The Dissertation Committee for Shant Harutunian
certifies that this is the approved version of the following dissertation:

Formal Verification of Computer Controlled Systems

Committee:

Warren A. Hunt, Jr., Supervisor

Jacob A. Abraham

Adnan Aziz

Craig M. Chase

Raul G. Longoria

Formal Verification of Computer Controlled Systems

by

Shant Harutunian, B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2007

Dedicated to the Harutunian family.

Formal Verification of Computer Controlled Systems

Publication No. ______________

Shant Harutunian, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Warren A. Hunt, Jr.

This dissertation discusses the application of formal verification methods to reasoning about the correctness of computer controlled systems. Due to the switching that is introduced by the computer controller, the differential equations associated with such systems may consist of discontinuous vector fields. The physical system may also experience switching due to physical interaction, such as impact. Such system models may exhibit an infinite number of switches in finite time. For example, using rigid body modeling assumptions, a bouncing ball may have infinitely many elastic impacts, but come to rest in finite time. Using nonstandard analysis, we present a model for computer controlled systems which accommodates discontinuous vector fields as well as infinite switches in finite time. This model includes both the semantics of the computer program as well as the ordinary differential equations governing the physical system behavior. We develop a nonstandard definition of a solution for such a model and formally prove that the solution exists.

Using this nonstandard definition of solution, we develop proof procedures whereby one may reason about safety and progress properties of the system. The soundness of these proof procedures is formally shown. We conclude with the presentation of a simple example computer controlled system using the presented model, for which safety and progress properties are shown using the respective proof procedures.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation addresses the application of formal verification methods to reason about the correctness of computer controlled systems. A computer controlled system may be regarded as consisting of a computer with several input/output signals connected to sensors and actuators in a physical system which the computer is to monitor and control. Example applications include control systems for automobiles, air crafts, food manufacturing facilities, and chemical plants. In each of these applications, a computer is used to sense signals from the associated process and send back signals to achieve some desired process behavior.

1.1 Interest and Scope

Our research interest is to model and formally reason about the correctness of *properties* of computer controlled systems. A property is some behavior that the control system is to exhibit. In linear control system analysis, Bounded Input Bounded Output (BIBO) stability is an example system property [47]. This property states that, for zero initial conditions, if the input signal to the control system is bounded, then the output signal is bounded as well.

For a computer controlled system, we assume that the physical system dynamics is expressed as a system of ordinary differential equations. Due to events which occur in the system, the physical system model may change from one set of differential equations to another. In particular, in this work, we are interested in the case when such changes in the physical system model are dictated by the output of some computer which is said to be the controller of the system. It is ideal to use a method for reasoning about the semantics of the computer program as well as the behavior of the dynamic system. This dissertation shows how a proof method used for reasoning about computer programs can be extended to reason about continuous dynamic systems with the aid of nonstandard analysis.

1.2 Motivation of this Work

The composition of some discrete subsystem, such as a computer, with a physical system is referred to as a hybrid system. In this dissertation, we are interested in the case where the discrete subsystem is a computer. Modeling a hybrid system requires a model of the physical system as well as the computer and the semantics of the program executing on the computer. Physical systems are typically modeled by ordinary differential equations. However, the equations associated with such systems may be discontinuous. While the observed behavior of some variable of the system, for example the position of an object in space, may be continuous, the first derivative (velocity) or second derivative (acceleration) may not be. Furthermore, it is possible that switching or assignment to the system variables may occur. Such activity may be a consequence of the physical system model, for example rigid body impact, or by the action of a computer controller. Hence, the model should take into account this switching and variable assignment. Due to its real valued and real time nature, a mathematical model of a physical system may exhibit infinitely many switches in finite time.

1.3 Objectives of this Work

Our objective for this work is to provide a model of a hybrid system that 1) models the computer and the semantics of its program, 2) models the ordinary differential equations of the physical system, 3) models physical systems with discontinuous vector fields, and 4) models events that occur over time, even if they should occur infinitely often over finite time.

In addition to providing a mathematic model for a hybrid system, we wish to provide a formal definition of its behavior over time, or its solution. We wish to show that this solution exists and to systematically prove properties about it. For example, we wish to prove if the value of a system variable remains within some bound, a safety property [48, 53], or to show that the solution reaches a particular state, a progress property [52].

1.4 Limitations of Existing Methods

Extensive research has been done in the area of modeling and formal reasoning about hybrid systems. Such models, however, typically regard the controller as a finite state machine. Each state of the finite state machine, or mode, has an associated physical system model, such as an ordinary differential equation or differential inclusion, governing the behavior of the physical system. Models and methods are also developed for switching systems. However, such models do not model the semantics of a computer program. In particular, should the program consist of arithmetic operations, as computer controllers typically do, such operations would be difficult to model as a finite state machine. Furthermore, switching may occur between physical system differential equations due to the change in output of the computer, or due to switching that inherently exists in the physical system models, for example due to impact [69]. In addition, the physical system model may consist of infinitely many switches over finite time. For example, a rigid body model of a ball bouncing with a coefficient of restitution less than 1, will have infinitely many impacts in finite time. Tools, such as hybrid automata, allow for modeling finite impacts in finite time only.

1.5 Outline of Dissertation

In chapter 2, we provide a brief survey of existing research in the area of hybrid systems. In chapter 3, we provide an extended background section that covers notation to be used in the dissertation, an introduction to ordinals, an introduction to nonstandard analysis, and an introduction to existing definitions of differential equation solutions. In particular, we clarify that the logic we will be using is based on the axiomatization of nonstandard analysis, by Edward Nelson, referred to as internal set theory [59].

In chapter 4, we apply nonstandard analysis in the modeling and reasoning about differential equations. We present a nonstandard proof of the existence and uniqueness theorem of differential equations. This chapter establishes a simple model and lemmas which we extend to a more general system model that we formally define for modeling hybrid systems in the succeeding chapters. We have mechanically proved the theorems and lemmas presented in this chapter using the ACL2r theorem prover [1, 29].

In chapter 5, we present a formal nonstandard model of a hybrid system. Section 5.1 presents the first contribution of this work which is a formal model of both the semantics of the computer program as well as the ordinary differential equations governing the behavior of the physical system. Section 5.1 also presents the second contribution of this research which is the modeling of physical systems with discontinuous vector fields. Section 5.3 presents the third contribution of this research which is the definition of a system solution for such a model. Section 5.5 presents a proof for the existence of such a solution.

In chapter 6, we present the fourth contribution of this research. The chapter presents formal methods whereby one may reason about safety and progress properties of the solution of the system model presented in chapter

5. Section 6.1 presents the proof procedure for proving safety properties and section 6.4 presents the soundness of this safety proof procedure. Section 6.5 presents the proof procedure for proving progress properties and section 6.8 presents the soundness of this progress proof procedure.

