To the Graduate Council:

I am submitting herewith a dissertation written by Jason Martin Carter entitled, "Sequence-Based Specification of Embedded Systems." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jesse H. Poore
Major Professor

We have read this dissertation
and recommend its acceptance:

Kenneth Stephenson

Michael G. Thomason

Lynne E. Parker

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and Dean of the
Graduate School

(Original signatures are on file with official student records.)

Sequence-Based Specification of Embedded Systems

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Jason Martin Carter
December 2009

Copyright © 2009 by Jason Martin Carter.
All rights reserved.

Dedication

To my wife, whose love and selfless support has sustained me during life's challenges and whose personal perseverance has motivated me to achieve more than I ever thought possible.

Acknowledgements

I would like to thank my adviser, Dr. Jesse H. Poore, for patience, guidance, and a tremendous amount of support during my years at the University of Tennessee.

I am also very grateful to my committee: Dr. Kenneth Stephenson, Dr. Michael G. Thomason, Dr. Lynne E. Parker, and Dr. Piotr Luszczek.

Over my years with the Software Quality Research Laboratory (SQRL), it has been a privilege to work with many talented people. I am grateful to all of them. Specifically, I would like to acknowledge Dr. Lan Lin. Lan is an exceptional computer science theoretician and a good friend. Thank you Lan for your unwavering support and assistance. Dr. Stacy Prowell is an outstanding professor and a tremendous example. Thank you Stacy for extending the invitation to work with SQRL. Dr. Krik Sayre is a remarkable software writer, and I share his passion. Thank you Kirk for helping me through practical application of the lab's methods. I cannot thank Mr. Tom Swain enough for providing opportunities. Finally, I would like to thank Mr. Bradford Smith for friendship and great conversation.

One of the most generous men I have ever met helped me achieve this goal. Although Mr. William Cash is not able to see the end result, I will never forget him.

I am grateful to my wife's parents, Bill and Pat Vaughn, for years of love and support. Their contribution to helping me achieve this goal was enormous.

Most importantly, I would like to thank my parents, Will and Judy Carter, for providing a strong foundation and a desire to improve and grow every day. I would not be where I am today without their love.

MATLAB®, Simulink®, and Stateflow® are registered trademarks of The MathWorks, Inc.
LabVIEW™ is a registered trademark of National Instruments, Inc.

Abstract

Software has become integral to the control mechanism of modern devices. From transportation and medicine to entertainment and recreation, embedded systems integrate fundamentally with time and the physical world to impact our lives; therefore, product dependability and safety are of paramount importance.

Model-based design has evolved as an effective way to prototype systems and to analyze system function through simulation. This process mitigates the problems and risks associated with embedding software into consumer and industrial products. However, the most difficult tasks remain: Getting the requirements right and reducing them to precise specifications for development, and providing compelling evidence that the product is fit for its intended use.

Sequence-based specification of discrete systems, using well-chosen abstractions, has proven very effective in exposing deficiencies in requirements, and then producing precise specifications for good requirements. The process ensures completeness, consistency, and correctness by tracing each specification decision precisely to the requirements. Likewise, Markov chain based testing has proven effective in providing evidence that systems are fit for field use.

Model-based designs integrate discrete and continuous behavior; models have both hybrid and switching properties. In this research, we extend sequence-based specification to explicitly include time, continuous functions, nondeterminism, and internal events for embedded real-time systems. The enumeration is transformed into an enumeration hybrid automaton that acts as the foundation for an executable model-based design and an algebraic hybrid I/O automaton with valuable theoretical properties. Enumeration is a step-wise problem solving technique that complements model-based design by converting ordinary requirements into precise specifications. The goal is a complete, consistent, and traceably correct design with a basis for automated testing.

Contents

1 Introduction 1

1.1 The Problem 1

1.2 Model-Based Design 2

1.3 Results and Contribution 4

2 System Characteristics 5

2.1 Sets and Number Systems 5

2.2 Relations and Functions 6

2.3 Dynamic Behavior 6

2.4 Signals 7

2.5 System Boundary 8

2.5.1 Internal Dynamics 10

2.6 System Types 11

2.6.1 Discrete Systems 11

2.6.2 Continuous Systems 11

2.6.3 Switching Systems 12

2.6.4 Hybrid Systems 13

2.7 Discrete Sequence-Based Specification 14

2.8 Applicability 16

3 Hybrid System Mathematics 17

3.1 Introduction 17

3.2 Intervals 19

3.3 Function Operations 20

3.4 Sequences . . . 20
3.5 Time . . . 21
3.6 Variables . . . 21
3.7 Valuations . . . 22
3.8 Trajectories . . . 23
3.8.1 Prefixes and Concatenation . . . 24
3.9 Hybrid Sequences . . . 24
3.9.1 Prefixes and Concatenation . . . 26
3.10 Hybrid Automata . . . 26
3.11 Trajectory and Hybrid Sequence Ordering . . . 29

4 Hybrid Sequence Enumeration . . . 31
4.1 Enumeration Elements . . . 32
4.1.1 Stimulus Vectors . . . 33
4.1.2 Trajectory Definitions . . . 35
4.1.3 Condition Vectors . . . 37
4.1.4 Hybrid Signals . . . 38
4.2 Hybrid Enumeration . . . 40
4.2.1 Relationship of $\mathcal{E}_{\mathcal{H}}$ to Discrete Enumeration . . . 42
4.2.2 Hybrid Enumeration Process . . . 44
4.2.3 Refinement . . . 52
4.2.4 Requirements Trace . . . 54
4.2.5 Relationship of $\mathcal{E}_{\mathcal{H}}$ to Hybrid Sequences . . . 55
4.2.6 Continuous Properties of Stimulus Vector Sequences . . . 57
4.2.6.1 Mode Invariant . . . 57
4.2.6.2 Point Modes . . . 58
4.3 Theorizable, Specifiable, Implementable, Realizable . . . 59
4.3.1 Infinite Hybrid Sequences . . . 59
4.3.2 Finite Time-Bounded Zeno Sequences . . . 60
4.3.3 Finite Admissible Sequences . . . 60
4.3.4 Finite Time-Bounded Closed Sequences . . . 61

5 Enumeration Hybrid Automata 62
5.1 Enumeration Hybrid Automata 62
5.1.1 Construction Algorithm: $\mathcal{E}_{\mathcal{H}}$ to $\mathcal{A}_{\mathcal{E}}$ 63
5.1.2 Relationship Between $\mathcal{A}_{\mathcal{E}}$ and M (Enumeration Mealy Machines) 68
5.1.3 Construction Algorithm: $\mathcal{A}_{\mathcal{E}}$ to $\mathcal{A}$ 69
5.1.4 Relationship Between $\mathcal{A}_{\mathcal{E}}$ and $\mathcal{A}$ 74

6 Hybrid Specification Implementations 77
6.1 Stateflow Implementations of $\mathcal{A}_{\mathcal{E}}$ 78
6.1.1 Implementing the State Machine Control 78
6.1.2 Implementing the Behavior Function $\mathcal{B}$ 80
6.1.3 Implementing the Condition Function $\mathcal{Q}$ 81
6.1.4 Resettable Timers Implementation 83
6.2 Simulink Fundamentals 85
6.2.1 Elements 85
6.2.2 Simulation 86
6.2.3 Solvers 87
6.2.4 Zero-Crossing Detection 87
6.3 Simulink Implementations of $\mathcal{A}_{\mathcal{E}}$ 88
6.3.1 Implementing the Behavior Function $\mathcal{B}$ 91
6.3.2 Implementing the Condition Function $\mathcal{Q}$ 93
6.4 Alternative Implementations 96

7 Conclusion 97
7.1 Tool Support 99
7.2 Future Research 100

References 102

Appendices 106

A Examples 107

A.1 Resettable Timers 107
A.1.1 Requirements 107
A.1.2 Enumeration Procedure 109
A.1.3 Tabular Enumeration 121
A.1.4 Constructing $\mathcal{A}_{\mathcal{E}}$ 124
A.2 Power Window 130
A.2.1 Requirements 130
A.2.2 Tabular Enumeration 132

Vita 141

List of Figures

2.1 A Dynamical System 7
2.2 Model and Software System Boundary 9
2.3 General System Model 10
2.4 Switching Behavior 12
2.5 Hybrid Behavior 14

3.1 A Hybrid Automaton 27
3.2 A Hybrid I/O Automaton 28

4.1 Process: Requirements to Constructive Enumeration ($\mathcal{E}_{\mathcal{H}}$) 32
4.2 Stimulus Vectors 34
4.3 Hybrid Signals 38
4.4 Stimulus Vector Sequence Extensions 43
4.5 Refinement Tree 53
4.6 Mode Invariant 57
4.7 Instantaneous Transitions 59
4.8 Hybrid Sequence Classes 60

5.1 Process: Enumeration ($\mathcal{E}_{\mathcal{H}}$) to Specification ($\mathcal{A}_{\mathcal{E}}$) to Algebra ($\mathcal{A}$) 63

6.1 Process: Specification ($\mathcal{A}_{\mathcal{E}}$) to Model-Based Design ($\mathcal{M}$) 78
6.2 Stateflow Implementation of the Resettable Timers Example 83
6.3 Resettable Timers 84
6.4 Discrete Output Signal $o[t]$ 84
6.5 Simulink Models 85
6.6 Simulink Implementation of a Signal Definition 86

6.7 Simulink Implementation ($\mathcal{M}$) of $\mathcal{A}_{\mathcal{E}}$ 88
6.8 Simulink Subsystem Components of $\mathcal{M}$ 89
6.9 Simulink Implementation of the Signal Definition Set in $\mathcal{M}$ 92
6.10 Simulink Implementation of the Transition Set in $\mathcal{M}$ 93
6.11 Simulink Implementation of the Condition Vector Set in $\mathcal{M}$ 94
6.12 Simulink Implementation of $\mathcal{Q}$ in $\mathcal{M}$ 95

A.1 Resettable Timers Refinement Tree 123

List of Notation and Conventions

General

$\triangleq$	Is defined as
$\leftarrow$	Takes the value of
A, B, C	Sets
$\varnothing$	The empty set
a, b, c	Elements of sets
$\mathbb{R}$	The set of real numbers
$\mathbb{Z}$	The set of integers
$\mathbb{N}$	The set of non-negative integers
$\mathbb{P}$	The set of positive integers
i, j, k	Indices
$\inf(A)$	The infimum (greatest lower bound) of A
$\sup(A)$	The supremum (least upper bound) of A
$\min(A)$	The minimum element in A
$\max(A)$	The maximum element in A
J	An interval ($J \subseteq \mathbb{R}$)
n	The size of a countable set ($n \in \mathbb{N}$)
$[n]$	$\{1, \ldots, n\}$
f, g	Functions
$dom(f)$	The domain of f
$range(f)$	The range of f
$f \circ g$	f composed with g
$f\|_A$	Domain restrict f to A
$f{\downarrow}A$	Domain restrict the functions in $range(f)$ to A
π_A	Projection function onto coordinate A
ρ	An equivalence relation

HIOA Algebra [13]

T	A time axis
$\mathsf{T}^{\geq 0}$	A non-negative time axis
t	An element of T
I	The set of input actions
O	The set of output actions
E	The set of external actions ($E = I \cup O$)
H	The set of internal actions
L	The set of locally controlled actions ($L = H \cup O$)
A	The set of actions ($A = E \cup H$)
U	The set of input variables
Y	The set of output variables
W	The set of external variables ($W = U \cup Y$)
X	The set of state variables
Z	The set of locally controlled variables ($Z = X \cup Y$)
V	The set of variables ($V = W \cup X$)
$val(V)$	The set of valuations for V
$\mathbf{v}$	A valuation in $val(V)$
Q	The set of states
Θ	The set of initial states ($\Theta \subseteq Q$)
$dtype(v)$	The set of atomic functions describing v's dynamic behavior ($v \in V$)
$type(v)$	The range of all functions in $dtype(v)$ ($v \in V$)
τ, υ	Trajectories ($\tau : J \rightarrow val(V)$ with J left-closed)
$\wp(\mathbf{v})$	A point trajectory ($\tau : [0,0] \rightarrow \{\, \mathbf{v} \,\}$)
$trajs(V)$	The set of all trajectories over the variables in V
$\tau.ltime$	$\sup(dom(\tau))$
$\tau.lval$	For $\tau : [0,t] \rightarrow val(V)$, $\tau(t)$
$\tau.fval$	$\tau(0)$
$\tau.lstate$	$\tau.lval\vert_X$
$\tau.fstate$	$\tau.fval\vert_X$
$\tau^\frown\upsilon$	The concatenation of trajectory τ with υ
$\mathcal{D}$	The set of hybrid automaton transitions
$\mathcal{T}$	The set of hybrid automaton trajectories
$\mathcal{H}$	A hybrid automaton ($W, X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T}$)
$\mathcal{A}$	A hybrid I/O automaton ($\mathcal{H}, U, Y, I, O$)
T	A set of trajectories

$finite(T)$	The bounded (finite) trajectories in T
$closed(T)$	The closed trajectories in T
$open(T)$	The open trajectories in T
$full(T)$	The full trajectories in T
α, β	Hybrid sequences ($\alpha = \tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \ldots$)
$\alpha.ltime$	$\sum_{i \in dom(\alpha)} \tau_i.ltime$
$\alpha.fval$	$head(\alpha).fval$
$\alpha.lval$	For closed α, $head(\alpha).lval$
$C^0(\mathsf{T})$	The set of continuous functions whose domain is T

Hybrid Enumeration

σ, ψ	Sequences
λ	The empty sequence
$head(\sigma)$	The first element in σ
$tail(\sigma)$	σ with the first element removed
$last(\sigma)$	The last element in finite σ
$init(\sigma)$	Finite σ with the last element removed
S	The stimulus set for a discrete sequence-based specification
S^*	The set of finite sequences over S
S^ω	The set of infinite sequences over S
M	An enumeration Mealy machine
S	The set of atomic stimulus vectors
$\boldsymbol{S}$	The set of stimulus vectors
$\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$	Stimulus vectors
$\boldsymbol{\sigma}$, $\boldsymbol{\psi}$	Stimulus vector sequences
B	A block in a partition of a set
χ_B	The atomic characteristic predicate for B
$\mathcal{U}$	The continuous state space
$h_{\mathcal{U}}$	The characteristic predicate for $\mathcal{U}$
h_Q	The characteristic predicate for $Q \subseteq \mathcal{U}$
ω	The illegal discrete response
E_x	An expression set for the variable x
R	The set of atomic condition vectors
R	The set of condition vectors
$\boldsymbol{r}$	A condition vector

$(\varphi, \boldsymbol{r})$	A controllable hybrid signal definition
$\boldsymbol{R}$	The set of controllable hybrid signal definitions
Ω	The illegal hybrid signal definition
$\mathcal{E}$	An enumeration $(\mathcal{E} : \boldsymbol{S}^* \to \boldsymbol{R} \times \boldsymbol{S}^*)$
$\boldsymbol{m}$	a mapping in $\mathcal{E}$: $(\boldsymbol{\sigma}, ((\varphi, \boldsymbol{r}), \boldsymbol{\psi}))$
$\mathcal{E}_{\mathcal{H}}$	A hybrid enumeration: $(\mathcal{E}, V, \Theta)$
$\mapsto$	$\pi_{\boldsymbol{R}} \circ \mathcal{E}$
$\rhd$	$\pi_{\boldsymbol{S}^*} \circ \mathcal{E}$
$\mapsto_{\Phi}$	$\pi_{\Phi} \circ \mapsto$
$\mapsto_{R}$	$\pi_{R} \circ \mapsto$
C_f	The codomain of the function f
s	A signal $(s : \mathsf{T} \to C_s)$
P_x	A set of characteristic predicates for C_x
$\mathcal{S}$	A set of signal names
u, x, y	Continuous-time input, state, and output signals, respectively
d, o	Discrete-time input and output signals, respectively
null	The null action (either input or output)
$\tilde{x}$	A surrogate state variable
$\dot{z}$, $\dot{x}$	The first time derivative of z and x, respectively
e	An expression
φ	A trajectory definition
Φ	The set of trajectory definitions
μ	The mode state variable
$\mathcal{A}_{\mathcal{E}}$	An enumeration hybrid automaton: $(Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$
$\mathcal{Q}$	The condition function of an enumeration hybrid automaton
$\mathcal{B}$	The behavior function of an enumeration hybrid automaton
$\mathcal{M}$	A model-based design

Chapter 1

Introduction

1.1 The Problem

Correctly designing complex software-intensive control systems remains a daunting task despite more than half a century of experience and thousands of successful systems. The problem is that software-intensive product development is still too costly and error prone. Developers are always seeking higher quality, lower cost, and shorter development cycles, usually while increasing functionality and complexity. This is surely a basis for errors, and developers are reluctant to release products that are safety critical or might be subject to expensive recall. Even after extensive and expensive measures are taken, products continue to fail in the field.

The Software Quality Research Lab (SQRL) continues to evolve a constructive approach to transforming an expression of product requirements into a precise specification of intended product behavior: a sufficiently precise specification from which much of the code can be generated automatically and automated testing methods can demonstrate that products are fit for their intended use. The result of this research is extension of sequence-based enumeration to directly handle time, continuity, and nondeterminism in embedded real-time systems. By capitalizing on the step-wise, problem solving enumeration process and its ability to elicit product requirements, models of embedded real-time systems can be produced that function as intended.

Microprocessors with software controlled sensors and actuators have essentially replaced analog control systems. An embedded system is an applied computer system with well-defined hardware and software constraints designed to perform a dedicated function that requires high levels of

dependability [20, p. 5]. An embedded system with timing constraints is a real-time system. With diminishing hardware constraints, the employment of embedded system software has become common place. From transportation and medicine to entertainment and recreation, embedded systems integrate fundamentally with time and the physical world to make our lives more safe, productive, and enjoyable.

1.2 Model-Based Design

No specific method of software development for embedded systems is widely acknowledged as the best or right way to work. Most companies hold all or part of their methodology, process, and tools proprietary. Although there are standards (IEEE, ISO, IEC, and others) and ratings (CMM, CMMI, for example), these are not uniformly respected or practiced. While various practices are generally recognized as good, or better than others, there is not a codified set of best practices and even if such practices existed, they would not be strictly followed.

Formal methods are used in some organizations, but are considered too expensive for the value received in all but the most safety-critical applications. The software engineering workforce capable of using formal methods is quite limited. Rigorous methods are somewhat more cost-effective, and sequence-based specification is considered rigorous. The most effective organizations follow a locally evolved process that combines elements of formal and rigorous methods with good tool support. Code generation and automated testing (hardware-in-the-loop, software-in-the-loop, model-in-the-loop) are typical in such organizations.

Model-based design has emerged as an effective way to prototype software-intensive systems and analyze system function through simulation. Automatic code generation and visual design components are also strengths of model-based design tools such as MATLAB/Simulink, LabVIEW, and others. These strengths help resolve discrepancies in requirements, facilitate verification and validation, and decrease overall development time. However, rapid design methods should supplement, not replace, the fundamentals of good design. As more features are integrated into the design at a faster pace, we must be sure unintended behavior is not introduced and the final result agrees with the product designer's intentions.

Software and physical device development are separable; however, integration of system com-

ponents after they have been developed separately can be problematic. Co-engineering improves this situation. A systems perspective taking into account the context of use enhances the dependability assurance case; dependability does not rely solely on the software component [8, p. 55]. During physical device design and manufacture, essential details are exposed that are needed to engineer correct control software. During software development, physical device interface requirements are also revealed. System prototyping using model-based design tools allows analysis of complete system performance without incurring the costs of physical component construction. In highly dependable systems, the embedded processor and its associated software seamlessly interact with the physical devices it monitors and controls. Getting one portion of the system correct is not enough; the entire system must operate correctly as a whole.

A model-based design is a graphical data flow program whose structure has roots in engineering block diagrams and signal flow graphs [15]. Model blocks are composed to create subsystems. Instead of line-by-line program execution, data flow programs evaluate a set of subsystems which replace blocks of code at each time step producing new output data for each subsystem; subsystem output is a function of the input from the previous time step and the time between steps. A model-based design simulation is a sequence of subsystem evaluations at critical time steps. The block diagram paradigm opens system design to a more diverse group of developers; however, more developers could exacerbate the inconsistencies within a model.

A model-based design simulates discrete and continuous behavior operating together; models have the properties of hybrid and switching systems. An event that occurs instantaneously in time is an example of discrete behavior; the finite or countably infinite set of event times makes the behavior discrete. Behavior that evolves over an uncountable interval of time is continuous. The set of states that characterize continuous system behavior may or may not be countable.

Sequence-based specification proceeds from an enumeration of sequences over a well-defined set of software inputs mapping each sequence to a response. The specification method is well suited to the design of embedded control software. Model-based designs include software, models of physical devices, and the connections between these components. This research complements model-based design with sequence-based methods by inserting a systematic specification process between requirements and model development. This will help engineers get the design right.

1.3 Results and Contribution

Discrete sequence-based specification cannot capture the continuous aspects of hybrid and switching systems. For this, we integrate continuous functions into our extended specification. To specify complete system models, identification of internal events is also important. We introduce internal (autonomous) events using internal system variables into the stimulus set. Real-time system properties are made explicit in the specification by resettable clock variables that advance at the same rate as real time and internal events based on clock values. Defining the stimulus set for a discrete sequence-based specification requires appropriate abstractions be made prior to system specification; sometimes the abstraction is not obvious. By making continuous system variables and the functions that describe their behavior explicit in the specification, internal and external stimulus refinement based on these variables can be used when necessary.

We extend existing sequence-based methods to address the characteristics of hybrid and switching systems with the following contributions:

- The elements needed to produce hybrid specifications with real-time properties are prescribed.
- A hybrid enumeration is defined and the constructive enumeration process is presented.
- A specification called an enumeration hybrid automaton is defined and an algorithm for its construction is given.
- The mathematical basis for our specification method is the hybrid I/O automaton (HIOA) [9, 13]. A constructive algorithm that produces a HIOA from our specification is presented and then used to prove a hybrid enumeration satisfies the HIOA axioms.
- Finally, using the enumeration hybrid automaton, algorithms to construct model-based designs in the Simulink and Stateflow design languages are presented.

Before establishing these results, several system models are explored and those characteristics that lend themselves to specification by sequence enumeration are identified. Next, we introduce the theory of hybrid I/O automata [9,13] and use this model as the mathematical foundation for our specification method. Finally, the definitions, enumeration process, and construction algorithms are presented. In summary, a process is defined for constructing complete, consistent, and traceably correct designs of hybrid and switching systems.

Chapter 2

System Characteristics

> A system is defined by a set of distinctive relationships among a group of components that interact with one another and their environment through the exchange of energy, matter, and/or information [17].

We target for specification systems that include interacting human users, control software, subsystems, and models of physical devices. The dynamic behavior of physical devices depends on time. The interaction between system components is often time sensitive as well. The state of a system in time is the minimal amount of historical system information required to predict future system behavior [27]. A hybrid I/O automaton (HIOA) is a system model that incorporates time, continuous dynamic functions, interactions in the form of stimuli and responses, and varying forms of nondeterminism [13]. Specially formed sequences are used to describe these behaviors in a HIOA. The framework for our specification method is a HIOA because it is sequence based and it includes the structures necessary to address behavior missing in our discrete method. In this chapter, systems are classified based on the characteristics that lend themselves to specification using sequences. They are discussed in the context of HIOA, a general definition for a dynamical system.

2.1 Sets and Number Systems

A set with finite or countably infinite cardinality is discrete. We use capital letters (e.g., A, B) to denote sets and lower case letters (e.g., a, b) to denote elements of sets. We use $\subseteq$ to denote the

subset relation and $\subsetneq$ to denote the proper subset relation. The relative complement of set B in A is $A - B$. We use $\triangleq$ to show two objects are equivalent by definition. $\mathbb{R}$ is the set of real numbers. $\mathbb{Z}$ is the set of integers; $\mathbb{N} = \{\, a \in \mathbb{Z} \mid a \geq 0 \,\}$; $\mathbb{P} = \mathbb{N} - \{\, 0 \,\}$. For all $n \in \mathbb{N}$, $[n] = \{\, 1, 2, \ldots, n \,\}$; $[n] \subseteq \mathbb{P}$ and $[0] = \varnothing$. $\mathbb{P} \subsetneq \mathbb{N} \subsetneq \mathbb{Z} \subsetneq \mathbb{R}$.

2.2 Relations and Functions

A relation on sets A and B is any subset of $A \times B$. For $(a, b) \in A \times B$, a is the first coordinate of the pair and b is the second coordinate of the pair. Relations are denoted using appropriate symbols (e.g., $\leq$). A total function from set A to set C, denoted $f : A \to C$, is a relation where for each $a \in A$ there exists one and only one $c \in C$ that satisfies $(a, c) \in f$. A is the domain of f, denoted $dom(f)$. C is the codomain of f. When needed, we use C_f to distinguish both a codomain and the function. The range of f, or $range(f)$, is the subset of C whose elements appear as the second coordinates of the ordered pairs of f. A partial function is a function where elements from a proper subset of A are found in the first coordinates of the pairs that make up f. We refer to this set as the domain of the partial function. Functions are denoted with the lower case letters f, g. When codomain C contains n-tuples for $n > 1$, bold face type is used, i.e., $\boldsymbol{f}$, $\boldsymbol{g}$. Unless noted otherwise, functions are total: $dom(f) = A$.

2.3 Dynamic Behavior

> An evolution rule that defines a *trajectory* as a function of a single parameter (time) on a set of states (the state space) is a *dynamical system* [16, p. 105].

Following [13, p. 17], we define time, denoted T, as a subgroup of $(\mathbb{R}, +)$. Although negative time is counter-intuitive it is necessary in the mathematical description of dynamical system behavior. For our purposes, we differentiate between an evolution rule and a trajectory. A trajectory is a function of time, describing how a system behaves between stimuli. An evolution rule is a function of time that describes complete system behavior; the result of our extended enumeration is an evolution rule. In other words, an evolution rule incorporates trajectories and how system behavior changes after each stimulus. The evolution rule may be deterministic or stochastic. Our goal is

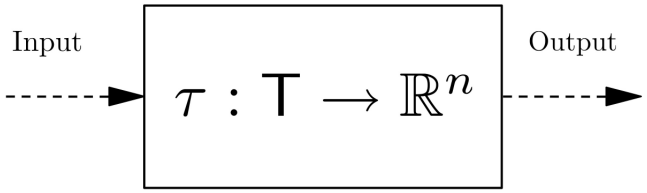


Figure 2.1: A Dynamical System

to build deterministic models in Simulink (or similar systems) that incorporate both software and models of physical devices. A deterministic evolution rule establishes a unique consequent state from every other system state [16, p. 106]. We focus primarily on deterministic rules.

Figure 2.1 depicts a dynamical system with trajectory τ. The trajectory's domain is time and the codomain is an n-dimensional space over the real numbers. In Figure 2.1, the arrows represent potential system interaction with the environment.

To provide sufficient detail to implement a specification as a model-based design, internal behavior is included explicitly in a system's evolution rule. A HIOA includes definitions for trajectory components called dynamic types; these definitions can be used for model-based design implementation. In this respect, we break from discrete sequence-based specification which transforms sequences with specific properties into a finite set of states upon completion of sequence enumeration. In the extended sequence-based specification method presented here, we examine, augment, and map state information as sequence enumeration progresses using trajectories. Additionally, internal stimuli are explicitly discovered and defined whereas these stimuli would be inferred abstractions in discrete sequence-based methods. Since internal and external details are included as part of the specification, a black box perspective can be obtained by restricting the final representation to an external set of variables and an external stimulus set.

2.4 Signals

A *signal* is a function of one or more independent variables. Signals convey information and are not limited to input and output; internal system behavior can also be characterized as a signal [14, p. 3]. When time is a signal's single independent variable, it is also a trajectory. Signals may be stochastic but the function definition suits our deterministic aims. Two types of time-based signals are fundamental in the discussion of hybrid systems.

Definition 2.1. *Continuous-time signal* s is a total function $s : \mathsf{T} \rightarrow C_s$ [14, p. 6].

We denote signals using lower-case script letters: u denotes a continuous-time input signal, y denotes a continuous-time output signal, and x denotes a continuous-time state (internal) signal. u, y, and x are also referred to as input, output, and state variables respectively. $u(t)$ is the value of continuous-time input signal u at time $t \in \mathsf{T}$; the parentheses in $u(t)$ distinguish signal u as continuous-time (rather than discrete-time).

Definition 2.2. *Discrete-time signal* s is a partial function $s : \mathsf{T} \rightarrow C_s$ with countable $dom(s)$. For $t \notin dom(s)$, $s[t] = null$ [14, p. 6].

$o[0]$ is the value of discrete-time output signal o at $t = 0$; the brackets in $o[0]$ distinguish signal o as discrete-time. Discrete-time signals may result from sampling a continuous signal at fixed or variable time increments. The range of a discrete-time signal is not restricted to a finite set.

2.5 System Boundary

The system boundary encapsulates those system components that define its state and clearly identifies signals in the environment that contribute to overall system behavior. Software is produced to accomplish a task on a digital computer; an input-output relationship exists and the number of states is finite. A model of a physical system, on the other hand, may operate autonomously within its environment over an arbitrary state space. Furthermore, a model of a physical system may not produce output. Consider an ideal spring. For this example, the arrows in Figure 2.1 are eliminated and τ models the spring's oscillation over time. $\tau(t)$ describes the spring's motion and position as a tuple. The values in this tuple define the spring's state at time t. The respective scopes of software and model designers are different. A model's system boundary may identify both continuous-time and discrete-time signals and the system specification must be able to define these details.

Embedded system software must be capable of processing continuous-time input signals. A quantizer performs the vital task of collecting and interpreting continuous-time input signals at the software system boundary [12, p. 11]. After the signal has been quantized, the software views each input as a discrete-time signal. The quantizer may produce the discrete-time signal by periodically

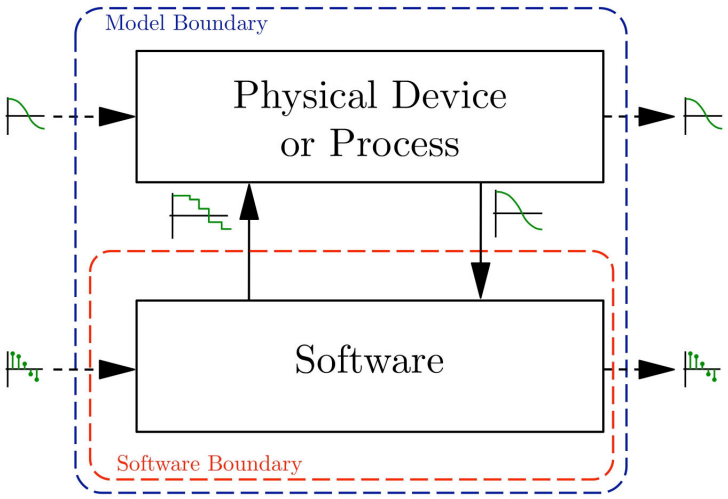


Figure 2.2: Model and Software System Boundary

interpreting a set of continuous-time signals. This interpretation must be sufficiently precise in the context of the application or failures may occur.

In a discrete sequence-based specification the values of continuous-time signals are represented with abstractions. For example, consider a system with two independent timers. Two events are defined, one for each timer reaching a specified value. The case when both timers reach their specified values together, however, is not represented. How to quantize each individual timer is clear from the requirements, and the implemented specification operates correctly when each timer reaches its specified threshold individually. However, when the timers reach their respective thresholds simultaneously, the system fails to perform a critical function. Recognizing the special case might be overlooked using abstraction. We provide a mechanism to address this problem in our process.

When the boundary includes software and models of physical devices or processes, the specification must include definitions for trajectories and a means to quantize continuous-time signals. Figure 2.2 illustrates the relationship between the software system boundary and the model system boundary.

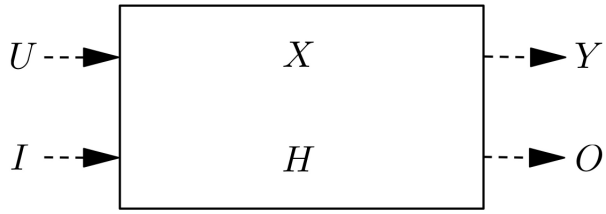


Figure 2.3: General System Model

Figure 2.3 presents a high-level view of the hybrid I/O automata model. The details of this model are presented in Chapter 3. This high-level view is consistent with Figure 2.2. The sets U, X, and Y contain the signal variables (u, x, and y respectively) that detail the behavior of physical devices and their interaction with the environment.

The sets I, H, and O detail discrete or instantaneous system behavior. Elements in I and H are discrete stimuli. Each stimulus in I is produced in the environment. H is the set of autonomous system stimuli; each element of this set is generated within the system boundary. Elements of I in conjunction with quantized data from the continuous-time signals in U and elements of H determine system function. The system responds to each stimulus with a discrete output or a change in continuous behavior. O is the set of outputs produced by the system at discrete instants in real time.

2.5.1 Internal Dynamics

The compelling reason to use discrete sequence-based specification is discovery of system state, transitions between states, and the output generated by each transition. The discovery results from serially examining a sufficient number of input sequences one at a time. The result is a precise mathematical function definition of the system that agrees with the final requirements document(s).

System trajectories are models of established physical laws defined by differential equations. Enumerating trajectories is unnecessary; models of their behavior are known. However, many systems use multiple trajectory definitions to describe overall system behavior or trajectory values may change instantaneously as a result of external or internal influence. Sequence enumeration will account for such behavior. For these reasons a sequence-based approach to writing model specifications is appropriate and useful.

2.6 System Types

The focus of this research is transforming requirements (in any form) into precise dynamical system specifications using sequences. Specifically, we find a sequence-based approach to be useful in defining discrete, switching, and hybrid system characteristics while ensuring requirements are complete.

2.6.1 Discrete Systems

A discrete event system may be described as a Mealy machine, a tuple [7, p. 43]:

$$M = (Q, \Sigma, \Delta, \delta, \nu, q_0). \tag{2.1}$$

Q is a finite set of system states; q_0 is the start state. Σ and Δ are the input and output alphabets, respectively. In terms of our signal discussion, Σ is the range of a discrete-time input signal; the same view holds for the output alphabet Δ. The function δ governs how state changes. The function ν governs how output is produced. There is no explicit definition of time in a Mealy machine; time is simply sequential. M could repeatedly process a fixed sequence from Σ^* starting at q_0 and produce the same result regardless of the time between elements in the input alphabet sequence.

2.6.2 Continuous Systems

A continuous system with input $\mathbf{u}$, state $\mathbf{x}$, and output $\mathbf{y}$ is defined by the equations [12, p. 4]:

$$\dot{\mathbf{x}} = \boldsymbol{f}(\mathbf{x}(t), \mathbf{u}(t), t), \quad \mathbf{x}(0) = \mathbf{x}_0 \tag{2.2}$$

$$\mathbf{y} = \boldsymbol{g}(\mathbf{x}(t), \mathbf{u}(t), t). \tag{2.3}$$

$\dot{\mathbf{x}}$ denotes the first derivative of $\mathbf{x}$ with respect to time. $\boldsymbol{f}$ is a vector field in $\mathsf{T} \times \mathbb{R}^{|\mathbf{x}|}$ produced by plotting each vector with initial point $(t, \mathbf{x})$ and terminal point $(t{+}1, \mathbf{x}{+}\dot{\mathbf{x}})$ in Euclidean space. Since $\dot{\mathbf{x}}$ defines the instantaneous rate of change at all points $(t, \mathbf{x})$, $\boldsymbol{f}$ describes all possible trajectories. Given an initial point, $\mathbf{x}_0$, a specific trajectory can be determined using $\boldsymbol{f}$. Meanwhile, $\boldsymbol{g}$ describes

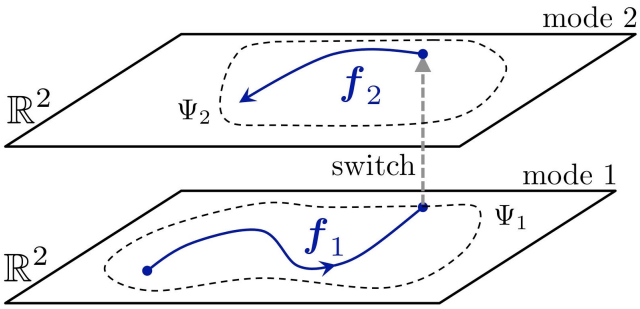


Figure 2.4: Switching Behavior

continuous system output as a function of state, input, and time. One purpose of enumeration is to discover a function. Equation 2.2 usually describes a set of known physical laws; enumeration is not required.

2.6.3 Switching Systems

The evolution rule of a switching system is defined using two or more vector fields to describe trajectories. State in a switching system includes a discrete component called system mode and a continuous component where a trajectory evolves over time [19, p. 6]. Each field, $\boldsymbol{f}$, distinguishes a mode. In an autonomous switching system, a threshold between two adjacent regions in the system's state space defines where system behavior changes fundamentally without influence from the environment. In a controlled switching system, behavior changes are based on input changes. A switching event is defined by properly identifying the threshold or input that triggers the switch and then defining a new vector field to describes how trajectories evolve over time after the switching event. Figure 2.4 illustrates switching in a two-dimensional continuous state space with two modes. The system's evolution rule (autonomous switching) is defined using a piecewise vector field and definitions for adjacent regions, Ψ_1 and Ψ_2, in the continuous state space [12, p. 8]:

$$\dot{\mathbf{x}} = \begin{cases} \boldsymbol{f}_1(\mathbf{x}(t), \mathbf{u}(t), t), \;\; \mathbf{x}(0) = \mathbf{x}_0 \text{ and } \mathbf{x} \in \Psi_1 \\ \boldsymbol{f}_2(\mathbf{x}(t), \mathbf{u}(t), t), \;\; \mathbf{x}(0) = \mathbf{x}_0 \text{ and } \mathbf{x} \in \Psi_2. \end{cases} \tag{2.4}$$

Consider mode 1 in Figure 2.4. For a deterministic evolution rule only one vector field can be associated with each mode. If this were not the case, at each point in Ψ_1 more than one vector could describe a trajectory's direction and velocity from that point. The dotted line perpendicular

to mode 1 in Figure 2.4 illustrates a switching event. This perpendicular transition indicates that continuous state remains constant through the switch. If a transition includes a discontinuity in the continuous state space, the switching system is also hybrid.

Enumeration supported by requirements can be used to identify switching events. System behavior before and after a switching event can be defined using Equations 2.2 and 2.3.

2.6.4 Hybrid Systems

The exact definition of a hybrid system is not settled: "No common definition of a hybrid system is available." [12, p. 3] When system behavior does not fit nicely into either discrete system theory or continuous system theory we must use a different method of analysis [12, p. 4]. We adopt Hypothesis 1 from [12, p. 5] to define hybrid behavior:

> Consider a dynamical system subject to some continuous input $\mathbf{u}$. The basic hybrid phenomenon is a combination of continuous state changes and abrupt state jumps.

In a hybrid system, the rate at which a continuous variable changes value at certain instants in real time is undefined; these times are determined by boundaries in the system's state space. When system state reaches the boundary it is no longer continuous, jumping to a new state from which system state continues to evolve. Hybrid behavior is illustrated in Figure 2.5. When the trajectory reaches the boundary of Ψ_1, its state changes discontinuously and trajectory evolution continues. The vector field $\boldsymbol{f}_1$ governs trajectory behavior before and after the jump. Trajectories in a pure switching system change their fundamental behavior but remain continuous. In a hybrid system, discontinuities exist with or without a fundamental change in trajectory behavior after the state jump.

