123, 124

**IVP**  Initial Value Problem.  77, 83, 88, 89, 100, 113

**NG-TEST**  Next Generation Test Methods for Active Safety Functions.  119, 120

**ODE**  Ordinary Differential Equation.  35, 83, 88, 89, 100, 113

**TTC**  Time-To-Collision.  120, 123
Index

Simulink, 28
Stateflow, 28

acausal, 29
activated, 79
active equations, 105
approximations, 18
auxiliary function, 76

block diagrams, 28
Booleans with the trivial topology, 68
bottom, 179
bounded model checking, 33

chain, 179
children, 39
closed, 67
closure, 67
complement, 67
complete lattice, 178
continuous, 179
continuous modes, 52
delay differential equation, 27
differential equation, 25
differential-algebraic equations, 27
discrete modes, 52
discrete states, 27

differential-algebraic equations, 27
discrete modes, 52
discrete states, 27

enclosure interpreter, 102
closure semantics, 102
end-time enclosure, 79
event detection, 31
event handling, 31
events, 30
evolution, 74

field, 26
field expression, 66
flow relation, 71
formal verification, 32

greatest element, 179
greatest lower bound, 178
guard, 52
guards, 27

hazard, 114
hom-set, 68
hybrid automata, 27
hybrid system, 25
infimum, 178  
iintegrator, 31  
interval box contractor, 109  
invariant, 27  
isotone, 78  
join, 178  
jump relation, 71  
lattice, 178  
least element, 179  
least upper bound, 178  
locations, 27  
lower bound, 178  
mathematical models, 18  
meet, 178  
model checking, 32  
Modelica, 29  
modes, 27  
monotone, 179  
monotone order preserving, 179  
umerical integrators, 31  
numerical primitive, 76  
onopen intervals, 67  
onopen set, 67  
over-approximations, 104  
pairwise infimum, 179  
pairwise supremum, 179  
parent, 39  
partially ordered set, 178  
path condition, 49  
Picard iterator, 109  
poiset, 178  
property, 32  
rang, 86  
rang enclosure, 79  
reachability, 32  
real numbers with the standard topology, 67  
rigorous simulation, 20  
risk, 114  
safety integrity level, 114  
safety property, 113  
Scott-continuous, 179  
signals, 29  
4simulation, 29  
simulation trace, 30  
solution, 73  
solver, 31  
stateflow charts, 28  
supremum, 178  
tactics, 33  
testing, 32  
theorem proving, 32  
time step, 31  
timed enclosure, 75  
top, 179  
topological closure, 67  
topological space, 67  
topology, 67  
traditional semantics, 96  
trajectory, 30  
transition relation, 72  
upper bound, 178
validated numerical integrators, 31
validated numerics, 31
value enclosure, 74
verification, 29