In chapter 7, we present a sample system consisting of a computer and a simple model of a physical system. We use the hybrid system model defined in chapter 5 to present a model of the sample system. We then use the proof procedures defined in chapter 6 to show both safety and progress properties about this sample system.

In chapter 8, we conclude with remarks about our work to date and provide direction by which it may be extended.

Chapter 2

Existing Research in Formal Verification of Control Systems

In some special cases, a discrete time version of the linear control theory may hold for a subclass of computer programs [47]. However, in the general case, a computer controlled system may not be modeled by the state equations for linear systems. Researchers have modeled computer controlled systems as a transition system whose state variables are Boolean and real valued [35]. It is also useful to describe the computer controlled system as a composition of transition systems, namely the transition system(s) modeling the computer program(s), and transition system(s) that model the physical equipment in the process environment that the computer is interfacing with.

Since, in general, a computer controlled system may not be modeled by the state equations for linear systems, well established methods of stability for linear control systems cannot be applied. The computer also introduces switching of the system dynamics of the physical system; the differential equations used to describe the system change as the computer outputs change [12]. If one were to study the differential equations associated with each mode of the physical system, they may each be unstable. However, it is possible that the computer switches the modes of the physical system such that the overall system behavior is stable.

2.1 Verification of Control Systems in Industry

In industry, computer controlled systems are often analyzed and their correctness established through computer simulation of both the physical system and the computer program. Such simulation programs approximate the non-linear behavior exhibited by the equations associated with the physical equipment by using a Newton-Raphson approximation technique, for example. The computer programs also simulate the behavior of the system as dictated by the differential equations by running the system for a δ time step, where δ is small enough to capture, within some error, the next state of the system [60]. Such numerical techniques have been extremely useful in modeling the system prior to its construction. However, a key drawback in simulating computer controlled systems is that all execution paths of the computer algorithm may not be explored. While simulation is powerful in prototyping the equipment, the execution of a simulation is often time consuming; repeating the simulation for every execution path of the computer algorithm may be impractical.

When using simulation to verify system correctness, the model is *executed* on a given input and the output is observed. This procedure is repeated for each input to be tested. However, it may be impractical to run a simulation for all possible inputs to the system.

2.2 Existing Research in Formal Verification of Control Systems

Some methods have evolved in the research community of program verification that may help address the verification of control systems without the simulation of every execution path and every possible input. Rather than explicitly executing the system model for each input, researchers have developed methods whereby one uses logic to prove that the system does satisfy a property for all possible inputs.

2.3 Introduction to Existing Methods

Table 2.1 on the following page presents various verification methods. For digital hardware systems, a finite state transition system is used to model state machines. Temporal logics and model checking techniques have been used to prove properties about such systems. This research has been extended to include quantitative temporal reasoning, where the time duration between events is a discrete quantity: the number of transition steps between those events. In the research of timed automata, non-negative real valued variables, called *clocks*, are used to model time durations between events. In the research of hybrid automata, timed automata are extended to include real valued, time varying variables that are useful in representing physical quantities, such as temperature, pressure, or flow. The table shows the methods as supporting the boolean type. It should be noted that the methods do not exclusively support boolean variables, but support n nodes in their graph representation, where each node represents a value of the discrete state space. Since n nodes may be encoded with $\lceil log_2(n) \rceil$ boolean variables, and since it is common in propositional temporal model checking to represent the next state function using boolean functions, the convention of using boolean variables to represent the discrete space is adopted for this presentation.

Table 2.1: Features of Formal Methods Used to Reason About Control Systems

Formal Method	*Variable Types*	*Modeling of Time*	*Terminates for All Cases*	*Example Tool*
Propositional Temporal Logics	Boolean	Untimed	Yes	SMV
Quantitative Temporal Reasoning	Boolean	Discrete	Yes	Verus
Timed Automata	Boolean and Clocks	Real	Yes	Kronos
Hybrid Automata	Boolean and Real	Real	No	-
Linear Hybrid Automata	Boolean and Real	Real	No	Hytech

A computer controlled system is often modeled to consist of *discrete* and *continuous* transitions. A *discrete* transition is associated with a time varying variable, but the variable is only modeled to change at discrete points in time. For a discrete transition system, it may be the case that two discrete transitions occur one after the other, at times t_1 and t_2, where $t_1 < t_2$, and there is no intervening discrete transition that occurs at time t, where $t_1 < t < t_2$. A *discrete* transition is usually associated with the change of a variable of the computer program. We denote the non-negative reals as $\mathbb{R}^{\geq 0}$. A *continuous* time varying variable may be regarded as a function mapping

$\mathbb{R}^{\geq 0}$ to the power set of the domain of the variable. A *continuous* transition is associated with the change of such a variable. For any times t_1 and t_2, if a continuous transition occurs at t_1, and another at t_2, where $t_1 < t_2$, then there is a continuous transition that occurs at time t, where $t_1 < t < t_2$. A *continuous* transition is usually associated with the change of a physical system variable, such as temperature, pressure or flow. Each of the discussed formal methods have different approaches of modeling discrete and continuous transitions. Tables 2.2 and 2.3 present the transition types modeled by the various existing methods.

Table 2.2: Methods of Modeling Discrete Transitions

Formal Method	*Discrete Transition*
Propositional Temporal Logics	Boolean Function (BDD) Relating Current and Next State
Quantitative Temporal Reasoning	Boolean Function (BDD) Relating Current and Next State
Timed Automata	Graph Edge Relation and zero or more Clocks Reset
Hybrid Automata	Graph Edge Relation and boolean combination of equations/inequations which constrain the next state variables $\mathbf{X}'$ based on the current state variables $\mathbf{X}$
Linear Hybrid Automata	Graph Edge Relation and a conjunction of linear equations/inequations which constrain the next state variables $\mathbf{X}'$ based on the current state variables $\mathbf{X}$

In these tables, $\mathbf{X}$ denotes the finite set of continuous time varying

Table 2.3: Methods of Modeling Continuous Transitions

Formal Method	*Continuous Transition*
Propositional Temporal Logics	None
Quantitative Temporal Reasoning	None
Timed Automata	For a clock x, $\dot{x} = 1$
Hybrid Automata	A boolean combination of equations/inequations whose variables are in $\mathbf{X} \cup \dot{\mathbf{X}}$
Linear Hybrid Automata	A conjunction of linear equations/inequations whose variables are only in $\dot{\mathbf{X}}$

variables . If the cardinality of $\mathbf{X}$ is n, then $V = \mathbb{R}^n$. The set $\dot{\mathbf{X}}$ represents the set of first derivatives (with respect to time) of all the variables in $\mathbf{X}$. That is, if we let $\mathbf{X} = \{x_1, x_2, x_3, \ldots, x_n\}$, then $\dot{\mathbf{X}} = \{\dot{x_1}, \dot{x_2}, \dot{x_3}, \ldots, \dot{x_n}\}$. For $x \in \mathbf{X}$, x' represents the value of x after the discrete transition is taken. We let $\mathbf{X}' = \{x'_1, x'_2, x'_3, \ldots, x'_n\}$. Propositional temporal logic and quantitative temporal reasoning both have "None" under the *Continuous Transition* column heading. The system models associated with these methods do not have a representation for non-negative, real valued time and, hence, cannot represent continuous transitions.