As an example of hybrid behavior, consider a state variable representing a resettable clock. The vector field describing the trajectory of a clock variable that advances at the same rate as real time is always $f(t) = 1$. If the clock variable resets when its value reaches 100, the clock's trajectory exhibits autonomous hybrid behavior.

The states and stimulus events that cause hybrid jumps are critical to identify in a hybrid system specification and enumeration does identify them.

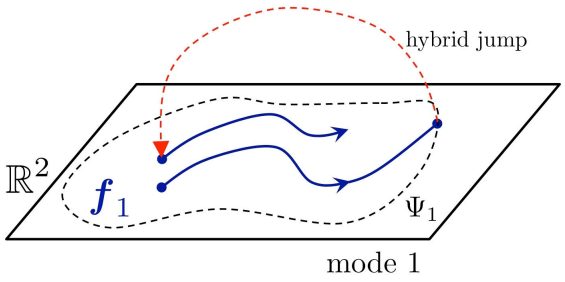


Figure 2.5: Hybrid Behavior

2.7 Discrete Sequence-Based Specification

The objective of the discrete sequence-based specification method is to discover a special Mealy machine that correctly models the requirements provided. The discovery process proceeds from a finite enumeration of stimulus sequences. Alternating between requirements and enumerated sequences, the process culminates in a complete, consistent, and traceably correct specification that matches an improved requirements document.

Let S be the finite set of system stimuli generated in the environment and R be the set of system responses. R always includes two special responses: ω and *null*. ω indicates that the associated stimulus sequence is physically impossible. *null* represents the absence of an observable output.

An enumeration is a partial function mapping a subset of S^* to a set of pairs. $(S^*, <)$ is a total order where $<$ is defined first by sequence length and then lexicographically. The first element of a pair in the range of the enumeration mapping is the response. The second element of the pair is an element in the partial function's domain; either a prior sequence by $<$ or the sequence itself. In the later case, we say the sequence is unreduced [10, Definition 4.6]. If the sequence is unreduced and illegal, it is not extended [10, Axiom 5]. If the sequence is unreduced and legal, the sequence describes unique future behavior and is extended by every element in S [10, Axiom 7]. Each extension is examined and associated with a response-sequence pair.

Two unreduced sequences σ and ψ are distinguishable if one is legal and the other is illegal or when σ and ψ are extended by the same stimulus and either their response is different or their reductions are themselves distinguishable [10, Definition 4.9]. σ and ψ are equivalent if they are not distinguishable [10]. σ can be reduced to ψ if they are equivalent and $\psi < \sigma$ [10, Definition 4.6]. A stimulus sequence is reducible if future stimulus extensions produce behavior identical to

the future behavior of an enumerated sequence earlier in the order. A missed equivalence by itself does not produce an incorrect enumeration [23, p. 422]. In this case, the enumeration will not be minimal, but that is preferred over making an incorrect reduction. Incorrect reductions produce incorrect behavior. The above definition of equivalence is Mealy equivalence; it is based on future behavior.

The illegal response ω is reserved for sequences that are impossible based on our knowledge of the current requirements and the physical world. Implicitly, illegal sequences identify trap states whose subsequent extensions remain illegal. A specification writer must use caution when classifying a sequence as illegal. Incorrect information about the system's operation and environment may lead to incorrect specification.

> When a system's environment is not well understood, requirements are hard to get right. A distinction must be made between truly impossible sequences and pathological events that are simply unexpected. Such unexpected sequences can be handled in two ways [4, p. 338]:
>
> 1. Define a response that puts the system in a safe state. Future legal sequence extensions produce appropriate legal behavior.
> 2. Define a response that halts, but does not damage the system; future sequence extensions are ignored. The user must correct the error before restarting the system.

Enumeration is complete when all sequences in the domain of $\mathcal{E}$ are reduced, legal and unreduced, or classified as illegal [10, Lemma 4.14]. When complete, the function

$$\mathcal{E} : S^* \to R \times S^* \tag{2.5}$$

provides sufficient information to produce an enumeration Mealy machine [10, Definition 5.1]. Each unreduced sequence in $\mathcal{E}$ represents a class of sequences in S^* and a state in the machine definition. A sequence is canonical if it is both unreduced and the least indistinguishable sequence in the order [10, Definition 4.10]. A canonical sequence truly characterizes a sequence of stimuli whose follow-on behavior is unique; there is no sequence earlier in the order that also characterizes this

future behavior. Canonical sequences are important because they represent the minimal information needed to specify a system correctly. Using canonical sequences, function composition, and projections, $\mathcal{E}$ can be transformed into an enumeration Mealy machine.

[23] provides an overview of the sequence-based specification process and [10] provides the formal axiomatized specification method along with algorithms to modify enumerations and convert them into alternative forms. For consistency we adopt the notation from [10] where possible.

Some code and model-based designers are hesitant to use sequence-based specification because it only handles explicitly the discrete part of the process and requires abstractions. They want to specify systems that include control software and the trajectories that describe the behavior of interacting physical devices. Extending the discrete enumeration process to include these features will make explicit the details needed for model implementation.

2.8 Applicability

Complex hybrid and switching systems can be enumerated using the systematic method outlined in successive chapters. When continuous system behavior can be defined using a system of first-order differential equations and functions of time, enumerating sequences that include both signals and events generates the series of questions necessary to reveal

- Internal autonomous system behavior,
- New trajectory definitions following a switch,
- Discrete system output,
- System state following a hybrid jump, and
- System mode changes.

By combining the fundamental characteristics of discrete, switching, and hybrid dynamical systems with a specification method well-suited to transform requirements into computer implementations, we can produce system models that are correct by construction. These provisions will allow specification writers to explicitly discover the interfaces between software and physical devices, resolve nondeterminism, and include descriptions of continuous real-time behavior. Using this specification, both model and software can be extracted to aid the embedded system development process.

Chapter 3

Hybrid System Mathematics

3.1 Introduction

There is a significant body of literature produced by computer scientists, mathematicians, and control engineers on specifying hybrid and switching systems. Some of the more well-known theories introduced in the computer science community are [1, 2, 6, 9, 13]. We form our specification method using complementary features from these works but have found the theory presented in [9, 13] to be especially well suited to specification by enumeration.

Timed automata were introduced in [2] and provide a foundation for specifying and analyzing real-time system behavior. This model is central to the Uppaal tool suite for verification of real-time systems [3]. A timed automaton describes timed behavior as a pair of possibly infinite sequences, (σ, τ), called a timed word. σ is a sequence over an alphabet, and τ is a monotonically increasing sequence of positive reals modeling time advance. The state of a timed automaton is a combination of values from a finite set of resettable clock variables advancing at the same rate as real time and a finite set of automaton modes. Mode-to-mode transitions are defined using an action alphabet similar to the stimulus and response sets combined and a set of predicates defined on the clock variables. The trajectories of a timed automaton are the advancing clocks.

More complex trajectories can be modeled by generalizing the expressions used to describe how a variable evolves over time. In the hybrid automata of [1], state is defined as a location and the values assigned to a set of data variables. The set of locations is finite; each data variable can assume an arbitrary number of values. A function called an activity maps the positive reals to data

states thereby specifying continuous system evolution. An activity is the same as a trajectory. As an example, the local behavior of trajectories for each data variable in a linear hybrid automaton is described using a constant vector field whose rate possibly changes between locations. The timed automata are a strict subset of the linear hybrid automata. A transition relation determines when the automaton may change location, but it does not force a transition. An exception set defines local data states that cannot be entered. An automaton location-to-location transition is required prior to its entering the data states in the exception set. The complement of the exception set is the invariant set. Activity continues within a location's invariant set until a transition or an exception occurs. In combination, the invariant set, exception set, and the transition relation allow varying degrees of nondeterminism to be modeled. A hybrid automaton run is defined as a sequence of 5-tuples that captures both instantaneous discrete transitions and the continuous evolution of the data variables. In [6] the theory introduced in [1] is expanded to include more complex trajectories.

The hybrid automata presented in [9, 13] are based on domain theory [24]. The theory of hybrid automata builds upon previous hybrid system theory and also uses a sequence as the central behavior descriptor. The behavior of each variable in the sets U, X, and Y(see Figure 2.3) is described as a signal, a function whose domain is T. Combinations of these functions define trajectories through an n-dimensional state space where n is the number of state variables in X. Using a set of actions equivalent to the stimulus and response sets combined, a hybrid sequence is used to describe system execution by alternating actions and trajectories. Well-defined operators are provided throughout the theory to facilitate abstraction and isolate system details. Finally, the prefix ordering relations of trajectories and hybrid sequences facilitate addressing continuity, so called "Zeno behavior", and infinite sequences from a domain-theoretic viewpoint.

The theory in [9, 13] approaches hybrid system verification from a high level of abstraction called a specification using induction to prove specific system properties without resolving every system detail. These properties cascade to system refinements called implementations as long as an implementation relation can be established. Our goal is to construct an implementable hybrid specification directly from requirements. To accomplish this goal, sequence-based specification is adapted to the hybrid sequences that define hybrid I/O automata (HIOA) behavior. The adapted process is presented in Chapter 4. The foundations of hybrid automata found in [9,13] are summarized in this chapter for completeness. The final hybrid sequence-based specification satisfies the

axioms of (discrete) sequence-based specification theory [10] and the axioms of the Lynch HIOA theory.

3.2 Intervals

Definition 3.1. Let $J \subseteq \mathbb{R}$ be nonempty and $\leq$ denote the less than or equal to relation:

1. b is an upper bound (lower bound) for J iff $b \in \mathbb{R}$ and $\forall a \in J,\ a \leq b$ ($b \leq a$, respectively.) If there exists an upper bound (lower bound) for J, J is bounded above (below).

2. c is the supremum (infimum) of J iff c is an upper bound (lower bound) for J and for all upper bounds (lower bounds) $b \in J$, $c \leq b$ ($b \leq c$, respectively.) The supremum, denoted $\sup(J)$, is also named the least upper bound. The infimum, denoted $\inf(J)$, is also named the greatest lower bound.

Definition 3.2. $J \subseteq \mathbb{R}$ is *convex* iff $\forall a, b \in J$, the line segment from a to b is a subset of J [28, p. 353].

Definition 3.3. An *interval*, $J \subseteq \mathbb{R}$, is a convex set described and denoted as follows [28]:

1. The following intervals are closed:
   (a) $[a, b] \triangleq \{\, c \in \mathbb{R} \mid a \leq c \leq b \,\}$.
   (b) $[a, \infty) \triangleq \{\, c \in \mathbb{R} \mid a \leq c \,\}$.
   (c) $(-\infty, b] \triangleq \{\, c \in \mathbb{R} \mid c \leq b \,\}$.

2. The following intervals are open:
   (a) $(a, b) \triangleq \{\, c \in \mathbb{R} \mid a < c < b \,\}$.
   (b) $(a, \infty) \triangleq \{\, c \in \mathbb{R} \mid a < c \,\}$.
   (c) $(-\infty, b) \triangleq \{\, c \in \mathbb{R} \mid c < b \,\}$.

3. The following intervals are half-open or equivalently half-closed:
   (a) $[a, b) \triangleq \{\, c \in \mathbb{R} \mid a \leq c < b \,\}$.
   (b) $(a, b] \triangleq \{\, c \in \mathbb{R} \mid a < c \leq b \,\}$.

4. a and b are called endpoints.

5. An interval is bounded or finite if $-\infty < a \leq b < \infty$.

$\mathbb{R}$ is considered both open and closed. A left-closed interval is either right-open, $[a, b)$, or right-closed, $[a, b]$. If J is left-closed, $\min(J)$ is the minimum element in J; likewise, for right-closed J, $\max(J)$ is the maximum element in J. An interval is called degenerate if its endpoints are equivalent. A closed degenerate is a point and an open degenerate is the empty set.

3.3 Function Operations

Definition 3.4. For function $f : A \to B$ and set C, $f|_C$ is a function g such that $dom(g) = dom(f) \cap C$ and $\forall c \in dom(g),\ g(c) = f(c)$. f is said to be restricted to C. For singleton sets, we abuse notation and let $f|_a$ be the same as $f|_{\{\, a \,\}}$.

Definition 3.5. Let A and C be sets and let B be a set of functions. For the function $f : A \to B$, $f \downarrow C$ is a new function g such that $dom(g) = dom(f)$ and $\forall a \in dom(g),\ g(a) = f(a)|_C$. For singleton sets, we let $f \downarrow a$ be the same as $f \downarrow \{\, a \,\}$.

Definition 3.6. Functions f and g are compatible if $f|_{dom(g)} = g|_{dom(f)}$. If f and g are compatible then $f \cup g$ is a function. Let F be a collection of compatible functions; $\bigcup_{f \in F} f$ is a function.

Definition 3.7. For sets A, B, and relation $D \subseteq A \times B$, the function $\pi_A : D \to A$ defined by $\pi_A((a, b)) = a$ is the projection on the first coordinate of the pairs in D. For clarity, we drop parentheses surrounding the pair when it is used as an argument for π_A.

Definition 3.8. For the finite collection $\{\, A_1, A_2, \ldots, A_n \,\}$, the set of n-tuples $D \subseteq A_1 \times \ldots \times A_n$, and $i \in [n]$, the function $\pi_{A_i} : D \to A_i$ defined by $\pi_{A_i}((a_1, \ldots, a_i, \ldots, a_n)) = a_i$ is the projection on the i^{th} coordinate of the tuples in D. Again, we drop the parentheses surrounding the n-tuple function argument.

3.4 Sequences

A sequence can be used to describe particular system behavior. From a finite enumeration of sequences over a finite set of events, we produce an automaton that defines complete system behavior as an infinite set of sequences.

Definition 3.9. A sequence over S is the function $\sigma : [n] \to S$ for some $n \in \mathbb{P}$. The length of σ, denoted $|\sigma|$, is n. If $dom(\sigma)$ is countably infinite, σ is an infinite sequence; otherwise, it is finite. $\lambda : \varnothing \to S$ defines the empty sequence.

Concatenation of finite sequence σ with sequence ψ is denoted $\sigma^\frown\psi$. If $\sigma = \psi|_{dom(\sigma)}$, we say σ is a prefix of ψ and denote this relationship $\sigma \leq \psi$. Given any sequence σ, $head(\sigma)$ represents the first element of σ and $tail(\sigma)$ represents σ with $head(\sigma)$ removed. If σ is finite, $last(\sigma)$ is σ's last element, and $init(\sigma)$ denotes σ with its last element removed.

The set of finite sequences over S is denoted S^* while the set of infinite sequences over S is denoted S^ω. We only enumerate finite sequences; therefore, we do not address infinite sequences. This is discussed in more detail in Section 3.11.

3.5 Time

Dynamic system behavior varies over time. From Chapter 2, time axis T is a subgroup of $(\mathbb{R}, +)$; $\mathsf{T}^{\geq 0} \triangleq \{\, t \in \mathsf{T} \mid t \geq 0 \,\}$. Intervals are used to define subsets of T.

Definition 3.10. For $K \subseteq \mathsf{T}$ and $t \in \mathsf{T}$, $K + t \triangleq \{\, t' + t \mid t' \in K \,\}$.

Definition 3.11. Let $K \subseteq T$, f be a function, and $dom(f) = K$. $f + t$ is the function with domain $K + t$ such that $\forall t' \in K + t$, $(\,(f + t)(t') = f(t' - t)\,)$.

By using T and not $\mathsf{T}^{\geq 0}$ in the above definitions, we can shift the interval K in either direction along T. In Definition 3.11, $f + t$ represents signal f shifted by t.

3.6 Variables

We use V to denote the universal set of variable names. In Section 2.5, the variable sets of the HIOA model were introduced. Each variable name represents a continuous-time signal.

Definition 3.12. A variable name $v \in \mathsf{V}$ has a static component and a dynamic component.

1. The static component $type(v)$ is a nonempty set of values.

2. The dynamic component $dtype(v)$ is a set of functions satisfying the following closure properties. The domain of each function in $dtype(v)$ is a left-closed interval of T; its codomain is $type(v)$.

   (a) Closure under time-shift:
   For each $f \in dtype(v)$ and $t \in \mathsf{T}$, $f + t \in dtype(v)$.

   (b) Closure under subinterval:
   For each $f \in dtype(v)$ and each left-closed interval $J \subseteq dom(f)$, $f|_J \in dtype(v)$.

   (c) Closure under pasting:
   Let $f_0, f_1, f_2, \ldots$ be a sequence of functions in $dtype(v)$ such that, for each nonfinal index i, $dom(f_i)$ is right-closed and $\max(dom(f_i)) = \min(dom(f_{i+1}))$. Then $f(t) \triangleq f_i(t)$, where i is the smallest index such that $t \in dom(f_i)$, is in $dtype(v)$.

Subinterval closure includes all functions derived through domain restriction. This includes degenerate domains and the function with the empty domain. Pasting closure produces piecewise functions that may contain discontinuities. Pasted functions are continuous from the left; the function derived using the pasting property is defined at each time where two adjacent functions in $dtype(v)$ are pasted by the left function. A set of functions closed under pasting includes functions that exhibit hybrid or switching behavior as described in [12].

Two specific variables types are described in [9, p. 18]. A variable is discrete when its *dtype* is the pasting closure of the set of all constant functions. $C^0(\mathsf{T})$ denotes the set of all continuous functions whose domain is time axis T. An analog variable is obtained by closing $C^0(\mathsf{T})$ under pasting.

3.7 Valuations

Definition 3.13. Given $V \subseteq \mathsf{V}$, $\mathbf{v}$ is a valuation for V and a function that maps each $v \in V$ to a value in $type(v)$.

The set of all possible valuations for V is denoted $val(V)$. If $V = \{ v_1, v_2 \}$ is a set of variable names and for each $i \in [2]$, $a_i \in type(v_i)$, then $\mathbf{v} = \{ (v_1, a_1), (v_2, a_2) \}$ is a valuation. The restriction operator isolates value assignments in a valuation. If $v_1 \in dom(\mathbf{v})$, $\mathbf{v}|_{v_1}$ is a new function that maps v_1 to $\mathbf{v}(v_1)$. When $V = \varnothing$, $\mathbf{v}$ is the special function with the empty domain.

An alternative notation for a valuation is a vector or tuple. For $v_i \in V$, $type(v_1) \times \ldots \times type(v_n)$ is a set of n-tuples each of which captures the same information as a valuation as long as we can properly index the tuple.

3.8 Trajectories

In Chapter 2, a trajectory was informally described as a function of time that describes continuous system behavior between two discrete stimuli. Specifically, a trajectory defines the evolution of valuations over time.

Definition 3.14. A J-trajectory for V is a function $\tau : J \to val(V)$. J is a left-closed interval of $\mathsf{T}^{\geq 0}$ with left endpoint equal to 0, and for each $v \in V$, $\tau{\downarrow}v \in dtype(v)$.

There are two special types of trajectories. When $V = \varnothing$, τ is a constant function mapping J to the set containing the single valuation with the empty domain. The only information trajectories of this type provide is elapsed time. Such trajectories are used in restrictions of hybrid sequences. A point trajectory has the domain $[0, 0]$ and a single valuation in the range. Point trajectories are denoted $\wp(\mathbf{v})$; $\mathbf{v}$ is the valuation in the range of $\wp(\mathbf{v})$. $trajs(V)$ is the set of all possible trajectories over the variables in V.

Definition 3.15. Let T be a set of trajectories. Elements of T are classified based on their domain:

- If $dom(\tau)$ is a bounded interval, then τ is *bounded*. $finite(T)$ is the set of bounded trajectories.
- If $dom(\tau)$ is bounded and closed, then τ is *closed*. $closed(T)$ is the set of closed trajectories.
- If $dom(\tau)$ is a bounded and right-open, then τ is *open*. $open(T)$ is the set of open trajectories.
- If $dom(\tau) = \mathsf{T}^{\geq 0}$, then τ is *full*. $full(T)$ is the set of full trajectories.

Let τ be a trajectory. The limit time of τ, $\tau.ltime$, is $\sup(dom(\tau))$. The initial valuation in τ is $\tau.fval$. If τ is closed, $\tau.lval \triangleq \tau(\tau.ltime)$.

3.8.1 Prefixes and Concatenation

Definition 3.16. Let τ and υ be trajectories. We say τ is a prefix of υ denoted $\tau \leq \upsilon$ iff $\tau = \upsilon|_{dom(\tau)}$.

Definition 3.17. Let $T \subseteq trajs(V)$. $pref(T) = \{\, \tau \in trajs(V) \mid \exists \upsilon \in T \ : \ \tau \leq \upsilon \,\}$.

$pref(T)$ is the set of all prefixes of each trajectory in T. If $pref(T) = T$, then T is prefix closed. The closed trajectories in the prefix closure are the trajectories realizable in a system simulation.

Definition 3.18. Let τ and τ' be trajectories for V with τ closed. The concatenation of τ and τ', denoted $\tau^\frown\tau'$, is defined by $\tau \cup (\tau'|_{(0,\infty)} + \tau.ltime)$.

Based on the closure properties of each variable name's *dtype*, concatenated trajectories are elements of $trajs(V)$. The concatenation is defined using two compatible functions; τ' is restricted to a left-open interval and time-shifted by the limit time of closed trajectory τ. The valuation used in the concatenated trajectory at $\tau.ltime$ is obtained from the first trajectory. Concatenation extends to a countably infinite sequence of trajectories provided that all nonfinal trajectories are elements of $closed(trajs(V))$.

The following special operators are defined to produce trajectory prefixes and time-shifted suffixes: $\tau \trianglelefteq t \triangleq \tau|_{[0,t]}$, $\tau \triangleleft t \triangleq \tau|_{[0,t)}$, and $\tau \trianglerighteq t \triangleq \tau|_{[t,\infty)} - t$.

3.9 Hybrid Sequences

Let A be the set of system actions and V be the set of system variable names.

Definition 3.19. A *hybrid sequence*, or an (A,V)-sequence, is a finite or infinite alternating sequence $\alpha = \tau_0\ a_1\ \tau_1\ a_2\ \tau_2\ \ldots$ with the following properties:

1. Each $\tau_i \in trajs(V)$.
2. Each $a_i \in A$.
3. If α is a finite sequence, it ends in a trajectory.
4. If τ_i is not the last trajectory, then τ_i is closed.

A hybrid sequence does not immediately satisfy Definition 3.9; however, any hybrid sequence could be re-indexed to satisfy this definition. The alternating property of a hybrid sequence is worth more attention. In Chapter 2, the fundamentals of transforming enumerated stimulus sequences into a discrete system specification were introduced. These enumerations do not provide sufficient information to specify a system that also incorporates trajectories. After the initial trajectory, a hybrid sequence always follows each action with a trajectory. This property of hybrid sequences makes enumerating action-trajectory pairs a suitable means to write hybrid and switching system specifications. Any hybrid sequence projected onto its actions is a well-defined sequence by Definition 3.9.

Point trajectories provide flexibility. For example, within a hybrid sequence an arbitrary number of action-point trajectory pairs can occur with zero time lapse. Although such an implementation is not realizable, this level of control exemplifies the expressiveness of the theoretical model in [13].

Some of the notation used for trajectories has similar meaning for hybrid sequences. The limit time of hybrid sequence α, denoted $\alpha.ltime$, is $\sum_{i \in dom(\alpha)} \tau_i.ltime$. $\alpha.fval \triangleq head(\alpha).fval$. If α is finite with a closed final trajectory, $\alpha.lval \triangleq last(\alpha).lval$.

Definition 3.20. A hybrid sequence α is

- *Time-bounded* if $\alpha.ltime$ is finite. A time-bounded hybrid sequence can be finite or infinite; if finite, its last trajectory could be open or closed.
- *Admissible* if $\alpha.ltime = \infty$. An admissible hybrid sequence can be finite or infinite; if finite, its last trajectory is open.
- *Closed* if α is a finite sequence and its final trajectory is closed.
- *Zeno* if α is neither closed nor admissible, that is, if α is time-bounded and is either an infinite sequence, or else a finite sequence ending with a trajectory whose domain is right-open.
- *Non-Zeno* if α is not Zeno.

3.9.1 Prefixes and Concatenation

Definition 3.21. Let $\alpha = \tau_0\ a_1 \ldots$ and $\beta = \upsilon_0\ b_1 \ldots$ be (A,V)-sequences. α is a prefix of β, denoted $\alpha \leq \beta$, provided that at least one of the following holds:

1. $\alpha = \beta$.
2. α is a finite sequence ending in some τ_k. For every i, $0 \leq i < k$, $\tau_i = \upsilon_i$ and $a_{i+1} = b_{i+1}$. $\tau_k \leq \upsilon_k$.

Definition 3.22.

Let α and α' be (A,V)-sequences with α closed. The concatenation of α and α', denoted $\alpha^\frown\alpha'$, is defined as $init(\alpha)(last(\alpha)^\frown head(\alpha'))tail(\alpha')$.

In the above definition $last(\alpha)^\frown head(\alpha')$ is the concatenation of two trajectories; therefore, the valuation used in the concatenated hybrid sequence at $last(\alpha).ltime$ is obtained from $last(\alpha)$. Concatenation is extended to a countably infinite sequence of hybrid sequences provided that all nonfinal hybrid sequences are closed.

3.10 Hybrid Automata

A HIOA provides the structures we target for specification using enumerated hybrid sequences. In this section, we define hybrid automata that make no distinction between inputs and outputs. Afterwords, we refine the hybrid automata definition to include this distinction [9, 13].

A hybrid automaton is an adaptable state machine that accommodates varying degrees of nondeterminism and distinguishes between continuous-time signals and discrete-time signals. Figure 3.1 illustrates the elements of a hybrid automaton.

Definition 3.23. A hybrid automaton $\mathcal{H}$ is a tuple $(W, X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$:

1. W is the set of external variables. X is the set of state (internal) variables. $W \cap X = \varnothing$. $V \triangleq W \cup X$.
2. $Q \subseteq val(X)$ is the set of states.
3. $\Theta \subseteq Q$ is the nonempty set of initial system states.

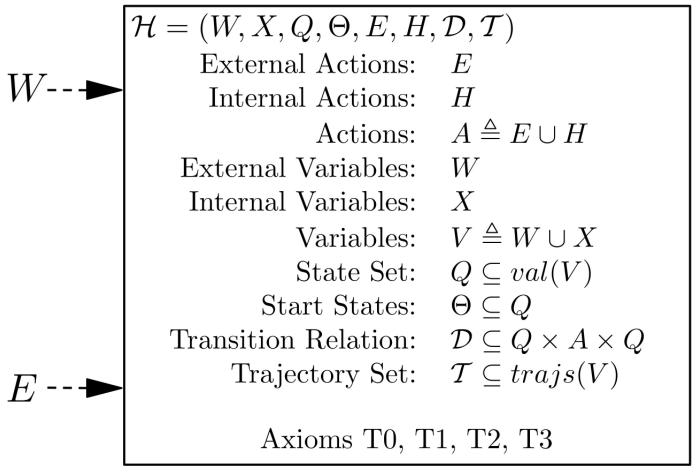


Figure 3.1: A Hybrid Automaton

4. E is the set of input (external) actions. H is the set of internal actions. $E \cap H = \varnothing$. $A \triangleq E \cup H$.

5. $\mathcal{D} \subseteq Q \times A \times Q$ is the set of discrete transitions.

6. $\mathcal{T}$ is the set of trajectories for V such that $\forall \tau \in \mathcal{T}$ and $t \in dom(\tau)$, $\tau(t)|_X \in Q$. Given $\tau \in \mathcal{T}$, $\tau.fstate \triangleq \tau.fval|_X$. If τ is closed, $\tau.lstate \triangleq \tau.lval|_X$. The following axioms hold for $\mathcal{T}$:

   T0. Existence of point trajectories. If $\mathbf{v} \in val(V)$ and $\mathbf{v}|_X \in Q$, then $\wp(\mathbf{v}) \in \mathcal{T}$.

   T1. Prefix closure. For every $\tau \in \mathcal{T}$ and every $\tau' \leq \tau,\ \tau' \in \mathcal{T}$.

   T2. Suffix closure. For every $\tau \in \mathcal{T}$ and every $t \in dom(\tau),\ \tau \trianglerighteq t \in \mathcal{T}$.

   T3. Concatenation closure. Let $\tau_0, \tau_1, \tau_2, \ldots$ be a sequence of trajectories in $\mathcal{T}$ such that, for each nonfinal index i, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$. Then $\tau_0 \frown \tau_1 \frown \tau_2 \ldots \in \mathcal{T}$.

Axiom T0 permits hybrid sequences consisting of an arbitrary number of adjacent point trajectories that define instantaneous automaton state changes. Axiom T0 also facilitates an arbitrary number of instantaneous actions with or without corresponding state changes. Implementation of a large system is simplified by composing smaller, more manageable, hybrid automata. Composition is critical for scalability. The time when one component automaton interrupts another may not be predictable. If a component automaton trajectory τ is interrupted at time t, Axiom T1 guarantees

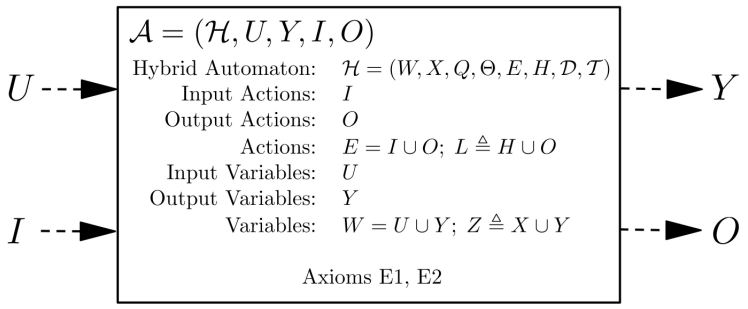


Figure 3.2: A Hybrid I/O Automaton

the prefix of τ is defined; Axiom T2 ensures if the interruption does not change the component's behavior the suffix of τ also exists.

The HIOA model is depicted in Figure 3.2. This model distinguishes input from output and adds two axioms to accommodate the distinction.

Definition 3.24. A hybrid I/O automaton $\mathcal{A}$ is a tuple $(\mathcal{H}, U, Y, I, O)$:

1. $\mathcal{H}$ is a hybrid automaton.

2. U and Y partition W into input and output variable sets. $Z \triangleq X \cup Y$. Z is the set of locally controllable variables.

3. I and O partition E into input and output action sets. $L \triangleq H \cup O$. These actions are locally controlled.

4. A HIOA satisfies two additional axioms:

   E1. Input Action Enabling. For every $\mathbf{x} \in Q$ and every $a \in I$, there exists $\mathbf{x}' \in Q$ such that $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$.

   E2. Input Trajectory Enabling. For every $\mathbf{x} \in Q$ and every $\upsilon \in trajs(U)$, there exists $\tau \in \mathcal{T}$ such that $\tau.fstate = \mathbf{x},\ \tau{\downarrow}U \leq \upsilon$, and either

      i. $\tau{\downarrow}U = \upsilon$, or

      ii. τ is closed and some $l \in L$ is enabled in $\tau.lstate$.

Since a system has no control over what happens in the environment, Axiom E1 guarantees the automaton can always handle every input action. Axiom E2 guarantees there exists a trajectory from every system state that accommodates every possible input trajectory (signal). If the trajectory can accept the input signal in its entirety, then E2 is satisfied; otherwise, there must be either an internal or output action available to close the trajectory. In the later case, the newly initiated trajectory must also satisfy E2.

A timed automaton is a hybrid automaton with $W = \varnothing$, and a timed I/O automaton is a timed automaton that distinguishes input actions from output actions.

3.11 Trajectory and Hybrid Sequence Ordering

In domain theory, a domain is a partially ordered set of objects having a least element where the ordering relation $a \leq b$ indicates that a approximates b or b has at least as much information as a. The least element represents the absence of information. The ordering of trajectories and hybrid sequences is founded in domain theory. To approximate open trajectories and infinite hybrid sequences, we require a domain to be complete. In a complete partial order, the sets whose elements sequentially improve upon the information provided by the previous element in the order have supremums in the partial order [24, p. 29].

Definition 3.25. A *partial order* is a set, D, along with a reflexive, transitive, and antisymmetric relation, $\leq$.

The antisymmetric property of a partial order can be satisfied vacuously; every pair of elements need not be comparable. Nonempty partial orders may contain subsets containing comparable and incomparable pairs but every pair has a common greatest element in the order. These are the directed subsets of a partial order.

Definition 3.26. Let $(D, \leq)$ be a partial order. A set $A \subseteq D$ is a *directed subset* if $A \neq \varnothing$ and whenever $a, b \in A$ there exists $c \in A$ such that $a \leq c$ and $b \leq c$.

A more restrictive form of a directed subset is a totally ordered set also called a chain; all elements in a chain are pairwise comparable.

Definition 3.27. A *chain* in partially ordered set $(D, \leq)$ is a subset $A \subseteq D$ such that for all $a, b \in A$, $a \leq b$ or $b \leq a$.

Given an open or closed trajectory $\tau \in \mathcal{T}$, a sequence of prefixes $\tau \trianglelefteq t$ for strictly increasing $t \in [0, \sup(dom(\tau)))$ is an example of a chain within the partial order of trajectories.

Definition 3.28. Let $(D, \leq, \perp)$ be a partial order with a least element $\perp$. $(D, \leq, \perp)$ is a *complete partial order* (cpo) if whenever $A \subseteq D$ is directed, $\sup(A)$ exists in D.

Definition 3.29. Let $(D, \leq, \perp)$ be a complete partial order. An element $a \in D$ is *compact* or finite if whenever $A \subseteq D$ is a directed set and $a \leq \sup(A)$, there is $x \in A$ such that $a \leq x$. We use D_0 to denote the set of compact elements in D.

Starting with the trajectory with the empty domain and ending with τ, the dynamic behavior of each element in the chain is more complete with respect to τ than the previous element. Informally, the objects capable of being used in computation are the compact elements. In the definition of a HIOA, we are particularly interested in the compact elements among its continuous components. Furthermore, we are interested in whether using those items alone is sufficient to specify the system's behavior completely.

From a computability perspective it is useful to work with complete partial orders where each element in the order can be approximated by a directed set of compact elements [24, p. 53].

Definition 3.30. A complete partial order $(D, \leq, \perp)$ is an *algebraic complete partial order* if for each $b \in D$, the set $D_0(b) = \{\, a \in D_0 \mid a \leq b \,\}$ is a directed set and $b = \sup(D_0(b))$.

Lemma 3.4 of [13] proves the set $trajs(V)$ with prefix ordering is an algebraic complete partial order whose compact elements are the closed trajectories. Lemma 3.6 of [13] proves the set of hybrid sequences with prefix ordering is an algebraic complete partial order whose compact elements are the closed hybrid sequences. The building blocks required to approximate any trajectory or hybrid sequence in a specification or model-based design are the closed trajectories and closed hybrid sequences. These are precisely the objects we address with our enumeration process to constructively define an automaton.

Chapter 4

Hybrid Sequence Enumeration

Construction of model-based designs requires specialized application knowledge. Our goal is to supplement the application expertise and provide a process that induces analysis of all and only those details needed to build a complete, consistent, and traceably correct specification. The hybrid I/O automaton, or HIOA, model in [13] provides the structures necessary to specify both the discrete and the continuous behavior simulated in a model-based design. However, the general, implementation independent form of a HIOA requires many details be explicitly defined, and this also requires special insight. The process of transforming requirements in any form into a model-based design or a HIOA is precisely defined in the chapters that follow.

In this chapter, we define an enumeration process that discovers the hybrid sequences required to describe dynamical system behavior. The enumeration provides the details needed for either an operational model or an algebraic hybrid automaton. Enumeration is possible using sequence abstraction and abstract values [23, p. 426], which we call enumeration elements. In our description, "atomic" refers to indivisible theoretical elements. The specific trajectories and actions that make up a hybrid sequence are atomic elements. The enumeration elements are stimulus vectors, condition vectors, and trajectory definitions. With definitions for these elements, a hybrid enumeration addresses the concerns voiced in industry with our discrete specification method and will aid engineers to produce and test embedded real-time systems. This chapter is a bridge from theory to application.

A hybrid enumeration is an intermediary between requirements and a specification called an enumeration hybrid automaton from which we can produce a HIOA or a model-based design as in

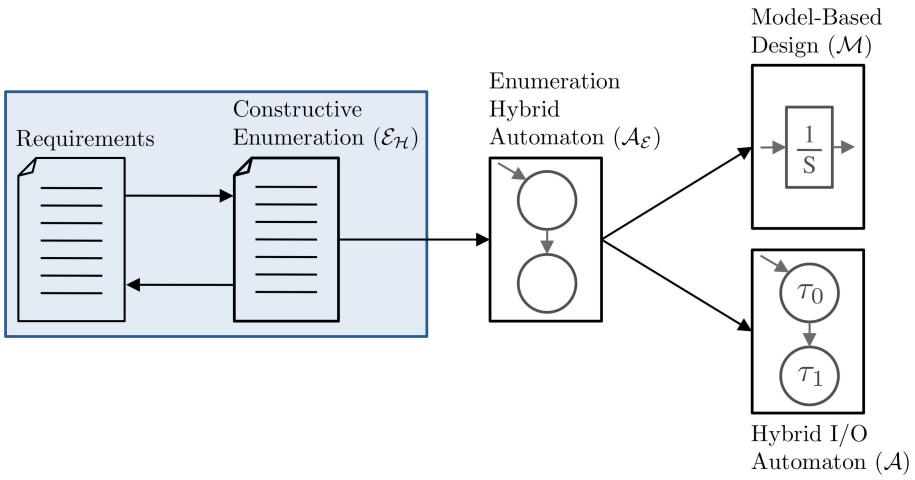


Figure 4.1: Process: Requirements to Constructive Enumeration ($\mathcal{E}_\mathcal{H}$)

Simulink. Figure 4.1 depicts the work flow. This approach differs from that taken in [13] in that we examine systems in detail for implementation and resolve nondeterminism. Although a lot of work is unavoidable, it is minimized by using a recursive abstraction-refinement process. During enumeration, new details emerge and are addressed with supplemental or corrective requirements to ensure the final product is a well-defined specification of intended requirements. We address a strict subset of the systems that can be specified in the algebra. By satisfying the axioms of the HIOA, we construct instances of Lynch algebras, and that theory becomes available for future use. The constructive method defined in this chapter produces a system specification that is complete, consistent, and traceably correct. Timing, continuity, and nondeterminism are addressed explicitly.

4.1 Enumeration Elements

We use enumeration to define the function $\mathcal{E} : \boldsymbol{S}^* \rightarrow \boldsymbol{R} \times \boldsymbol{S}^*$ (Equation 2.5). Stimulus vectors are enumerated, and for each stimulus vector sequence, we prescribe a trajectory definition - condition vector pair and an equivalent stimulus vector sequence. Each line in an enumeration presents a unique situation where the details of enumeration elements can be prescribed based on design requirements. As defined in [10], we isolate the coordinates $\boldsymbol{R}$ and $\boldsymbol{S}^*$ in the $range(\mathcal{E})$ using the projections: $\mapsto \triangleq \pi_{\boldsymbol{R}} \circ \mathcal{E}$, and $\rhd \triangleq \pi_{\boldsymbol{S}^*} \circ \mathcal{E}$. The sections that follow define each enumeration element used to produce $\mathcal{E}$ in terms of the atomic elements in a hybrid system.

4.1.1 Stimulus Vectors

A stimulus is any event that changes system state, trajectory, or output. We adapt the description of a stimulus set for a complex system presented in [21] and predicate refinements in [23, p. 420] to specification of hybrid sequences. The stimulus set for an embedded system is the cross product of a finite collection of sets. Vectors let us address events that occur simultaneously.