For timed and hybrid automata, the decision procedures represent the automata as a graph. The edges of this graph represent the discrete transitions of the system. The discrete transitions occur when the boolean variables change value. In addition, for timed automata, a discrete transition also occurs when zero or more clock variables are reset. For hybrid automata, the effect of a discrete transition on the real valued variables is characterized by predicates over the current state and next state variables, as discussed in tables 2.2 and 2.3.

The following sections discuss these methods and their application to computer controlled system verification in further detail.

2.4 Propositional Temporal Logics

Much work has been done in the formal verification and reasoning about transition systems. The transition system is effectively a graph where each node represents the state in the system and an edge represents a possible transition from one state to the next. Propositional temporal logics are used to state properties about such transition structures. Examples of propositional temporal logics include Computation Tree Logic (CTL), CTL*, CTL+, and PLTL [23]. Associated with these logics are algorithms whereby one can determine whether a given transition structure satisfies the property stated by a formula from the logic. This technique is referred to as model checking. Such logics have been useful in reasoning about the correctness of transition systems without having to execute every computable path in the system. Such logics assume that each computable path is infinite. That is, when reasoning about the correctness of the system, the system is observed from time $t = 0$ (initial state), to $t = \infty$. Indeed, such a computable path cannot be simulated, since it would require infinite time. Furthermore, even if one were to choose a reasonable time limit for simulation, the number of paths to be explored may be too large for simulation within reasonable time.

However, such propositional temporal logics assume that the transition system is a graph, where the nodes in the graph represent the possible values of the state variables of the system. For large systems, a computer implementation of the model checking decision procedure of the logic may result in too many nodes in the graph of the transition system, exhausting the computer memory resources available. Researchers have developed computer implementations of the propositional temporal model checking decision procedures whereby the transition system is represented symbolically in the memory of the computer, as apposed to explicitly as a graph [15, 16]. Such symbolic methods have attained reasonable success in reasoning about Boolean valued transition systems. These methods use, for example, Binary Decision Diagrams to represent the Boolean valued functions of the system [13, 14]. Alternatively, some propositional temporal logic decision procedures make use of Boolean satisfiability checkers [8, 22].

2.5 Quantitative Temporal Reasoning

The previously discussed propositional temporal logics do not model time. However, when reasoning about computer controlled systems, reasoning about time is required. Some extensions to propositional temporal logics have been proposed whereby time is modeled implicitly, namely as the number of steps taken in the state transition system. This method is commonly referred to as quantitative temporal reasoning [17–20, 26]. Using this method, one may state properties regarding the duration of time (number of steps) between events in the system. Since it reasons about time durations in terms of transition steps, this method uses a discrete time model.

2.6 Timed Automata

Other researchers have addressed the matter of modeling time by proposing a new theory of modeling timed systems, called timed automata [3, 4]. The model of this type of transition system includes Boolean valued functions for input, output, and state variables, as well as a representation of *clock* variables. Clock variables assume non-negative, real values. For a given system, all clock variables increase at the same fixed, non-negative rate. The system can reset a clock variable to zero, but it cannot write non-zero values to a clock variable.

Due to the introduction of real valued variables for representing clocks, the explicit representation of the reachable states from the initial state requires a graph with an infinite number of nodes. Unlike boolean variables, clock variables assume values from the set of non-negative real numbers and, if not reset, increase uniformly. A significant contribution of timed automata is a method of abstracting this infinite graph representation of the transition system to a finite graph representation. Symbolic methods have been applied to real time automata as well [7, 39, 54]. However, in general, the timed automata decision procedures have not enjoyed the utility and much of the success of the untimed, propositional temporal logic decision procedures. The timed automata graph representation, even when symbolic, results in memory exhaustion in the implementation of the model checking procedure, even for moderately simple systems [24]. Nonetheless, the research in timed automata has been valuable in presenting methods for reasoning about infinite state systems.

2.7 Hybrid Automata

Hybrid automata extends the ideas from timed automata [4, 34, 36, 37, 40–42]. While the timed automata model uses non-negative real valued variables called *clocks* to represent time, the hybrid automata model allows the real valued variables to represent time as well as the input, output, and state variables of the control system. This is a significant contribution in the modeling of control systems. This model allows for the representation of physical system quantities, such as temperature, pressure, or flow. However, unlike its real time version, the general hybrid automata decision procedure is not guaranteed to terminate. That is, given the initial state(s), if the transition system is symbolically executed, the decision procedure may not reach a fixed point; it may proceed interminably. This limitation is usually overcome, similar to the approach in system simulation, by assuming an upper bound on the time variable t.

A subset of hybrid automata theory called linear hybrid automata is used to implement a decision procedure for hybrid systems [30, 35]. Linear hybrid automata has an associated decision procedure whereby the transition system is represented as linear polynomial inequalities with existential and universal quantification. The software tool which implements this decision procedure is named Hytech [35, 67]. To allow for linear arithmetic reasoning, the input language for Hytech has restrictions on the defined transition system; the differential equations are posed as the first derivative of a state variable set equal to (or less than, or some other inequality operator) a linear polynomial only in terms of constants and the first derivatives of state variables. It should be noted that the first derivative of a variable is not allowed to be expressed directly in terms of the state variables of the system. That is, $dx/dt = x$ is not allowed. An interesting consequence of this representation is that systems may be described by differential equations as well as inequations. The approach also allows the logical conjunction of one or more such differential inequation. This allows the approximation of the physical equation of the system. For example, if dx/dt is a complicated equation, but dx/dt is upper and lower bounded by constants, then the system may be conservatively approximated by $dx/dt > L \wedge dx/dt < U$, where L and U are the constants for the lower and upper bounds, respectively.

At first, such approximations may seem crude. However, such approximations are accepted to allow for reasoning about the system for arbitrary values of the input variables. While a computer simulation of the actual differential equation would yield a more accurate behavior over time, one would have to repeat the simulation for each of the execution paths of the computer program. The theory of hybrid automata provides a framework within which to use logic and arithmetic to reason about the behavior of the system for a large number of values for the inputs (in some cases infinite values), without requiring repetition of simulations.

2.8 Theorem Proving

Theorem proving tools have been used to reason about correctness properties of real time and hybrid systems. In the work of Boyer and Moore, a model of a vehicle autopilot control program is described and stability properties are proved about it [11]. In the work of Shankar, a real time logic is formalized using the PVS theorem prover [49, 66]. The theorem prover is used to model a rail road crossing problem and Fischer's mutual exclusion protocol. The PVS theorem prover is also used by Mader, Wupper, and Bauerto to verify some properties about portions of a batch plant [50], based on the Verification of Hybrid System (VHS) Case Study 1 [71].

2.9 Nonstandard Analysis

In the work of Heinrich Rust, a nonstandard analysis approach is taken to discretize the real line in representing time [62–64]. This approach extends existing logics used for reasoning about real time systems by including the notion of an infinitesimal time step. In the work of Krob and Bliudze, models are introduced of discrete and continuous systems using nonstandard analysis [46]. In the work by Nakamura and Fusaoka, nonstandard analysis is applied to hybrid automata to allow for modeling of infinitely many switches in finite time [58]. In the work of Iwasaki and others, nonstandard analysis models are introduced and methods are developed for reasoning about such nonstandard hybrid systems [44].

2.10 Other Research

There has been other research in the area of formal verification of control systems. Alur [5] proposes a method whereby a hybrid automaton is abstracted using predicate abstraction techniques. In this approach, predicates about the infinite state space are proposed, and a conservative, finite abstraction of the transition system is derived, where the abstract state space is the values of the predicates for all possible values of the original state space. The paper by Alur also discusses heuristics whereby new predicates may be derived to further refine the state space based on examples of failed properties of the current abstraction. This approach effectively converts a hybrid automata into a transition structure that consists of booleans only, allowing the methods of propositional temporal logic or quantitative temporal reasoning to be used to reason about properties of the abstracted system. Another method of abstraction is based on numerical methods to simulate the system and using flow pipe approximations, as developed by Chutinan [21]. Other techniques regarding approximation and abstraction are discussed in [25, 55, 70].

In the work of Branicky [12], stability of dynamic systems is discussed. The paper is primarily concerned with switched systems where the system dynamics switch from one set of equations to another. The paper does not address the computer program or control system that initiates the switching, but assumes that the system switches finitely many times within a finite time period. The paper proposes the extension of Lyapunov stability to such switched systems.

In the work of Adams et al., the PVS theorem prover is extended with the Maple computer algebra system [2]. This extension allows reasoning about such functions as exponent, sine, and other transcendentals. The system also provides tools for checking continuity of functions. A similar approach is taken with the HOL theorem prover [32].

2.11 Summary of Existing Research in Formal Verification of Control Systems

While it is ideal to reason about computer controlled systems using the linear system theory briefly summarized earlier, it is unrealistic. Practical computer controlled systems do not exhibit linear system behavior and cannot be modeled as such. The current research methods provide some tools of reasoning about nonlinear control systems. These approaches are advantageous in providing automated decision procedures that show correctness of properties for a system. However, for some systems, these approaches cannot be used because state space explosion is encountered.

For the propositional temporal logic approach and the quantitative reasoning approach, Binary Decision Diagrams (BDD's) are used to represent boolean functions. However, BDD's do not encode multiplication very well;

the number of nodes required is exponential in the number of Boolean variables required to encode the multiplication (independent of the BDD variable ordering) [13]. In control systems, multiplication is used in the definition of systems. Even if a system is defined using a linear next state function, it is often the case that a state variable y has a next state equation of the form $y' = y + x$, where the variable x is not time varying, and y' is the value of y in the next state. Assuming y=0 at the initial state, after n cycles, the variable y would be encoded as xn, which requires multiplication to represent.

The approaches of timed automata and hybrid automata use nodes in a graph to represent the discrete states of the system. Hence, this graph would have to represent the discrete state space and arithmetic associated with the program's next state function. The explicit representation of the reachable discrete space as nodes in a graph is very inefficient, and can easily result in state explosion, since the number of discrete nodes may equal to the product of the size of the domain of each discrete program variable. A new verification method is required to allow for multiplication within the system model and allow for reasoning about correctness without incurring state space explosion.

The theorem proving approaches also have short comings. The work of Shankar formalizes a Real Time logic and is used to prove properties about timed models, however, these models do not include hybrid systems; further theory development is required for hybrid systems. In the work of Mader, Wupper and Bauer [50] concerning the batch plant, the verification group initially chose the theorem proving approach since they felt that model checking may result in a state explosion. However, theorem proving became an unmanageable task, requiring several proof obligations. The problem definition which the group attempted to formally model and verify consisted of several boolean valued variables, which resulted in many case splits in the proof process. The group points out that they were able to verify properties about the system by using a propositional temporal logic model checker named Spin, after performing some abstraction of the system model.

Application of nonstandard analysis to hybrid systems comprises a smaller body of research than that of the other areas presented. Research efforts in this area include development of new models of hybrid systems whereby the real line is discretized into infinitely many time steps, where each time step is of positive, infinitesimal duration [46, 62–64]. Some efforts go further by extending existing logics to formally reason about such models [64]. While these approaches provide models for describing hybrid systems, they do not address the modeling of differential equations with discontinuous vector fields. With the exception of the work on transfinite automata [58], these efforts also do not address physical system models which result in infinitely many switches in finite time. As with standard hybrid automata, the transfinite automata method represents the reachable discrete state space as nodes in a graph, which can result in state explosion.

Chapter 3

Background on Notation and Mathematical Models Used in this Work

In this chapter, we provide some background on the notation to be used in the remaining chapters as well as an introduction to mathematical models of physical systems and computer systems. We provide a brief introduction to nonstandard analysis and internal set theory. This theory shall form the basis of our work in reasoning about hybrid systems. We also provide a background about the ordinal numbers, which we will use later to establish decreasing measure functions in proving progress properties. Lastly, we present a brief introduction to some existing definitions of differential equation solutions, with attention to the case where the vector field is discontinuous.

3.1 Notation

We denote the set of integers as $\mathbb{Z}$, the set of non-negative integers (natural numbers) as $\mathbb{N}$, and the set of real numbers as $\mathbb{R}$. We also make use of the notation $\mathbb{R}_{\geq 0}$ to represent the set of non-negative real numbers, and the notation $\mathbb{R}_{>0}$ to represent the set of positive real numbers. We denote the floor of a real number x as $\lfloor x \rfloor$. For any real x, $\lfloor x \rfloor$ is an integer and satisfies the property $x - 1 < \lfloor x \rfloor \leq x$. The notation $\dot{V}(x)$ denotes the first derivative of the function V with respect to time.

We assume a vector takes on values in $\mathbb{R}^n$, where n is a positive integer. We represent a vector with its constituent elements as, $x = (x_1, x_2, \ldots, x_n)$, in tuple form, or as

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix},$$

in matrix form. The special symbol $\mathbf{0}$ denotes the constant vector all whose components are 0.