Definition 4.1. Given a non-empty set of signals $\mathcal{S} = \{ s_1, \ldots, s_n \}$ of the form $s_i : \mathsf{T} \to C_{s_i}$, the set of *atomic vectors* over $\mathcal{S}$ is given by

$$C_{s_1} \times \ldots \times C_{s_n}. \tag{4.1}$$

The signals named in $\mathcal{S}$ may be continuous-time or discrete-time signals following Definitions 2.1 and 2.2 respectively. Let $d : \mathsf{T} \to I$ be the discrete-time input signal. I always includes the special element *null* that represents the absence of an external input action at a time t in T. No two elements of I can occur simultaneously since d is a function. Each state variable x in X names a continuous-time signal whose codomain is C_x. In the algebra, $x(t)$ is an element of $type(x)$.

Definition 4.2. The set of *atomic stimuli* S of a hybrid system is an atomic vector set over $X \cup \{ d \}$.

Variables named in U are not used to define S. It is safer to include a surrogate input variable in X that follows and constrains the behavior of an input variable in U; the values of state variables can be controlled whereas the values of input variables are uncontrollable by the system being designed.

Figure 4.2 illustrates a set of atomic stimuli. The codomains of the signals label each column; dotted lines partition codomains. For $B \subseteq C_x$, the atomic characteristic predicate of B is χ_B. If ρ is an arbitrary equivalence relation defined on C_x, $C_x/\rho = \{ B_1, \ldots, B_j \}$ is the set of blocks induced by ρ. The characteristic predicate for a block in C_x/ρ defines a subset of C_x. In practice, the characteristic predicate used to define a block in C_x/ρ may be written in various ways. For example, $\chi_{x \neq 100}$ makes clear both the variable name and the block of C_x the predicate defines.

Definition 4.3. The *continuous state space* $\mathcal{U}$ of a system is the set of atomic vectors over X; values in the tuples of $\mathcal{U}$ are ordered lexicographically by variable name. The characteristic predicate of $\mathcal{U}$ is $h_{\mathcal{U}}$.

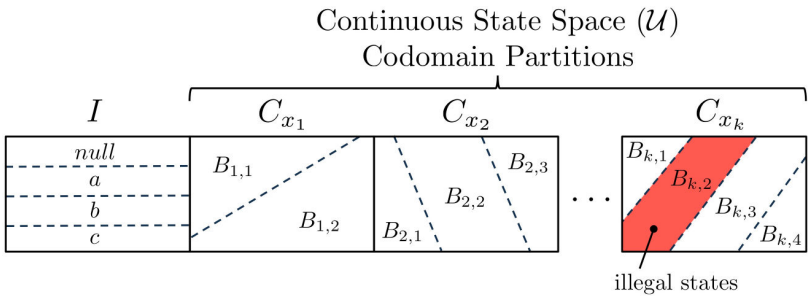


Figure 4.2: Stimulus Vectors

The characteristic predicate of $Q \subseteq \mathcal{U}$ is h_Q. This predicate is defined as a conjunction of atomic characteristic predicates. For example with $k = 2$, $h_Q \triangleq \chi_{x_1=10} \wedge \chi_{x_2>0}$ denotes the subset of the continuous state space where x_1 is exactly 10 and x_2 is positive. When understood by the context, we use h_Q when referring to the set Q.

In embedded real-time systems of interest, S is uncountable and requires a representation that facilitates enumeration. The consolidation of elements in S to describe identical behavior is called abstraction, while partitioning subsets of S to describe different behaviors is called refinement. Characteristic predicates are used for both abstraction and refinement. In our treatment of stimuli, I is always partitioned into labeled blocks as in Figure 4.2. The shaded block represents a subset of codomain C_{x_k} that is not reachable. These unreachable or "illegal" states may be "safely" excluded from the codomain of the corresponding continuous-time signal. Continuing with the previous example, $(null, h_Q)$ is the abstraction of S characterized by three blocks: $\{\, null \,\}$, $[10, 10]$, and $(0, \infty)$; the final two blocks identify a subset of $\mathcal{U}$. The set of actions and the set of characteristic predicates needed to express an infinite state system as a finite automaton constitute an abstraction of S.

Definition 4.4. Given finite I, X, and a partition of $\mathcal{U} = \{\, Q_1, \ldots, Q_n \,\}$ defined with characteristic predicate set $P = \{\, h_{Q_1}, \ldots, h_{Q_n} \,\}$, a set of *stimulus vectors* over I and X is given by

$$\boldsymbol{S} = I \times P. \tag{4.2}$$

In Figure 4.2 with $k = 3$, 96 distinct stimulus vectors are depicted, 24 of which are illegal. The enumeration process considers the infinite set of finite stimulus vector sequences by successive abstraction and refinement managed by characteristic predicates and Boolean operations on them.

4.1.2 Trajectory Definitions

In order to address dynamic systems, we use functions that describe known physical laws whose behavior depends explicitly on time. Well-defined functions, including the laws of physics, do not require enumeration. In every hybrid sequence, each action is followed immediately by a trajectory (see Definition 3.14). We capitalize on this structure by mapping enumerated stimulus vector sequences to trajectory definition-condition vector pairs to incorporate well-defined continuous functions into the specification.

For each system variable $v \in V$, $dtype(v)$ is the set of atomic signals that contains as a subset the signals needed to specify dynamic system behavior. The enumeration elements used to describe this set of signals are equations and condition vectors.

Definition 4.5. An *expression*, denoted e, is an operation on real number constants, time (t), and signals named in $X \cup U$ evaluated at t.

Definition 4.6. A *signal definition* is an equation of the form $v(t) = e$ or $\dot{v} = e$ where $\dot{v}$ is the first time-derivative of the signal $v(t)$ and e is an expression.

The continuous behavior specified in a hybrid enumeration follows the definition presented in Section 2.6.2. Based on this definition, we distinguish several signal definition types.

Definition 4.7. A *state variable signal definition* for $x \in X$ is of the form $x(t) = e$ or $\dot{x} = e$ where expression e is an operation restricted to signals named in X.

Since the system being designed does not control input signals, variables in U are excluded from state variable signal definitions. In state variable signal definition $\dot{x} = e$, e is a vector field. When e is integrable, the differential equation can be solved given an initial condition $x(0)$. A system describable by a higher order explicit differential equation can be decomposed into a system of equivalent first-order differential equations. The signal definition $\dot{x} = 0$ defines discrete state variable behavior (see Section 3.6); its value remains fixed at its initial condition during every trajectory.

Definition 4.8. A *surrogate input variable* $\tilde{x}$ has a signal definition $\tilde{x}(t) = f(u(t))$ where $u \in U$ and f is a function. $C_{\tilde{x}} = C_u$, and $\tilde{x} \in X$.

For example, if u is said to be reasonable with $u(t) \in [a, b]$, then the surrogate signal definition

$$\tilde{x}(t) = \begin{cases} a & u(t) \leq a \\ u(t) & a < u(t) < b \\ b & u(t) \geq b \end{cases} \tag{4.3}$$

might be used to ensure $\tilde{x}$ does not reproduce any values outside of $[a, b]$. A surrogate input variable allows safe use of an input signal. Actions defined on surrogate input variables replace actions otherwise caused by input signals. For example, a sensor may report pressure to a system. Although pressure behavior may be understood, erratic pressure or erroneous sensor readings are possible and should not be allowed to produce a fault in the system being designed. To complete the example, the actual $u(t)$ value remains available to report outside of the system boundary.

For output variables, we follow Equation 2.3 in the definition of continuous systems.

Definition 4.9. An *output variable signal definition* for $y \in Y$ is of the form $y(t) = e$ where e is an expression.

A system must be capable of reporting unmodified input signals to its environment; therefore, an expression used in an output variable signal definition may include signals named in X and U.

$\mathcal{T}$ is the set of atomic trajectories required to define initial system behavior and the system's behavior following any action. The enumeration elements used to describe the trajectories in $\mathcal{T}$ are trajectory definitions.

Definition 4.10. A *trajectory definition* φ is a $|Z|$-tuple; each coordinate is a signal definition for a locally controllable variable in Z. Variable name determines the order in φ.

Only the locally controllable variables in Z require explicit signal definitions. Continuous input signals are not controlled by the system being designed. Accordingly, no assumptions can be made during system specification about trajectories over variables in U.

Definition 4.11. Φ is the set of *trajectory definitions*.

We say that two trajectories are *behaviorally equivalent* if they have identical trajectory definitions. This does not imply the trajectories have the same initial condition or range, only that the signal definitions that define their behavior are the same. If an action changes a system's trajectory definition, the system exhibits switching behavior (see Section 2.6.3).

4.1.3 Condition Vectors

The transition relation of a hybrid automaton is $\mathcal{D} \subseteq Q \times A \times Q$ (Definition 3.23). For a triple in $\mathcal{D}$, the first coordinate is the state of the system when the action in A occurs and closes the current trajectory τ. We use $last = \tau.ltime$ and $x(last) = \tau.lstate(x)$ in our enumeration elements to identify the time in the trajectory when an action occurs and the value of x at that time, respectively. The third coordinate is the first state of the trajectory following the action in A.

Since the set of discrete system outputs, O, is a subset of A in a HIOA, a subset of $\mathcal{D}$ defines how discrete output is generated. In our specifications, a discrete output is always the instantaneous, externally available result of an input or internal action. In other words, an expression (possibly *null*) characterizes discrete output for every sequence of stimulus vectors. Let $o : \mathsf{T} \rightarrow O$ be the discrete-time output signal that results from an input or internal action. O includes *null* to represent the absence of an observable discrete output at a time t in T and ω to indicate the sequence of atomic stimuli is physically impossible.

Definition 4.12. The set of *atomic conditions* R of a hybrid system is an atomic vector set over $X \cup \{ o \}$.

An atomic condition is an instantaneous discrete output and an internal system state; it is the result of an applied atomic stimulus and current system state.

Definition 4.13. An *expression set* E_s is a finite set containing the expressions used to compute the value of signal s at $t = 0$ and one or more special elements to include *null*. Each expression in E_s operates on real number constants and signals named in X evaluated at *last*.

Definition 4.14. Given the collection of expression sets $E = \{ E_o, E_{x_1}, \ldots, E_{x_k} \}$ where $\omega \in E_o$, the set of *condition vectors* over O and $X = \{ x_1, \ldots, x_k \}$ is given by

$$R \subseteq E_o \times E_{x_1} \times \ldots \times E_{x_k}. \tag{4.4}$$

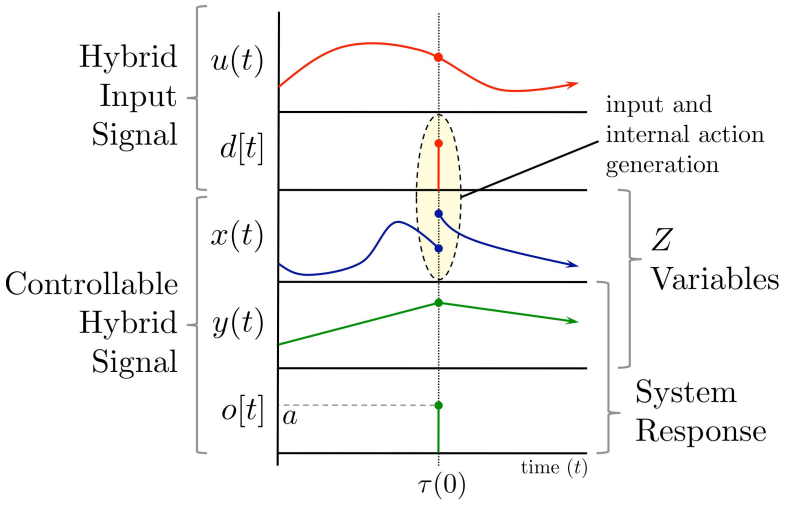


Figure 4.3: Hybrid Signals

Use of *null* in condition vector coordinate x indicates $x(0)$ is defined explicitly using the signal definition for x. When the signal definition is of the form $\dot{x} = e$, $x(0)$ is defined with $x(last)$ when x is continuous across the transition, or $x(0)$ is defined by an expression.

In the first condition vector definition used in a hybrid enumeration, discrete output is always *null*, and $x(last)$ indicates nondeterministic selection of $x(0)$ from a set of possible initial values for x.

Each condition vector computes atomic condition values in a specific way. For example, let $\boldsymbol{r} \in R$ and $\boldsymbol{r} = (x_1(last)+x_2(last),\ x_1(last),\ 2(x_2(last)),\ 8)$. The discrete output is the sum of the values of two state variables prior to the transition. State variable x_1 is continuous through the transition, x_2 jumps to twice its previous value, and x_3 always starts from 8 after the transition.

4.1.4 Hybrid Signals

Definition 4.15. A *hybrid signal* is a composite of a continuous-time and a discrete-time signal.

In this section, we relate the atomic elements of a hybrid system to the first element of the pairs in the range of a hybrid enumeration. A trajectory-action pair can be characterized as a hybrid signal. In Figure 4.3, $o[t]$ is the discrete-time output signal defined by $o[0] = a$ and for all $t \in dom(\tau),\ t > 0,\ o[t] = null$. Together, continuous-time signal $y(t)$ and discrete-time signal $o[t]$ form a hybrid signal. In a hybrid system, input and internal actions initiate new atomic hybrid

signals. This concept is illustrated in Figure 4.3 where $d[t]$ and $x(t)$ produce hybrid and switching behavior (x jumps in value and changes trajectory at $t = 0$) as well as changes in discrete ($o[0]$ is a spike) and continuous output ($y(t)$ starts to decrease at $t = 0$). The continuous portion of the controllable hybrid signal is only defined for the variables in Z based on Equations 2.2 and 2.3.

Definition 4.16. Given $trajs(Z)$ and O, the set of *controllable atomic hybrid signals* of a hybrid system is a subset of

$$trajs(Z) \times O. \tag{4.5}$$

For the pair $(\tau, a) \in trajs(Z) \times O$, the discrete output a is only detectable in the environment at $t = 0$ of τ. The enumeration elements that characterize the set of controllable atomic hybrid signals are defined in Sections 4.1.2 and 4.1.3.

Definition 4.17. The set of *controllable hybrid signal definitions* is given by

$$\boldsymbol{R} \subseteq \Phi \times R. \tag{4.6}$$

The pairs in $\boldsymbol{R}$ contain sufficient information to describe system behavior following every sequence of stimulus vectors. For the pair $(\varphi, \boldsymbol{r}) \in \boldsymbol{R}$, the discrete output at $t = 0$ is computed using the first coordinate of the condition vector $\boldsymbol{r}$. The remaining coordinates compute initial conditions as needed to produce a specific trajectory from φ.

We frequently use two projections of $\boldsymbol{R}$:

$$\pi_{\Phi} : \boldsymbol{R} \rightarrow \Phi \tag{4.7}$$

$$\pi_{R} : \boldsymbol{R} \rightarrow R \tag{4.8}$$

The enumeration projection function $\mapsto$ (Section 2.7) is extended to incorporate trajectory definitions and condition vectors: $\mapsto_{\Phi} \triangleq \pi_{\Phi} \circ \mapsto$, and $\mapsto_{R} \triangleq \pi_{R} \circ \mapsto$.

Definition 4.18. For stimulus vector sequence $\boldsymbol{\sigma}$, a *null hybrid signal definition* $(\varphi_0, \boldsymbol{r}_0)$ satisfies the following properties:

1. If $\boldsymbol{\sigma} = \lambda$, $\varphi_0 = \mapsto_\Phi (\lambda)$; otherwise, $\varphi_0 = \mapsto_\Phi (init(\boldsymbol{\sigma}))$.
2. $\pi_o(\boldsymbol{r}_0) = null$.
3. For all $x \in X$, if $\pi_x(\varphi_0) = (\dot{x} = e)$, then $\pi_x(\boldsymbol{r}_0) = x(last)$; otherwise, $\pi_x(\boldsymbol{r}_0) = null$.

A *null* hybrid signal definition exhibits neither switching (trajectory definition does not change) nor hybrid behavior (continuity maintained across transitions). These conditions are established using the trajectory definition associated with the stimulus vector sequence's prefix. A *null* hybrid signal definition does not change system behavior.

Definition 4.19. An *illegal hybrid signal definition* $\Omega = (\varphi, \boldsymbol{r})$ satisfies the following properties:

1. $\pi_o(\boldsymbol{r}) = \omega$.
2. For all $x \in X$, $\pi_x(\varphi) = (\dot{x} = 0)$ and $\pi_x(\boldsymbol{r}) = x(last)$.
3. For all $y \in Y$, $\pi_y(\varphi) = (y(t) = y(last))$.

An illegal hybrid signal definition fixes the value of each state and output variable for all $t \in \mathsf{T}^{\geq 0}$ and produces the illegal discrete response ω. $\boldsymbol{R}$ may or may not contain the *null* or illegal hybrid signal definition.

Function composition is used to obtain the expressions associated with specific coordinates of either element in the pair. For example, the discrete and instantaneous output value that results from an applied stimulus can be computed using the expression produced by the function: $\pi_o \circ \pi_R$.

4.2 Hybrid Enumeration

A hybrid enumeration contains the details necessary to produce an abstract state machine called an enumeration hybrid automaton. Referring to Figure 2.3 for the variable names, we start with a set of discrete input actions (I), input variables (U), output variables (Y), and state variables (X) gathered from the initial requirements and assume nothing about system dynamics or discrete

output. An appropriate codomain, e.g., $\mathbb{R}$, $\mathbb{N}$, $\{\,\texttt{on}, \texttt{off}\,\}$, $[0, 10]$, etc. is defined for each variable in V. An enumeration builds from these essential sets, the construction of the enumeration elements introduced in the previous sections, and such documented requirements as may be available. New information or insight at any point necessitate restarting the enumeration process.

Definition 4.20. Given an enumeration $\mathcal{E} : \boldsymbol{S}^* \to \boldsymbol{R} \times \boldsymbol{S}^*$, a set of system variables $V = U \cup X \cup Y$, and a set of start states Θ. A *hybrid enumeration* $\mathcal{E}_{\mathcal{H}}$ is a triple:

$$(\mathcal{E}, V, \Theta). \tag{4.9}$$

There are three phases in the hybrid enumeration process. The first two phases, declaration and initialization, define the system variables, the set of initial system states, and the initial trajectory definition inferred from existing requirements. The sequence enumeration phase produces the remaining mappings that make up the partial function $\mathcal{E}$.

ConstructHybridEnumeration(Requirements)

1. Declaration
2. Initialization
3. Sequence Enumeration

end ConstructHybridEnumeration

The DefineMapping procedure uses recursion to define the mappings in $\mathcal{E}$ in four phases.

DefineMapping($\boldsymbol{m}, D, C, X'$)

1. Trajectory Definition
2. Condition Vector Definition
3. **if** necessary **then** Refinement
4. **else** Sequence Reduction

end DefineMapping

4.2.1 Relationship of $\mathcal{E}_{\mathcal{H}}$ to Discrete Enumeration

The basic iterative process of sequence enumeration remains unchanged from discrete sequence-based specification except for the level of detail, addition of trajectory definitions for continuity and timing, and explicit inclusion of a recursive process to define finite abstractions of a system's continuous state space. Notation remains consistent with discrete sequence-based specification theory and the theory of hybrid automata with the following exceptions: the letters x, u, and v are used as variable names and not stimuli and sequences of stimuli.

Definition 4.21. Given $\mathcal{E}_{\mathcal{H}}$ with $\mathcal{E} : \boldsymbol{S}^* \rightarrow \boldsymbol{R} \times \boldsymbol{S}^*$, the *distinguishability* of unreduced stimulus vector sequences $\boldsymbol{\sigma}$ and $\boldsymbol{\psi}$ in $dom(\mathcal{E})$ is defined as follows:

1. If $\mapsto_{\Phi} (\boldsymbol{\sigma}) \neq \mapsto_{\Phi} (\boldsymbol{\psi})$, then $\boldsymbol{\sigma}$ and $\boldsymbol{\psi}$ are distinguishable.
2. If $\mapsto (\boldsymbol{\sigma}) = \Omega$ and there exists $\boldsymbol{a} \in \boldsymbol{S}$ such that $\mapsto (\boldsymbol{\psi}.\boldsymbol{a}) \neq \Omega$, then $\boldsymbol{\sigma}$ and $\boldsymbol{\psi}$ are distinguishable.
3. If there exists $\boldsymbol{a} \in \boldsymbol{S}$ such that $\mapsto (\boldsymbol{\sigma}.\boldsymbol{a}) \neq \mapsto (\boldsymbol{\psi}.\boldsymbol{a})$, then $\boldsymbol{\sigma}$ and $\boldsymbol{\psi}$ are distinguishable.
4. If $\boldsymbol{\sigma}$ and $\boldsymbol{\psi}$ are distinguishable and there exists $\boldsymbol{a} \in \boldsymbol{S}$ such that for unreduced $\boldsymbol{c}, \boldsymbol{d} \in dom(\mathcal{E})$, $\boldsymbol{c}.\boldsymbol{a} \rhd \boldsymbol{\sigma}$ and $\boldsymbol{d}.\boldsymbol{a} \rhd \boldsymbol{\psi}$, then $\boldsymbol{c}$ and $\boldsymbol{d}$ are distinguishable.
5. No two sequences are distinguishable except by a finite number of applications of the previous rules.

This definition of distinguishability is adapted from [10, Definition 4.9] and covers all cases of distinguishability among sequences in $dom(\mathcal{E})$.

Definition 4.22. Two stimulus vector sequences are *equivalent* if they are not distinguishable.

Each enumerated stimulus vector sequence starting with the empty sequence is mapped to a controllable hybrid signal definition and an equivalent sequence of stimulus vectors. Unlike discrete sequence-based specification, the empty sequence in a hybrid enumeration does not always map to a fixed *null* object; however, by Definition 4.18, the empty sequence always maps to the *null* hybrid signal definition. This definition along with Θ may characterize more than one initial trajectory. The form of each mapping added to $\mathcal{E}$ is $(\boldsymbol{\sigma}, ((\varphi, \boldsymbol{r}), \boldsymbol{\psi}))$ where $\boldsymbol{\sigma}$ is the sequence of stimulus vectors

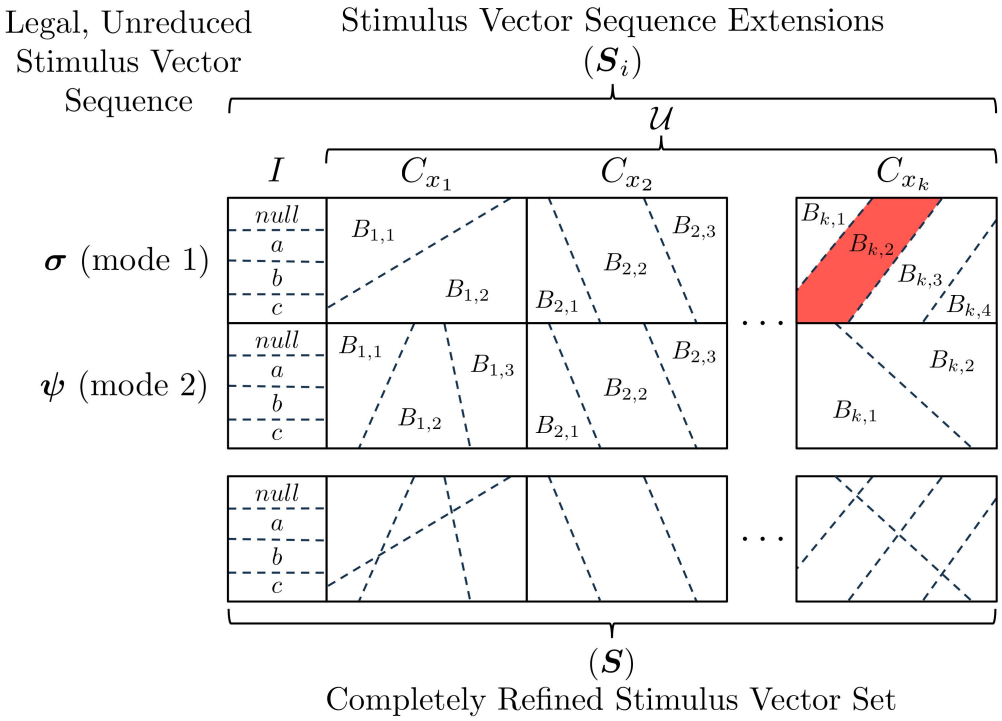


Figure 4.4: Stimulus Vector Sequence Extensions

being mapped, $(\varphi, \boldsymbol{r})$ is the controllable hybrid signal definition that includes a trajectory definition and a condition vector, and $\boldsymbol{\psi}$ is an equivalent stimulus vector sequence possibly equal to $\boldsymbol{\sigma}$.

Discrete sequence-based specification assumes S is initially well-defined (although subject to change). To make enumeration efficient and handle variables that operate over infinite sets, predicate refinement is essential. Refinements are defined using a recursive process to complete mappings not definable at a higher level of abstraction. Each unreduced and legal stimulus vector sequence $\boldsymbol{\sigma}$ represents a mode i and is extended by the set of stimulus vectors $\boldsymbol{S}_i$. In Figure 4.4, $\boldsymbol{\sigma}$ represents mode 1. It is extended by 96 stimulus vectors created using the partition of I and the codomains of each variable in X based on $k = 3$. $\boldsymbol{\psi}$ represents mode 2, and it is extended by 72 different stimulus vectors. The partition of I used to create each $\boldsymbol{S}_i$ is always the same; however, the partition of $\mathcal{U}$ may change as shown in Figure 4.4.

Let $P_{x,i}$ be the set of atomic characteristic predicates that partition C_x for the set of stimulus vectors $\boldsymbol{S}_i$ for mode i. For a system with n modes, the conjunctions formed using the tuples in $P_{x,1} \times \ldots \times P_{x,n}$ partition C_x. This set of atomic characteristic predicates along with similarly produced predicate sets for the remaining variables in X are used to define a complete refinement

of $\mathcal{U}$ with respect to the system being enumerated. In the third row of Figure 4.4, the complete refinement of $\mathcal{U}$ is created by superimposing mode 1's partition of $\mathcal{U}$ on mode 2's partition of $\mathcal{U}$. Using I and the complete refinement of $\mathcal{U}$, the completely refined stimulus vector set $\boldsymbol{S}$ can be created. With this $\boldsymbol{S}$, $\mathcal{E}$ may be updated, and $\mapsto$ and $\triangleright$ can be extended to make $\mathcal{E}$ a total function over the domain $\boldsymbol{S}^*$.

4.2.2 Hybrid Enumeration Process

ConstructHybridEnumeration returns a hybrid enumeration based on requirements. The procedure takes a single parameter, the requirements document(s), which may be modified by making additions, deletions, or revisions. Procedure steps are numbered. Comments precede the applicable procedure steps. Where useful, the line number that ends or begins an algorithm segment is enclosed in square brackets in the line's right margin.

In the constructive process, the following conventions are used: for $\mathbf{v} \in \times_{i=1}^{n} type(v_i)$, $\pi_{v_1}(\mathbf{v})$ produces the value associated with variable v_1, while $\mathbf{v}(v_1) \leftarrow a_1$ assigns the value a_1 to v_1's coordinate in the tuple $\mathbf{v}$.

ConstructHybridEnumeration(Requirements)

Elements (variable names) in the sets I, U, Y, and X are identified from existing requirements (see Figure 2.3).

1.1 Construct set I from input actions and include *null*

Input signals are not controlled by the system being specified. Surrogate input variables (see Definition 4.8) are added to X for variables in U to facilitate controlled software function. The set U is the continuous input interface between system and environment.

1.2 Construct set U from input variables

1.3 Construct set Y from output variables

1.4 Construct set X from state variables and include surrogates for variables in U

Z is defined for reference.

1.5 $Z \triangleq X \cup Y$

Define the codomain of each variable's associated signal, e.g., $\mathbb{R}$, $\mathbb{N}$, $\{\,\texttt{on}, \texttt{off}\,\}$, $[0, 10]$, *etc.*

1.6 **for** each $z \in Z$

1.7 For surrogate input variable z with $z(t) = f(u(t))$, define codomain C_u and let $C_z \leftarrow C_u$

1.8 define codomain C_z for non-surrogate z

1.9 **end for**

1.10 $\mathcal{U} \leftarrow \times_{x \in X} C_x$

The mapping for the empty stimulus vector sequence λ *is always included first in an enumeration. The initial condition vector is an* $(|X|+1)$*-tuple with null entries. The initial trajectory definition is a* $|Z|$*-tuple with null entries. We use* $\boldsymbol{m}$ *to denote a mapping in* $\mathcal{E}$*. Each aspect of the initial mapping* $\boldsymbol{m}$ *will be examined in the process to ensure it satisfies existing or new requirements.* λ *is always reduced to itself.*

2.1 $\boldsymbol{r} \leftarrow (null, \ldots, null)$

2.2 $\varphi \leftarrow (null, \ldots, null)$

2.3 $\boldsymbol{m} \leftarrow (\lambda, ((\varphi, \boldsymbol{r}), \lambda))$

Define the set of initial continuous states as $|X|$*-tuples ordered by variable name.*

2.4 **for** each $x \in X$

2.5 define $A_x \subseteq C_x$

2.6 **end for**

2.7 $\Theta \leftarrow \times_{x \in X} A_x$

Based on existing or derived requirements, define the initial signal definition for each controllable variable in Z*. A first time-derivative may only be used if the variable is in* X*;* $z(last)$ *represents the nondeterministic selection of an initial condition from* A_z *(see Section 4.1.3).*

2.8 **for** each $z \in Z$ in lexicographical order [2.20]

2.9 **case** z's signal definition [2.19]

2.10 **when** $z \in X$ and $\dot{z} = e$

2.11 $\varphi(z) \leftarrow \dot{z} = e$ over X and t

2.12 $\quad\quad\quad\quad \boldsymbol{r}(z) \leftarrow z(last)$

2.13 $\quad\quad\quad$ **when** $z \in X$ and $z(t) = e$

2.14 $\quad\quad\quad\quad \varphi(z) \leftarrow z(t) = e$ over X and t

2.15 $\quad\quad\quad$ **when** $z \in X$ and z is a surrogate input variable

Each surrogate signal definition must ensure safe system operation for all possible values of u (see Definition 4.8)

2.16 $\quad\quad\quad\quad \varphi(z) \leftarrow z(t) = f(u(t))$ for $u \in U$

2.17 $\quad\quad\quad$ **when** $z \in Y$

2.18 $\quad\quad\quad\quad \varphi(z) \leftarrow z = e$ over X, U, and t

2.19 $\quad\quad$ **end case** [2.9]

2.20 $\quad$ **end for** [2.8]

2.21 $\quad \mathcal{E} \leftarrow \{\, \boldsymbol{m} \,\}$

E *collects extensible stimulus vector sequences.*

2.22 $\quad \boldsymbol{E} \leftarrow \{\, \lambda \,\}$

Sequence enumeration proceeds by sequence length, l.

3.1 $\quad l \leftarrow 0$

3.2 $\quad$ **while** ($\exists \boldsymbol{\sigma} \in \boldsymbol{E}$ such that $|\boldsymbol{\sigma}| = l$)

3.3 $\quad$ **do** [3.12]

Each extensible sequence is extended by every element of I paired with the characteristic predicate for the entire continuous state space $\mathcal{U}$. Future refinement of $\mathcal{U}$ may be necessary. Inclusion of null in I forces discovery of internal actions and the invariant set discussed in Section 3.1.

3.4 $\quad\quad$ **for** each $\boldsymbol{\sigma} \in \boldsymbol{E}$ of length l [3.10]

3.5 $\quad\quad\quad$ **for** each $a \in I$ [3.9]

3.6 $\quad\quad\quad\quad \boldsymbol{a} \leftarrow (a, h_{\mathcal{U}})$

A new default mapping is constructed using the null hybrid signal definition $(\varphi_0, \boldsymbol{r}_0)$ *(Definition 4.18). The procedure* DefineMapping *returns the set of mappings needed to ensure each component of the default null hybrid signal definition* $(\varphi_0, \boldsymbol{r}_0)$ *and the sequence reduction is re-examined and defined consistently based on existing or derived requirements. When the characteristic predicate* $h_{\mathcal{U}}$ *is refined new mappings are added to the set returned by* DefineMapping.

3.7 $\quad\quad\quad \boldsymbol{m} \leftarrow (\boldsymbol{\sigma}.\boldsymbol{a}, ((\varphi_0, \boldsymbol{r}_0), \boldsymbol{\sigma}))$

3.8 $\quad\quad\quad \mathcal{E} \leftarrow \mathcal{E} \cup$ DefineMapping $(\boldsymbol{m},\ Z,\ \{o\},\ X)$

3.9 $\quad\quad$ **end for** [3.5]

3.10 $\quad$ **end for** [3.4]

3.11 $\quad l \leftarrow l + 1$

3.12 **end while do** [3.3]

Enumeration is complete when all the extensible sequences in $dom(\mathcal{E})$ *are extended.*

3.13 **return** $(\mathcal{E},\ (U \cup Z),\ \Theta)$

end ConstructHybridEnumeration

DefineMapping is a recursive procedure that refines the predicate in stimulus vector $\boldsymbol{a}$ in the sequence $\boldsymbol{\sigma}.\boldsymbol{a}$ as necessary to completely define the controllable hybrid signal definition $(\varphi, \boldsymbol{r})$. DefineMapping returns a set of mappings that ensures the characteristic predicates used in refined mapped sequences cover the entire continuous state space, $\mathcal{U}$, and whose definitions are complete.

The set $\boldsymbol{E}$ has global scope; legal, unreduced stimulus vector sequences are added to $\boldsymbol{E}$ in DefineMapping. DefineMapping takes the following parameters:

1. $\boldsymbol{m} = (\boldsymbol{\sigma}.\boldsymbol{a}, ((\varphi, \boldsymbol{r}), \boldsymbol{\sigma}))$ is the mapping being defined with $\boldsymbol{a} = (h_Q, a)$. Initially it is defined using the *null* hybrid signal definition. If DefineMapping is called recursively, h_Q will be a refinement of the previous predicate paired with a.

2. D contains the locally controllable variables whose signal definitions remain undefined; initially D is equal to Z.

3. The parameter C contains the names of condition vector coordinates that need to be defined.

4. X' is the set of state variable names whose atomic characteristic predicate(s) in h_Q may be refined further.

DefineMapping($\boldsymbol{m}, D, C, X'$)

1.1 **for** each $z \in D$ in lexicographical order [1.21]

When possible based on existing requirements, derived requirements, and the current refinement h_Q, write the signal definitions for the variables in D.

1.2 **case** definability of z's signal definition [1.20]

1.3 **when** illegal

Recursion base: $\boldsymbol{\sigma}.\boldsymbol{a}$ is illegal.

1.4 **return** IllegalMapping($\boldsymbol{m}$)

1.5 **when** definable

The signal definition for z is complete and must be removed from D.

1.6 $D \leftarrow D - \{ z \}$

1.7 **case** z's signal definition [1.18]

1.8 **when** $z \in X$ and $\dot{z} = e$

Only variable names in X may have a differential equation as a signal definition. If the signal is defined in this way, add the variable name to the set whose coordinate in the condition vector must be defined. By default, it is assumed to be continuous.

1.9 $C \leftarrow C \cup \{ z \}$

1.10 $\varphi(z) \leftarrow \dot{z} = e$ over X and t

1.11 $\boldsymbol{r}(z) \leftarrow z(last)$

1.12 **when** $z \in X$ and $z(t) = e$

1.13 $\varphi(z) \leftarrow z(t) = e$ over X and t

1.14 **when** $z \in X$ and z is a surrogate input variable

Each surrogate definition is a function of actual input (see Definition 4.8).

1.15 $\varphi(z) \leftarrow z(t) = f(u(t))$ for $u \in U$

1.16 **when** $z \in Y$

1.17 $\varphi(z) \leftarrow z = e$ over X, U, and t

1.18 **end case** [1.7]

There may be signal definitions in φ that remain undefined; the refinement process will provide a way to complete φ.

1.19 **when** undefinable for the current refinement h_Q

1.20 **end case** [1.2]

1.21 **end for** [1.1]

Define the discrete output $o[0]$ and the initial condition $x(0)$ for each variable in X whose signal is defined by a differential equation.

2.1 **for** each $v \in C$ in lexicographical order [2.15]

Determine whether existing or newly derived requirements support defining the condition vector entry for v.

2.2 **case** definability of v's value at $t = 0$ of φ [2.14]

2.3 **when** illegal

Recursion base: $\boldsymbol{\sigma}.\boldsymbol{a}$ is illegal.

2.4 **return** IllegalMapping($\boldsymbol{m}$)

2.5 **when** definable

The condition vector definition for the variable named v will be completed and must be removed from C.

2.6 $C \leftarrow C - \{ v \}$

Either v is the discrete-time signal o, or v is a continuous-time signal previously defined in φ with a differential equation. At this point, $\pi_v(\boldsymbol{r}) = null$ in the first case and $\pi_v(\boldsymbol{r}) = v(last)$ in the second case. If these default definitions are inconsistent with existing or derived requirements, they must be changed.

2.7 **if** $v(0) = e$ or $v[0] = e$

2.8 **then**

2.9 $\boldsymbol{r}(z) \leftarrow e$ over X and *last*

2.10 **else**

2.11 keep the previous default definition

2.12 **end if**

There may be conditions in $\boldsymbol{r}$ *that remain undefined; the refinement process will provide a way to complete* $\boldsymbol{r}$.

2.13 **when** undefinable for the current refinement h_Q

2.14 **end case** [2.2]

2.15 **end for** [2.1]

When each variable's signal definition and condition vector coordinate are defined for the current mapping, candidates for sequence reduction may be considered. Either the sequence in the mapping is reducible or extensible.

3.1 **if** $C \cup D = \varnothing$ [4.14]

3.2 **then** [3.12]

Candidates for sequence reduction are the legal, extensible sequences in $\boldsymbol{E}$.

3.3 **for** each $\boldsymbol{\psi} \in \boldsymbol{E}$ [3.9]

Determining sequence equivalence is based on existing or derived requirements. Two sequences are equivalent if they are behaviorally equivalent and all future sequence extensions of both have identical controllable hybrid signal definitions. A single inconsistency is sufficient to eliminate the candidate from further consideration. It is safer to extend a sequence than to make an incorrect equivalence assignment.

3.4 **if** $\boldsymbol{\sigma}.\boldsymbol{a}$ and $\boldsymbol{\psi}$ are equivalent

3.5 **then**

3.6 $\boldsymbol{\sigma}.\boldsymbol{a} \rhd \boldsymbol{\psi}$

Recursion base: $\boldsymbol{\sigma}.\boldsymbol{a}$ *is legal and reduced.*

3.7 **return** $\{\boldsymbol{m}\}$

3.8 **end if**

3.9 **end for** [3.3]

3.10 $\boldsymbol{\sigma}.\boldsymbol{a} \rhd \boldsymbol{\sigma}.\boldsymbol{a}$

3.11 $\boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{\boldsymbol{\sigma}.\boldsymbol{a}\}$

Recursion base: $\boldsymbol{\sigma}.\boldsymbol{a}$ *is legal and extensible.*

3.12 **return** $\{\boldsymbol{m}\}$ [3.2]

The controllable hybrid signal definition $(\varphi, \boldsymbol{r})$ *is not defined completely since* $C \cup D \neq \varnothing$*. Refinement of the continuous state space is necessary.* X' *contains those continuous state variables that may be used to eliminate ambiguities in the controllable hybrid signal definition by refining the predicate* h_Q*. In the case where* X' *is empty, the stimulus vector sequence is illegal.*

4.1 **else** [4.13]

4.2 **for** each $x \in X'$ in lexicographical order [4.13]

The characteristic predicate h_Q *may contain multiple atomic characteristic predicates for variable* x *due to previous refinement.*

4.3 **for** each atomic χ_x used in the conjunction h_Q [4.12]

4.4 **if** χ_x can be refined into a set χ'_x to define consistent elements in the controllable hybrid signal definition $(\varphi, \boldsymbol{r})$

4.5 **then**

Behavior previously defined is carried over into mapping $\boldsymbol{m}$ *and* $\boldsymbol{m}'$ *in* $(\varphi, \boldsymbol{r})$*.*

4.6 $\boldsymbol{a} \leftarrow (a, h_Q \wedge \chi'_x)$

4.7 $\boldsymbol{m} \leftarrow (\boldsymbol{\sigma}.\boldsymbol{a}, ((\varphi, \boldsymbol{r}), \boldsymbol{\sigma}))$

To ensure complete coverage of $\mathcal{U}$*, the complement of the refinement* χ'_x *is used to create a second mapping that may be refined further in variable* x*.*

4.8 $\boldsymbol{a}' \leftarrow (a, h_Q \wedge \overline{\chi'_x})$

4.9 $\boldsymbol{m}' \leftarrow (\boldsymbol{\sigma}.\boldsymbol{a}', ((\varphi, \boldsymbol{r}), \boldsymbol{\sigma}))$

The recursive step.