For a given vector $x \in \mathbb{R}^n$, we denote its norm as $\| x \|$. The norm function is the usual Euclidean norm where, for a vector x of n components, its norm is defined as:

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

The norm function always returns a non-negative value. For two vectors x and y in $\mathbb{R}^n$, the norm function satisfies the triangle inequality:

$$\|x + y\| \leq \|x\| + \|y\| .$$

The dot product of two vectors, x and y in $\mathbb{R}^n$, is represented as $x \bullet y$ and defined as follows:

$$x \bullet y = \sum_{i=1}^{n} x_i \, y_i.$$

We will use the logical operators $\wedge$, $\vee$, $\neg$. The operator $\wedge$ denotes "logical AND" or conjunction. The operator $\vee$ denotes "logical OR" or disjunction. The operator $\neg$ denotes negation. The formula $P \rightarrow Q$ is read "P implies Q" and is logically equal to $\neg P \vee Q$. The formula $\forall_x P(x)$ is read "for all x" the predicate P(x) holds. The formula $\exists_x P(x)$ is read "there exists an x" for which the predicate P(x) holds. In the event that the quantification is over sets, then the variable is upper case. For example, $\forall_X P(X)$ is read as "for all sets X" the predicate P(X) holds.

For sets U and V, their union is represented as $U \cup V$, their intersection as $U \cap V$, and the subtraction of V from U as $U \backslash V$.

For reals a and b, we represent intervals over the real line as follows:

$$\begin{array}{lll} [a,b] & \text{iff} & \{x \in \mathbb{R} : a \leq x \leq b\}, \\ (a,b] & \text{iff} & \{x \in \mathbb{R} : a < x \leq b\}, \\ [a,b) & \text{iff} & \{x \in \mathbb{R} : a \leq x < b\}, \text{ and} \\ (a,b) & \text{iff} & \{x \in \mathbb{R} : a < x < b\}, \end{array} \tag{3.1}$$

where a and b are defined such that the corresponding set is nonempty.

For integers a and b such that $a \leq b$, we represent intervals over the integers as follows:

$$[a..b] \quad \text{iff} \quad \{x \in \mathbb{Z} : a \leq x \leq b\}. \tag{3.2}$$

3.2 Model of the Physical System

We model a physical system of n variables as a system of n ordinary, autonomous, first order differential equations:

$$dx_i/dt = f_{x_i}(x_1, \ldots, x_n), \tag{3.3}$$

where x_i is a real valued system variable, and i is an integer ranging from 1 to n, for n a standard positive integer. For each x_i, we define a function f_{x_i} which is the derivative of x_i with respect to time. We denote the system state as the vector x, where the i^{th} component of x is x_i. We refer to the collection of functions f_{x_i} as the vector field and represent them as the vector valued function $f : \mathbb{R}^n \mapsto \mathbb{R}^n$, where the i^{th} component of $f(x)$ is $f_{x_i}(x)$.

A function f is said to be Lipschitz continuous if there exists a finite, real valued constant L, referred to as the Lipschitz constant, such that for any two points x and y in $\mathbb{R}^n$, the following condition holds:

$$\|f(x) - f(y)\| \leq L \; \|x - y\| \; . \tag{3.4}$$

Given the initial condition x_0, we denote the solution of the system of differential equations as a function $\phi : \mathbb{R} \mapsto \mathbb{R}^n$, where $\phi(t)$ returns the state at time $t \in \mathbb{R}_{\geq 0}$ and $\phi(0) = x_0$. Since the solution ϕ satisfies the differential equation, then its derivative exists with respect to time and is continuous with respect to time. Sometimes we will use the notation $\phi(x_0, t)$ to denote the solution at non-negative time t with the solution satisfying the initial condition $\phi(x_0, 0) = x_0$.

A system of differential equations has a unique solution in some domain D if the vector field is Lipschitz continuous over points in D. Under this Lipschitz continuity criteria, the unique solution is guaranteed to exist only if the initial state is in D and the system state remains in D up to time t. That is, the unique solution is guaranteed to exist only for time t where for all t', such that $0 \leq t' \leq t$, $\phi(t') \in D$. We should note that the Lipschitz continuity criteria is a sufficient, but not necessary, condition for existence and uniqueness of a solution. In practice, the solution function ϕ of a system of differential equations represents the behavior of the physical system. Specifically, it represents the values of the system variables given some time $t \geq 0$. Throughout this dissertation, the properties of the physical system are assumed to be those as represented by the mathematical model of the system [6].

3.3 Stability of a Physical System

For an autonomous system $\dot{x} = f(x)$, we say that a point q is a critical point if $f(q) = \mathbf{0}$. The point q is a stable critical if it is a critical point and, for all solutions of the system that start sufficiently close to q, stay close to q [9]. Specifically, for every $\alpha > 0$, there exists a $\delta > 0$ where at least one solution $\phi(t)$ satisfying

$$\parallel \phi(0) - q \parallel < \delta \tag{3.5}$$

exists and every such solution $\phi(t)$ also satisfies, for all $t \geq 0$,

$$\parallel \phi(t) - q \parallel < \alpha. \tag{3.6}$$

The point q is an asymptotically stable critical point if it is a stable critical point and, for $0 < \delta' < \delta$, if

$$\parallel \phi(0) - q \parallel < \delta' \tag{3.7}$$

exists then every such solution $\phi(t)$ also satisfies

$$\lim_{t \to \infty} \phi(t) = q. \tag{3.8}$$

3.4 Lyapunov Stability Method

Methods are available for reasoning about the solution of a system without determining the solution itself. Among these, the most notable in control systems is the Lyapunov stability method. Using this method, one generates a Lyapunov function $V : \mathbb{R}^n \mapsto \mathbb{R}$. If the Lyapunov function V has a continuous first partial derivative and satisfies the following properties:

$$\begin{aligned} V(x) = 0, \quad \text{for } x &= \mathbf{0} \\ V(x) > 0, \quad \text{for } x &\neq \mathbf{0} \end{aligned}$$

$$\begin{aligned} \dot{V}(x) = 0, \quad \text{for } x &= \mathbf{0} \\ \dot{V}(x) < 0, \quad \text{for } x &\neq \mathbf{0}, \end{aligned}$$

then the solution ϕ will asymptotically approach $\mathbf{0}$ as t increases [9]. We note that the function $\dot{V}$ may be derived by use of the chain rule for multivariable functions as follows:

$$\dot{V}(x) = \nabla V \bullet f(x).$$

That is, the derivate of the proposed Lyapunov function V with respect to time may be expressed as the vector dot product of the gradient of V (represented as ∇V) and the vector field f. The gradient of V is the vector of partial derivatives of V with respect to each system variable x_i:

$$\nabla V = \left(\frac{\partial V}{\partial x_1}, \ldots, \frac{\partial V}{\partial x_n} \right)^T .$$

The interesting result from this approach is that a characterization of the system solution is achieved, namely asymptotic approach to $\mathbf{0}$, without determining the solution itself. This approach is widely used in the research community, along with semi-definite programming, for the case where the system of differential equations is linear [10, 45]. These semi-definite programming techniques are successful in determining asymptotic stability of linear systems in thousands of variables. An area of active research is the use of non-convex programming techniques to reason about stability of non-linear systems.