4.10 **return** DefineMapping($\boldsymbol{m}, D, C, X' - \{x\}$) $\cup$
DefineMapping($\boldsymbol{m}', D, C, X'$)

4.11 **end if**

4.12 **end for** [4.3]

4.13 **end for** [4.1, 4.2]

4.14 **end if** [3.1]

There are no variables in X'. Refinement is not possible, and the controllable hybrid signal definition is not fully defined based on the requirements. $\boldsymbol{\sigma}.\boldsymbol{a}$ is illegal.

4.15 **return** IllegalMapping($\boldsymbol{m}$)

end DefineMapping

The function IllegalMapping constructs and returns an illegal mapping to add to the enumeration. An illegal mapping fixes the values of variables in Z at their value when an illegal sequence of stimulus vectors occurred.

IllegalMapping($\boldsymbol{m}$)

1.1 $\boldsymbol{\sigma}.\boldsymbol{a} \mapsto \Omega$

1.2 $\boldsymbol{\sigma}.\boldsymbol{a} \rhd \boldsymbol{\sigma}.\boldsymbol{a}$

1.3 **return** $\{\boldsymbol{m}\}$

end IllegalMapping

4.2.3 Refinement

Refinement is a depth-first binary search of continuous state space; maximum depth is $|X| + 1$. Figure 4.5 shows refinement of the extended sequence $\boldsymbol{\sigma}.\boldsymbol{a}$. At the refinement tree's root, $\boldsymbol{a}$ is in its most abstract form $(a, h_{\mathcal{U}})$. By starting with $h_{\mathcal{U}}$, the enumeration process is made more efficient. Branching represents further refinement of $h_{\mathcal{U}}$ using a single variable, x. In each right branch,

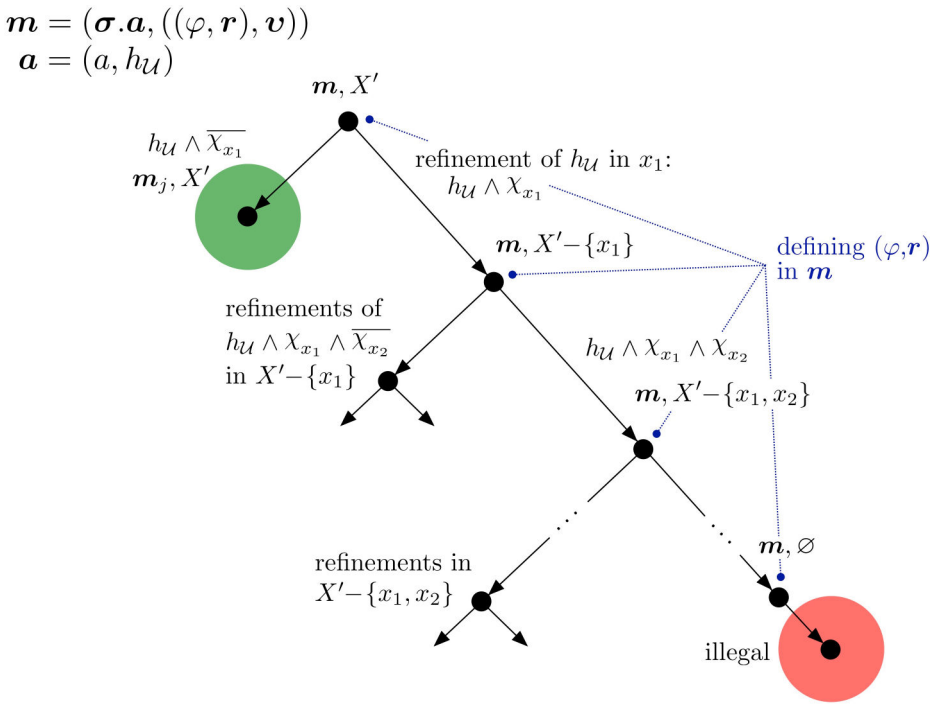


Figure 4.5: Refinement Tree

a single atomic characteristic predicate χ_x is defined to address inconsistencies in $\boldsymbol{m}$. One or more additional mappings are created in the left branch using the atomic characteristic predicate's complement $\overline{\chi_x}$. Each leaf in the tree represents either a legal or illegal mapping. The right most path from the root shows the refinement needed to define the controllable hybrid signal definition $(\varphi, \boldsymbol{r})$. Existing definitions in $(\varphi, \boldsymbol{r})$ are passed on to new mappings associated with the refined stimulus vector sequences in Steps 4.7 and 4.9 of DefineMapping while incomplete definitions are tracked in the parameters C and D of DefineMapping. At the root, the variable x_1 used to refine $h_{\mathcal{U}}$ is removed from X' since the new atomic characteristic predicate associated with x_1 is specifically defined to address inconsistencies in $\boldsymbol{m}$. The second mapping developed using the complement may be refined further in x_1, so X' is not modified in the call to DefineMapping. All left branches of the refinement tree can be refined further in some $x \in X$. All right branches have one or more specially designed characteristic predicates in X. The mapping $\boldsymbol{m}$ is examined one last time when $X' = \varnothing$. If $(\varphi, \boldsymbol{r})$ remains incomplete, $\boldsymbol{m}$ is illegal.

The recursive step of DefineMapping is taken in Step 4.10. This step is illustrated as refinement tree branching. The union of the mapping sets developed from the refined pair is returned.

The total number of mappings returned by a call to DefineMapping is equivalent to the number of leaves in a refinement sub-tree; this includes illegal mappings. The tree in Figure 4.5 depicts j mappings; the highlighted illegal mapping $\boldsymbol{m}$ is the first mapping defined and the highlighted legal mapping $\boldsymbol{m}_j$ is the last mapping defined in the initial call to DefineMapping.

The base of the recursion is determined by [10, Lemma 4.14, p. 8] that shows every sequence in $dom(\mathcal{E})$ is legal and reduced (Step 3.7), legal and extensible (Step 3.12), or illegal (Steps 1.4, 2.4, and 4.15).

4.2.4 Requirements Trace

Design decisions must be made at specific steps in the enumeration process. This is one of the most important aspects of the sequence-based method; instead of being faced with multiple design decisions at once, the enumeration process targets where design decisions must be made. The low-level detail in a hybrid enumeration makes this especially clear. As decisions are made through analysis of requirements, a link between enumeration elements and requirements is established. When requirements are missing the design decision identified during enumeration is reconciled with the product designer (or customer) and new requirements are derived.

Signal interfaces and variable names (the sets I, U, Y, X, and their elements) must be supported by requirements (Steps 1.1–1.4). The codomain of each controllable system variable is also requirements based (Steps 1.6–1.9).

In a discrete enumeration, the empty sequence is always mapped to *null* as a rule of the method; only mappings for extensions of the empty sequence are dictated by requirements. In the hybrid enumeration initialization (phase 2), the set of initial continuous states must be supported by requirements (Steps 2.4–2.7) in addition to the initial trajectory definition (Steps 2.8–2.14).

The design decisions that remain occur in the DefineMapping procedure. Deciding whether or not a signal definition (Step 1.2) or an expression in the condition vector (Step 2.2) is definable as well as the enumeration element definitions (Steps 1.3–1.16 and 2.3–2.14) is based on requirements. This includes legality decisions. Sequence equivalence is based on requirements (Step 3.4).

When an element of the controllable hybrid signal definition cannot be decided, refinement is necessary to resolve ambiguity. The characteristic predicate developed to eliminate the ambiguity must be backed by requirements (Step 4.4).

4.2.5 Relationship of $\mathcal{E}_{\mathcal{H}}$ to Hybrid Sequences

A hybrid enumeration is a template for a set of hybrid sequences. Hybrid sequence $\alpha = \tau_0\, a_1\, \tau_1 \ldots \tau_k\, a_{k+1}\, \tau_{k+1}$ is an *execution fragment* if it satisfies the properties [13, p. 26]:

1. Each τ_i is an element of $\mathcal{T}$.
2. If τ_i is not the last trajectory in α, then a_{i+1} transitions $\tau_i.lstate$ to $\tau_{i+1}.fstate$.

When the first state of τ_0 is an element of Θ, the execution fragment is an *execution* [13, p. 26]. We show the production of an execution α from a hybrid enumeration to establish the real-time relationship between application and theory given the following information:

1. An initial state $\mathbf{x}_0 \in \Theta$
2. An input sequence of action-time pairs $\mathbf{e} = (a_1, t_1), (a_2, t_2), \ldots, (a_k, t_k)$ such that $a_i \in I - \{\, null \,\}$, $t_0 = 0$, $t_i \in \mathsf{T}^{\geq 0}$, and $t_{i-1} \leq t_i$ for $i \in [k]$
3. An input trajectory from $trajs(U)$

A simulated execution starts at $t_0 = 0$. Simulation time t is the amount of time elapsed from t_0. Atomic stimuli characterize evolving system state. A stimulus vector $\boldsymbol{a} = (a, h_Q)$ defines an action. $\boldsymbol{a}$ is enabled at time t when h_Q is satisfied by system state and discrete time input signal $d[t] = a$. When $a = null$, $\boldsymbol{a}$ is enabled as long as h_Q is satisfied. During a simulation, the action defined by $\boldsymbol{a}$ is triggered the instant it is enabled. Let t_i be the simulation time when stimulus vector $\boldsymbol{a}_i$ is first enabled and the i^{th} action is triggered.

In the Lynch algebra, the infimum of every trajectory's domain is 0. The domain of the trajectory that follows the i^{th} action is $[0, t_{i+1} - t_i]$, $[0, t)$, or $[0, \infty)$. In the first case, action $i+1$ defined by stimulus vector $\boldsymbol{a}_{i+1}$ in the corresponding sequence of stimulus vectors closes trajectory τ_i. In the second case, τ_i is undefined at t. In the final case, τ_i is a full trajectory. In the last two cases, the finite executions defined by the enumeration are Zeno and admissible, respectively. Since we control continuous input signals in U by using surrogates in X, this particular type of Zeno hybrid sequence does not arise in enumeration automata.

To show the relationship between a hybrid enumeration and an execution, we use an input sequence $\mathbf{e}$ and the mappings in $\mathcal{E}$ to construct a sequence of stimulus vectors $\boldsymbol{\sigma}$ and the corresponding execution α. With mapping $\boldsymbol{m} = (\lambda, ((\varphi, \boldsymbol{r}), \lambda))$, the initial conditions for trajectory definition φ are provided in $\mathbf{x}_0$. The trajectory definition φ and the initial conditions are sufficient to produce the domain of τ_0 only if $\mathbf{e}$ is the empty sequence and no internal action is generated by the behavior of φ as simulation time t runs without bound. In this special case, $\boldsymbol{\sigma} = \lambda$ and $\alpha = \tau_0$. Otherwise, the domain of τ_0 is undetermined until the next action. The predicates in stimulus vector extensions for each $a \in I$ cover $\mathcal{U}$; therefore, as t progresses from 0 to t_1 an internal action ($a = null$) will always be enabled at $t \leq t_1$. One such stimulus vector corresponds to the invariant set discussed in Section 3.1 (this special case is discussed in detail in Section 4.2.6). Ignoring the invariant case, if h_Q is satisfied at $t < t_1$ in stimulus vector $\boldsymbol{a}_1 = (null, h_Q)$, an internal action precedes the action a_1 in the input sequence. This internal action determines the current construction of $\boldsymbol{\sigma}$ and α independent of the pair (a_1, t_1) in $\mathbf{e}$. Otherwise, $t = t_1$, and stimulus vector $\boldsymbol{a}_1 = (a_1, h_Q)$ determines how $\boldsymbol{\sigma}$ and α are constructed. In either case, the domain of τ_0 is $[0, t]$ and at simulation time t the stimulus vector sequence $\boldsymbol{\sigma} = \boldsymbol{a}_1$ corresponds to execution $\alpha = \tau_0$.

As the execution progresses, $\mathbf{x}_0$ is updated using $\boldsymbol{r}$ from the mapping $\boldsymbol{m} = (\boldsymbol{\sigma}, ((\varphi, \boldsymbol{r}), \boldsymbol{\psi}))$ and $\tau_i.lstate$ where $i = |\boldsymbol{\sigma}| - 1$. Trajectory behavior is established using $\mathbf{x}_0$ and φ. System state is updated as simulation time progresses until either the characteristic predicate of a stimulus vector in $\boldsymbol{S}$ whose first component is *null* is satisfied or the time in the next input sequence pair is reached. When $\boldsymbol{\sigma} \not\triangleright \boldsymbol{\sigma}$ in enumeration mapping $\boldsymbol{m}$, $\boldsymbol{\sigma}$ is replaced with the equivalent (earlier) sequence $\boldsymbol{\psi}$. Future mappings are selected in the partial function $\mathcal{E}$ using the established equivalence relation on stimulus vector sequences.

The times assigned to each t_i for a sequence in $\boldsymbol{S}^*$ are mathematically totally ordered; $t_0 \leq t_1 \leq t_2 \leq \ldots \leq t_k$. By introducing time and internal actions it is possible to produce executions that are unexpected. Surrogate input variables provide a way to avoid one type of Zeno hybrid sequence. Executions containing a sequence of point trajectory - internal action pairs must be considered as well. This presents another type of Zeno hybrid sequence that cannot be avoided using surrogates. Depending on the function $\mathcal{E}$ and the simulation parameters, there could be an uncountable number of internal actions between two pairs in $\mathbf{e}$.

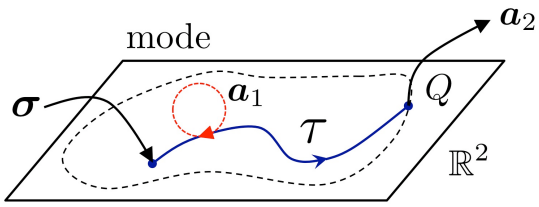


Figure 4.6: Mode Invariant

4.2.6 Continuous Properties of Stimulus Vector Sequences

By introducing timing, continuity, and internal actions into the enumeration, unintended continuous behavior may be introduced into the specification. In this section, we investigate properties of sequences of stimulus vectors whose first coordinate is *null* by showing their relationship to executions.

Definition 4.23. Let $\boldsymbol{\sigma}, \boldsymbol{\sigma}' \in \boldsymbol{S}^*$. $\boldsymbol{\sigma}'$ is a *subsequence* of $\boldsymbol{\sigma}$ if there exists a monotonically increasing $f : dom(\boldsymbol{\sigma}') \to dom(\boldsymbol{\sigma})$ such that $\boldsymbol{\sigma}'(i) = \boldsymbol{\sigma}(f(i))$ and $f(i+1) = f(i) + 1$ for all $i \in dom(\boldsymbol{\sigma}')$ [9, p. 15].

Definition 4.24. Stimulus vector subsequence $\boldsymbol{\sigma}'$ is *autonomous* if $\forall i \in dom(\boldsymbol{\sigma}')$, $\boldsymbol{\sigma}'(i) = (null, h_Q)$; h_Q defines a subset of $\mathcal{U}$.

An autonomous stimulus vector subsequence of length one is called an autonomous stimulus vector. Let $\beta = \tau_i \, a \, \tau_{i+1}$ be the final hybrid sequence fragment associated with a stimulus vector sequence ending in autonomous stimulus vector $\boldsymbol{a}$. h_Q is satisfied by $\tau_i.lstate$ triggering the transition to $\tau_{i+1}.fstate$. The stimulus vector $\boldsymbol{a}$ defines an internal action in H (Definition 3.23).

4.2.6.1 Mode Invariant

Definition 4.25. Stimulus vector sequence $\boldsymbol{\sigma}.\boldsymbol{a}$ where $\boldsymbol{a}$ is autonomous defines a *mode invariant* if $\mapsto (\boldsymbol{\sigma}.\boldsymbol{a}) = (\varphi_0, \boldsymbol{r}_0)$ and $\boldsymbol{\sigma}.\boldsymbol{a} \rhd \boldsymbol{\sigma}$.

For the mode defined by unreduced sequence $\boldsymbol{\sigma}$, the internal action defined by $\boldsymbol{a}_1$ is enabled in h_Q. Q is shown in Figure 4.6 as the subset of $\mathbb{R}^2$ enclosed by the dotted line. Q is the invariant set of the mode defined by $\boldsymbol{\sigma}$. The system progresses continuously according to the *null* hybrid signal definition $(\varphi_0, \boldsymbol{r}_0)$ in Q; τ is produced using the trajectory definition φ_0, and $\boldsymbol{r}_0$ ensures the

continuity of each variable in X. Although the enumeration contains this mapping, the associated transition may not be required in certain implementations. Along trajectory τ, $\boldsymbol{a}_1$ is continuously enabled. The stimulus vector $\boldsymbol{a}_2$ defines the action that exits the mode defined by $\boldsymbol{\sigma}$.

If all the properties of a mode invariant (Definitions 4.25 and 4.19) hold except for the *null* discrete output property, o is no longer a discrete-time signal; the system produces discrete output for all $t \in dom(\tau)$ making $dom(o)$ uncountable. This would be an error in an enumeration. If condition vector coordinate x differs from $x(last)$ with corresponding signal definition $\dot{x} = e$, the signal definition is contradicted by condition vector expression $e' \neq x(last)$. This would also be an enumeration error. To avoid these errors, the hybrid enumeration process is designed to correctly construct mode invariant sequences by default, starting with the initial mapping defined in Step 3.7.

4.2.6.2 Point Modes

Executions that contain a subsequence of point trajectory - internal action pairs may be generated by legal stimulus vector sequences in the domain of $\mathcal{E}$. Executions of this type are generated by autonomous subsequences. We examine two types:

1. An instantaneous, finite sequence of mode transitions that is acyclic.

2. An instantaneous, finite sequence of mode transitions that form a cycle.

Consider stimulus vector sequence $\boldsymbol{\sigma}.\boldsymbol{a}.\boldsymbol{b}.\boldsymbol{c}$ where $\boldsymbol{a}.\boldsymbol{b}.\boldsymbol{c}$ is an autonomous subsequence. $\boldsymbol{\sigma}$, $\boldsymbol{\sigma}.\boldsymbol{a}$, $\boldsymbol{\sigma}.\boldsymbol{a}.\boldsymbol{b}$, and $\boldsymbol{\sigma}.\boldsymbol{a}.\boldsymbol{b}.\boldsymbol{c}$ are legal extensible stimulus vector sequences. Figure 4.7 illustrates this hybrid automaton structure. Let execution, $\tau_0 \ldots \wp(\mathbf{x}_i)\ a\ \wp(\mathbf{x}_{i+1})\ b\ \wp(\mathbf{x}_{i+2})\ c\ \wp(\mathbf{x}_{i+3})$ be a consequence of this sequence of stimulus vectors. Type 1 behavior is characterized by the characteristic predicates in $\boldsymbol{a}, \boldsymbol{b}$, and $\boldsymbol{c}$ being satisfied immediately by $\mathbf{x}_i$, $\mathbf{x}_{i+1}$, and $\mathbf{x}_{i+2}$, respectively. Type 2 behavior occurs when $\boldsymbol{\sigma}.\boldsymbol{a}.\boldsymbol{b}.\boldsymbol{c} \rhd \boldsymbol{\sigma}$.

When the values of $\mathbf{x}_i$, $\mathbf{x}_{i+1}$, and $\mathbf{x}_{i+2}$ in the figure change gradually, the cycle may eventually terminate. For example, when $\mathbf{x}_{i+2}$ triggers the action defined by stimulus vector $\boldsymbol{d}$ and $\boldsymbol{\sigma}.\boldsymbol{a}.\boldsymbol{b}.\boldsymbol{d} \not\rhd \boldsymbol{\sigma}$ the cycle is broken. This type of system behavior is sometimes called chattering [19, p. 15-23]. Chattering is a potentially valuable design feature. For example, anti-locking automobile brakes exhibit chattering behavior. If the values of $\mathbf{x}_i$, $\mathbf{x}_{i+1}$, and $\mathbf{x}_{i+2}$ do not change, such values are called

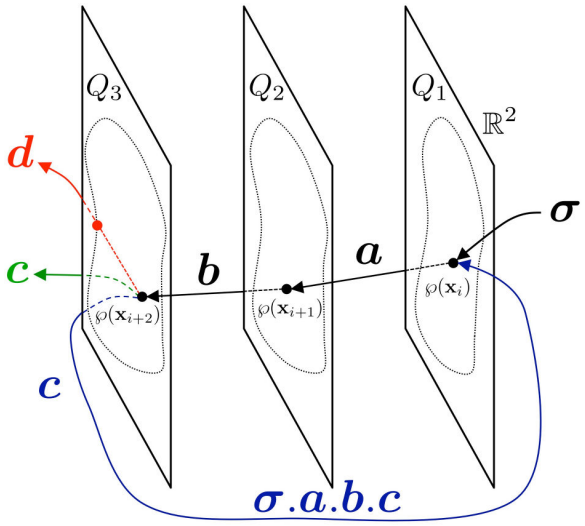


Figure 4.7: Instantaneous Transitions

pinnacles [19, p. 15-21]; the cycle will continue autonomously unless interrupted by a discrete input action. When an interrupt is not part of the design the system exhibits Zeno behavior that results from a type 2 autonomous subsequence.

4.3 Theorizable, Specifiable, Implementable, Realizable

The essence of Zeno's paradox is one must reach the half-way point to a goal prior to reaching the goal itself; therefore, the goal can never be reached. This is an especially discouraging paradox. Figure 4.8 classifies hybrid sequences based on length and their Zeno characteristics as defined in [13, p. 18]. With respect to a hybrid enumeration, an enumeration hybrid automaton, a hybrid I/O automaton, and an implemented model-based design, we classify the executions derivable by enumeration using the terms theorizable, specifiable, implementable, and realizable.

4.3.1 Infinite Hybrid Sequences

$\boldsymbol{S}^*$ is the infinite set of finite stimulus vector sequences and the domain of $\mathcal{E}$ in $\mathcal{E}_\mathcal{H}$. $\boldsymbol{S}^\omega$ is the set of infinite stimulus vector sequences. An execution derived from a stimulus vector sequence of length n has length $2n+1$. Infinite executions in $\boldsymbol{S}^\omega$ whether Zeno or not are classified as theorizable and cannot be specified using a finite enumeration. These classes are shown in light gray in the bottom two boxes of Figure 4.8.

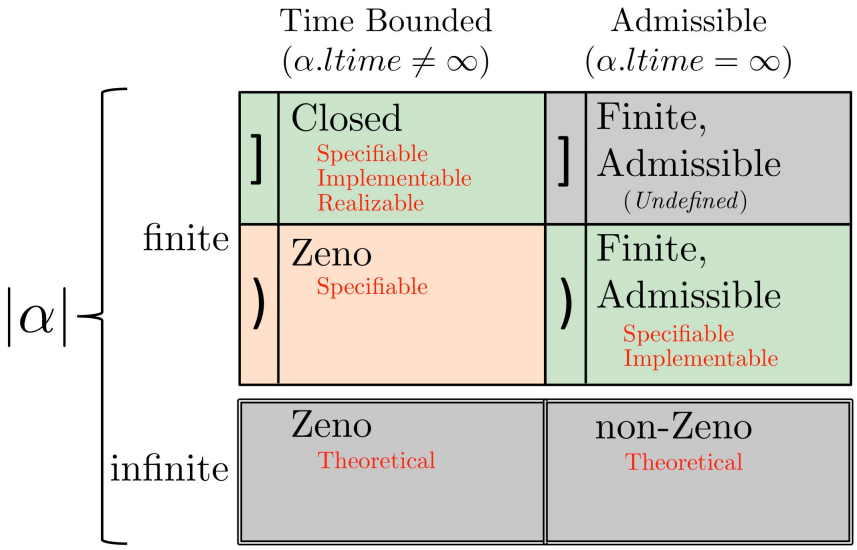


Figure 4.8: Hybrid Sequence Classes

4.3.2 Finite Time-Bounded Zeno Sequences

Finite Zeno executions end in a trajectory that is undefined at some t and therefore has a right-open domain. This class of executions is highlighted in red in Figure 4.8. Surrogate input variables (Definition 4.8) are used to address this type of Zeno execution.

A second type of specifiable Zeno execution is usually associated with infinite hybrid sequences; however, as discussed in Section 4.2.6.2, executions characterized by a finite cycle of mode changes where time does not advance is also specifiable in a finite enumeration. This type of Zeno behavior can be avoided by ensuring there always exists a discrete input action to interrupt the unwanted behavior. For example, a manual emergency stop in an otherwise autonomous system.

4.3.3 Finite Admissible Sequences

Finite, admissible executions are the ideal target of our specification method (specifiable, implementable). In a specification that defines only finite admissible executions, the domain of every trajectory definition is $\mathsf{T}^{\geq 0}$. The ranges of system signals are bounded such that execution does not terminate unexpectedly. $[0, \infty]$ is not a defined interval in $\mathbb{R}$; therefore, the upper right box in Figure 4.8 is gray and labeled "undefined."

4.3.4 Finite Time-Bounded Closed Sequences

The difference between realizable hybrid sequences and unrealizable hybrid sequences is based on the limitations of a digital computer. A primary goal is to correctly construct specifications and then implement models whose hybrid sequences are *capable* of being run forever. Unfortunately, either the computer running the simulation or the person monitoring the simulation wears out and the simulation terminates. This leaves us with the set of realizable hybrid sequences which is the set of finite, closed executions.

Chapter 5

Enumeration Hybrid Automata

5.1 Enumeration Hybrid Automata

An enumeration hybrid automaton, denoted $\mathcal{A}_{\mathcal{E}}$, is a state machine produced from a hybrid enumeration ($\mathcal{E}_{\mathcal{H}}$). Figure 5.1 illustrates the workflow described in this chapter where we convert $\mathcal{E}_{\mathcal{H}}$ into $\mathcal{A}_{\mathcal{E}}$ and $\mathcal{A}_{\mathcal{E}}$ into a HIOA. An enumeration hybrid automaton is an implementable specification that incorporates continuous functions, time, and exhibits deterministic behavior beyond the initial system state.

Definition 5.1. An *enumeration hybrid automaton* $\mathcal{A}_{\mathcal{E}}$ is a tuple:

$$(Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B}). \tag{5.1}$$

1. Q is the set of legal system states.
2. $\Theta \subseteq Q$ is the set of start states.
3. I is the finite set of discrete input actions; $null \in I$.
4. $V = U \cup X \cup Y$ is the finite set of system variables. For each $v \in V$, $type(v)$ is the set of values assignable to v.
5. R is the finite set of condition vectors developed during enumeration.
6. Φ is the finite set of trajectory definitions developed during enumeration.

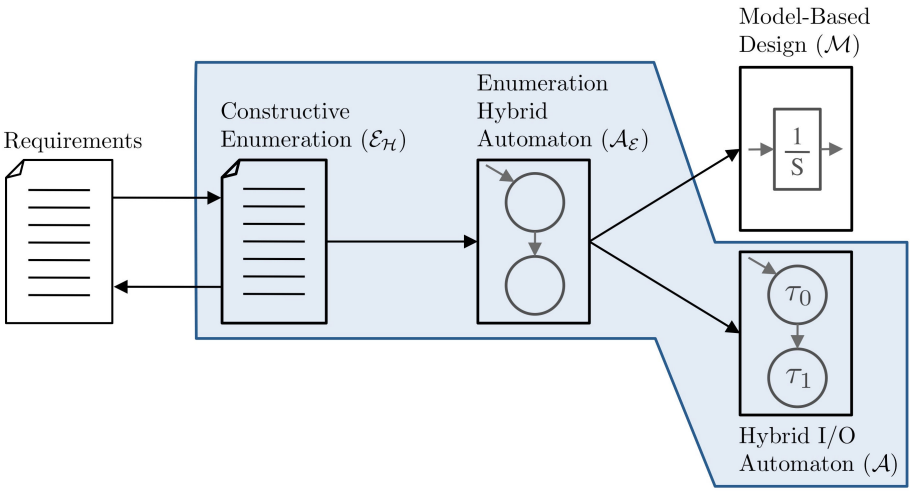


Figure 5.1: Process: Enumeration ($\mathcal{E}_\mathcal{H}$) to Specification ($\mathcal{A}_\mathcal{E}$) to Algebra ($\mathcal{A}$)

7. Mode variable $\mu \in X$, and $\mu : \mathsf{T}^{\geq 0} \to \mathbb{N}$ is the signal definition for μ.

8. $\mathcal{Q} : Q/\rho \times I \to R$ is the condition function mapping blocks of Q to conditions vectors. Q/ρ is the finite partition of Q discovered during enumeration.

9. $\mathcal{B} : \mathit{type}(\mu) \to \Phi$ is the behavior function mapping system modes to trajectory definitions.

In $\mathcal{A}_\mathcal{E}$, we capitalize on surrogate input variables to eliminate the Zeno behaviors caused by executions terminating in an open trajectory (see Definition 3.15). The function $\mathcal{Q}$ maps blocks induced by ρ paired with a discrete input action, possibly *null*, to condition vectors. Each condition vector maps the atomic states in a block to an instantaneous discrete output, possibly *null*, and the next atomic system state. Each mode is associated with a single trajectory definition by $\mathcal{B}$.

5.1.1 Construction Algorithm: $\mathcal{E}_\mathcal{H}$ to $\mathcal{A}_\mathcal{E}$

The algorithm ConstructA$_\mathcal{E}$ constructs $\mathcal{A}_\mathcal{E}$ from $\mathcal{E}_\mathcal{H}$. During construction, unreduced stimulus vector sequences (modes) are mapped by f to non-negative integers in the codomain of the continuous-time signal $\mu : \mathsf{T}^{\geq 0} \to \mathbb{N}$. f is constructed and used in ConstructA$_\mathcal{E}$, but it is not part of the definition of $\mathcal{A}_\mathcal{E}$. For the duration of every trajectory, μ is a discrete variable that does not change (see Section 3.6). The range of each system signal $\mathit{type}(v)$ is defined in $\mathcal{A}_\mathcal{E}$ using unreduced stimulus vector sequences and the characteristic predicates in extensions of those sequences; the set of legal system states Q is defined using the set of signal ranges.

CONSTRUCT$\mathcal{A}_{\mathcal{E}}$($\mathcal{E}_{\mathcal{H}}$)

The mode variable μ identifies discrete system state. Lexicographically μ is first in the variable order. The signal $\mu : \mathsf{T}^{\geq 0} \to \mathbb{N}$ describes the behavior of this variable.

1.1 declare mode variable μ

1.2 $X \leftarrow X \cup \{ \mu \}$

1.3 $C_{\mu} \leftarrow \mathbb{N}$

Initialize the continuous state space.

1.4 $\mathcal{U} \leftarrow \times_{x \in X} C_x$

Define a function f to map the set of unreduced sequences to a finite set of non-negative integers that represent system mode.

1.5 $f \leftarrow \varnothing$

Establish a type set for each state variable. This set will be the range of the associated variable's signals.

1.6 **for** each $x \in X$

1.7 $\quad type(x) \leftarrow \varnothing$

1.8 **end for**

Define the type set for each external variable (see Definition 3.23). Since the values of these variables are not completely controllable, the codomain is used.

1.9 **for** each $w \in W$

1.10 $\quad type(w) \leftarrow C_w$

1.11 **end for**

Each enumeration hybrid automaton has a set of trajectory definitions (Φ), condition vectors (R), and two functions: $\mathcal{B}$ relates modes to trajectory definitions; $\mathcal{Q}$ defines the follow-on condition vectors.

1.12 $\Phi \leftarrow \varnothing$

1.13 $\mathcal{B} \leftarrow \varnothing$

1.14 $R \leftarrow \varnothing$

1.15 $\mathcal{Q} \leftarrow \varnothing$

The set of legal start states.

1.16 $Q \leftarrow \varnothing$

1.17 $i \leftarrow 0$

Each mapping in the partial function $\mathcal{E}$ has the form $\boldsymbol{m} = (\boldsymbol{\sigma}, ((\varphi, \boldsymbol{r}), \boldsymbol{\psi}))$. The previous mode for $\boldsymbol{\sigma} \neq \lambda$ is based on the prefix of $\boldsymbol{\sigma}$, $init(\boldsymbol{\sigma})$. The order ensures $f(\lambda) = 1$.

2.1 **for** each $\boldsymbol{m}$ in $\mathcal{E}$ ordered by sequence length and then lexicographically [2.38]

Every sequence in $dom(\mathcal{E})$ is either legal and reduced, legal and unreduced, or illegal. If a sequence is unreduced (legal or illegal), it is transformed into state data in $range(f)$.

2.2 **if** $\triangleright (\boldsymbol{\sigma}) \notin dom(f)$ [2.15]

2.3 **then**

The stimulus vector sequence $\boldsymbol{\sigma}$ is illegal (see Definition 4.19). The integer 0 represents a common illegal mode. Illegal stimulus vector sequences are equivalent (Definition 4.22).

2.4 **if** $\mapsto (\boldsymbol{\sigma}) = \Omega$ [2.10]

2.5 **then**

2.6 $f(\triangleright (\boldsymbol{\sigma})) \leftarrow 0$

The stimulus vector sequence $\boldsymbol{\sigma}$ is legal.

2.7 **else**

2.8 $i \leftarrow i + 1$

2.9 $f(\triangleright (\boldsymbol{\sigma})) \leftarrow i$

2.10 **end if** [2.4]

Each mode has a trajectory definition. Modify the trajectory definition to reflect the mode variable's behavior; it is always a discrete variable (see Section 4.1.2).

2.11 $\varphi \leftarrow \pi_{\Phi} \mapsto (\boldsymbol{\sigma})$

2.12 $\varphi(\mu) \leftarrow \dot{\mu} = 0$

2.13 $\Phi \leftarrow \Phi \cup \{ \varphi \}$

2.14 $\mathcal{B} \leftarrow \mathcal{B} \cup \{ (f(\triangleright (\boldsymbol{\sigma})), \varphi) \}$

2.15 **end if** [2.2]

The empty sequence is used to define the set of start states.

2.16 **if** $\boldsymbol{\sigma} = \lambda$ [2.30]

2.17 **then** [2.21]

The set of start states must incorporate mode. Redefine the set of start states using Θ *from* $\mathcal{E}_{\mathcal{H}}$ *and* $\mathcal{U}$.

2.18 h_{Θ} is the characteristic predicate for Θ in $\mathcal{E}_{\mathcal{H}}$

2.19 $\chi_{\mu} \triangleq \mu = f(\lambda)$

2.20 $h_Q \triangleq h_{\Theta} \wedge \chi_{\mu}$

2.21 $\Theta \leftarrow \{ \mathbf{x} \in \mathcal{U} \mid h_Q(\mathbf{x}) \}$ [2.17]

2.22 **else** [2.29]

Define the atomic characteristic predicate for the system mode where the action takes place.

2.23 $\chi_{\mu} \triangleq \mu = f(init(\boldsymbol{\sigma}))$

Projection is used to access h_Q *in the stimulus vector (see Definition 4.4).*

2.24 $h_Q \triangleq \pi_P(last(\boldsymbol{\sigma})) \wedge \chi_{\mu}$

2.25 $a \leftarrow \pi_I(last(\boldsymbol{\sigma}))$

2.26 $\boldsymbol{r} \leftarrow \pi_R \mapsto (\boldsymbol{\sigma})$

2.27 $\boldsymbol{r}(\mu) \leftarrow f(\triangleright (\boldsymbol{\sigma}))$

2.28 $R \leftarrow R \cup \{ \boldsymbol{r} \}$

2.29 $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{ ((h_Q, a), \boldsymbol{r}) \}$ [2.22]

2.30 **end if** [2.16]

Add legal states to the type sets and Q.

2.31 **if** $\boldsymbol{\sigma} \not\mapsto \Omega$ [2.37]

2.32 **then**

Update the type set definition (legal values) for each variable in X including μ. χ_x may be the conjunction of two or more atomic characteristic predicates based on the refinement process.

2.33 **for** each $x \in X$

2.34 $type(x) \leftarrow type(x) \cup \{\, x \in C_x \mid \chi_x(x) \,\}$

2.35 **end for**

$Q \subseteq \mathcal{U}$ is the set of legal system states. Update Q based on h_Q.

2.36 $Q \leftarrow Q \cup \{\, \mathbf{x} \in \mathcal{U} \mid h_Q(\mathbf{x}) \,\}$

2.37 **end if** [2.31]

2.38 **end for** [2.1]

Q is defined. All the states in $\mathcal{U} - Q$ are illegal. Define $\boldsymbol{r}$ using the definition for the illegal hybrid sequence (Definition 4.19) and add the condition function mapping for the set of illegal states.

3.1 $\boldsymbol{r}(o) \leftarrow \omega$

3.2 **for** each $x \in X$

3.3 $\boldsymbol{r}(x) \leftarrow x(last)$

3.4 **end for**

3.5 $R \leftarrow R \cup \{\, \boldsymbol{r} \,\}$

3.6 **for** each $a \in I$

3.7 $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\, ((h_{\mathcal{U}-Q},\, a),\, \boldsymbol{r}) \,\}$

3.8 **end for**

3.9 **return** $(Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$

end CONSTRUCT$\mathcal{A}_\mathcal{E}$

5.1.2 Relationship Between $\mathcal{A}_\mathcal{E}$ and M (Enumeration Mealy Machines)

An Enumeration Mealy Machine, M, as defined in [10, Definition 5.1, p. 10] is a Mealy machine that satisfies five conditions that place the set of Enumeration Mealy Machines in one-to-one correspondence with the set of finite discrete enumerations. The theorems below show the relationship between $\mathcal{A}_\mathcal{E}$ and M.

Theorem 5.1. $\mathcal{A}_\mathcal{E}$ = CONSTRUCT$\mathcal{A}_\mathcal{E}$($(\mathcal{E}, \varnothing, \varnothing)$) *is an Enumeration Mealy Machine* $M = (Q', \Sigma, \Delta, \delta, \nu, q_0)$.

Proof. In $\mathcal{E}_\mathcal{H} = (\mathcal{E}, \varnothing, \varnothing)$, $X = \varnothing$; therefore, $X = \{\, \mu \,\}$ in $\mathcal{A}_\mathcal{E}$. For $\mu \in X$ in $\mathcal{A}_\mathcal{E}$, let $n = |type(\mu)|$ and $type(\mu) = \{\, 0, 1, \ldots, n-1 \,\}$. Let $Q' = \{\, q_i \mid i \in type(\mu) \,\}$. Let $f : type(\mu) \to Q'$ map modes in $\mathcal{A}_\mathcal{E}$ to states in M as follows:

$$f(i) = \begin{cases} q_{n-1} \text{ where } n = |type(\mu)| & i = 0 \\ q_{i-1} & i > 0 \end{cases}$$

$\Theta = \{\, 1 \,\}$ (Step 2.1 in CONSTRUCT$\mathcal{A}_\mathcal{E}$). $q_0 = f(1)$. Let $\Sigma = I$. In $\mathcal{A}_\mathcal{E}$, $R \subseteq E_o \times E_\mu$. $X = \{\, \mu \,\}$; therefore, E_o is a finite set. Let $\Delta = E_o$. By Definition 4.14, $\{\, null, \omega \,\} \subseteq E_o$. For each mapping in $\mathcal{Q}$, the map $(((\mu = i), a), \boldsymbol{r}) \mapsto ((f(i), a), f(\pi_\mu(\boldsymbol{r})))$ defines δ in M, and the map $(((\mu = i), a), \boldsymbol{r}) \mapsto ((f(i), a), \pi_o(\boldsymbol{r}))$ defines ν. For all $i \in type(\mu)$, $\mathcal{B}(i) = (\dot{\mu} = 0)$; therefore, μ changes only during the transitions in $\mathcal{Q}$.

Condition 1: There are n states in M; q_{n-1} is the illegal state.

Condition 2: Every sequence in $\mathcal{E}$ is reduced or extensible; therefore, M is connected.

Condition 3: The relationship between canonical sequence ordering and states in M is satisfied by Step 2.1 in CONSTRUCT$\mathcal{A}_\mathcal{E}$.

Conditions 4, 5: The properties of the illegal state are satisfied by Steps 2.1-2.9 in CONSTRUCT$\mathcal{A}_\mathcal{E}$.

□

Theorem 5.2. *Given Enumeration Mealy Machine* $M = (Q', \Sigma, \Delta, \delta, \nu, q_0)$*, an* $\mathcal{A}_{\mathcal{E}} = (Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$ *can be constructed with equivalent behavior.*

Proof. Let $f : Q' \to \mathbb{N}$ map states in M to modes in $\mathcal{A}_{\mathcal{E}}$ as follows:

$$f(q_i) = \begin{cases} 0 & \exists a \in \Sigma \text{ such that } (\nu(q_i, a) = \omega \wedge \delta(q_i, a) = q_i) \\ i + 1 & \text{Otherwise} \end{cases}$$

Let $V = X$ and $X = \{ \mu \}$. Let $Q = range(f)$, $type(\mu) = Q$, and $E_\mu = type(\mu) \cup \{ null \}$. Let $\Theta = \{ f(q_0) \}$. Let $E_o = \Delta$. By the definition of an Enumeration Mealy Machine, $\{ null, \omega \} \subseteq \Delta$. Let $R = E_o \times E_\mu$. Let $I = \Sigma \cup \{ null \}$.

For the mappings in δ and ν, the map $((q, a), q')$, $((q, a), r) \mapsto (((\mu = f(q)), a), \boldsymbol{r})$ where $\pi_\mu(\boldsymbol{r}) = f(q')$ and $\pi_o(\boldsymbol{r}) = r$ defines the function $\mathcal{Q}$ in $\mathcal{A}_{\mathcal{E}}$. The state variable signal definition for μ is $\dot{\mu} = 0$; $\varphi = (\dot{\mu} = 0)$; $\Phi = \{ \varphi \}$, and $\mathcal{B} = \{ (i, \varphi) \mid i \in type(\mu) \}$. □

5.1.3 Construction Algorithm: $\mathcal{A}_{\mathcal{E}}$ to $\mathcal{A}$

To prove our specification satisfies the theoretical properties of a HIOA, ConstructA is presented to transform the enumeration elements in $\mathcal{A}_{\mathcal{E}}$ into the atomic elements in $\mathcal{A}$. The result is a well-defined HIOA where the axioms are preserved.

Lemma 3.4 of [13, p. 15] proves $trajs(V)$ with prefix ordering is an algebraic complete partial order. ConstructA only constructs compact trajectories from $\mathcal{A}_{\mathcal{E}}$. This constructed set of compact trajectories approximates every trajectory in $trajs(V)$ by Definition 3.30.

$\mathcal{A}_{\mathcal{E}}$ is deterministic after establishing the initial start state. One trajectory definition in $\mathcal{A}_{\mathcal{E}}$ determines the precise behavior of $\mathcal{A}$ from any state in Q as a trajectory. If more than one trajectory definition are associated with a single mode, nondeterminism with respect to trajectories would be introduced during the construction of $\mathcal{A}$. The time of the next action determines the domain of the trajectory. The *null* element in I is removed, and the set H is defined by naming the pairs in the domain of $\mathcal{Q}$ that have the form $(h_Q, null)$. O is defined as the range of discrete-time signal o. There remains an element of nondeterminism in $\mathcal{A}$ with respect to output actions. Given a state in Q, an output action may exist in $\mathcal{D}$ that is triggered by an action in $I \cup H$ other than intended. By completing the set A, $\mathcal{D}$ can be defined.

$\mathbf{x}$, $\mathbf{u}$, and $\mathbf{y}$ are valuations constructed from atomic vectors. The relationship between atomic vectors and valuations is established. Given a set of atomic vectors (Definition 4.1) over $\mathcal{S}$ with $s_i \in \mathcal{S}$ and $v_i \in C_{s_i}$ for $i \in [|\mathcal{S}|]$, the map from atomic vectors to valuations is defined by

$$(v_1, \ldots, v_n) \mapsto \{\, (s_1, v_1), \ldots, (s_n, v_n) \,\}. \tag{5.2}$$

In the construction algorithm, $\tilde{X}$ is the set of surrogate input variables in X, $\tilde{\boldsymbol{f}}$ denotes the composite surrogate signal definition associated with φ, and $\tilde{\mathbf{x}} \in val(\tilde{X})$. Let $(\,(h_Q, a), \boldsymbol{r}\,) \in \mathcal{Q}$ and $\varphi \in \Phi$ define the trajectory prior to the transition. If for all $x \in X$, $\pi_x(\boldsymbol{r}) = x(last)$ when $\pi_x(\varphi) = (\dot{x} = e)$ and $\pi_x(\boldsymbol{r}) = null$ otherwise, then $\boldsymbol{r}$ is invariant.

CONSTRUCT$\mathcal{A}$($\mathcal{A}_\mathcal{E}$)

The input and internal action sets.

1.1 $O \leftarrow \varnothing$

1.2 $H \leftarrow \varnothing$

An index for internal action names.

1.3 $i \leftarrow 0$

The h_Q of the pairs in the domain of $\mathcal{Q}$ partition Q. A mapping in $\mathcal{Q}$ defines either an invariant transition or an action transition.

1.4 **for** each $(\,(h_Q,\, a),\, \boldsymbol{r}\,) \in \mathcal{Q}$ [1.59]

Transitions and trajectories are constructed from every state in Q. Given $\mathbf{x}$, $\wp(\mathbf{x})$ is defined.

1.5 **for** each $\mathbf{x} \in \{\, \mathbf{x} \in \mathcal{U} \mid h_Q(\mathbf{x}(\mu), \mathbf{x}(x_1), \ldots, \mathbf{x}(x_k)) \,\}$, $k = |X| - 1$ [1.58]

Each trajectory is constructed from the trajectory definition. Since there is only one trajectory definition for each mode, this eliminates nondeterminism in the constructed $\mathcal{A}$.

1.6 $\varphi \leftarrow \mathcal{B}(\mathbf{x}(\mu))$

1.7 **if** $a = null$ and $\boldsymbol{r}$ is invariant [1.57]

1.8 **then** [1.26]

Construct the trajectories in $\mathcal{T}$ starting with a point trajectory; $\tau = \wp(\mathbf{v})$. Thus, $\wp(\mathbf{x})$ is defined.

1.9 $t \leftarrow 0$

1.10 **do** [1.26]

The trajectory over X is fixed; this includes the surrogates. There is a many-to-one relationship between input trajectories and surrogate trajectories. $\mathcal{T}$ must include all trajectories that satisfy the signal definitions in φ.

1.11 **for** each $\tau{\downarrow}U \in trajs(U)$ such that $dom(\tau{\downarrow}U) = [0,t]$ and for all $t' \in [0,t]$, $\tilde{\boldsymbol{f}}(\tau{\downarrow}U(t')) = \tau{\downarrow}\tilde{X}(t')$ [1.23]

$\tau{\downarrow}U$ and $\tau{\downarrow}\tilde{X}$ are fixed over the domain $[0,t]$.

1.12 **for** each $z \in Z$ and z is not a surrogate [1.21]

1.13 **case** membership of z [1.20]

1.14 **when** $z \in X$ and $\pi_z(\varphi)$ is a differential equation

1.15 $\tau{\downarrow}z(t) \leftarrow$ solution of $\pi_z(\varphi)$ at t with $\mathbf{x}(z)$ and $\tau{\downarrow}X(t)$

1.16 **when** $z \in X$ and $\pi_z(\varphi)$ is an equation

1.17 $\tau{\downarrow}z(t) \leftarrow$ solution of $\pi_z(\varphi)$ at t with $\tau{\downarrow}X(t)$

1.18 **when** $z \in Y$

1.19 $\tau{\downarrow}z(t) \leftarrow$ solution to $\pi_z(\varphi)$ at t with $\tau{\downarrow}U(t)$ and $\tau{\downarrow}X(t)$

1.20 **end case** [1.13]

1.21 **end for** [1.12]

τ has been completely defined over the domain $[0,t]$. $\tau.lstate$ is fixed for every $\tau{\downarrow}W \in trajs(W)$ used to construct τ.

1.22 $\mathcal{T} \leftarrow \mathcal{T} \cup \{\tau\}$

1.23 **end for** [1.11]

After all trajectories with domain $[0,t]$ are added to $\mathcal{T}$, h_Q is checked to determine whether the system remains in the invariant set or an action in I or H has occurred.

1.24 $\mathbf{x}' \leftarrow \tau.lstate$

1.25 $t \leftarrow$ the next greater $t \in \mathsf{T}^{\geq 0}$ (strict order)

1.26 **while** $h_Q(\mathbf{x}'(\mu), \mathbf{x}'(x_1), \ldots, \mathbf{x}'(x_k))$ [1.8, 1.10]

1.27 **else** [1.56]

The point trajectory over X is fixed; this includes the surrogates. There is a many-to-one relationship between input point trajectories and the point trajectory over $\tilde{X}$. $\mathcal{T}$ must include all point trajectories that satisfy the signal definitions in φ.

1.28 **for** each $\wp(\mathbf{u}) \in trajs(U)$ such that $\tilde{\boldsymbol{f}}(\wp(\mathbf{u})(0)) = \wp(\tilde{\mathbf{x}})(0)$ [1.33]

1.29 **for** each $y \in Y$

1.30 $\wp(y)(0) \leftarrow \pi_y(\varphi)$ evaluated at $t = 0$ with $\wp(\mathbf{u})$ and $\wp(\mathbf{x})$

1.31 **end for**

1.32 $\mathcal{T} \leftarrow \mathcal{T} \cup \{\, \wp(\mathbf{v}) \,\}$

1.33 **end for** [1.28]

Use the condition vector mapping to define the initial atomic system state following the input or internal action. When a state variable's signal definition is a differential equation, construct the next atomic state value using the condition vector. Otherwise, define the next atomic state value explicitly at $t = 0$ using the signal definition.

1.34 **for** each $x \in X$ [1.41]

1.35 **if** $\pi_x(\boldsymbol{r}) \neq null$ [1.40]

1.36 **then**

1.37 $\mathbf{x}'(x) \leftarrow \pi_x(\boldsymbol{r})$ evaluated with $\mathbf{x}$

1.38 **else**

1.39 $\mathbf{x}'(x) \leftarrow$ the solution of $\pi_x(\varphi)$ at $t = 0$

1.40 **end if** [1.35]

1.41 **end for** [1.34]

1.42 **if** $a = null$ [1.50]

Name an internal action. Then, add it to the set of internal actions. Add the transition caused by this action to the set of transitions.

1.43 **then**

1.44 $i \leftarrow i + 1$

1.45 $h_i \triangleq h_Q$

1.46 $H \leftarrow H \cup \{ h_i \}$

1.47 $\mathcal{D} \leftarrow \mathcal{D} \cup \{ (\mathbf{x}, h_i, \mathbf{x}') \}$

1.48 **else**

Add an input action.

1.49 $\mathcal{D} \leftarrow \mathcal{D} \cup \{ (\mathbf{x}, a, \mathbf{x}') \}$

1.50 **end if** [1.42]

1.51 **if** $\pi_o(\boldsymbol{r}) \neq null$ [1.56]

1.52 **then**

Add an output action. $\mathbf{x}$ *represents* $x(last)$ *for each* $x \in X$*. This is the only nondeterminism that is in* $\mathcal{A}$*.*

1.53 $o[0] \leftarrow \pi_o(\boldsymbol{r})$ evaluated with $\mathbf{x}$

1.54 $O \leftarrow O \cup \{ o[0] \}$

1.55 $\mathcal{D} \leftarrow \mathcal{D} \cup \{ (\mathbf{x}', o[0], \mathbf{x}') \}$

1.56 **end if** [1.27, 1.51]

1.57 **end if** [1.7]

1.58 **end for** [1.5]

1.59 **end for** [1.4]

The dtype sets contain all the atomic signals used to form the trajectories in $\mathcal{T}$*.*

1.60 **for** each $\tau \in \mathcal{T}$

1.61 **for** each $v \in V$

1.62 $dtype(v) \leftarrow dtype(v) \cup \{ \tau \downarrow v \}$

1.63 **end for**

1.64 **end for**

1.65 $I \leftarrow I - \{ null \}$

1.66 $W \triangleq U \cup Y$

1.67 $E \triangleq I \cup O$

1.68 **return** $(W, X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$

end CONSTRUCT$\mathcal{A}$

5.1.4 Relationship Between $\mathcal{A}_\mathcal{E}$ and $\mathcal{A}$

In this section, we show that $\mathcal{A}_\mathcal{E}$ satisfies the HIOA axioms of Definition 3.23 and Definition 3.24. Only output action nondeterminism remains in the constructed $\mathcal{A}$. Trajectory behavior is selected deterministically in the constructed $\mathcal{A}$ by inclusion of the mode variable in elements of Q. Certain types of Zeno behavior is also eliminated in $\mathcal{A}_\mathcal{E}$ through use of surrogate input variables.

Lemma 5.1. CONSTRUCT$\mathcal{A}$ $(\mathcal{A}_\mathcal{E})$ *where* $\mathcal{A}_\mathcal{E} = (Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$ *satisfies Axiom T0.*

Proof. Let $\mathbf{v}$ be an arbitrary valuation in $val(V)$ and $\mathbf{v}|_X \in Q$. Let $\mathbf{x} = \mathbf{v}|_X$. The predicates of the pairs in the domain of $\mathcal{Q}$ partition Q; therefore, there exists an $(\,(h_Q,\, a),\ \boldsymbol{r}\,) \in \mathcal{Q}$ (Step 1.4) such that h_Q is satisfied by $\mathbf{x}$. The mapping $(\,(h_Q,\, a),\ \boldsymbol{r}\,)$ defines an invariant set or a transition. In the first case, the first iteration through the loop starting at Step 1.10 and ending at Step 1.26 constructs every point trajectory where $\mathbf{x} \in Q$. In the second case, Steps 1.28 to 1.33 constructs every point trajectory where $\mathbf{x} \in Q$ and $\mathbf{x}$ initiates an automaton transition. For arbitrary $\mathbf{x}$, every $\mathbf{w} \in val(W)$ is used to define $\mathbf{v} \in val(V)$. $\wp(\mathbf{v})$ is added to $\mathcal{T}$. □

Lemma 5.1 shows the many-to-one relationship between the point trajectories in $trajs(W)$ and the point trajectories in $trajs(X)$. The automaton can report the exact value of any input signal as an output signal. Surrogate input variables ensure the internal automaton behavior is controlled. If an action transitions the automaton to a state where another action is enabled, Lemma 5.1 proves a point trajectory exists in $\mathcal{T}$ that allows the two actions to occur simultaneously. During simulation, event times are strictly ordered and every trajectory in the algebra has a non-degenerate domain; the point trajectories of HIOA Axiom T0 would not not exist.

Lemma 5.2. CONSTRUCT$\mathcal{A}$ $(\mathcal{A}_\mathcal{E})$ *where* $\mathcal{A}_\mathcal{E} = (Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$ *satisfies Axiom T1.*

Proof. Let τ be an arbitrary trajectory constructed from $\varphi \in \Phi$ (Step 1.6) and τ' be an arbitrary prefix of τ. If $\tau.ltime = 0$, Lemma 5.1 completes the proof; otherwise, $\tau.ltime > 0$. The loop starting at Step 1.10 defines every prefix of τ and adds it to $\mathcal{T}$. □

Lemma 5.3. CONSTRUCT$\mathcal{A}(\mathcal{A}_\mathcal{E})$ *where* $\mathcal{A}_\mathcal{E} = (Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$ *satisfies Axiom T2.*

Proof. Let τ be an arbitrary trajectory constructed from $\varphi \in \Phi$ (Step 1.6 and the loop starting at Step 1.10) and $t \in dom(\tau)$ be an arbitrary time. If $\tau.ltime = 0$, Lemma 5.1 completes the proof; otherwise, $\tau.ltime > 0$. τ is constructed using the invariant set and $t \in [0, \tau.ltime]$. Let $\mathbf{x}' = \tau(t)|_X$. $\mathbf{x}'$ either satisfies h_Q in Step 1.26 or it does not. In the first case, $\mathbf{x}'$ also satisfies the characteristic predicate in Step 1.5. Let $\mathbf{x} = \mathbf{x}'$. The suffix of τ with initial state $\mathbf{x}$, is constructed in the loop starting at Step 1.10 and ending at Step 1.26. In the second case, $\mathbf{x}'$ satisfies a transition characteristic predicate. $\wp(\mathbf{x}')$ is the suffix of τ constructed and added to $\mathcal{T}$ in Steps 1.28 - 1.33. □

Lemma 5.4. CONSTRUCT$\mathcal{A}(\mathcal{A}_\mathcal{E})$ *where* $\mathcal{A}_\mathcal{E} = (Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$ *satisfies Axiom T3.*

Proof. Let $\tau_0, \tau_1, \ldots, \tau_n$ be a finite sequence of arbitrary trajectories constructed from $\varphi \in \Phi$ such that for non-final index i, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.lstate$. We construct the concatenation as τ. $\tau_i.lstate|_\mu = \tau_{i+1}.lstate|_\mu$ for $i \in [n-1]$; therefore, all the trajectories have the same trajectory definition. If for any non-final index i, a transition is caused by a in $\tau_i.lstate$ satisfying a pair in the domain of $\mathcal{Q}$, the result is either a discrete output or an invariant transition. The first case is a contradiction by the definition of a discrete-time signal; therefore, for all non-final i, $\tau_i.lstate$ satisfies an invariant h_Q producing τ. □

Theorem 5.3. CONSTRUCT$\mathcal{A}(\mathcal{A}_\mathcal{E})$ *is a Hybrid Automaton.*

Proof. By Lemmas 5.1, 5.2, 5.3, and 5.4. □

We now consider the Hybrid I/O Automata axioms of Definition 3.23.

Lemma 5.5. CONSTRUCT$\mathcal{A}(\mathcal{A}_\mathcal{E})$ *where* $\mathcal{A}_\mathcal{E} = (Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$ *satisfies Hybrid I/O Automata Axiom E1 of Definition 3.24.*

Proof. Let $\mathbf{x} \in Q$ and $a \in I - \{ null \}$. There exists an $(\,(h_Q,\, a),\ \boldsymbol{r}\,) \in \mathcal{Q}$ such that $h_Q(\mathbf{x}(\mu), \mathbf{x}(x_1), \ldots, \mathbf{x}(x_k))$ by the definition of $\mathcal{Q}$. By Step 1.4, the transition is defined in CONSTRUCT$\mathcal{A}(\mathcal{A}_\mathcal{E})$. □

In the enumeration process, we introduce surrogate input variables and their signal definitions to handle unexpected input signal behavior (Definition 4.8). For surrogate signal definition $\tilde{x}(t)$ defined on $u \in U$, an internal action is implicitly defined by $\tilde{x}(t)$ at any time t where $\tilde{x}(t) \neq u(t)$; the implicit action changes $u(t)$ to $\tilde{x}(t)$ at time t.

Lemma 5.6. CONSTRUCT$\mathcal{A}(\mathcal{A}_{\mathcal{E}})$ *where* $\mathcal{A}_{\mathcal{E}} = (Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$ *satisfies Hybrid I/O Automata Axiom E2 of Definition 3.24.*

Proof. Let $\mathbf{x} \in Q$ and $\tau{\downarrow}U \in trajs(U)$. Let $\tau{\downarrow}\tilde{X}$ be the trajectory produced by φ using variables in $\tilde{X}$. If $\tau{\downarrow}\tilde{X} = \tau{\downarrow}U$, then E2 Case 1 is satisfied; otherwise, there exists some internal action(s) realized by one or more surrogate input variable signal definitions f that transforms $\tau{\downarrow}U$ into $\tau{\downarrow}\tilde{X}$. This set of internal actions is enabled for all $\mathbf{x}$ in all $\tau \in \mathcal{T}$ by Step 1.11 and Step 1.28. □

Theorem 5.4. CONSTRUCT$\mathcal{A}(\mathcal{A}_{\mathcal{E}})$ *is a Hybrid I/O Automaton (Definition 3.24).*

Proof. By Theorem 5.3, Lemma 5.5, and Lemma 5.6. □

Chapter 6

Hybrid Specification Implementations

This chapter examines Simulink and Stateflow implementations of enumeration hybrid automata specifications. With a complete hybrid enumeration, the model building problem has been prescribed to the point that it can be generated algorithmically. The process outlined in Figure 6.1 places the design focus on systematic interpretation and refinement of requirements guided by the enumeration procedure. This work eliminates errors and improves the implementation.

The sections in this chapter relate the structures of an enumeration hybrid automaton to the Simulink and Stateflow design languages. A transformation algorithm is provided for each language. Various techniques are used in industry to produce implementations in Simulink or Stateflow. The techniques presented in this chapter have been developed to show the links between the specification and the model implementation.

With imperative program design, sequential program statements produce the desired program function. Stateflow charts are based on classical state machine definitions and their function follows the sequential programming paradigm. Stateflow implementation of an enumeration hybrid automaton is presented first. A Simulink model, on the other hand, is a data-flow program in which each model block is computed when necessary at each simulation time step; computations appear to be performed in parallel. The approach used to construct Simulink block diagrams differs from that of Stateflow; it is presented second.

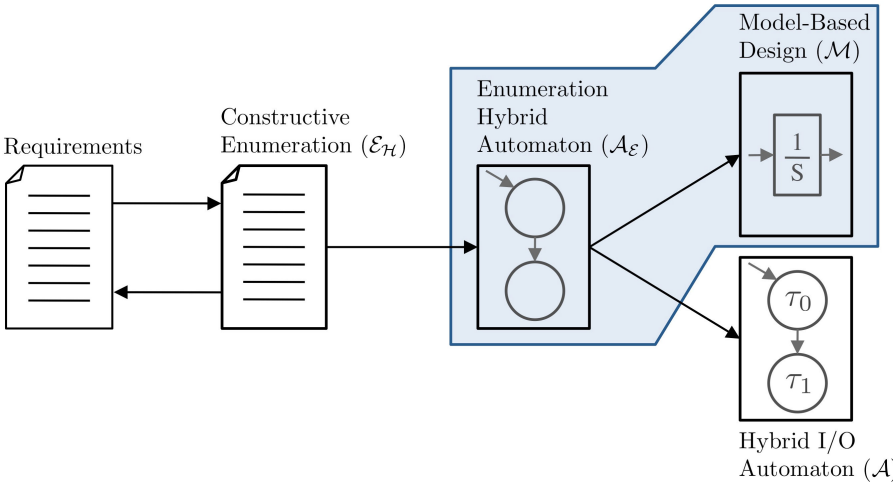


Figure 6.1: Process: Specification ($\mathcal{A}_\mathcal{E}$) to Model-Based Design ($\mathcal{M}$)

6.1 Stateflow Implementations of $\mathcal{A}_\mathcal{E}$

Stateflow charts are executable graphical representations of finite automata based on Statecharts [5]. The modeling fundamentals of Stateflow are those of state machines. In Stateflow, Mealy (see Section 2.6.1) and Moore [7, p. 42] machines can be expressed as well as machines that include continuous behavior. Stateflow charts are often used as finite state control components in a Simulink model. The technique described in this section does not take that approach. Instead, we isolate hybrid behavior in a single Stateflow chart. In other respects, the chart follows Stateflow's Mealy machine semantics. The term "state" in the Stateflow documentation refers to a mode of the $\mathcal{A}_\mathcal{E}$, which is only one component of system state.

In the implementation algorithms, the `monospaced` typeface is used for Simulink and Stateflow keywords, object names, and object properties. The *math* typeface is used for structures in $\mathcal{A}_\mathcal{E}$ and algorithm data structures.

6.1.1 Implementing the State Machine Control

The algorithm ConstructStateflow produces Stateflow chart $\mathcal{M}$ using a breadth-first search of the underlying finite automaton graph structure in $\mathcal{A}_\mathcal{E}$. The set of modes in $\mathcal{A}_\mathcal{E}$ guides the construction process. When constructing a Stateflow chart that models both discrete and continuous behavior, the chart's "update method" parameter must be set to "continuous". The derivatives that

define continuous model behavior can only be computed in mode `during` actions [26, pp. 14-27–14-29]. Variables in X have local scope and are not visible outside of $\mathcal{M}$. Variables in Y are visible outside of $\mathcal{M}$; their "scope parameter" is set to "output". The function ConstructModelMode returns a mode object with a defined `during` action. The function ConstructModelTransition returns a transition object to add to $\mathcal{M}$.

ConstructStateflow($\mathcal{A}_\mathcal{E}$)

mode_constructed is a function that identifies the modes in $\mathcal{A}_\mathcal{E}$ that have been previously visited during the search of the automaton's structure.

1.1 $\mathcal{M} \leftarrow \varnothing$

1.2 $mode_constructed \leftarrow \varnothing$

1.3 **for** each $m \in type(\mu)$

1.4 $\quad mode_constructed(m) \leftarrow false$

1.5 **end for**

The breadth-first search starts from mode 1; Mode 1 is always defined in $\mathcal{A}_\mathcal{E}$. Define the start state in $\mathcal{M}$. This may be a value selected nondeterministically from Θ.

1.6 add ConstructModelMode(1, $\mathcal{B}(1)$) to $\mathcal{M}$

1.7 add ConstructStartState(Θ) to $\mathcal{M}$

M is the set of constructed modes whose exit transitions have not been added to $\mathcal{M}$.

1.8 $M \leftarrow \{ 1 \}$

1.9 $mode_constructed(1) \leftarrow true$

1.10 **while** $M \neq \varnothing$ [1.23]

1.11 $\quad m \in M$

1.12 $\quad M \leftarrow M - \{ m \}$

Create model transitions for each mapping in $\mathcal{Q}$ whose source mode is m.

1.13 **for** each ((h_Q, a), $\boldsymbol{r}$) $\in \mathcal{Q}$ such that $\chi_\mu(m)$ [1.22]

1.14 $m' \leftarrow \pi_\mu(\boldsymbol{r})$

If the destination mode has not been visited, the transitions from that mode have not been added to $\mathcal{M}$. Construct the newly discovered mode and add it to $\mathcal{M}$. Then add m' to M so the set of transitions from that mode are added to $\mathcal{M}$.

1.15 **if** not *mode_constructed*(m')

1.16 **then**

1.17 add ConstructModelMode($m', \mathcal{B}(m')$) to $\mathcal{M}$

1.18 $M \leftarrow M \cup \{ m' \}$

1.19 *mode_constructed*(m') $\leftarrow$ *true*

1.20 **end if**

1.21 add ConstructModelTransition(m, m', $((h_Q, a)$, $\boldsymbol{r})$) to $\mathcal{M}$

1.22 **end for** [1.13]

1.23 **end while** [1.10]

1.24 **return**($\mathcal{M}$)

end ConstructStateflow

6.1.2 Implementing the Behavior Function $\mathcal{B}$

For each mode m in *type*(μ), the `during` action in $\mathcal{M}$ is defined using $\varphi = \mathcal{B}(m)$. The "update method" parameter for state variable x whose signal definition is a differential equation is set to "continuous" in $\mathcal{M}$. Stateflow adds the variable `x_dot` to $\mathcal{M}$ automatically. `x_dot` is used to define the behavior of x in a mode's `during` action; the expression in the signal definition may be used directly. The "update method" parameter for variables whose signal definitions are simple equations is set to "discrete" in $\mathcal{M}$. An Embedded MATLAB Function or Stateflow Graphical Function may be used to implement surrogate input variable signal definitions.

ConstructModelMode(m, φ)

1.1 create a new `mode` using m

1.2 **for** each $z \in Z$ [1.12]

1.3 **case** z's signal definition in φ [1.11]

1.4 **when** $\dot{z} = e$

1.5 append `z_dot=` e to the `during` action of `mode`

1.6 **when** $z(t) = e$

1.7 append `z=` e to the `during` action of `mode`

1.8 **when** z is a surrogate for u

1.9 construct surrogate function f as an embedded MATLAB or a Stateflow function.

1.10 append `z=f(u(t))` to the `during` action of `mode`

1.11 **end case** [1.3]

1.12 **end for** [1.2]

1.13 **return** (`mode`)

end ConstructModelMode

When all the `during` actions are removed and the "scope parameter" of each variable in X is changed to "external," $\mathcal{M}$ becomes a Mealy machine representative of the embedded system control.

6.1.3 Implementing the Condition Function $\mathcal{Q}$

Each mapping in $\mathcal{Q}$ associates a transition condition (h_Q, a) with a condition vector $\boldsymbol{r}$. The general syntax for a Stateflow transition consists of four parts: `event[condition]{condition_action}/transition_action`. Each part is optional. To comply with Stateflow Mealy machine semantics, we use only the `condition` and `condition_action` parts of the transition definition [26, p. 6-9]. `condition_actions` are executed when the source mode is active and the `condition` is satisfied. The expressions in $\boldsymbol{r}$ are implemented in the `condition_action`. Since the values of variables whose coordinates in $\boldsymbol{r}$ are set to $x(last)$ remain unchanged during the transition, they are not

implemented in the `condition_action`. The function ConstructModelTransition transforms mappings in $\mathcal{Q}$ into transitions in $\mathcal{M}$.

ConstructModelTransition(m, m', $((h_Q, a), \boldsymbol{r})$)

1.1 create a new `transition`
1.2 `transition.source` $\leftarrow m$
1.3 `transition.destination` $\leftarrow m'$
1.4 `transition.condition` $\leftarrow [\texttt{input} = a \;\&\&\; h_Q]$
1.5 **if** $\pi_o(\boldsymbol{r}) \neq null$
1.6 **then**
1.7 append `o:=` $\pi_o(\boldsymbol{r})$ to the `condition_action` of `transition`
1.8 **end if**
1.9 **for** each $x \in X$ [1.14]
1.10 **if** ($\pi_x(\boldsymbol{r}) \neq null \wedge \pi_x(\boldsymbol{r}) \neq x(last)$)
1.11 **then**
1.12 append `x:=` $\pi_x(\boldsymbol{r})$ to the `condition_action` of `transition`
1.13 **end if**
1.14 **end for** [1.9]
1.15 **return** (`transition`)

end ConstructModelTransition

Stateflow imposes an order on transition execution. The order resolves conflicting transitions that represent implemented nondeterminism. The order is either set explicitly by the designer or implicitly based on a set of rules [26, pp. 3-57–3-68]. By ensuring characteristic predicates are pairwise disjoint, any unintended behavior that may result from these rules is eliminated.

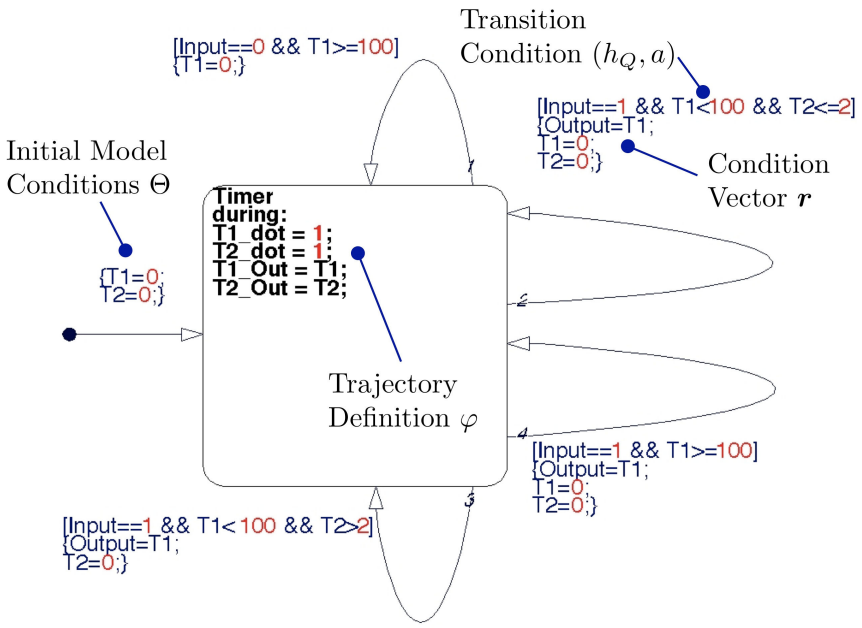


Figure 6.2: Stateflow Implementation of the Resettable Timers Example

6.1.4 Resettable Timers Implementation

Appendix A.1 contains the requirements, enumeration, and construction process of the enumeration hybrid automaton of an example with two resettable timers. Figure 6.2 is the Stateflow implementation ($\mathcal{M}$) of the resettable timers example annotated to show the relationship between specification and implementation. In Figure 6.2, the `during` keyword in the Timer mode prefixes the continuous action taken between discrete system transitions. In the figure, the trajectory definition governs continuous system behavior between discrete actions while in the Timer mode.

In Figure 6.2, the pair $(\chi_{\mu=\mathsf{Timer}} \wedge \chi_{\mathtt{T1}<100} \wedge \chi_{\mathtt{T2}\leq 2},\ \mathtt{button})$ is shown in its implemented form. The atomic characteristic predicate $\chi_{\mu=\mathsf{Timer}}$ designates the transition's source. The discrete action `button` is identified using the integer index 1. The remainder of the transition condition is defined using h_Q. The entire characteristic predicate may be used; however, $\chi_{\mu=\mathsf{Timer}}$ is made redundant by occupation of the Timer mode.

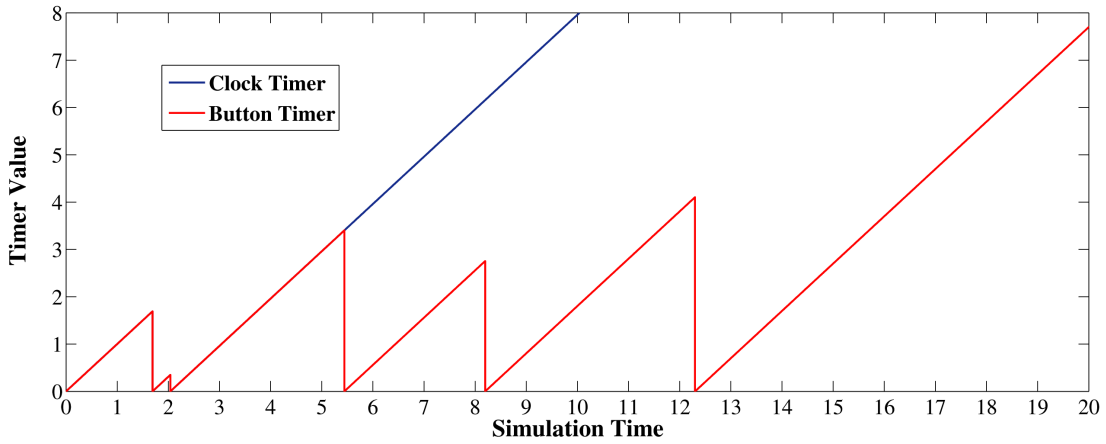


Figure 6.3: Resettable Timers

Two output variables, Clock_Timer and Button_Timer, were added to $\mathcal{M}$ to illustrate "update method" parameter effects. The signals associated with these variables for a system run are shown in Figure 6.3. At simulation time 1.691 and 2.040, the button press event resets both timers. As can be seen in the figure, both continuous-time signals are continuous despite the discrete "update method" parameter setting.

Discrete-time signals that satisfy Definition 2.2 cannot be represented in Simulink and Stateflow; each system variable will have a value at each continuous time step during a simulation of $\mathcal{M}$. During a simulation, for example, the discrete-time output signal $o[t]$ is plotted in Figure 6.4. The spikes in the plot are the instantaneous values of the Clock_Timer sent to the display; for all other simulation times the value 0 represents *null*.

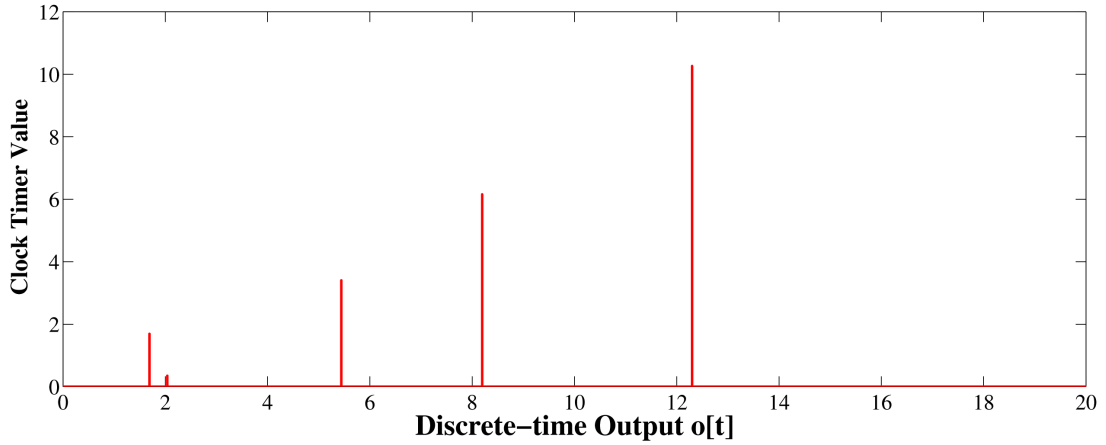


Figure 6.4: Discrete Output Signal $o[t]$

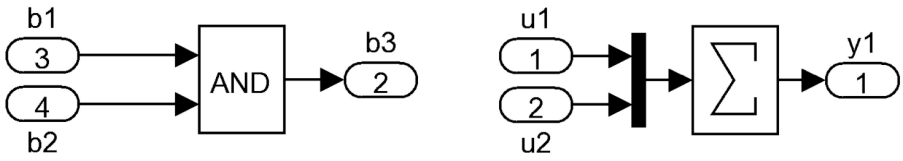


Figure 6.5: Simulink Models

6.2 Simulink Fundamentals

Enumeration hybrid automata and HIOA incorporate discrete and continuous mathematics into their respective definitions. Simulink is a graphical tool and a graphical language used to construct, simulate, and analyze model-based designs [25, p. 2]. A model-based design simulation cannot represent the continuous mathematics in these models with fidelity. Every simulation is intrinsically discrete. Solutions to differential equations are approximated using a variety of ordinary differential equation (ODE) solvers. Accuracy is determined by several factors: the type of solver, the simulation step size, round-off error, and the technique and parameter settings for zero-crossing detection (zero-crossing is discussed in Section 6.2.4). Once a model has been constructed from a specification, different results can be produced by varying the many available parameters in the Simulink software.