Other variations based on Lyapunov functions are used to determine system instability, where the solution grows without bound as time increases [9].

3.5 Model of the Computer System

Theoretically, a computer system with a program is modeled as a Turing machine with a tape of instructions. The behavior of this Turing Machine is what we would like to reason about. Mathematically, the semantics of this Turing Machine may be modeled as a partial function whose domain and range are the Natural numbers: $k : \mathbb{N} \mapsto \mathbb{N}$. Specifically, given the initial state $\mathtt{x}$ of the Turing Machine, the function returns the state after the machine halts (or terminates). Since the function k is partial, it is not required that the machine halt for every input. If the machine does terminate for every input, then the corresponding function is said to be total.

In this dissertation, we will be concerned only with those computer programs whose mathematical models are total functions. Specifically, we model a computer program as a recursive function defined as follows:

$$\mathtt{run}(\mathtt{x},\mathtt{n}) = \begin{cases} \mathtt{x}, & \text{for } \mathtt{n} = 0 \\ \mathtt{run}(\mathtt{step}(\mathtt{x}), \mathtt{n} - \mathtt{1}), & \text{for } \mathtt{n} > 0, \end{cases}$$

where n is assumed to be a Natural number. The function `step` is assumed to be a total function which, given the current computer state `x`, returns the computer state after one step of execution.

In computer verification, two types of properties may be shown about a computer program: Safety and Progress [48, 52, 53]. A safety property is true for the initial state of the program, and remains true after executing each step. A sufficient method by which to show a safety property P holds for our model of a computer program is to show that if a state `x` satisfies P, then `step(x)` also satisfies P.

A progress property, informally, implies the computer program will eventually do something useful. Formally, a progress property states that some state `y` is reachable from the initial program state `x`. In our model, we may show a progress property Q holds by defining a measure function `m` which returns a Natural number and has the following properties:

$$\begin{array}{ll} \mathtt{m}(\mathtt{x}) = 0, & \text{when } \mathtt{Q(x)} \text{ is true} \\ \mathtt{m}(\mathtt{step}(\mathtt{x})) < \mathtt{m}(\mathtt{x}) & \text{otherwise,} \end{array}$$

where `x` is the initial computer state. This method may be generalized by allowing the measure function `m` to evaluate to an ordinal, up to ϵ_0, rather than a Natural number.

3.6 Ordinals

The ordinals can be considered as an extension of the natural numbers. We regard each element in the set of natural numbers as a finite ordinal. We add a new element denoted as ω to this set such that ω is larger than any natural number. We call ω a transfinite ordinal. We can add another transfinite ordinal denoted $\omega + 1$ such that $\omega < \omega + 1$, where $<$ is the well founded relation on the ordinals. We can continue this process of adding successively larger transfinite ordinals.

The ordinals are well founded. That is, they are totally ordered and there exists no infinitely decreasing sequence of ordinals. There are finite ordinals and transfinite ordinals. A transfinite ordinal is larger than any finite ordinal. The smallest ordinal is 0. The smallest transfinite ordinal is denoted as ω. We caution the reader that, although the ordinals extend the natural numbers, ordinal arithmetic should not be confused with arithmetic on natural numbers. For example, according to ordinal arithmetic, $1 + \omega = \omega$, but $\omega < \omega + 1$.

So far, we have extended the natural numbers with the transfinite ordinal ω and the larger transfinite ordinal $\omega + 1$. The set can be further extended with larger ordinals:

$$\omega + 2,\ \omega + 3,\ \omega + 4, \ldots, \omega + \omega, \tag{3.9}$$

shown in ascending order. The transfinite ordinal $\omega + \omega$ is denoted as $\omega 2$. We can continue to construct larger transfinite ordinals and add them to the set.

The following illustrates transfinite ordinals, in ascending order, that may be added:

$$\omega 2,\ \omega 3,\ \omega 4, \ldots,\ \omega^2,\ \omega^3,\ \omega^4, \ldots,\ \omega^\omega,\ \omega^{\omega^\omega},\ \omega^{\omega^{\omega^\omega}}, \ldots,\ \epsilon_0, \tag{3.10}$$

with ϵ_0 the largest ordinal shown. While the smallest ordinal is 0, there is no largest ordinal. We will only consider the ordinals up to ϵ_0. Further information regarding ordinals may be found in [51] or in a text on set theory [31].

We will make use of the fact, for non-zero finite ordinals p, q, non-zero ordinals α_2, β_2, and for ordinals α_1, β_1 such that $\alpha_1 < \omega^{\alpha_2}$ and $\beta_1 < \omega^{\beta_2}$, that:

$$(\omega^{\alpha_2} p + \alpha_1) < (\omega^{\beta_2} q + \beta_1) \ \text{ iff } \ \begin{cases} \alpha_2 < \beta_2, \text{ or} \\ \alpha_2 = \beta_2 \wedge p < q, \text{ or} \\ \alpha_2 = \beta_2 \wedge p = q \wedge \alpha_1 < \beta_1. \end{cases} \tag{3.11}$$

3.7 Formal Definition of a Limit

For a real valued function $f : \mathbb{R} \mapsto \mathbb{R}$, and $x, a, l \in \mathbb{R}$, a formula consisting of a limit may be defined as follows:

$$\lim_{x \to a} f(x) = l. \tag{3.12}$$

Weierstrass provided a formal definition for such a formula using first order logic:

$$\forall_{\epsilon > 0} \exists_{\delta > 0}\ |x - a| < \delta \rightarrow |f(x) - l| < \epsilon, \tag{3.13}$$

where ϵ and δ are real.

3.8 Nonstandard Analysis

Nonstandard analysis, developed by Robinson [61], is based on a formalism whereby mathematical objects may be regarded as standard or nonstandard. In particular, we are interested in the set of real numbers, $\mathbb{R}$. In an effort to axiomatize nonstandard analysis, Nelson developed Internal Set Theory [59]. Internal set theory contains all the axioms and definitions of set theory, as well as the added formal predicate *standard*, and three axioms: the transfer principle, the idealization principle, and the standardization principle. Internal set theory does not add any object to set theory, but only the predicate *standard*. Any formula which does not contain the predicate standard, is called *internal*. If a formula is not internal, then it is *external*. In particular, for the set of real numbers $\mathbb{R}$, we regard some of the elements in $\mathbb{R}$ as standard, and others as not standard, or nonstandard. Any object that we can uniquely define in mathematics, without the use of the predicate standard, is considered standard. Some examples of standard objects are the natural numbers, ordinals, π, e, $\sqrt{2}$, and the set of real numbers.