6.2.1 Elements

A Simulink model is a directed graph whose edges are signals and whose vertices are computational elements [25, pp. 2–4]. The signals that flow from one vertex to another can be multidimensional. Simulink contains a vast library of blocks used to transform signals. In our examples, we use a small subset of the available blocks; those that perform the logic and the mathematical operations needed to define characteristic predicates and expressions (see Definition 4.5). Figure 6.5 shows two Simulink models. The model on the left computes the conjunction of continuous-time Boolean-valued signals $b1(t)$ and $b2(t)$. The model on the right computes the sum of continuous-time signals $u1(t)$ and $u2(t)$ after they have been converted into a vector signal; $y1(t) = u1(t)+u2(t)$. Subsystem blocks are used for hierarchical model organization and contain a collection of model blocks some of which may be additional subsystem blocks. Two types of subsystem blocks are used in Simulink. A non-virtual subsystem is conditionally executed; a virtual subsystem is used for model organization.

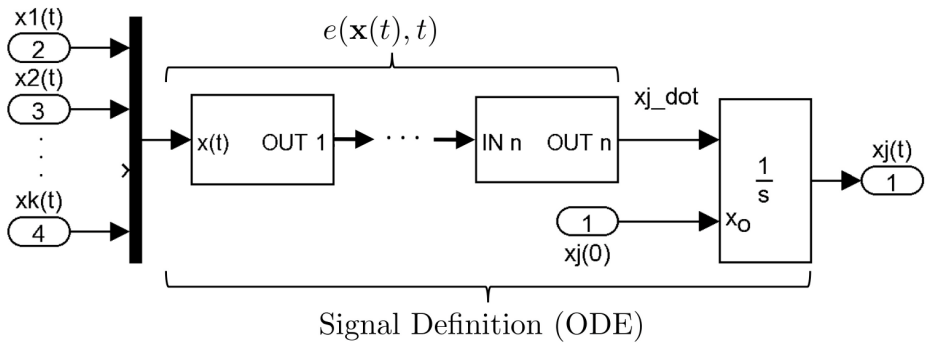


Figure 6.6: Simulink Implementation of a Signal Definition

The outline of a subsystem block delineates a system boundary as discussed in Section 2.5.

The Simulink block diagram shown in Figure 6.6 models a signal definition for x_j where $j \in [k]$ and $k = |X|$. The blocks in the path from left to right model the expression $e(x_1(t), \ldots, x_k(t), t)$. This expression paired with the initial condition xj(0) are solved using an ODE solver to produce the output xj(t). With the integrator block and the initial condition removed, an algebraic equation is modeled.

6.2.2 Simulation

A Simulink model simulation is computed by iterating over a sequence of major time steps (not necessarily fixed) and at each time step iterating over an ordered subset of the computational model blocks (not all blocks will require updating at every major time step). At each major time step, the output of each computational block is computed in order. Next, blocks having states, e.g., integrators, are computed using the designated solver. To improve accuracy, blocks requiring state updates may require additional iterative computation involving minor time steps within the major time step. For each time step (major or minor), state information is used to detect discontinuities. A discontinuity results from hybrid or switching behavior and is detected using a zero-crossing detection algorithm. The last step in the simulation cycle is to compute the next major time step [25].

6.2.3 Solvers

The accuracy of a model simulation is determined by the selected ODE solver and its parameters. Time step size may be modulated or fixed. Smaller step size improves simulation accuracy while increasing the number of necessary computations. Variable rate solvers improve efficiency; these solvers vary step size based on a variable's rate of change [25, pp. 2–5]. Larger step sizes are used for variables whose derivatives change slowly. Signal definitions for discrete state variables are transformed using discrete blocks, e.g., Z-transform blocks (unit delay). Discrete blocks have a sample time parameter. The state of every discrete block in the model must be computed for each multiple of its sample time parameter.

6.2.4 Zero-Crossing Detection

Zero-crossing detection algorithms attempt to produce simulations that precisely identify discontinuous system behavior. Blocks that employ zero-crossing algorithms use special detection variables computed using state variables and block parameters [25, pp. 2–27]. The algorithm identifies a time interval using the simulation times before and after the detection variable changes sign. The interval includes the exact time of the zero-crossing event. The algorithm refines the interval by changing its bounds to identify the approximate time and state of the discontinuity. Depending on algorithm and simulation parameters, zero-crossings can be missed; this occurs when the stimulation time step passes over a sign change. A zero-crossing algorithm parameter halts chattering or excessive time interval refinement. Two types of zero-crossing algorithms are available in Simulink.

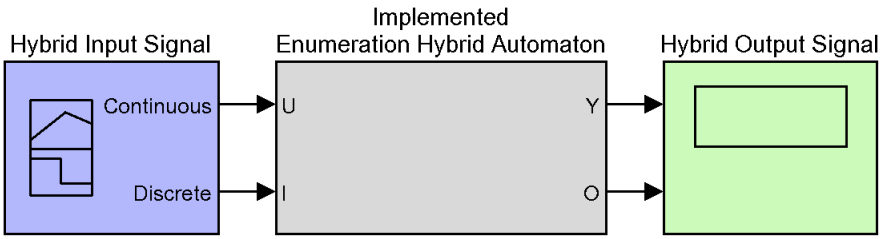


Figure 6.7: Simulink Implementation ($\mathcal{M}$) of $\mathcal{A}_{\mathcal{E}}$

6.3 Simulink Implementations of $\mathcal{A}_{\mathcal{E}}$

The Simulink model $\mathcal{M}$ contains the top-level subsystem `Main` that has the interfaces of a HIOA (see Figure 2.3). In Figure 6.7, `Main` is the gray subsystem located between the hybrid input signal generator (see Section 4.1.4) and the hybrid output signal displayed using a scope block. $\mathcal{M}$ contains four component subsystems as depicted in Figure 6.8:

1. The transition subsystem contains a subsystem for each element in $dom(\mathcal{Q})$. Each component subsystem computes the conditions in (h_Q, a) and outputs an index when those conditions are satisfied; 0 is the output when the conditions are not satisfied.

2. The state machine subsystem implements $\mathcal{Q}$ as a look-up table using a matrix. The object returned from the look-up table, $\boldsymbol{r}$, is realized using a tuple of indices; the indices in this tuple identify a condition vector implementation.

3. The condition vector subsystem implements $range(\mathcal{Q})$ or the expressions used to define the tuples in R. Given $x \in X \cup \{ o \}$, $|E_x| = |range(\pi_x \circ \mathcal{Q})|$ is the number of expressions required to compute the initial state for x following any system transition.

4. The trajectory definition subsystem implements each trajectory definition in Φ, the range of $\mathcal{B}$. Given $z \in Z$, $|range(\pi_z \circ \mathcal{B})|$ is the number of signal definitions required to define the trajectory definitions in Φ.

In ConstructSimulink, μ, *signal_constructed*, *transition_index*, *condition_index*, and *condition_constructed* are global. The procedures ConstructModelMode and ConstructModelTransition modify this set of global objects.

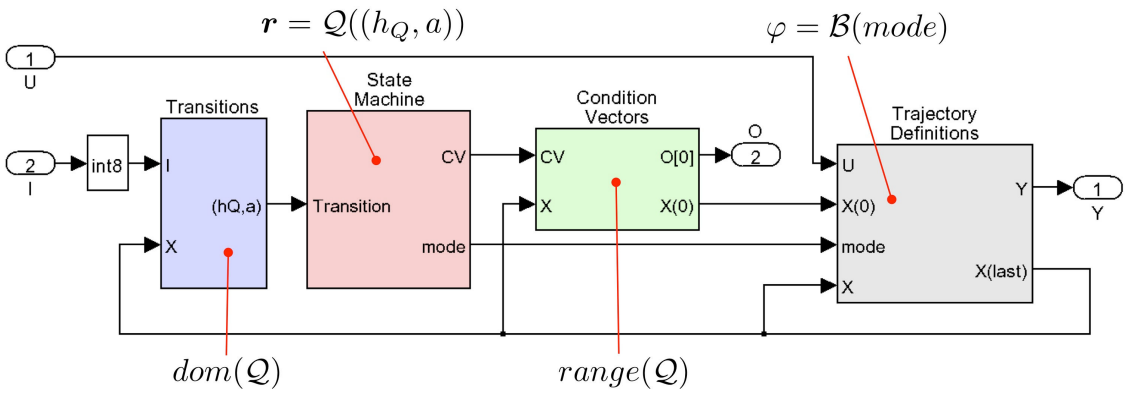


Figure 6.8: Simulink Subsystem Components of $\mathcal{M}$

ConstructSimulink($\mathcal{A}_\mathcal{E}$)

1.1 $\mathcal{M} \leftarrow \varnothing$

1.2 add subsystem `Main` to $\mathcal{M}$

1.3 add subsystem `Transitions`, `StateMachine`, `ConditionVectors`, and `TrajectoryDefinitions` to subsystem `Main`

Declare bookkeeping data structures. Computational elements are only added to the model one time.

1.4 $mode_constructed \leftarrow \varnothing$

1.5 **for** each $m \in type(\mu)$

1.6 $\quad mode_constructed(m) \leftarrow false$

1.7 **end for**

1.8 $transition_index \leftarrow 0$

1.9 $condition_constructed \leftarrow \varnothing$

1.10 $condition_index \leftarrow \varnothing$

1.11 $signal_constructed \leftarrow \varnothing$

A transition matrix is used to implement the function $\mathcal{Q}$. It has $|X| + 1$ rows and $|dom(\mathcal{Q})|$ columns.

1.12 define `transition_matrix`

1.13 add `transition_matrix` to $\mathcal{M}$

Initialize the bookkeeping function condition_constructed that takes a variable name and an expression and returns an expression index. The index is used in the transition matrix to identify the expression. A 0 index indicates the subsystem has not been constructed.

1.14 **for** $v \in X \cup \{ o \}$ [1.20]

1.15 add subsystem `vCondition` to subsystem `ConditionVectors`

1.16 $condition_index(v) \leftarrow 0$

1.17 **for** $e \in range(\pi_v \circ \mathcal{Q})$

1.18 $condition_constructed(v, e) \leftarrow 0$

1.19 **end for**

1.20 **end for** [1.14]

Initialize the bookkeeping function signal_constructed that takes a variable name and a signal definition and indicates if the subsystem implementing that signal definition has been constructed.

1.21 **for** $z \in Z$ [1.26]

1.22 add subsystem `zSignal` to subsystem `TrajectoryDefinitions`

1.23 **for** $equation \in range(\pi_z \circ \mathcal{B})$

1.24 $signal_constructed(z, equation) \leftarrow false$

1.25 **end for**

1.26 **end for** [1.21]

The breadth-first search starts from mode 1; Mode 1 is always defined in $\mathcal{A}_\mathcal{E}$.

1.27 ConstructModelMode($\mathcal{M}$, 1, $\mathcal{B}(1)$)

M is the set of modes whose exit transitions have not been added to $\mathcal{M}$.

1.28 $M \leftarrow \{ 1 \}$

1.29 $mode_constructed(1) \leftarrow true$

1.30 **while** $M \neq \varnothing$ [1.43]

1.31 $m \in M$

1.32 $M \leftarrow M - \{ m \}$

Construct transitions for each mapping in $\mathcal{Q}$ whose source mode is m.

1.33 **for** each ((h_Q, a), $\boldsymbol{r}$) $\in \mathcal{Q}$ such that $\chi_\mu(m)$ [1.42]

1.34 $m' \leftarrow \pi_\mu(\boldsymbol{r})$

If the destination mode has not been visited, the transitions from that mode have not been added to $\mathcal{M}$. Construct the newly discovered mode and add it to $\mathcal{M}$. Then add m' to M so the set of transitions from that mode are added to $\mathcal{M}$.

1.35 **if** not *mode_constructed*(m')

1.36 **then**

1.37 ConstructModelMode($\mathcal{M}$, m', $\mathcal{B}(m')$)

1.38 $M \leftarrow M \cup \{ m' \}$

1.39 *mode_constructed*(m') $\leftarrow$ *true*

1.40 **end if**

1.41 ConstructModelTransition($\mathcal{M}$, m, m', $((h_Q, a), \boldsymbol{r})$)

1.42 **end for** [1.33]

1.43 **end while** [1.30]

Define the start state in $\mathcal{M}$. Start states are discrete feedback block parameters.

1.44 ConstructStartState(Θ)

1.45 **return**($\mathcal{M}$)

end ConstructStateflow

6.3.1 Implementing the Behavior Function $\mathcal{B}$

Figure 6.9 is one subsystem in the Trajectory Definitions subsystem of Figure 6.8. The model implements a set of signal definitions. Each gray subsystem models a signal definition in the set and follows the structure in Figure 6.6. The function $\mathcal{B}$ is realized by using mode variable predicates that enable exactly one conditionally executed subsystems; a mode enabling signal definition is annotated in the figure. The implemented signal definition for T1-dot of the resettable timers

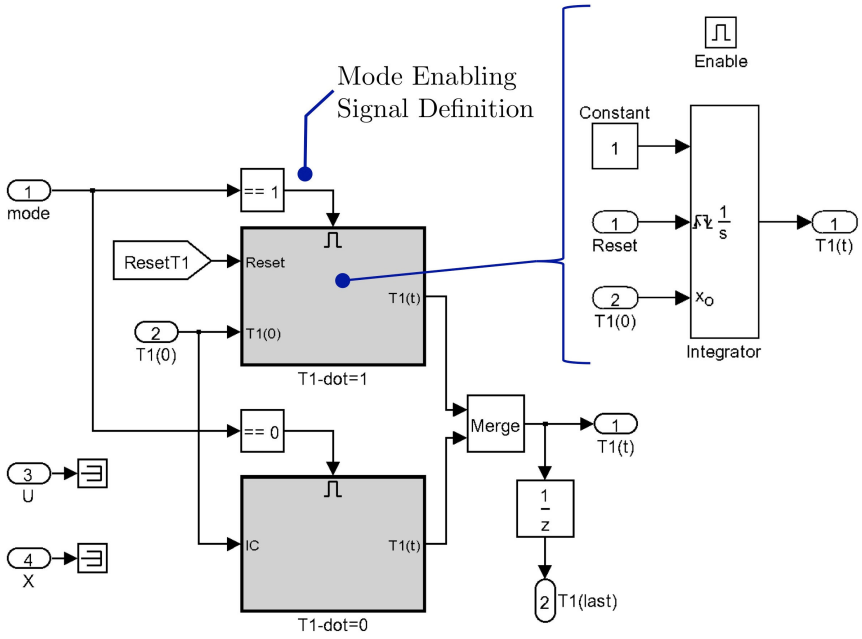


Figure 6.9: Simulink Implementation of the Signal Definition Set in $\mathcal{M}$

example in Appendix A is expanded on the right. Detailed construction of signal definitions is performed by the function ConstructSignalDefinition.

ConstructModelMode($\mathcal{M}$, m, φ)

A trajectory definition is associated with each model mode.

1.1 **for** each $z \in Z$ [1.7]

1.2 **if** not $signal_constructed(z, \pi_z(\varphi))$

1.3 **then**

1.4 add ConstructSignalDefinition$(\pi_z(\varphi), m)$ to subsystem `zSignal`

1.5 $signal_constructed(z, \pi_z(\varphi)) \leftarrow true$

1.6 **end if**

1.7 **end for** [1.1]

end ConstructModelMode

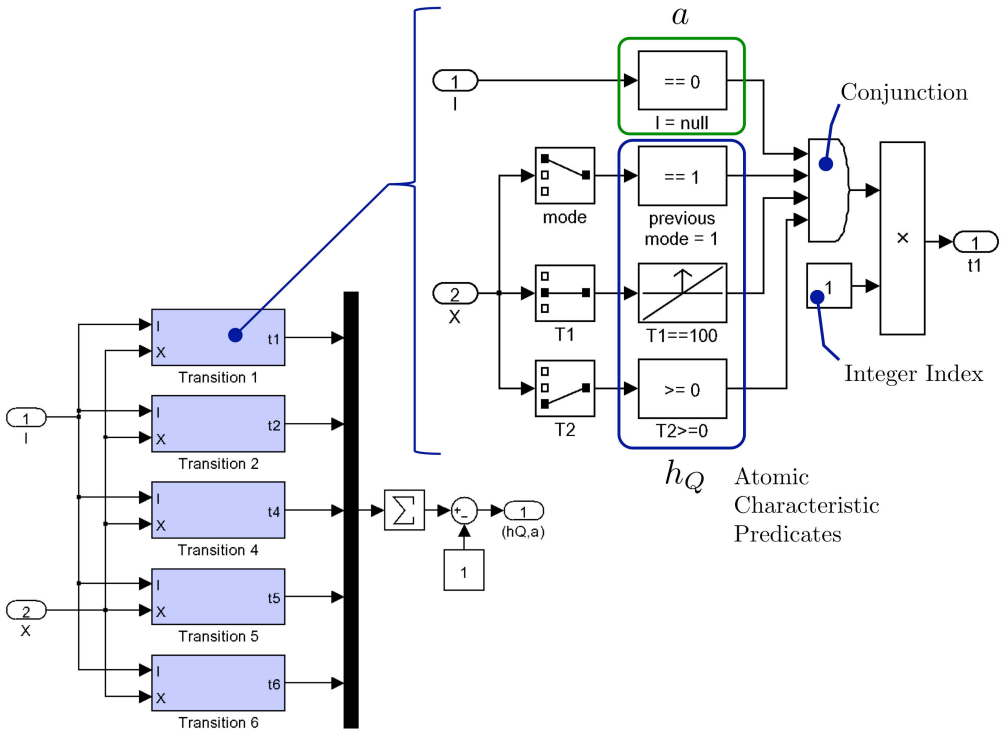


Figure 6.10: Simulink Implementation of the Transition Set in $\mathcal{M}$

6.3.2 Implementing the Condition Function $\mathcal{Q}$

Figure 6.10 shows the details of the Transitions subsystem of Figure 6.8. The model implements the set of transitions (blue subsystems) that make up $dom(\mathcal{Q})$. The expanded view of Transition 1 models (h_Q, a) and assigns it an index of 1 when the conjunction is satisfied. Each atomic characteristic predicate in h_Q is implemented using a single block. The transition condition is completed by adding the predicate $i[t] = a$ for the discrete-time signal i. The figure shows a transition model in the resettable timers example. ConstructTransition constructs the details in the expanded view.

In $\mathcal{A}_\mathcal{E}$ of the resettable timers example, there are nine transitions. Figure 6.10 shows a model with only five transitions. Mathematically, the specification is correct; $dom(\mathcal{Q})$ is formed using a partition of the system's state space and the input action set. Nine subsystem can be constructed; however, during system simulation the illegal mode will be entered due to the limitations of an intrinsically discrete model implementation. This behavior is correct. In order to produce a simulation that behaves as expected, blocks with zero-crossing detection are used and certain transitions

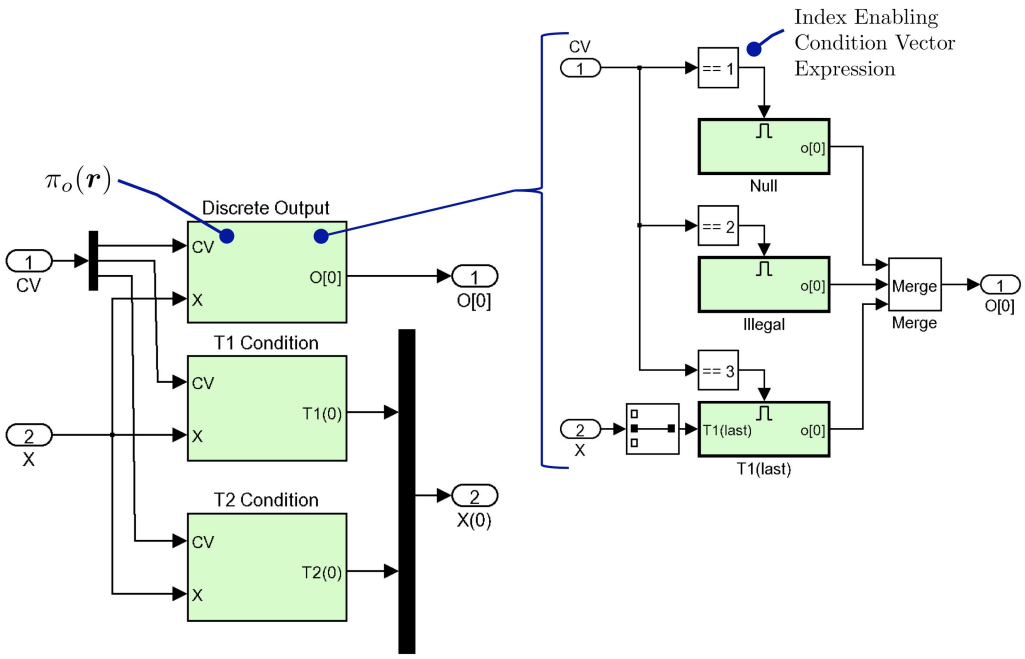


Figure 6.11: Simulink Implementation of the Condition Vector Set in $\mathcal{M}$

are eliminated from the model implementation. Blocks must be eliminated due to the round-off error associated with zero-crossing detection that causes multiple transitions to be triggered.

Figure 6.11 shows the details of the Condition Vectors subsystem of Figure 6.8. The model on the left side of the figure implements the set of condition vectors R in the resettable timers example. The subsystem Discrete Output contains models of each expression needed to compute $o[0]$. The expanded view shows the subsystem for each expression. The function ConstructCondition constructs the details in a subsystem added to

ConstructModelTransition($\mathcal{M}$, m, m', $((h_Q, a), \boldsymbol{r})$)

Each transition is implemented as a subsystem. transition_index references a column in the `transition_matrix`.

1.1 $transition_index \leftarrow transition_index + 1$

1.2 add ConstructTransition((h_Q, a), $transition_index$) to subsystem `Transitions`

j indexes the row in `transition_matrix` *that contains the condition vector expression index.*

1.3 $j \leftarrow 1$

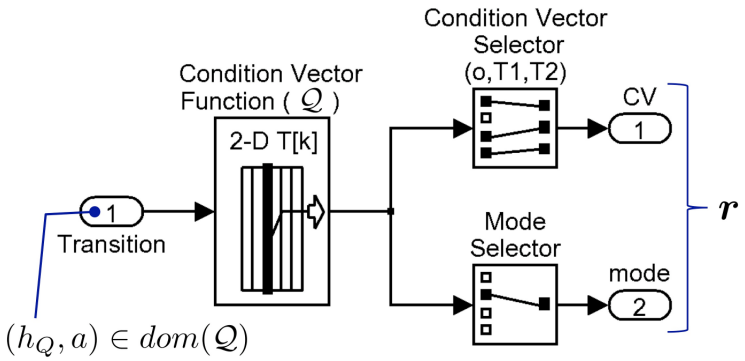


Figure 6.12: Simulink Implementation of $\mathcal{Q}$ in $\mathcal{M}$

1.4 **for** each $v \in X \cup \{ o \}$ ordered lexicographically [1.13]

When the condition vector expression $\pi_v(\boldsymbol{r})$ has not been implemented (0 value), define its index, construct the expression model, and add the newly constructed model to the appropriate subsystem.

1.5 **if** $condition_constructed(v, \pi_v(\boldsymbol{r})) = 0$ [1.10]

1.6 **then**

1.7 $condition_index(v) \leftarrow condition_index(v) + 1$

1.8 $condition_constructed(v, \pi_v(\boldsymbol{r})) \leftarrow condition_index(v)$

1.9 add ConstructCondition($\pi_v(\boldsymbol{r})$) to subsystem `vCondition`

1.10 **end if** [1.5]

Add the index of the condition vector expression to the `transition_matrix`*.*

1.11 `transition_matrix`$(j, transition_index) \leftarrow condition_constructed(v, \pi_v(\boldsymbol{r}))$

1.12 $j \leftarrow j + 1$

1.13 **end for** [1.4]

end ConstructModelTransition

The function $\mathcal{Q}$ is modeled in Figure 6.12. The condition vector, $\boldsymbol{r} = \mathcal{Q}((h_Q, a))$, is retrieved from the transition matrix used to define the Condition Vector Function block in the figure. The indices needed to select condition expressions are selected and used by the Condition Vectors subsystem. The mode is passed to the Trajectory Definitions subsystem.

6.4 Alternative Implementations

For a given $\mathcal{A}_{\mathcal{E}}$, there are many possible implementations in many design languages. In Simulink for example, the state machine subsystem can be implemented using only logic. This increases the number of model blocks, but the model will still be correct. Although not investigated, it should be possible to generate implementations in Uppaal [3] and LabVIEW, for example.

The methods outlined in this chapter highlight the relationship between the structures in $\mathcal{A}_{\mathcal{E}}$ and the final implementation. The enumeration makes automatic implementation possible and eliminates guesswork in hand-crafted implementations.

Chapter 7

Conclusion

Hybrid automata and model-based designs share common characteristics; they both are capable of modeling discrete software function coacting with physical processes. The hybrid specification process we have introduced complements the model-based design process. It provides a critical link between requirements and completed model through detailed analysis of the application to be implemented as an embedded real-time system.

This research extends existing sequence-based specification to explicitly address continuity, time, nondeterminism, and internal events. We use the hybrid I/O automaton model developed in [9,13] as a framework and prove our method produces well-defined HIOA. As the industry evolves with competing embedded system design methods, we offer a unique perspective: Our systematic and constructive approach maps sequences of time-sensitive events based on hybrid signals to models of internal dynamic behavior and hybrid output signals. Timing constraints may be modeled using continuous-time signals that evolve at the same rate as real time. By including functions that model state evolution, complete system behavior and internal actions may be defined and simulated.

We design deterministic systems since complete, consistent, and correct control software function is essential in safety and mission critical systems. An enumeration hybrid automaton is deterministic after the initial start state has been established. The enumeration process discovers an equivalence relation using unreduced sequences and refinements of the continuous state space to ensure one and only one automaton transition is possible at any instant in time. The refinement-abstraction process over a finite-dimensional (but potentially infinite) state space allows us to address simultaneous events and define system properties at various levels of abstraction.

A summary of the contributions follows:

- The definitions and process needed to construct precise specifications of hybrid systems based on requirements
- The consequential improvement in statement of requirements
- An algorithm for automated conversion from enumeration to specification
- An algorithm for automated conversion from specification to algebraic HIOA
- Algorithms for generation of Simulink and Stateflow models
- A mathematical model generated by the workflow with useful properties:
  - Composition
  - Potential for automated assertion proofs
  - Implementation independence
- Compatibility with existing Software Quality Research Laboratory (SQRL) processes and tools
  - Discrete enumeration process
  - Markov chain automated testing process

Two examples are included in Appendix A. These examples illustrate that step-by-step adherence to the hybrid enumeration process produces a precise specification. The meticulous examination of requirements and sequential system events ensures all the artifacts of the design process agree. The first example shows in detail the connection between the mathematical description of the enumeration process and the practical working process.

The second, larger example includes more hybrid system features. In this example, we go directly from the statement of requirements to the tabular enumeration. For this derivative of the Power Window Example [18], eight additional requirements were needed to complete the enumeration. In the enumeration, 46 refinement steps were needed to create the six mappings that define

the modes of the power window system and the 71 mappings that define the stimulus vector sets needed to define system behavior in those modes.

This table would be provided to programmers for product implementation. A Simulink model would be generated to serve as an oracle for automated testing, product simulation, and system validation. When compared to typical design methods, the effectiveness of the method is compelling.

There is always the question of scalability. One's first reaction to the size of the enumeration table is that the method is impractical. Realizing that every possible situation required in an implementation has been presented in a well-organized tabular form is the better reaction. It is exactly this set of situations that must be discovered and analyzed to ensure the system is correct. The process described herein clearly separates the process of requirements refinement and specification development from the programming and implementation tasks. Enumeration must be done by domain experts and product engineers because their knowledge is essential to make the decisions prompted by enumeration. The tabular form is suitable for review with customers, management, and engineers alike. The table contains all and only the details needed by programmers who would correctly implement the requirements; subsequent software development can be done with full attention to implementation matters.

By keeping system boundary size reasonable and using HIOA composition theorems, larger composite systems can be created from component hybrid specifications. This is one benefit of strict adherence to an independent mathematical model.

7.1 Tool Support

Enumeration methods are only practical with effective tool support and model composition. A sequence-based specification tool called Protoseq has been developed by SQRL that is very effective for discrete enumerations. The prototype implements the change-in-requirements features developed by Lin [11]. Research in string rewriting offers additional features to improve efficiency. This prototype is being enhanced at the Fraunhofer IESE. Also, an enumeration tool has been developed by Verum and is in commercial use. There is no doubt about the ability to bring significant tool support to bear on the process.

Although the tables in the appendix were produced using spreadsheet software, suitable tool

support for hybrid enumeration will require further enhancement of Protoseq. An examination of the examples in the appendix will immediately convey the impression that substantial tool support is possible. From the starting point forward, the tool will ensure that details derivable through the method are provided automatically. Furthermore, the tool will maintain the mathematical integrity of the underlying model by prompting the designer for each step of the workflow. The tool will detect the need for additional table blocks and automatically supply data where possible, while highlighting the incomplete details. Domain experts will then be able to refine that state space to produce a complete table row. As with the current version of Protoseq, complete documentation will be maintained as enumeration progresses and can be generated on demand. Connection to the automated testing process of JUMBL [22] will be maintained.

7.2 Future Research

There are two tracks for future research. The first and most important is field experience in creating specifications and testing frameworks for real products. The second research focus is further independent work in the mathematics and algorithmics to find more advancements that can be brought to application. This includes techniques such as zero-crossing detection and simulation of Zeno behavior. We must continue the feedback loop between the field and the laboratory, so that field experience will inform tool development and algorithms, while research advancements will inform application in the field. Some specific ideas follow.

- Algorithms for Stateflow and Simulink are included here, but additional algorithms could be devised to generate Uppaal and other models as needed.
- Connections could be made to commercial requirements and documentation tools.
- A hybrid specification defines a set of hybrid sequences. From this population of hybrid sequences, test cases can be generated. Hybrid sequences explicitly include real-time information that can be used in the testing regimen. Since a hybrid specification models both trajectories and actions, trajectory information may be used as a portion of the test oracle.
- Treat a component hybrid specification as a trajectory. Activate this component specification in one or more modes of a higher level specification, forming a specification hierarchy.

- Integrate nondeterministic behavior into specification trajectory definitions while retaining deterministic control. As models are evolved into products, comparisons between product, model, and specification can be analyzed to improve the overall design and better approximate modeled behavior of the real-world plant.
- Develop refinement and abstraction change algorithms that agree with the specification change algorithms from [11].
- Integration of the HIOA hybrid sequence restriction operation into a complete or ongoing hybrid enumeration.
- Elimination / minimization of theoretical Zeno behavior caused by loops of point trajectories.

This research was undertaken because discrete sequence-based enumeration does not directly address time, nondeterminism, and continuity. Heretofore, we invented abstract stimuli to deal with those matters based on the experience and intuition of the developers with risk of error and rework in the enumeration. The extension to hybrid enumeration does address these matters directly by facilitating more precise analysis and specification.

It is very reassuring that one can identify from the hybrid enumeration the precise abstractions that would be needed to revert to the simpler discrete enumeration. The end result is the same: a complete, consistent and traceably correct specification that supports code generation and automated testing.

References

References

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-hsin Ho. Hybrid automata an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, LNCS, pages 209–229. Springer–Verlag, 1993.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] Jason M. Carter and Jesse H. Poore. Sequence-based specification of feedback control systems in Simulink®. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 332–345, New York, NY, USA, 2007. ACM.

[5] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[6] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[7] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[8] Daniel Jackson, Martyn Thomas, Lynette Millett, and Editors. *Software for Dependable Systems: Sufficient Evidence?* The National Academies Press, Washington, DC, 2007.

[9] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The theory of timed I/O automata. Technical Report TR-2003-015, Massachusettes Institute of Technology, 2005.

[10] Lan Lin, Stacy J. Prowell, and Jesse H. Poore. An axiom system for sequence-based specification. *Theoretical Computer Science*, pages 1–17, July 2009.

[11] Lan Lin, Stacy J. Prowell, and Jesse H. Poore. The impact of requirements changes on specification and state machines. *Software — Practice and Experience*, 39(6):573–610, 2009.

[12] Jan Lunze. *Modelling, Analysis, and Design of Hybrid Systems*, volume 279/2002 of *Lecture Notes in Control and Information Sciences*, chapter What is a Hybrid System?, pages 3–14. Springer Berlin / Heidelberg, Cambridge, UK, January 2002.

[13] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata. Technical Report MIT-LCS-TR-827c, MIT Laboratory for Computer Science, 2001.

[14] Mrinal Mandal and Amir Asif. *Continuous and Discrete Time Signals and Systems*. Cambridge University Press, Cambridge, New York, first edition, 2007.

[15] S. J. Mason. Feedback theory – some properties of signal flow graphs. *Proceedings of the IRE*, 41(9):1144–1156, Sept 1953.

[16] James D. Meiss. *Differential Dynamical Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, first edition, 2007.

[17] Carl Mitcham, editor. *Encyclopedia of Science, Technology, and Ethics*, volume 4. Gale Cengage, Farmington Hills, MI, 2005.

[18] Pieter J. Mosterman. MATLAB® file exchange: Power window system. Internet, 2002.

[19] Pieter J. Mosterman. *Hybrid Dynamic Systems: Modeling and Execution (in CRC Handbook of Dynamic System Modeling)*, chapter 15, page (in publication). Chapmand and Hall/CRC Press, Boca Raton, FL, 2007.

[20] Tammy Noergaard. *Embedded Systems Architecture - A Comprehensive Guide for Engineers and Programmers*. Elsevier, Oxford, UK, 2005.

[21] Stacy J. Prowell. *Sequence-Based Software Specification*. PhD thesis, The University of Tennessee, Knoxville, Tennessee, December 1996.

[22] Stacy J. Prowell. JUMBL: a tool for model-based statistical testing. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 337.3, Washington, DC, USA, 2003. IEEE Computer Society.

[23] Stacy J. Prowell and Jesse H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29(5):417–429, May 2003.

[24] Viggo Stoltenberg-Hansen, Ingid Lindström, and Edward R. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, Cambridge, UK, first edition, 1994.

[25] The MathWorks™. *Smulink® 7 User's Guide*. The MathWorks, Inc., Natick, MA, 2008. This is an electronic document.

[26] The MathWorks™. *Stateflow® and Stateflow Coder® 7 User's Guide*. The MathWorks, Inc., Natick, MA, 2008. This is an electronic document.

[27] LaMar K. Timothy and Blair E. Bona. *State Space Analysis: An Introduction*. McGraw-Hill Book Company, New York, NY, first edition, 1968.

[28] William R. Wade. *An Introduction To Analysis*. Pearson Prentise Hall, Upper Saddle River, NJ, third edition, 2004.

Appendices

Appendix A

Examples

A.1 Resettable Timers

The resettable timers example uses two timers and a single input button. One timer's value can be displayed by depressing a button and resets automatically to 0 when it reaches 100. Another timer tracks the amount of time between button presses. If the button timer is 2 seconds or less, then both timers reset.

In this example, refinement is required to define consistent system behavior when specific combinations of timer values and button state occur. Enumeration provides the mechanism to identify these details and tie them to requirements. This example demonstrates how a sequential approach discovers individual coordinates of condition vectors and signal definitions at different levels of abstraction. Furthermore, a discrete-time signal is used to send a clock's current value to a display. This signal's range is not a finite set; however, the signal produces output at discrete instants in time. A detailed outline of the process is given as well as the tabular format. The tabular format is more concise.

A.1.1 Requirements

1. The resettable timers system consists of a control logic unit, two timers, an interface to a single external button, and an interface to a single output display.

2. Initially, both timers are set to 0 and the button is not activated.

(a) (D) Initially, Timer 2 is activated and incrementing.

3. Timer 1 tracks time from 0 to 100 units at a rate of 1 unit/sec. It always resets to 0 when it reaches 100 units.

4. Timer 2 tracks the time between consecutive external button activations and advances at a rate of 1 unit/sec.

   (a) If the button is never pushed, Timer 2 will increase without bound.

5. External button activation produces the following effects:

   (a) The value of Timer 1 will be sent to the display.

   (b) If the value Timer 2 is less than or equal to 2 units, Timer 1 will reset to 0 units.

   (c) Timer 2 will reset to 0 units.

The (D) following Requirement 2(a) above indicates that this requirement was not included in the original statement of requirements but was derived or discovered as an augmentation to the requirements through the enumeration process.

A.1.2 Enumeration Procedure

The resettable timers requirements are converted into enumeration mappings using the hybrid enumeration process in Section 4.2.2. Headers indicate the procedure phase of either ConstructHybridEnumeration or DefineMapping. The steps of the process are indicated in the left column. When decisions must be made, the requirements needed to proceed are listed in the right column: R(1) means Item 1 of Section A.1.1, for example. When "Method" is used in the justification column, some or all of the process step could be automated with tool support.

Resettable Timers Enumeration Steps

Step	Process	Justification
1	**Declaration**	
1.1	$I \leftarrow \{\, null, \texttt{button} \,\}$	R(1), Method
1.2	$U \leftarrow \varnothing$	R(1)
1.3	$Y \leftarrow \varnothing$	R(1)
1.4	$X \leftarrow \{\, \texttt{T1}, \texttt{T2} \,\}$	R(1, 3, 4)
1.5	$Z \triangleq \{\, \texttt{T1}, \texttt{T2} \,\}$	Method
1.8	$C_{\texttt{T1}} \leftarrow \mathbb{R}^{\geq 0}$	R(3)[1]
1.8	$C_{\texttt{T2}} \leftarrow \mathbb{R}^{\geq 0}$	R(4)
1.10	$\mathcal{U} \leftarrow \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$	Method
2	**Initialization**	
2.1	$\boldsymbol{r} \leftarrow (null, null, null)$	Method
2.2	$\varphi \leftarrow (null, null)$	Method
2.3	$\boldsymbol{m}_0 \leftarrow (\lambda,\ ((\varphi, \boldsymbol{r}),\ \lambda))$	Method
2.5	$A_{\texttt{T1}} \leftarrow [0, 0]$	R(2)
2.5	$A_{\texttt{T2}} \leftarrow [0, 0]$	R(2)
2.7	$\Theta \leftarrow \{\, (0, 0) \,\}$	Method
2.8	$z \in \{\, \texttt{T1}, \texttt{T2} \,\}$	
	$z \leftarrow \texttt{T1}$	Method
2.11	$\varphi(\texttt{T1}) \leftarrow \dot{\texttt{T1}} = 1$	R(3)
2.12	$\boldsymbol{r}(\texttt{T1}) \leftarrow \texttt{T1}(last)$	Method
2.8	$z \leftarrow \texttt{T2}$	Method

continued on the next page

[1] We make the assumption timer values cannot be negative; $\mathbb{R}$ may be used and the process will construct a correct specification with a larger set of illegal sequences.