We are careful to point out that, while we use the term nonstandard to describe a function or mathematical object whose definition depends on the predicate standard, we use the term *external* to describe a formula or predicate that depends on the predicate standard.

With the aid of the predicate standard, we may define other predicates. For example, we define an infinitesimal as a real number whose absolute value is less than the absolute value of every non-zero standard number. There are infinitely many such infinitesimals in $\mathbb{R}$. Division by an infinitesimal results in a nonstandard number whose magnitude is larger than any standard number. We call such a number *large*. If a number is not large, then it is *limited*.

We should note an important distinction in notation between that of Robinson and Nelson. In the work by Robinson, the set denoted by the symbol $\mathbb{R}$ consists of only the standard real numbers, and the set denoted by the symbol $^*\mathbb{R}$ is an *enlargement* of $\mathbb{R}$, and consists of both the standard and nonstandard numbers. In some literature, the set $^*\mathbb{R}$ is referred to as the set of *hyperreals* [33].

In this dissertation, we will adopt the convention introduced by Nelson [59] where the set $\mathbb{R}$, although it consists of nonstandard elements, is the usual set of real numbers. The predicate standard simply allows one to distinguish some reals as standard, and others as nonstandard.

3.8.1 The Principles of Internal Set Theory

In addition to the axioms and definitions of set theory, internal set theory contains the formal predicate *standard*, and three axioms: the transfer principle, the idealization principle, and the standardization principle. Using these added principles and the predicate standard, one may formally reason about the standard and nonstandard mathematical objects.

3.8.1.1 Transfer Principle

We will use the notation by Nelson [59], where $\forall^{\text{st}}x$ is to be interpreted as "for all standard x", and $\exists^{\text{st}}x$ is to be interpreted as "there exists a standard x". For an internal predicate P, the transfer principle states:

$$\forall_t^{\text{st}}(\ \forall_x^{\text{st}} P(t,x) \text{ iff } \forall_x P(t,x)\ ).$$

Hence, for an internal predicate P, if we show P is true for all standard x and t, we may conclude that P holds for all x, standard and nonstandard. The variable t may be regarded as a parameter. As shown in [59], since $\neg\forall\neg P(x)$ iff $\exists P(x)$,

$$\forall_t^{\text{st}}(\ \exists_x^{\text{st}} P(t,x) \text{ iff } \exists_x P(t,x)\ ).$$

The above is referred to as the dual form of the transfer principle [59].

The dual form of the transfer principle is useful in showing that an object x is standard. We begin by constructing an internal predicate P which uniquely recognizes x; that is, $P(x)$ holds and $P(y) \rightarrow (y = x)$. By the dual form of the transfer principle, if P is true for some object, then it must be true for a standard object. But since P uniquely recognizes x, then x must be standard. Therefore, if one can produce an internal predicate P which uniquely recognizes an object x, then x is standard.

3.8.1.2 Idealization Principle

The second principle of internal set theory is idealization. We will adopt the notation that $\forall^{\text{finst}}$ is to be interpreted as "for all finite, standard sets."

For an internal predicate P, the idealization principle states:

$$\forall_X^{\text{finst}} \exists_y \forall_{x \in X} P(x,y) \text{ iff } \exists_y \forall_x^{\text{st}} P(x,y). \tag{3.14}$$

The idealization principle allows one to show the existence of nonstandard objects. Intuitively, the idealization principle states that we can "fix" only a finite number of objects at a time. If we assume that x takes on values from some standard set Y, 3.14 states that showing $\exists_y P(x,y)$ for all standard $x \in Y$ is the same as showing $\exists_y P(x,y)$ for all standard $x \in X$, for any standard finite set X such that $X \subseteq Y$.

For example, suppose we define $P(x,y)$ as $x \neq y$. Assume that x and y range over the set $\mathbb{R}$. For every finite standard set $S \subseteq \mathbb{R}$ of real numbers, there is a $y \in R$, where $\forall_{x \in S} x \neq y$. By the idealization principle, there must exist a y, not necessarily standard, where for every standard $x \in R$, $y \neq x$. Hence, y is not standard, since it is not equal to any of the standard elements in R.

Due to the idealization and transfer principles, the following theorem may be derived:

Theorem 3.8.1. *A standard set R is finite iff every element of R is standard.*

This theorem is proved in [59]. By this theorem, one may also conclude that any infinite standard set must consist of nonstandard elements. For example, the set $\mathbb{R}$ is a set that consists of infinite standard elements, since each such element may be uniquely defined without using the predicate standard. But since $\mathbb{R}$ is a standard set which consists of infinitely many standard elements, then, in internal set theory, it must also consist of nonstandard elements.

Theorem 3.8.2. *A non-empty standard set has at least one standard element.*

Proof. Let R be a non-empty standard set. The predicate $\exists x \in R$ is an internal predicate which is true. By the dual form of the transfer principle, there exists a standard x such that $x \in R$. Therefore, R has at least one standard element. $\square$

3.8.1.3 Standardization Principle

The third principle of internal set theory is *standardization*. The standardization principle states, for sets X and Y:

$$\forall^{\text{st}} X \exists^{\text{st}} Y \forall^{\text{st}} x (x \in Y \text{ iff } x \in X \wedge P(x)), \tag{3.15}$$

where the predicate P may be internal or external. Intuitively, this principle allows one to define a standard set Y, such that $Y \subseteq X$, by way of a nonstandard predicate P. Suppose there exists another standard set G which also satisfies the predicate P for every standard element in X. Then G and Y have the same standard elements. From internal mathematics, we know that two sets G and Y are equal iff they have the same elements:

$$\forall_{x \in (G \cup Y)}\ x \in G \text{ iff } x \in Y. \tag{3.16}$$

By the transfer principle, two standard sets G and Y are equal iff they have the same standard elements:

$$\forall^{\text{st}}_{x \in (G \cup Y)}\ x \in G \text{ iff } x \in Y. \tag{3.17}$$

Therefore, for a given set X and predicate P, the set Y is unique. We will use the standardization principle extensively throughout our presentation. In Nelson's notation, [59], the set Y is represented as ${}^{S}\{x \in X : P(x)\}$. This may be read as "The standard set Y whose every standard element is a standard element in X that satisfies P [59]." By applying the standardization principle, we may define some standard set Y. However, if this standard set Y is infinite, then by theorem 3.8.1, it must consist of nonstandard elements. In particular, if x is standard, then we know that $x \in Y$ iff $P(x)$; but if x is not standard, it may be that $P(x)$ does not hold, but $x \in Y$.