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
2.11	$\varphi(\texttt{T2}) \leftarrow \dot{\texttt{T2}}=1$	R(2a,4)
2.12	$\boldsymbol{r}(\texttt{T2}) \leftarrow \texttt{T2}(last)$	Method
2.21	$\mathcal{E} \leftarrow \{\, \boldsymbol{m}_0 \,\}$	Method
2.22	$\boldsymbol{E} \leftarrow \{\, \lambda \,\}$	Method
3	**Sequence Enumeration:** Extending λ	
3.1	$l \leftarrow 0$	Method
3.2	Length 0 sequences in $\boldsymbol{E}$: $\{\, \lambda \,\}$	Method
3.4	$\boldsymbol{\sigma} \in \{\, \lambda \,\}$	
	$\boldsymbol{\sigma} \leftarrow \lambda$	Method
3.5	$a \in \{\, null, \texttt{button} \,\}$	
	$a \leftarrow null$	Method
3.6	$\boldsymbol{a}_1 \leftarrow (null, (\chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	Method
3.7	$\boldsymbol{m}_1 \leftarrow (\boldsymbol{a}_1,\ (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (null, \texttt{T1}(last), \texttt{T2}(last))),\ \lambda))$	Method
3.8	$\mathcal{E} \leftarrow \{\, \boldsymbol{m}_0 \,\} \cup$ DefineMapping $(\boldsymbol{m}_1, Z, \{o\}, X)$	Method

DefineMapping: $\boldsymbol{m}_1 = ((null, (\chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (null, \texttt{T1}(last), \texttt{T2}(last))), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \texttt{T1}, \texttt{T2} \,\}$	
	$z \leftarrow \texttt{T1}$	Method
1.19	T1 signal definition not definable.	R(3)[2]
1.1	$z \leftarrow \texttt{T2}$	Method
1.5	T2 signal definition definable.	R(4, 4a)
1.6	$D \leftarrow \{\, \texttt{T1} \,\}$	Method
1.8	$\texttt{T2} \in X$ and $\dot{\texttt{T2}}=1$	R(4)
1.9	$C \leftarrow \{\, o, \texttt{T2} \,\}$	Method
1.10	$\varphi(\texttt{T2}) \leftarrow \dot{\texttt{T2}}=1$	R(4)
1.11	$\boldsymbol{r}(\texttt{T2}) \leftarrow \texttt{T2}(last)$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, o, \texttt{T2} \,\}$	
	$v \leftarrow o$	Method

continued on the next page

[2] T1 only acts over [0,100].

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
2.5	Discrete output $o[0]$ is definable.	R(5,5a)[3]
2.6	$C = \{\, \mathtt{T2} \,\}$	Method
2.11	Keep default definition: $o[0] = \mathit{null}$	R(5,5a)
2.1	$v \leftarrow \mathtt{T2}$	Method
2.5	T2 initial condition definable.	R(4a, 5c)[4]
2.6	$C \leftarrow \varnothing$	Method
2.9	$\boldsymbol{r}(\mathtt{T2}) \leftarrow \mathtt{T2}(\mathit{last})$	R(4, 4a, 5c)[5]
3	**Reduction**	
3.1	$C \cup D = \{\, \mathtt{T1} \,\}$	Refinement Required
4	**Refinement**	
4.2	$x \in \{\, \mathtt{T1}, \mathtt{T2} \,\}$	
	$x \leftarrow \mathtt{T1}$	Method
4.3	$\chi_{\mathtt{T1}} \in \{\, \chi_{\mathtt{T1}\in\mathbb{R}^{\geq 0}} \,\}$	Method
4.4	Refine $\chi_{\mathtt{T1}\in\mathbb{R}^{\geq 0}}$: $\chi'_{\mathtt{T1}=100}$	R(3)
4.6	$\boldsymbol{a}_1 \leftarrow (\mathit{null}, (\chi'_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$	Method[6]
4.7	$\boldsymbol{m}_1 \leftarrow (\boldsymbol{a}_1,\ (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), \mathtt{T2}(\mathit{last}))),\ \lambda))$	Method[7]
4.8	$\boldsymbol{a}_2 \leftarrow (\mathit{null}, (\chi_{\mathtt{T1}\neq 100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$	Method[8]
4.9	$\boldsymbol{m}_2 \leftarrow (\boldsymbol{a}_2,\ (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), \mathtt{T2}(\mathit{last}))),\ \lambda))$	Method
4.10	return DefineMapping$(\boldsymbol{m}_1, \{\mathtt{T1}\}, \varnothing, \{\mathtt{T2}\}) \cup$	
	DefineMapping$(\boldsymbol{m}_2, \{\mathtt{T1}\}, \varnothing, \{\mathtt{T1}, \mathtt{T2}\})$	Method

DefineMapping: $\boldsymbol{m}_1 = ((\mathit{null}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), \mathtt{T2}(\mathit{last}))), \lambda))$		
1	**Trajectory Definition**	
1.1	$z \in \{\, \mathtt{T1} \,\}$	
	$z \leftarrow \mathtt{T1}$	Method
1.5	T1 signal definition definable.	R(3)
1.6	$D \leftarrow \varnothing$	Method

continued on the next page

[3]The value of T1 is only displayed when the button is pushed.
[4]Timer T2 is only reset with the button.
[5]Requirements support default assignment.
[6]Redefining $\boldsymbol{a}_1$ using the previous instance.
[7]Redefining $\boldsymbol{m}_1$ using the previous instance.
[8]New action. $\overline{\chi'_{\mathtt{T1}=100}} = \chi_{\mathtt{T1}\neq 100}$.

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
1.8	$\mathtt{T1} \in X$ and $\dot{\mathtt{T1}} = 1$	R(3)
1.9	$C \leftarrow \{\, \mathtt{T1} \,\}$	Method
1.10	$\varphi(\mathtt{T1}) \leftarrow \dot{\mathtt{T1}} = 1$	R(3)
1.11	$\boldsymbol{r}(\mathtt{T1}) \leftarrow \mathtt{T1}(last)$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, \mathtt{T1} \,\}$	
	$v \leftarrow \mathtt{T1}$	Method
2.5	T1 initial condition definable.	R(3)
2.6	$C \leftarrow \varnothing$	Method
2.9	$\boldsymbol{r}(\mathtt{T1}) \leftarrow 0$	R(3)
3	**Reduction**	
3.1	$C \cup D = \varnothing$	Method
3.3	$\boldsymbol{\psi} \in \{\, \lambda \,\}$	
	$\boldsymbol{\psi} \leftarrow \lambda$	Method
3.4	$(null, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$ and λ are equivalent	R(3,4,5)
3.6	$(null, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \rhd \lambda$	Method
3.7	$\boldsymbol{m}_1 \leftarrow (\boldsymbol{a}_1, (((\dot{\mathtt{T1}} = 1, \dot{\mathtt{T2}} = 1), (null, 0, \mathtt{T2}(last))), \lambda))$	Method
3.7	return $\{\, \boldsymbol{m}_1 \,\}$	Method
4	**Continue Refinement:** $(null, (\chi_{\mathtt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{\, \boldsymbol{m}_1 \,\} \cup$	
	DefineMapping$(\boldsymbol{m}_2, \{\mathtt{T1}\}, \varnothing, \{\mathtt{T1}, \mathtt{T2}\})$	Method
DefineMapping: $\boldsymbol{m}_2 = ((null, (\chi_{\mathtt{T1}\neq 100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\mathtt{T1}} = 1, \dot{\mathtt{T2}} = 1), (null, \mathtt{T1}(last), \mathtt{T2}(last))), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \mathtt{T1} \,\}$	
	$z \leftarrow \mathtt{T1}$	Method
1.19	T1 signal definition not definable.	R(3)
2	**Condition Vectors**	
2.1	$v \in \varnothing$	Method
3	**Reduction**	
3.1	$C \cup D = \{\, \mathtt{T1} \,\}$	Refinement Required

continued on the next page

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
4	**Refinement**	
4.2	$x \in \{\, \mathtt{T1}, \mathtt{T2} \,\}$	
	$x \leftarrow \mathtt{T1}$	Method
4.3	$\chi_{\mathtt{T1}} \in \{\, \chi_{\mathtt{T1}\neq 100} \,\}$	Method
4.4	Refine $\chi_{\mathtt{T1}\neq 100}$: $\chi'_{\mathtt{T1}\in[0,100)}$	R(3)
4.6	$\boldsymbol{a}_2 \leftarrow (\mathit{null}, (\chi'_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$	Method
4.7	$\boldsymbol{m}_2 \leftarrow (\boldsymbol{a}_2, (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), \mathtt{T2}(\mathit{last}))), \lambda))$	Method
4.8	$\boldsymbol{a}_3 \leftarrow (\mathit{null}, (\chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$	Method
4.9	$\boldsymbol{m}_3 \leftarrow (\boldsymbol{a}_3, (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), \mathtt{T2}(\mathit{last}))), \lambda))$	Method
4.10	return DefineMapping$(\boldsymbol{m}_2, \{\mathtt{T1}\}, \varnothing, \{\mathtt{T2}\}) \cup$	
	DefineMapping$(\boldsymbol{m}_3, \{\mathtt{T1}\}, \varnothing, \{\mathtt{T1}, \mathtt{T2}\})$	Method
DefineMapping: $\boldsymbol{m}_2 = ((\mathit{null}, (\chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), \mathtt{T2}(\mathit{last}))), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \mathtt{T1} \,\}$	
	$z \leftarrow \mathtt{T1}$	Method
1.5	T1 signal definition definable.	R(3)
1.6	$D \leftarrow \varnothing$	Method
1.8	$\mathtt{T1} \in X$ and $\dot{\mathtt{T1}}=1$	R(3)
1.9	$C \leftarrow \{\, \mathtt{T1} \,\}$	Method
1.10	$\varphi(\mathtt{T1}) \leftarrow \dot{\mathtt{T1}}=1$	R(3)
1.11	$\boldsymbol{r}(\mathtt{T1}) \leftarrow \mathtt{T1}(\mathit{last})$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, \mathtt{T1} \,\}$	
	$v \leftarrow \mathtt{T1}$	Method
2.5	T1 initial condition is definable.	R(3)
2.6	$C \leftarrow \varnothing$	Method
2.9	$\boldsymbol{r}(\mathtt{T1}) \leftarrow \mathtt{T1}(\mathit{last})$	R(3, 5b)
3	**Reduction**	
3.1	$C \cup D = \varnothing$	Method
3.3	$\boldsymbol{\psi} \in \{\, \lambda \,\}$	
	$\boldsymbol{\psi} \leftarrow \lambda$	Method

continued on the next page

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
3.4	$(null, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$ and λ are equivalent.	R(3,4,5)
3.6	$(null, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})) \rhd \lambda$	Method
3.7	$\boldsymbol{m}_2 \leftarrow (\boldsymbol{a}_2, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (null, \texttt{T1}(last), \texttt{T2}(last))), \lambda))$	Method
3.7	return $\{\, \boldsymbol{m}_2 \,\}$	Method
4	**Continued Refinement:** $(null, (\chi_{\texttt{T1}\neq 100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{\, \boldsymbol{m}_2 \,\} \cup$	
	DefineMapping$(\boldsymbol{m}_3, \{\texttt{T1}\}, \varnothing, \{\texttt{T1}, \texttt{T2}\})$	Method
DefineMapping: $\boldsymbol{m}_3 = ((null, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (null, \texttt{T1}(last), \texttt{T2}(last))), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \texttt{T1} \,\}$	
	$z \leftarrow \texttt{T1}$	Method
1.19	T1 signal definition is illegal.	R(3)[9]
1.4	return IllegalMapping($\boldsymbol{m}_3$)	R(3)
IllegalMapping: $\boldsymbol{m}_3 = ((null, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (null, \texttt{T1}(last), \texttt{T2}(last))), \lambda))$		
1.1	$\boldsymbol{a}_3 \mapsto ((\dot{\texttt{T1}}=0, \dot{\texttt{T2}}=0), (\omega, \texttt{T1}(last), \texttt{T2}(last)))$	Method
1.2	$\boldsymbol{a}_3 \rhd \boldsymbol{a}_3$	Method
1.3	return $(\boldsymbol{m}_3)$	Method
4	**Continued Refinement:** $(null, (\chi_{\texttt{T1}\neq 100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{\, \boldsymbol{m}_2, \boldsymbol{m}_3 \,\}$	Method
4	**Continued Refinement:** $(null, (\chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{\, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3 \,\}$	Method
3	**Sequence Enumeration:** Extending λ	
3.8	$\mathcal{E} \leftarrow \{\, \boldsymbol{m}_0, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3 \,\}$	Method
3.5	$a \leftarrow \texttt{button}$	Method
3.6	$\boldsymbol{a}_4 \leftarrow (\texttt{button}, (\chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	Method
3.7	$\boldsymbol{m}_4 \leftarrow (\boldsymbol{a}_4,\ (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (null, \texttt{T1}(last), \texttt{T2}(last))),\ \lambda))$	Method
3.8	$\mathcal{E} \leftarrow \{\, \boldsymbol{m}_0, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3 \,\} \cup$	
	DefineMapping($\boldsymbol{m}_4, Z, \{o\}, X$)	Method

continued on the next page

[9] T1's signal definition does not operate in the range $(100, \infty)$.

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
DefineMapping: $\boldsymbol{m}_4 = ((\texttt{button}, (\chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), \texttt{T2}(\mathit{last}))), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \texttt{T1}, \texttt{T2} \,\}$	
	$z \leftarrow \texttt{T1}$	Method
1.19	T1 signal definition not definable.	R(3)
1.1	$z \leftarrow \texttt{T2}$	Method
1.5	T2 signal definition definable.	R(4)
1.6	$D \leftarrow \{\, \texttt{T1} \,\}$	Method
1.8	$\texttt{T2} \in X$ and $\dot{\texttt{T2}}=1$	R(4)
1.9	$C \leftarrow \{\, o, \texttt{T2} \,\}$	Method
1.10	$\varphi(\texttt{T2}) \leftarrow \dot{\texttt{T2}}=1$	R(4)
1.11	$\boldsymbol{r}(\texttt{T2}) \leftarrow \texttt{T2}(\mathit{last})$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, o, \texttt{T2} \,\}$	
	$v \leftarrow o$	Method
2.13	Discrete output $o[0]$ not definable.	R(3,5a)[10]
2.1	$v \leftarrow \texttt{T2}$	Method
2.5	T2 initial condition is definable	R(4, 4a)
2.6	$C \leftarrow \{\, o \,\}$	Method
2.9	$\boldsymbol{r}(\texttt{T2}) \leftarrow 0$	R(4, 5c)
3	**Reduction**	
3.1	$C \cup D = \{\, o, \texttt{T1} \,\}$	Refinement Required
4	**Refinement**	
4.2	$x \in \{\, \texttt{T1}, \texttt{T2} \,\}$	
	$x \leftarrow \texttt{T1}$	Method
4.3	$\chi_{\texttt{T1}} \in \{\, \chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}} \,\}$	Method
4.4	Refine $\chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}}$: $\chi'_{\texttt{T1}=100}$	R(3)
4.6	$\boldsymbol{a}_4 \leftarrow (\texttt{button}, (\chi'_{\texttt{T1}=100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	Method
4.7	$\boldsymbol{m}_4 \leftarrow (\boldsymbol{a}_4, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), 0)), \lambda))$	Method
4.8	$\boldsymbol{a}_5 \leftarrow (\texttt{button}, (\chi_{\texttt{T1}\neq 100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	Method

continued on the next page

[10] Would be $\texttt{T1}(\mathit{last})$ but $\texttt{T1}(\mathit{last}) > 100$ could occur with current refinement.

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
4.9	$\boldsymbol{m}_5 \leftarrow (\boldsymbol{a}_5, (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), 0)), \lambda))$	Method
4.10	return DefineMapping$(\boldsymbol{m}_4, \{\mathtt{T1}\}, \{o\}, \{\mathtt{T2}\}) \cup$	
	DefineMapping$(\boldsymbol{m}_5, \{\mathtt{T1}\}, \{o\}, \{\mathtt{T1}, \mathtt{T2}\})$	Method
DefineMapping: $\boldsymbol{m}_4 = ((\mathtt{button}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathit{null}, \mathtt{T1}(\mathit{last}), 0)), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \mathtt{T1} \,\}$	
	$z \leftarrow \mathtt{T1}$	Method
1.5	T1 signal definition definable.	R(3)
1.6	$D \leftarrow \varnothing$	Method
1.8	$\mathtt{T1} \in X$ and $\dot{\mathtt{T1}}=1$	R(3)
1.9	$C \leftarrow \{\, o, \mathtt{T1} \,\}$	Method
1.10	$\varphi(\mathtt{T1}) \leftarrow \dot{\mathtt{T1}}=1$	R(3)
1.11	$\boldsymbol{r}(\mathtt{T1}) \leftarrow \mathtt{T1}(\mathit{last})$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, o, \mathtt{T1} \,\}$	
	$v \leftarrow o$	Method
2.6	$C \leftarrow \{\, \mathtt{T1} \,\}$	Method
2.9	$\boldsymbol{r}(o) \leftarrow \mathtt{T1}(\mathit{last})$	R(5,5a)
2.1	$v \leftarrow \mathtt{T1}$	Method
2.6	$C \leftarrow \varnothing$	Method
2.9	$\boldsymbol{r}(\mathtt{T1}) \leftarrow 0$	R(3)
3	**Reduction**	
3.1	$C \cup D = \varnothing$	No Refinement
3.3	$\boldsymbol{\psi} \in \{\, \lambda \,\}$	
	$\boldsymbol{\psi} \leftarrow \lambda$	Method
3.4	$(\mathtt{button}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$ and λ are equivalent	R(3,4,5)
3.6	$(\mathtt{button}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \rhd \lambda$	Method
3.7	$\boldsymbol{m}_4 \leftarrow (\boldsymbol{a}_4, (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (\mathtt{T1}(\mathit{last}), 0, 0)), \lambda))$	Method
3.7	return $\{\, \boldsymbol{m}_4 \,\}$	Method
4	**Continued Refinement** $(\mathtt{button}, (\chi_{\mathtt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$	

continued on the next page

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
4.10	return $\{\, \boldsymbol{m}_4 \,\} \cup$	
	DefineMapping($\boldsymbol{m}_5, \{\, \texttt{T1} \,\}, \{o\}, \{\, \texttt{T1}, \texttt{T2} \,\}$)	Method

DefineMapping: $\boldsymbol{m}_5 = ((\texttt{button}, (\chi_{\texttt{T1}\neq 100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), 0)), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \texttt{T1} \,\}$	
	$z \leftarrow \texttt{T1}$	Method
1.19	T1 signal definition not definable.	R(3)
2	**Condition Vectors**	
2.1	$v \in \{\, o \,\}$	
	$v \leftarrow o$	Method
2.13	Discrete output $o[0]$ not definable.	R(3,5a)
3	**Reduction**	
3.1	$C \cup D = \{\, o, \texttt{T1} \,\}$	Refinement Required
4	**Refinement**	
4.1	$x \in \{\, \texttt{T1}, \texttt{T2} \,\}$	
	$x \leftarrow \texttt{T1}$	Method
4.3	$\chi_{\texttt{T1}} \in \{\, \chi_{\texttt{T1}\neq 100} \,\}$	Method
4.4	Refine $\chi_{\texttt{T1}\neq 100}$: $\chi'_{\texttt{T1}\in[0,100)}$	R(3)
4.6	$\boldsymbol{a}_5 \leftarrow (\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	Method
4.7	$\boldsymbol{m}_5 \leftarrow (\boldsymbol{a}_5, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), 0)), \lambda))$	Method
4.8	$\boldsymbol{a}_6 \leftarrow (\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	Method
4.9	$\boldsymbol{m}_6 \leftarrow (\boldsymbol{a}_6, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), 0)), \lambda))$	Method
4.10	return DefineMapping($\boldsymbol{m}_5, \{\texttt{T1}\}, \{o\}, \{\texttt{T2}\}$) $\cup$	
	DefineMapping($\boldsymbol{m}_6, \{\texttt{T1}\}, \{o\}, \{\texttt{T1}, \texttt{T2}\}$)	Method

DefineMapping: $\boldsymbol{m}_5 = ((\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), 0)), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{\, \texttt{T1} \,\}$	
	$z \leftarrow \texttt{T1}$	Method
1.5	T1 signal definition definable.	R(3)
1.6	$D \leftarrow \varnothing$	Method

continued on the next page

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
1.8	$\mathtt{T1} \in X$ and $\dot{\mathtt{T1}} = 1$	R(3)
1.9	$C \leftarrow \{\, \mathtt{T1} \,\}$	Method
1.10	$\varphi(\mathtt{T1}) \leftarrow \dot{\mathtt{T1}} = 1$	R(3)
1.11	$\boldsymbol{r}(\mathtt{T1}) \leftarrow \mathtt{T1}(last)$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, o, \mathtt{T1} \,\}$	
	$v \leftarrow o$	Method
2.6	$C \leftarrow \{\, \mathtt{T1} \,\}$	Method
2.9	$\boldsymbol{r}(o) \leftarrow \mathtt{T1}(last)$	R(5a)
2.1	$v \leftarrow \mathtt{T1}$	Method
2.13	T1 initial conditions not definable.	R(5,5b)
3	**Reduction**	
3.1	$C \cup D = \{\, \mathtt{T1} \,\}$	Refinement Required
4	**Refinement**	
4.1	$x \in \{\, \mathtt{T2} \,\}$	
	$x \leftarrow \mathtt{T2}$	Method
4.3	$\chi_{\mathtt{T2}} \in \{\, \chi_{\mathtt{T2} \in \mathbb{R}^{\geq 0}} \,\}$	Method
4.4	Refine $\chi_{\mathtt{T2} \in \mathbb{R}^{\geq 0}}$: $\chi'_{\mathtt{T2} \leq 2}$	R(5,5b)
4.6	$\boldsymbol{a}_5 \leftarrow (\mathtt{button}, (\chi_{\mathtt{T1} \in [0,100)} \wedge \chi'_{\mathtt{T2} \leq 2}))$	Method
4.7	$\boldsymbol{m}_5 \leftarrow (\boldsymbol{a}_5, (((\dot{\mathtt{T1}} = 1, \dot{\mathtt{T2}} = 1), (\mathtt{T1}(last), \mathtt{T1}(last), 0)), \lambda))$	Method
4.8	$\boldsymbol{a}_7 \leftarrow (\mathtt{button}, (\chi_{\mathtt{T1} \in [0,100)} \wedge \chi_{\mathtt{T2} > 2}))$	Method
4.9	$\boldsymbol{m}_7 \leftarrow (\boldsymbol{a}_7, (((\dot{\mathtt{T1}} = 1, \dot{\mathtt{T2}} = 1), (\mathtt{T1}(last), \mathtt{T1}(last), 0)), \lambda))$	Method
4.10	return DefineMapping$(\boldsymbol{m}_5, \varnothing, \{\mathtt{T1}\}, \varnothing) \cup$	
	DefineMapping$(\boldsymbol{m}_7, \varnothing, \{\mathtt{T1}\}, \{\mathtt{T2}\})$	Method
DefineMapping: $\boldsymbol{m}_5 = ((\mathtt{button}, (\chi_{\mathtt{T1} \in [0,100)} \wedge \chi_{\mathtt{T2} \leq 2})), (((\dot{\mathtt{T1}} = 1, \dot{\mathtt{T2}} = 1), (\mathtt{T1}(last), \mathtt{T1}(last), 0)), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \varnothing$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, \mathtt{T1} \,\}$	
	$v \leftarrow \mathtt{T1}$	Method
2.6	$C \leftarrow \varnothing$	Method

continued on the next page

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
2.9	$\boldsymbol{r}(\texttt{T1}) \leftarrow 0$	R(5,5b)
3	**Reduction**	
3.1	$C \cup D = \varnothing$	No Refinement
3.3	$\boldsymbol{\psi} \in \{\, \lambda \,\}$	
	$\boldsymbol{\psi} \leftarrow \lambda$	Method
3.4	$(\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\leq 2}))$ and λ are equivalent	R(3,4,5)
3.6	$(\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\leq 2})) \rhd \lambda$	Method
3.7	$\boldsymbol{m}_5 \leftarrow (\boldsymbol{a}_5, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\texttt{T1}(\mathit{last}), 0, 0)), \lambda))$	Method
3.7	return $\{\, \boldsymbol{m}_5 \,\}$	Method
4	**Continued Refinement** $(\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{\, \boldsymbol{m}_5 \,\} \cup$ DefineMapping$(\boldsymbol{m}_7, \varnothing, \{\texttt{T1}\}, \{\texttt{T2}\})$	Method
DefineMapping: $\boldsymbol{m}_7 = ((\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}>2})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\texttt{T1}(\mathit{last}), \texttt{T1}(\mathit{last}), 0)), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \varnothing$	Method
2	**Condition Vectors**	
2.1	$v \in \{\, \texttt{T1} \,\}$	
	$v \leftarrow \texttt{T1}$	Method
2.6	$C \leftarrow \varnothing$	Method
2.9	$\boldsymbol{r}(\texttt{T1}) \leftarrow \texttt{T1}(\mathit{last})$	R(5,5b,4a)
3	**Reduction**	
3.1	$C \cup D = \varnothing$	No Refinement
3.3	$\boldsymbol{\psi} \in \{\, \lambda \,\}$	
	$\boldsymbol{\psi} \leftarrow \lambda$	Method
3.3	$(\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}>2}))$ and λ are equivalent	R(3,4,5)
3.6	$(\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}>2})) \rhd \lambda$	Method
3.7	$\boldsymbol{m}_7 \leftarrow (\boldsymbol{a}_7, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\texttt{T1}(\mathit{last}), \texttt{T1}(\mathit{last}), 0)), \lambda))$	Method
3.7	return $\{\, \boldsymbol{m}_7 \,\}$	Method
4	**Continued Refinement** $(\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{\, \boldsymbol{m}_5, \boldsymbol{m}_7 \,\}$	Method
4	**Continued Refinement** $(\texttt{button}, (\chi_{\texttt{T1}\neq 100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	

continued on the next page

Resettable Timers Enumeration Steps (*continued*)

Step	Process	Justification
4.10	return $\{ \boldsymbol{m}_5, \boldsymbol{m}_7 \} \cup$	
	DefineMapping($\boldsymbol{m}_6, \{\texttt{T1}\}, \{o\}, \{\texttt{T1}, \texttt{T2}\}$)	Method
DefineMapping: $\boldsymbol{m}_6 = ((\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), 0)), \lambda))$		
1	**Trajectory Definitions**	
1.1	$z \in \{ \texttt{T1}, \texttt{T2} \}$	
	$z \leftarrow \texttt{T1}$	Method
1.19	T1 signal definition is illegal.	R(3)[11]
1.4	return IllegalMapping($\boldsymbol{m}_6$)	R(3)
IllegalMapping: $\boldsymbol{m}_6 = ((\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})), (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\mathit{null}, \texttt{T1}(\mathit{last}), 0)), \lambda))$		
1.1	$\boldsymbol{a}_6 \mapsto ((\dot{\texttt{T1}}=0, \dot{\texttt{T2}}=0), (\omega, \texttt{T1}(\mathit{last}), \texttt{T2}(\mathit{last})))$	Method
1.2	$\boldsymbol{a}_6 \rhd \boldsymbol{a}_6$	Method
1.3	return $(\boldsymbol{m}_6)$	Method
4	**Continued Refinement** $(\texttt{button}, (\chi_{\texttt{T1}\neq 100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{ \boldsymbol{m}_5, \boldsymbol{m}_7, \boldsymbol{m}_6 \}$	Method
4	**Continued Refinement** $(\texttt{button}, (\chi_{\texttt{T1}\in\mathbb{R}^{\geq 0}} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$	
4.10	return $\{ \boldsymbol{m}_4, \boldsymbol{m}_5, \boldsymbol{m}_7, \boldsymbol{m}_6 \}$	Method
3	**Sequence Enumeration:** Extending λ	
3.8	$\mathcal{E} \leftarrow \{ \boldsymbol{m}_0, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3, \boldsymbol{m}_4, \boldsymbol{m}_5, \boldsymbol{m}_6, \boldsymbol{m}_7 \}$	Method
3.11	$l \leftarrow 1$	Method
3.4	No extensible length 1 stimulus vector sequences	Method
3.13	return $(\mathcal{E}, \{ \texttt{T1}, \texttt{T2} \}, \{ (0,0) \})$	Method

[11] T1's signal definition does not operate in the range $(100, \infty)$.

A.1.3 Tabular Enumeration

During practical application of the process in Section A.1.2, information derived from the process is recorded in tabular form. The table that follows was developed from the detailed process outlined in Section A.1.2. The table has been annotated with arrows to indicate the workflow within each table row: columns are filled in left to right, and in each column as many details as possible are added from top to bottom. Each detail in the table is traced to one or more requirements; requirement tags are added to the "Reqs" column. In the four step columns of the tabular form, a ConstructHybridEnumeration process step is preceded with C, a DefineMapping process step is preceded with D, and an IllegalMapping process step is preceded with I. The table tracks every detail of the process in Section A.1.2.

Canonical Sequences (Mode) | Illegal Sequences | Reducible Sequences

Invariant Sequence | Refinement Step | Red Text: To be resolved enumeration elements.

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
lambda										O	null	method	C 2.1	lambda	method	C2.3
		T1	= 0	2	C 2.5	T1-dot(t)	1	3	C 2.11	T1(0)	T1(last)	method	C 2.12			
		T2	= 0	2	C 2.5	T2-dot(t)	1	2a, 4	C 2.11	T2(0)	T2(last)	method	C 2.12			

The workflow for each row proceeds according to the arrows indicated above. The above row for the empty sequence establishes the set of initial system states and the initial system trajectory.

lambda	null									O	null	5, 5a	D 2.11	lambda	method	C3.7
		T1	[0,inf)	method	C 3.6	T1-dot(t)	1	3	D 1.19	T1(0)	T1(last)	method	C 3.7			
		T2	[0,inf)	method	C 3.6	T2-dot(t)	1	4, 4a	D 1.10	T2(0)	T2(last)	4, 4a, 5c	D 2.9			

The definition that is shown in red is determined from the prefix sequence (in this case lamda); it is the default definition. White colored rows represent intermediate work that must be completed using refinement.

lambda	null									O	null	5, 5a	D 4.7	lambda	3, 4, 5	D3.4
		T1	=100	3	D 4.4	T1-dot(t)	1	3	D 1.10	T1(0)	0	3	D 2.9			
		T2	[0,inf)	method	D 4.6	T2-dot(t)	1	4, 4a	D 4.7	T2(0)	T2(last)	4, 4a, 5c	D 4.7			

Only incomplete definitions (outlined in blue) must be added in a mapping of a refined stimulus vector sequence. When the mapping is complete, colors are used to differentiate sequence types.

lambda	null									O	null	5, 5a	D 4.9	lambda	method	D4.9
		T1	≠100	method	D 4.8	T1-dot(t)	1	3	D 1.19	T1(0)	T1(last)	method	D 2.1			
		T2	[0,inf)	method	D 4.9	T2-dot(t)	1	4, 4a	D 4.9	T2(0)	T2(last)	4, 4a, 5c	D 4.9			

lambda	null									O	null	5, 5a	D 4.7	lambda	3, 4, 5	D3.4
		T1	[0,100)	3	D 4.4	T1-dot(t)	1	3	D 1.10	T1(0)	T1(last)	3, 5b	D 2.9			
		T2	[0,inf)	method	D 4.6	T2-dot(t)	1	4	D 4.7	T2(0)	T2(last)	4, 4a, 5c	D 4.7			

lambda	null									O	omega	method	I 1.1	Illegal	3	I1.2
		T1	(100,inf)	method	D 4.8	T1-dot(t)	0	3	D 1.19	T1(0)	T1(last)	method	I 1.1			
		T2	[0,inf)	method	D 4.8	T2-dot(t)	0	method	I 1.1	T2(0)	T2(last)	method	I 1.1			

lambda	button									O	null	3, 5a	D 2.13	lambda	method	C3.7
		T1	[0,inf)	method	C 3.6	T1-dot(t)	1	3	D 1.19	T1(0)	T1(last)	method	C 3.7			
		T2	[0,inf)	method	C 3.6	T2-dot(t)	1	4	D 1.10	T2(0)	0	4, 5c	D 2.9			

lambda	button									O	T1(last)	5, 5a	D 2.9	lambda	3, 4, 5	D 3.4
		T1	=100	3	D4.4	T1-dot(t)	1	3	D 1.10	T1(0)	0	3	D 2.9			
		T2	[0,inf)	method	D4.6	T2-dot(t)	1	4	D 4.7	T2(0)	0	4, 5c	D 4.7			

lambda	button									O	null	3, 5a	D 2.13	lambda	method	D4.9
		T1	≠100	method	D 4.8	T1-dot(t)	1	3	D 1.19	T1(0)	T1(last)	method	D 1.19			
		T2	[0,inf)	method	D 4.8	T2-dot(t)	1	4	D 4.9	T2(0)	0	4, 5c	D 4.9			

lambda	button									O	T1(last)	5a	D 2.9	lambda	3, 4, 5.	D4.7
		T1	[0,100)	3	D 4.4	T1-dot(t)	1	3	D 1.10	T1(0)	T1(last)	5, 5b	D 2.13			
		T2	[0,inf)	method	D 4.6	T2-dot(t)	1	4	D 4.7	T2(0)	0	5, 5c	D 4.7			

lambda	button									O	T1(last)	5a	D 4.7	lambda	3, 4, 5	D 3.4
		T1	[0,100)	method	D 4.4	T1-dot(t)	1	3	D 4.7	T1(0)	0	5, 5b	D 2.9			
		T2	≤2	5, 5b.	D 4.6	T2-dot(t)	1	4	D 4.7	T2(0)	0	5, 5c	D 4.7			

lambda	button									O	T1(last)	5a	D 4.9	lambda	3, 4, 5	D 3.4
		T1	[0,100)	method	D 4.8	T1-dot(t)	1	3	D 4.9	T1(0)	T1(last)	5, 5b	D 2.9			
		T2	>2	method	D 4.8	T2-dot(t)	1	4	D 4.9	T2(0)	0	5, 5c	D 4.9			

lambda	button									O	omega	method	I 1.1	Illegal	3	I 1.2
		T1	(100,inf)	method	D 4.8	T1-dot(t)	0	method	D 1.19	T1(0)	T1(last)	method	I 1.1			
		T2	[0,inf)	method	D 4.8	T2-dot(t)	0	method	I 1.1	T2(0)	T2(last)	method	I 1.1			

In Section 4.2.3, the refinement process was described. Figure A.1 illustrates a refinement tree (an application of the general structure shown in Figure 4.5) generated from the tabular enumeration of the resettable timers; tree nodes are labeled with the sequences mapped in the partial function $\mathcal{E}$. The colors used in the tree have the same meaning as the colors in the enumeration table. A black node without color corresponds to the white blocks. The dotted lines in the figure represent extension of the empty sequence, λ. The right refinement subtree has the root sequence $(null, \chi_{\texttt{T1}\in[0,\infty)} \wedge \chi_{\texttt{T2}\in[0,\infty)})$, the first white block. Three mappings are defined during subsequent refinement: two legal mappings and one illegal mapping. The left refinement subtree has the root sequence $(\texttt{button}, \chi_{\texttt{T1}\in[0,\infty)} \wedge \chi_{\texttt{T2}\in[0,\infty)})$. Four mappings are defined during subsequent refinement: three legal mappings and one illegal mapping.

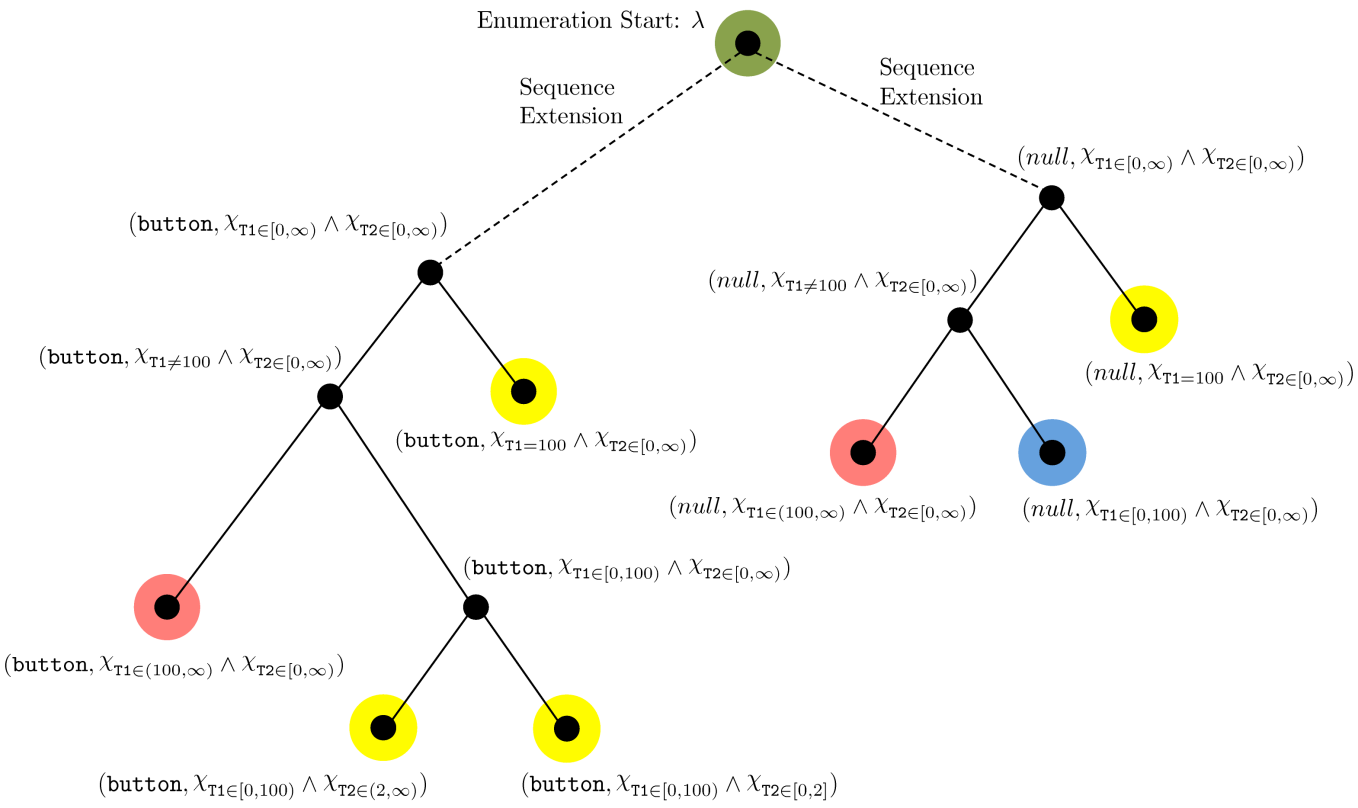


Figure A.1: Resettable Timers Refinement Tree

A.1.4 Constructing $\mathcal{A}_{\mathcal{E}}$

The hybrid enumeration of the resettable timers example is converted into an enumeration hybrid automaton using the algorithm in Section 5.1.1. The complete automaton maps a finite set of predicate - action pairs to response vectors. Both the predicates and the response vectors are augmented with a mode as determined by the unreduced stimulus vector sequences. Trajectory definitions are unique to each mode. Indices are used to show the addition of elements to the trajectory definition set, condition vector set, and the predicates used in the function $\mathcal{Q}$.

Resettable Timers Automaton Construction Steps

Step	Process
1	**Initialization**
1.2	$X \leftarrow \{ \mu, \texttt{T1}, \texttt{T2} \}$
1.3	$C_\mu \leftarrow \mathbb{N}$
1.4	$\mathcal{U} \leftarrow \mathbb{N} \times \mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0}$
1.5	$f \leftarrow \varnothing$
1.7	$type(\mu) \leftarrow \varnothing$
1.7	$type(\texttt{T1}) \leftarrow \varnothing$
1.7	$type(\texttt{T2}) \leftarrow \varnothing$
1.9	$W \leftarrow \varnothing$
1.12	$\Phi \leftarrow \varnothing$
1.13	$\mathcal{B} \leftarrow \varnothing$
1.14	$R \leftarrow \varnothing$
1.15	$\mathcal{Q} \leftarrow \varnothing$
1.16	$Q \leftarrow \varnothing$
1.17	$i \leftarrow 0$
2	**Construction**
2.1	$\boldsymbol{m} \in \{ \boldsymbol{m}_0, \boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3, \boldsymbol{m}_4, \boldsymbol{m}_5, \boldsymbol{m}_6, \boldsymbol{m}_7 \}$
2.1	$\boldsymbol{m}_0 \leftarrow (\lambda, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (null, \texttt{T1}(last), \texttt{T2}(last))), \lambda))$
2.2	$\lambda \notin dom(f)$
2.4	λ is legal
2.8	$i \leftarrow 1$
2.9	$f(\lambda) \leftarrow 1$

continued on the next page

Resettable Timers Automaton Construction Steps (*continued*)

Step	Process
2.11	$\varphi_0 \leftarrow (\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1)$
2.12	$\varphi_0 \leftarrow (\dot{\mu}=0, \dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1)$
2.13	$\Phi \leftarrow \{\, \varphi_0 \,\}$
2.14	$\mathcal{B} \leftarrow \{\, (1, \varphi_0) \,\}$
2.16	$\boldsymbol{\sigma} = \lambda$
2.18	$h_\Theta \triangleq \chi_{\mathtt{T1}=0} \wedge \chi_{\mathtt{T2}=0}$
2.19	$\chi_\mu \triangleq \mu = 1$
2.20	$h_{Q_0} \triangleq \chi_{\mu=1} \wedge \chi_{\mathtt{T1}=0} \wedge \chi_{\mathtt{T2}=0}$
2.21	$\Theta \leftarrow \{\, (1, 0, 0) \,\}$
2.31	λ is legal
2.34	$type(\mu) \leftarrow [1, 1]$
2.34	$type(\mathtt{T1}) \leftarrow [0, 0]$
2.34	$type(\mathtt{T2}) \leftarrow [0, 0]$
2.36	$Q \leftarrow \{\, (1, 0, 0) \,\}$
2.1	$\boldsymbol{\sigma} \leftarrow (null, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$
	$\boldsymbol{m}_1 \leftarrow (\boldsymbol{\sigma}, (((\dot{\mathtt{T1}}=1, \dot{\mathtt{T2}}=1), (null, 0, \mathtt{T2}(last))), \lambda))$
2.2	$\lambda \in dom(f)$
2.16	$(null, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \neq \lambda$
2.23	$\chi_\mu \triangleq \mu = 1$[12]
2.24	$h_{Q_1} \triangleq \chi_{\mu=1} \wedge \chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}$
2.25	$a \leftarrow null$
2.26	$\boldsymbol{r}_1 \leftarrow (null, 0, \mathtt{T2}(last))$
2.27	$\boldsymbol{r}_1 \leftarrow (null, 1, 0, \mathtt{T2}(last))$
2.28	$R \leftarrow \{\, \boldsymbol{r}_1 \,\}$
2.29	$\mathcal{Q} \leftarrow \{\, ((h_{Q_1}, null), \boldsymbol{r}_1) \,\}$
2.31	$(null, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$ is legal
2.34	$type(\mu) \leftarrow [1, 1]$
2.34	$type(\mathtt{T1}) \leftarrow \{\, 0, 100 \,\}$
2.34	$type(\mathtt{T2}) \leftarrow [0, \infty)$
2.36	$Q \leftarrow [1, 1] \times \{\, 0, 100 \,\} \times [0, \infty)$

continued on the next page

[12] The prefix sequence is λ.

Resettable Timers Automaton Construction Steps (*continued*)

Step	Process
2.1	$\boldsymbol{\sigma} \leftarrow (null, (\chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$
	$\boldsymbol{m}_2 \leftarrow (\boldsymbol{\sigma}, (((\dot{\mathtt{T}}1=1, \dot{\mathtt{T}}2=1), (null, \mathtt{T1}(last), \mathtt{T2}(last))), \lambda))$
2.2	$\lambda \in dom(f)$
2.16	$(null, (\chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \neq \lambda$
2.23	$\chi_\mu \triangleq \mu = 1$
2.24	$h_{Q_2} \triangleq \chi_{\mu=1} \wedge \chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}$
2.25	$a \leftarrow null$
2.26	$\boldsymbol{r}_2 \leftarrow (null, \mathtt{T1}(last), \mathtt{T2}(last))$
2.27	$\boldsymbol{r}_2 \leftarrow (null, 1, \mathtt{T1}(last), \mathtt{T2}(last))$
2.28	$R \leftarrow \{ \boldsymbol{r}_1, \boldsymbol{r}_2 \}$
2.29	$\mathcal{Q} \leftarrow \{ ((h_{Q_1}, null), \boldsymbol{r}_1), ((h_{Q_2}, null), \boldsymbol{r}_2) \}$
2.31	$(null, (\chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$ is legal
2.34	$type(\mu) \leftarrow [1, 1]$
2.34	$type(\mathtt{T1}) \leftarrow [0, 100]$
2.34	$type(\mathtt{T2}) \leftarrow [0, \infty)$
2.36	$Q \leftarrow [1, 1] \times [0, 100] \times [0, \infty)$
2.1	$\boldsymbol{\sigma} \leftarrow (null, (\chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$
	$\boldsymbol{m}_3 \leftarrow (\boldsymbol{\sigma}, (((\dot{\mathtt{T}}1=0, \dot{\mathtt{T}}2=0), (\omega, \mathtt{T1}(last), \mathtt{T2}(last))), \boldsymbol{\sigma}))$
2.2	$(null, (\chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \notin dom(f)$
2.4	$(null, (\chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$ is illegal
2.9	$f((null, (\chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))) \leftarrow 0$
2.11	$\varphi_1 \leftarrow (\dot{\mathtt{T}}1=0, \dot{\mathtt{T}}2=0)$
2.12	$\varphi_1 \leftarrow (\dot{\mu}=0, \dot{\mathtt{T}}1=0, \dot{\mathtt{T}}2=0)$
2.13	$\Phi \leftarrow \{ \varphi_0, \varphi_1 \}$
2.14	$\mathcal{B} \leftarrow \{ (1, \varphi_0), (0, \varphi_1) \}$
2.16	$(null, (\chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \neq \lambda$
2.23	$\chi_\mu \triangleq \mu = 1$
2.24	$h_{Q_3} \triangleq \chi_{\mu=1} \wedge \chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}$
2.25	$a \leftarrow null$
2.26	$\boldsymbol{r}_3 \leftarrow (\omega, \mathtt{T1}(last), \mathtt{T2}(last))$
2.27	$\boldsymbol{r}_3 \leftarrow (\omega, 0, \mathtt{T1}(last), \mathtt{T2}(last))$

continued on the next page

Resettable Timers Automaton Construction Steps (*continued*)

Step	Process
2.28	$R \leftarrow \{\, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3 \,\}$
2.29	$\mathcal{Q} \leftarrow \{\, ((h_{Q_1}, \mathit{null}), \boldsymbol{r}_1), ((h_{Q_2}, \mathit{null}), \boldsymbol{r}_2), ((h_{Q_3}, \mathit{null}), \boldsymbol{r}_3) \,\}$
2.31	$(\mathit{null}, (\chi_{\mathtt{T1}>100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$ is illegal
2.1	$\boldsymbol{\sigma} \leftarrow (\mathtt{button}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$
	$\boldsymbol{m}_4 \leftarrow (\boldsymbol{\sigma}, (((\dot{\mathtt{T}}1=1, \dot{\mathtt{T}}2=1), (\mathtt{T1}(\mathit{last}), 0, 0)), \lambda))$
2.2	$\lambda \in \mathit{dom}(f)$
2.16	$(\mathtt{button}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \neq \lambda$
2.23	$\chi_\mu \triangleq \mu = 1$
2.24	$h_{Q_4} \triangleq \chi_{\mu=1} \wedge \chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}$
2.25	$a \leftarrow \mathtt{button}$
2.26	$\boldsymbol{r}_4 \leftarrow (\mathtt{T1}(\mathit{last}), 0, 0)$
2.27	$\boldsymbol{r}_4 \leftarrow (\mathtt{T1}(\mathit{last}), 1, 0, 0)$
2.28	$R \leftarrow \{\, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4 \,\}$
2.29	$\mathcal{Q} \leftarrow \{\, ((h_{Q_1}, \mathit{null}), \boldsymbol{r}_1), ((h_{Q_2}, \mathit{null}), \boldsymbol{r}_2), ((h_{Q_3}, \mathit{null}), \boldsymbol{r}_3), ((h_{Q_4}, \mathtt{button}), \boldsymbol{r}_4) \,\}$
2.31	$(\mathtt{button}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}}))$ is legal
2.34	$\mathit{type}(\mu) \leftarrow [1, 1]$
2.34	$\mathit{type}(\mathtt{T1}) \leftarrow [0, 100]$
2.34	$\mathit{type}(\mathtt{T2}) \leftarrow [0, \infty)$
2.36	$Q \leftarrow [1, 1] \times [0, 100] \times [0, \infty)$
2.1	$\boldsymbol{\sigma} \leftarrow (\mathtt{button}, (\chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\leq 2}))$
	$\boldsymbol{m}_5 \leftarrow (\boldsymbol{\sigma}, (((\dot{\mathtt{T}}1=1, \dot{\mathtt{T}}2=1), (\mathtt{T1}(\mathit{last}), 0, 0)), \lambda))$
2.2	$\lambda \in \mathit{dom}(f)$
2.16	$(\mathtt{button}, (\chi_{\mathtt{T1}=100} \wedge \chi_{\mathtt{T2}\in\mathbb{R}^{\geq 0}})) \neq \lambda$
2.23	$\chi_\mu \triangleq \mu = 1$
2.24	$h_{Q_5} \triangleq \chi_{\mu=1} \wedge \chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}\leq 2}$
2.25	$a \leftarrow \mathtt{button}$
2.26	$\boldsymbol{r}_5 \leftarrow (\mathtt{T1}(\mathit{last}), 0, 0)$
2.27	$\boldsymbol{r}_5 \leftarrow (\mathtt{T1}(\mathit{last}), 1, 0, 0)$
2.28	$R \leftarrow \{\, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4 \,\}$ [13]
2.29	$\mathcal{Q} \leftarrow \{\, ((h_{Q_1}, \mathit{null}), \boldsymbol{r}_1), ((h_{Q_2}, \mathit{null}), \boldsymbol{r}_2), ((h_{Q_3}, \mathit{null}), \boldsymbol{r}_3), ((h_{Q_4}, \mathtt{button}), \boldsymbol{r}_4),$

continued on the next page

[13] $\boldsymbol{r}_5$ is already an element of R.

Resettable Timers Automaton Construction Steps (*continued*)

Step	Process
	$((h_{Q_5}, \texttt{button}), \boldsymbol{r}_5)\,\}$
2.31	$(\texttt{button}, (\chi_{\texttt{T1}=100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$ is legal
2.34	$type(\mu) \leftarrow [1,1]$
2.34	$type(\texttt{T1}) \leftarrow [0,100]$
2.34	$type(\texttt{T2}) \leftarrow [0,\infty)$
2.36	$Q \leftarrow [1,1] \times [0,100] \times [0,\infty)$
2.1	$\boldsymbol{\sigma} \leftarrow (\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$
	$\boldsymbol{m}_6 \leftarrow (\boldsymbol{\sigma}, (((\dot{\texttt{T1}}=0, \dot{\texttt{T2}}=0), (\omega, \texttt{T1}(last), \texttt{T2}(last))), \boldsymbol{\sigma}))$
2.2	$(\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})) \notin dom(f)$
2.4	$(\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$ is illegal
2.9	$f((\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))) \leftarrow 0$
2.11	$\varphi_2 \leftarrow (\dot{\texttt{T1}}=0, \dot{\texttt{T2}}=0)$
2.12	$\varphi_2 \leftarrow (\dot{\mu}=0, \dot{\texttt{T1}}=0, \dot{\texttt{T2}}=0)$
2.13	$\Phi \leftarrow \{\, \varphi_0, \varphi_1 \,\}$[14]
2.14	$\mathcal{B} \leftarrow \{\, (1,\varphi_0), (0,\varphi_1) \,\}$[15]
2.16	$(\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}})) \neq \lambda$
2.23	$\chi_\mu \triangleq \mu = 1$
2.24	$h_{Q_6} \triangleq \chi_{\mu=1} \wedge \chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}$
2.25	$a \leftarrow \texttt{button}$
2.26	$\boldsymbol{r}_6 \leftarrow (\omega, \texttt{T1}(last), \texttt{T2}(last))$
2.27	$\boldsymbol{r}_6 \leftarrow (\omega, 0, \texttt{T1}(last), \texttt{T2}(last))$
2.28	$R \leftarrow \{\, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4 \,\}$ [16]
2.29	$\mathcal{Q} \leftarrow \{\, ((h_{Q_1}, null), \boldsymbol{r}_1), ((h_{Q_2}, null), \boldsymbol{r}_2), ((h_{Q_3}, null), \boldsymbol{r}_3), ((h_{Q_4}, \texttt{button}), \boldsymbol{r}_4),$
	$((h_{Q_5}, \texttt{button}), \boldsymbol{r}_5), ((h_{Q_6}, \texttt{button}), \boldsymbol{r}_6)\,\}$
2.31	$(\texttt{button}, (\chi_{\texttt{T1}>100} \wedge \chi_{\texttt{T2}\in\mathbb{R}^{\geq 0}}))$ is illegal
2.1	$\boldsymbol{\sigma} \leftarrow (\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}>2}))$
	$\boldsymbol{m}_7 \leftarrow (\boldsymbol{\sigma}, (((\dot{\texttt{T1}}=1, \dot{\texttt{T2}}=1), (\texttt{T1}(last), \texttt{T1}(last), 0)), \lambda))$
2.2	$\lambda \in dom(f)$
2.16	$(\texttt{button}, (\chi_{\texttt{T1}\in[0,100)} \wedge \chi_{\texttt{T2}>2})) \neq \lambda$

continued on the next page

[14] φ_2 is already an element of Φ.
[15] $(0,\varphi_2) = (0,\varphi_1)$.
[16] $\boldsymbol{r}_6$ is already an element of R.

Resettable Timers Automaton Construction Steps (*continued*)

Step	Process
2.23	$\chi_\mu \triangleq \mu = 1$
2.24	$h_{Q_7} \triangleq \chi_{\mu=1} \wedge \chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}>2}$
2.25	$a \leftarrow \mathtt{button}$
2.26	$\boldsymbol{r}_7 \leftarrow (\mathtt{T1}(last), \mathtt{T1}(last), 0)$
2.27	$\boldsymbol{r}_7 \leftarrow (\mathtt{T1}(last), 1, \mathtt{T1}(last), 0)$
2.28	$R \leftarrow \{\, \boldsymbol{r}_0, \boldsymbol{r}_1, \boldsymbol{r}_3, \boldsymbol{r}_4, \boldsymbol{r}_7 \,\}$
2.29	$\mathcal{Q} \leftarrow \{\, ((h_{Q_1}, null), \boldsymbol{r}_1), ((h_{Q_2}, null), \boldsymbol{r}_2), ((h_{Q_3}, null), \boldsymbol{r}_3), ((h_{Q_4}, \mathtt{button}), \boldsymbol{r}_4),$ $((h_{Q_5}, \mathtt{button}), \boldsymbol{r}_5), ((h_{Q_6}, \mathtt{button}), \boldsymbol{r}_6), ((h_{Q_7}, \mathtt{button}), \boldsymbol{r}_7) \,\}$
2.31	$(\mathtt{button}, (\chi_{\mathtt{T1}\in[0,100)} \wedge \chi_{\mathtt{T2}>2}))$ is legal
2.34	$type(\mu) \leftarrow [1, 1]$
2.34	$type(\mathtt{T1}) \leftarrow [0, 100]$
2.34	$type(\mathtt{T2}) \leftarrow [0, \infty)$
2.36	$Q \leftarrow [1, 1] \times [0, 100] \times [0, \infty)$
3	**Illegal Mappings**
3.1	$\boldsymbol{r}_8 \leftarrow (\omega)$
3.3	$\boldsymbol{r}_8 \leftarrow (\omega, 0, \mathtt{T1}(last), \mathtt{T2}(last))$ [17]
3.5	$R \leftarrow \{\, \boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4, \boldsymbol{r}_7, \boldsymbol{r}_8 \,\}$
2.29	$\mathcal{Q} \leftarrow \{\, ((h_{Q_1}, null), \boldsymbol{r}_1), ((h_{Q_2}, null), \boldsymbol{r}_2), ((h_{Q_3}, null), \boldsymbol{r}_3), ((h_{Q_4}, \mathtt{button}), \boldsymbol{r}_4),$ $((h_{Q_5}, \mathtt{button}), \boldsymbol{r}_5), ((h_{Q_6}, \mathtt{button}), \boldsymbol{r}_6), ((h_{Q_7}, \mathtt{button}), \boldsymbol{r}_7),$ $((\chi_{\mu=0} \vee \chi_{\mathtt{T1}\notin[0,100]} \vee \chi_{\mathtt{T2}\notin\mathbb{R}^{\geq 0}}, null), \boldsymbol{r}_8), ((\chi_{\mu=0} \vee \chi_{\mathtt{T1}\notin[0,100]} \vee \chi_{\mathtt{T2}\notin\mathbb{R}^{\geq 0}}, \mathtt{button}), \boldsymbol{r}_8) \,\}$ [18]
3.9	return $(Q, \Theta, I, V, R, \Phi, \mathcal{Q}, \mathcal{B})$

[17] The illegal mode is a trap mode; the second coordinate is $\mu(last)$ that will always be 0.
[18] Use disjunctive form for illegal state set.

A.2 Power Window

The power window example is an adaptation of a similar example in [18]. The adapted requirements listed below are used to construct a hybrid enumeration. For this example, we forgo the detailed enumeration procedure and use the tabular form that includes the same steps. The tabular workflow proceeds from left to right, column to column, and in each column as many details as possible are added from top to bottom. Each detail in the table is traced to one or more requirements; requirement tags are added to the "Reqs" column.

A.2.1 Requirements

1. A power window system includes a window, a drive motor, a processor for control software, two timers, and a single 3 position control switch.

   (a) The system is initially powered.

   (b) The window can be positioned anywhere from down (0 cm) to up (50 cm). Initially the window is not moving and its position is unknown.

   (c) (D) Initially both timers are set to 0 and stopped.

   (d) (D) Initially the switch is in the intermediate position.

2. **Control Switch:** The control switch issues interrupts in the form of a discrete-time signal indicating driver commands. An up interrupt is issued when the switch is initially deflected toward the front of the car; a down interrupt is issued when the switch is initially deflected toward the rear of the car. When the switch initially returns to the spring-loaded intermediate position an intermediate interrupt is issued. Identical consecutive interrupts should be treated as a single interrupt.

   (a) **Intermediate Interrupt:** This interrupt will stop the drive motor except when the system is in the automatic mode.

      i. An intermediate interrupt will always be issued during transitions between up and down interrupts.

(b) **Up Interrupt:** This interrupt activates the drive motor to move the window up at a rate of 10 cm/sec except as follows:

i. Window motion stops when the window is up.

(c) **Down Interrupt:** This interrupt activates the drive motor to move the window down at a rate of 10 cm/sec except as follows:

i. Window motion stops when the window is down.

3. **Automatic Mode:** A switch timer keeps track of how long the switch is in either the up or the down position. Window motion will continue in the previously commanded direction when an intermediate interrupt (the control switch is released to the intermediate position) is issued and the timer's value is 0.5 seconds or less.

(a) Automatic motion stops when the window reaches the up or down position.

(b) Automatic motion stops when an up or down interrupt is issued.

(c) (D) An up or down interrupt that initiates motion starts the switch timer from 0.

(d) (D) The switch timer stops when window motion stops.

(e) (D) An intermediate interrupt stops the switch timer.

4. **Emergency Mode:** An emergency timer tracks the duration window movement commands continuously attempt to drive the window in one direction. As a safety measure, the motor will stop and a discrete-time emergency signal will be issued if the motor has not moved the window to the up or down position as commanded and 6 seconds or more have elapsed on the emergency timer. This overrides all other system behavior.

(a) (D) Initial window motion and changes in window direction start the emergency timer from 0.

(b) (D) The emergency timer stops when the window stops.

(c) (D) System power must be turned off to exit the emergency mode.

The (D) following requirements above indicates that these requirements were not included in the original statement of requirements but were derived or discovered as an augmentation to the requirements through the enumeration process.

A.2.2 Tabular Enumeration

In the four step columns, a ConstructHybridEnumeration process step is preceeded with C, a DefineMapping process step is preceeded with D, and an IllegalMapping process step is preceeded with I. The colors indicate sequence types and refinement stages.

Canonical Sequences (Mode) | Illegal Sequences | Reducible Sequences

Invariant Sequence | Refinement Step | Red Text: Unresolved enumeration elements.

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
lambda										o[0]	null	method	C 2.1	lambda	method	C 2.3
		eTime	[0,0]	1, 1c(D)	C 2.5	eTime-dot(t)	0	1, 1c(D)	C 2.11	eTime(0)	eTime(last)	method	C 2.12			
		sTime	[0,0]	1, 1c(D)	C 2.5	sTime-dot(t)	0	1, 1c(D)	C 2.11	sTime(0)	sTime(last)	method	C 2.12			
		wPosit	[0,50]	1, 1b	C 2.5	wPosit-dot(t)	0	1, 1b	C 2.11	wPosit(0)	wPosit(last)	method	C 2.12			
lambda	null									o[0]	null	4	D 2.11	lambda	1b, 1c(D)	D 3.4
		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	1c(D)	D 1.10	eTime(0)	eTime(last)	1c(D)	D 2.11			
		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	1c(D)	D 1.10	sTime(0)	sTime(last)	1c(D)	D 2.11			
		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	1b	D 1.10	wPosit(0)	wPosit(last)	1, 1b	D 2.11			
lambda	down									o[0]	null	4	D 2.13	lambda	method	C 3.7
		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	4, 4a(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13			
		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3c(D), 3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13			
		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13			
lambda	down									o[0]	null	method	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
		eTime	[0,inf)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11			
lambda	down									o[0]	null	method	D 2.11	----	2c	D 3.10
		eTime	[0,inf)	method	D 4.8	eTime-dot(t)	1	4a(D)	D 1.10	eTime(0)	0	4a(D)	D 2.9			
		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3c(D)	D 1.10	sTime(0)	0	3c(D)	D 2.9			
		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2c	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11			
lambda	int									o[0]	null	4	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	2a	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11			
lambda	up									o[0]	null	4	D 2.13	lambda	method	C 3.7
		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	4, 4a(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13			
		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3c(D), 3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13			
		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13			
lambda	up									o[0]	null	method	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
		eTime	[0,inf)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11			
lambda	up									o[0]	null	method	D 2.11	----	2b	D 3.10
		eTime	[0,inf)	method	D 4.8	eTime-dot(t)	1	4a(D)	D 1.10	eTime(0)	0	4a(D)	D 2.9			
		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3c(D)	D 1.10	sTime(0)	0	3c(D)	D 2.9			
		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2b	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11			
down	null									o[0]	null	4	D 2.13	down	method	C 3.7
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		
(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	-10	2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50]		
down	null									o[0]	null	4	D 2.13	down	method	C 3.7
[0,inf)		eTime	[6,inf)	4	D 4.4	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf)		
(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	0	2ci, 4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11	(0,50]		
down	null									o[0]	emergency	4	D 2.9	----	4	D 3.10
[0,inf)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf)		sTime	[0,inf)	3	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50]		wPosit	(0,50]	4	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
down	null									o[0]	null	4	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9			
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
(0,50]		wPosit	[0,0]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9			
down	null									o[0]	null	4	D 2.11	down	method	C 3.7
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		
(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	-10	2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50]		
down	null									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11			
down	null									o[0]	null	method	D 4.9	down	2c, 2ci, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf)		
(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11	(0,50]		
down	down									o[0]	null	4	D 2.13	down	method	C 3.7
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		
(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	-10	2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50]		
down	down									o[0]	null	4	D 2.13	down	method	C 3.7
[0,inf)		eTime	[6,inf)	4	D 4.4	eTime-dot(t)	0	2ci, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf)		
(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	0	2ci, 4	D 1.10	wPosit(0)	wPosit(last)	2ci	D 2.11	(0,50]		
down	down									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7	[0,inf),[6,inf)		
[0,inf)		sTime	[0,inf)	3	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf),[0,inf)		
(0,50]		wPosit	(0,50]	4	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7	(0,50],(0,50)		
down	down									o[0]	null	4	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9			
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
(0,50]		wPosit	[0,0]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9			
down	down									o[0]	null	4	D 2.11	down	method	C 3.7
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		
(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	-10	2ci, 4	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50]		
down	down									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11			
down	down									o[0]	null	method	D 4.9	down	2c, 2ci, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf)		
(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11	(0,50]		
down	int									o[0]	null	4	D 2.13	down	method	C 3.7
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3c(D), 3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf)		
(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	-10	2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50]		
down	int									o[0]	null	4	D 2.11	down	method	C 3.7
[0,inf)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf)		
(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	-10	2ci, 3	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50]		
down	int									o[0]	null	method	D 4.7	down	method	C 3.7
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,0.5]	3	D 4.4	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf)		
(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	-10	2ci, 3	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50]		
down	int									o[0]	null	method	D 4.7	lambda	1b, 1c(D), 2, 3, 4	D 3.4

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	2c, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	[0,0.5]	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.11			
down	int									o[0]	null	method	D 4.9	----	3	D 3.10
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	2c, 4	D 1.10	eTime(0)	eTime(last)	4	D 2.11			
[0,inf)		sTime	[0,0.5]	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2c, 3	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11			
down	int									o[0]	null	method	D 4.9	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	(0.5,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2a, 3	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.11			
down	int									o[0]	null	4	D 2.13	down	method	C 3.7
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	2ci, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf)		
(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2ci, 4b(D)	D 1.10	wPosit(0)	wPosit(last)	2ci	D 2.11	(0,50]		
down	int									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7	[0,inf).[6,inf)		
[0,inf)		sTime	[0,inf)	3	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf).[0,inf)		
(0,50]		wPosit	(0,50]	4	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7	(0,50].(0,50)		
down	int									o[0]	null	4	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9			
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
(0,50]		wPosit	[0,0]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9			
down	up									o[0]	illegal	method	I 1.1	Illegal	2ai	I 1.2
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	2ai	D 1.4	eTime(0)	eTime(last)	method	I 1.1			
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	method	I 1.1	sTime(0)	sTime(last)	method	I 1.1			
(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	method	I 1.1	wPosit(0)	wPosit(last)	method	I 1.1			
up	null									o[0]	null	4	D 2.13	up	method	C 3.7
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		
[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	10	2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50)		
up	null									o[0]	null	4	D 2.13	up	method	C 3.7
[0,inf)		eTime	[6,inf)	4	D 4.4	eTime-dot(t)	0	4bD)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf)		
[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	0	2bi, 4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11	[0,50)		
up	null									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7	[0,inf).[6,inf)		
[0,inf)		sTime	[0,inf)	3	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf).[0,inf)		
[0,50)		wPosit	[0,50)	2bi, 4	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7	(0,50].(0,50)		
up	null									o[0]	null	4	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9			
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
[0,50)		wPosit	[50,50]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9			
up	null									o[0]	null	4	D 2.11	up	method	C 3.7
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		
[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	10	2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50)		
up	null									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11			
up	null									o[0]	null	method	D 4.9	up	2b, 2bi, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf)		
[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11	[0,50)		
up	down									o[0]	illegal	method	I 1.1	Illegal	2ai	I 1.2
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	2ai	D 1.4	eTime(0)	eTime(last)	method	I 1.1			

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	method	I 1.1	sTime(0)	sTime(last)	method	I 1.1			
[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	method	I 1.1	wPosit(0)	wPosit(last)	method	I 1.1			
up	int									o[0]	null	4	D 2.13	up	method	C 3.7
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D), 3c(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf)		
[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	10	2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50)		
up	int									o[0]	null	4	D 2.11	up	method	C 3.7
[0,inf)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf)		
[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	10	2bi, 3	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50)		
up	int									o[0]	null	method	D 4.7	up	method	C 3.7
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,0.5]	3	D 4.4	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf)		
[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	10	2bi, 3	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50)		
up	int									o[0]	null	method	D 4.7	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	2b, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	[0,0.5]	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b	D 2.11			
up	int									o[0]	null	method	D 4.9	----	3	D 3.10
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	2b, 4	D 1.10	eTime(0)	eTime(last)	4	D 2.11			
[0,inf)		sTime	[0,0.5]	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2b, 3	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b	D 2.11			
up	int									o[0]	null	method	D 4.9	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	(0.5,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2b, 3	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b	D 2.11			
up	int									o[0]	null	4	D 2.13	up	method	C 3.7
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	2bi, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf)		
[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2bi, 4b(D)	D 1.10	wPosit(0)	wPosit(last)	2bi	D 2.11	[0,50)		
up	int									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7	[0,inf),[6,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf),[0,inf)		
[0,50)		wPosit	[0,50)	4	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7	(0,50],(0,50)		
up	int									o[0]	null	4	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9			
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
[0,50)		wPosit	[50,50]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9			
up	up									o[0]	null	4	D 2.13	up	method	C 3.7
[0,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		
[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	10	2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50)		
up	up									o[0]	null	4	D 2.13	up	method	C 3.7
[0,inf)		eTime	[6,inf)	4	D 4.4	eTime-dot(t)	0	2bi, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf)		
[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	0	2bi, 4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b, 2bi	D 2.11	[0,50)		
up	up									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7	[0,inf),[6,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf),[0,inf)		
[0,50)		wPosit	[0,50)	2bi, 4	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7	(0,50],(0,50)		
up	up									o[0]	null	4	D 2.11	lambda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9			
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9			
[0,50)		wPosit	[50,50]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9			
up	up									o[0]	null	4	D 2.11	up	method	C 3.7
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf)		

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	10	2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50)		
up	up									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf)		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11			
up	up									o[0]	null	method	D 4.9	up	2b, 2bi, 4	D 3.4
[0,inf)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf)		
[0,inf)		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf)		
[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2b	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11	[0,50)		
down.null	null									o[0]	null	4	D 2.11	down.null	4	D 3.4
[0,inf).[6,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[6,inf)		
[0,inf).[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b, 2c 4	D 2.11	(0,50].(0,50]		
down.null	down									o[0]	null	4	D 2.11	down.null	4	D 3.4
[0,inf).[6,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[6,inf)		
[0,inf).[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b, 2c 4	D 2.11	(0,50].(0,50]		
down.null	int									o[0]	null	4	D 2.11	down.null	4	D 3.4
[0,inf).[6,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[6,inf)		
[0,inf).[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b, 2c 4	D 2.11	(0,50].(0,50]		
down.null	up									o[0]	null	4	D 2.11	down.null	4	D 3.4
[0,inf).[6,inf)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[6,inf)		
[0,inf).[0,inf)		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	0	4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b, 2c 4	D 2.11	(0,50].(0,50]		
down.int	null									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	2ci, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	-10	2ci, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	null									o[0]	null	4	D 2.11	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	2ci, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	-10	2ci, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	null									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50].(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.11			
down.int	null									o[0]	null	method	D 4.9	down.int	3	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2c	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.11	(0,50].(0,50]		
down.int	null									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	2ci, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2ci, 4	D 1.10	wPosit(0)	wPosit(last)	2ci, 4	D 2.11	(0,50].(0,50]		
down.int	null									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50].(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
down.int	null									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		
down.int	down									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	-10	2ci, 4	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	down									o[0]	null	4	D 2.11	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	-10	2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	down									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
(0,50].(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11			
down.int	down									o[0]	null	method	D 4.9	down	2c, 2ci, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3c(D)	D 1.10	sTime(0)	0	3c(D)	D 2.9	[0,inf)		
(0,50].(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2c	D 1.10	wPosit(0)	wPosit(last)	2c	D 2.11	(0,50]		
down.int	down									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	4, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2ci, 4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11	(0,50].(0,50]		
down.int	down									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50].(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
down.int	down									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		
down.int	int									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	2ci, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	-10	2ci, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	int									o[0]	null	4	D 2.11	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	2ci, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	-10	2ci, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	int									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50].(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.11			
down.int	int									o[0]	null	method	D 4.9	down.int	3	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2c	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.11	(0,50].(0,50]		
down.int	int									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	2ci, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2ci, 4	D 1.10	wPosit(0)	wPosit(last)	2ci, 4	D 2.11	(0,50].(0,50]		
down.int	int									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50].(0,50]		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
down.int	int									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		
down.int	up									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
(0,50].(0,50]		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	-10	2b, 2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	up									o[0]	null	4	D 2.11	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	4, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	-10	2b, 2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	(0,50].(0,50]		
down.int	up									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
(0,50].(0,50]		wPosit	[50,50]	2ci	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11			
down.int	up									o[0]	null	method	D 4.9	up	2b, 2bi, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	0	4c(D)	D 2.9	[0,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3	D 1.10	sTime(0)	0	3c(D)	D 2.9	[0,inf)		
(0,50].(0,50]		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2b	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11	[0,50)		
down.int	up									o[0]	null	4	D 2.13	down.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	4, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,0.5]		
(0,50].(0,50]		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2bi, 4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2b	D 2.11	(0,50].(0,50]		
down.int	up									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
(0,50].(0,50]		wPosit	[50,50]	2ci	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
down.int	up									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
(0,50].(0,50]		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		
up.int	null									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	2bi, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	10	2bi, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	null									o[0]	null	4	D 2.11	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	2bi, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	10	2bi, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	null									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
[0,50).[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b	D 2.11			
up.int	null									o[0]	null	method	D 4.9	up.int	3	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2b	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b	D 2.11	[0,50).[0,50)		
up.int	null									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	2bi, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2bi, 4	D 1.10	wPosit(0)	wPosit(last)	2bi, 4	D 2.11	[0,50).[0,50)		
up.int	null									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
[0,50).[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
up.int	null									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
[0,50).[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		
up.int	down									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
[0,50).[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	10	2c, 2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	down									o[0]	null	4	D 2.11	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	4, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	10	2c, 2ci	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	down									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
[0,50).[0,50)		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	2ci	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.11			
up.int	down									o[0]	null	method	D 4.9	down	2c, 2ci, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	0	4c(D)	D 2.9	[0,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3	D 1.10	sTime(0)	0	3c(D)	D 2.9	[0,inf)		
[0,50).[0,50)		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	-10	2c	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2c	D 2.13	(0,50]		
up.int	down									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	4, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2ci, 4	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2c	D 2.11	[0,50).[0,50)		
up.int	down									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
[0,50).[0,50)		wPosit	[0,0]	2ci	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
up.int	down									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
[0,50).[0,50)		wPosit	(0,50]	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		
up.int	int									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	2bi, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3	D 1.10	sTime(0)	sTime(last)	3	D 2.11	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	10	2bi, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	int									o[0]	null	4	D 2.11	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	2bi, 4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	10	2bi, 4b(D)	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	int									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
[0,50).[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b	D 2.11			
up.int	int									o[0]	null	method	D 4.9	up.int	3	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2b	D 1.10	wPosit(0)	wPosit(last)	1, 2, 2a, 2b	D 2.11	[0,50).[0,50)		
up.int	int									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	2bi, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2bi, 4	D 1.10	wPosit(0)	wPosit(last)	2bi, 4	D 2.11	[0,50).[0,50)		
up.int	int									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
[0,50).[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
up.int	int									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
[0,50).[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		
up.int	up									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,inf)	method	C 3.6	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	C 3.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		

Prefix Sequence	*Stimulus*	*Characteristic Predicates*				*Trajectory Definition*				*Condition Vector*				*Equivalent Sequence*		
	I	Var	Blocks	Reqs	Step	Var	exp	Reqs	Step	Var	exp	Reqs	Step	Sequence	Reqs	Step
[0,50).[0,50)		wPosit	[0,50]	method	C 3.6	wPosit-dot(t)	10	2bi, 4	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	up									o[0]	null	4	D 2.11	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[0,6)	4	D 4.4	eTime-dot(t)	1	4b(D)	D 1.19	eTime(0)	eTime(last)	method	D 2.13	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.19	sTime(0)	sTime(last)	method	D 2.13	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.6	wPosit-dot(t)	10	2bi	D 1.19	wPosit(0)	wPosit(last)	method	D 2.13	[0,50).[0,50)		
up.int	up									o[0]	null	method	D 4.7	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.6	eTime-dot(t)	0	4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11			
[0,50).[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	2bi	D 1.10	wPosit(0)	wPosit(last)	2bi	D 2.11			
up.int	up									o[0]	null	method	D 4.9	up	2b, 2bi, 4	D 3.4
[0,inf).[0,6)		eTime	[0,6)	method	D 4.8	eTime-dot(t)	1	4	D 1.10	eTime(0)	eTime(last)	4	D 2.11	[0,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	1	3c(D)	D 1.10	sTime(0)	0	3c(D)	D 2.9	[0,inf)		
[0,50).[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	10	2b	D 1.10	wPosit(0)	wPosit(last)	2b	D 2.11	[0,50)		
up.int	up									o[0]	null	4	D 2.13	up.int	method	C 3.7
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	4, 4b(D)	D 1.10	eTime(0)	eTime(last)	4b(D)	D 2.11	[0,inf).[0,6)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	3d(D)	D 1.10	sTime(0)	sTime(last)	3d(D)	D 2.11	[0,inf).[0,0.5]		
[0,50).[0,50)		wPosit	[0,50]	method	D 4.8	wPosit-dot(t)	0	2bi, 4	D 1.10	wPosit(0)	wPosit(last)	2b	D 2.11	[0,50).[0,50)		
up.int	up									o[0]	null	4	D 2.11	lamda	1b, 1c(D), 2, 3, 4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.6	eTime-dot(t)	0	method	D 4.7	eTime(0)	eTime(last)	method	D 4.7			
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.6	sTime-dot(t)	0	method	D 4.7	sTime(0)	sTime(last)	method	D 4.7			
[0,50).[0,50)		wPosit	[50,50]	2bi	D 4.4	wPosit-dot(t)	0	method	D 4.7	wPosit(0)	wPosit(last)	method	D 4.7			
up.int	up									o[0]	emergency	4	D 2.9	down.null	4	D 3.4
[0,inf).[0,6)		eTime	[6,inf)	method	D 4.8	eTime-dot(t)	0	method	D 4.9	eTime(0)	eTime(last)	method	D 4.9	[0,inf).[6,inf)		
[0,inf).[0,0.5]		sTime	[0,inf)	method	D 4.8	sTime-dot(t)	0	method	D 4.9	sTime(0)	sTime(last)	method	D 4.9	[0,inf).[0,inf)		
[0,50).[0,50)		wPosit	[0,50)	method	D 4.8	wPosit-dot(t)	0	method	D 4.9	wPosit(0)	wPosit(last)	method	D 4.9	(0,50].(0,50]		

Vita

Jason Martin Carter was born in Dayton, Ohio, on June 27, 1968. After completing Morristown-Hamblem High School East, Morristown, TN in 1986, he entered the United State Naval Academy and graduated with a Bachelor of Science degree in Computer Science in 1990. He was designated a Naval Aviator in August of 1992 and flew missions all over the world until 1997. From 1997 to 2000, Lieutenant Carter was responsible for applications, process improvement, and advancements in technology for admissions at the United States Naval Academy. After leaving active duty in 2000, Jason was a general manager for a small business and then a supervisor for an automated production facility in the automotive industry. From January 2005 to December 2009, he attended the University of Tennessee, Knoxville and worked in the Software Quality Research Laboratory (SQRL). Jason continues to serve in the United States Navy Reserve and was promoted to the rank of Commander in 2007. He received his Master's of Science Degree in Computer Science in May 2007 and Doctor of Philosophy Degree in Computer Science in December of 2009.