We should point out that if P is an internal predicate, then there is no need to apply the standardization principle to show that a standard set Y exists. We already know, by the subset axiom of set theory, that there exists $Y \subseteq X$, denoted $Y = \{x : x \in X \wedge P(x)\}$, such that $x \in Y$ iff $x \in X \wedge P(x)$. Since such a set Y is defined without the use of the predicate standard, then it is standard.

3.8.2 Nonstandard Analysis Definitions and Theorems

In table 3.8.2, we present some useful definitions and theorems based on nonstandard analysis which we will be using in this dissertation. In the formulas of the table, x and y are assumed to be real. We denote the predicate infinitesimal as *infsml*, and the predicate limited as *limtd*. In **DEF1**, the use of the predicate standard is explicit in the definition of infinitesimal. All the definitions directly, or indirectly, make use of the predicate standard and are therefore external. The items labeled **TH1** through **TH11** are theorems which can be derived based on the definitions **DEF1** through **DEF4**, the definition of the function st, the principles of internal set theory, and internal mathematics.

Table 3.1: Some Definitions and Theorems based on Nonstandard Analysis

DEF1	$\mathit{infsml}(x) \overset{\text{def.}}{=} \text{for all standard } y > 0,\ \lvert x\rvert \le y$
DEF2	$\mathit{large}(x) \overset{\text{def.}}{=} x \neq 0 \land \mathit{infsml}(1/x)$
DEF3	$\mathit{limtd}(x) \overset{\text{def.}}{=} \neg\, \mathit{large}(x)$
DEF4	$x \simeq y \overset{\text{def.}}{=} \mathit{infsml}(x-y)$
TH1	$\mathit{standard}(x) \rightarrow \mathit{st}(x) = x \ \land\ \mathit{limtd}(x)$
TH2	$\mathit{st}(-x) = -\mathit{st}(x)$
TH3	$\mathit{st}(x+y) = \mathit{st}(x) + \mathit{st}(y)$
TH4	$\mathit{limtd}(x) \land \mathit{limtd}(y) \rightarrow \mathit{st}(x\ y) = \mathit{st}(x)\ \mathit{st}(y)$
TH5	$\mathit{standard}(x) \land \mathit{standard}(y) \rightarrow \mathit{standard}(x+y) \ \land\ \mathit{standard}(x-y)$
TH6	$\mathit{standard}(x) \land \mathit{standard}(y) \rightarrow \mathit{standard}(xy) \land (y \neq 0 \rightarrow \mathit{standard}(x/y))$
TH7	$x \le y \rightarrow \mathit{st}(x) \le \mathit{st}(y)$
TH8	$\mathit{infsml}(x)$ iff $\mathit{st}(x) = 0$
TH9	$\mathit{st}(\mathit{st}(x)) = \mathit{st}(x)$
TH10	$x \in \mathbb{Z} \rightarrow (\mathit{limtd}(x) \text{ iff } \mathit{standard}(x))$
TH11	$\mathit{limtd}(x) \rightarrow \mathit{standard}(\mathit{st}(x))$

(3.18)

The function *st* denotes *standard part* and is one which we will use extensively. Intuitively, for a limited number $x \in \mathbb{R}$, this function returns that standard number which is infinitesimally close to x. We may define such a function as shown in [59].

3.8.3 Proof Strategy Using the Standardization and Transfer Principles

The standardization and transfer principles may be used in a proof strategy whereby, if some external predicate $P_{ext}(x)$ is shown to be true for standard x, then we may conclude that there exists an internal predicate $P_{int}(x)$ which is true for all x. Suppose we can show that an external predicate P_{ext} is true for all standard objects. It is tempting to use the transfer principle to conclude that P_{ext} is true for all nonstandard objects, but the transfer principle can only be applied to internal predicates. By standardization, we know that there exists a standard set Y whose standard elements satisfy P_{ext}. Since Y is a standard set, there exists an internal predicate P_{int} which is true only for elements in Y. Furthermore, the predicate P_{int} has the same truth value as P_{ext} for all standard objects. If we have shown that P_{ext} is true for all standard objects, then P_{int} is true for all standard objects; but the formula "P_{int} is true for all standard objects" is not internal, since it uses the predicate standard. We apply the transfer principle to conclude that P_{int} is true for all objects.

Alternatively, we may use such a proof strategy to model a nonstandard function F_{ns} with a standard function F_{std}, where F_{ns} and F_{std} have the same standard domain D and standard range Ω, and $F_{ns}(x) = F_{std}(x)$ for standard x.

Theorem 3.8.3. *For $F_{ns} : D \mapsto \Omega$ a nonstandard function, suppose we have shown that for every standard $x \in D$, $F_{ns}(x)$ exists and is standard. We may then conclude that there exists a unique standard function F_{std} which is a total mapping and for standard $x \in D$, $F_{ns}(x) = F_{std}(x)$.*

Proof. Let the predicate $P_{ext} \overset{\text{def.}}{=} y = F_{ns}(x)$. Let $X = D \times \Omega$. We let ν be a tuple (x, y), with $x \in D$ and $y \in \Omega$. A standard element $\nu = (x, y)$ is one where x is a standard element of D and y is a standard element of Ω. We may use the standardization principle to show that a standard set $Y \subseteq D \times \Omega$ exists:

$$Y = {}^{S}\{\nu : \nu \in X \wedge P_{ext}(\nu)\}. \tag{3.19}$$

Furthermore, since Y is standard and since for every standard $(x, y_1) \in Y$ and standard $(x, y_2) \in Y$ we have that $y_1 = F_{ns}(x) = y_2$, then by the transfer principle, for every $(x, y_1) \in Y$ and every $(x, y_2) \in Y$, $y_1 = y_2$. Since for every standard $x \in D$, there exists a y, namely $y = F_{ns}(x)$, where $(x, y) \in Y$, then, by transfer, for every $x \in D$, there exists a y where $(x, y) \in Y$. Therefore, for every $x \in D$, there is only one $y \in \Omega$ such that $(x, y) \in Y$. Since Y is standard, we can symbolically represent it as the standard mapping F_{std}. Since Y is unique for the given F_{ns}, F_{std} is unique for the given F_{ns}. $\square$

By theorem 3.8.3, it is shown that the standardization principle extends to showing the existence of standard functions. Therefore, throughout the dissertation, we will use the standardization principle to show the existence of standard functions.

3.8.4 Example Application of Nonstandard Analysis

To explore the use of nonstandard analysis, we present an example. Suppose we want to take the derivative of $f(x) = x^2$. We know that the derivative of x^2 is $2x$, a standard function. Let us derive this result using nonstandard analysis. By the definition of the derivative, we have:

$$g(x) = \frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{3.20}$$

In using nonstandard analysis, we may interpret $\epsilon \to 0$ as a non-zero infinitesimal, or $\epsilon \neq 0 \wedge st(\epsilon) = 0$. By the definition of f, and assuming non-zero infinitesimal ϵ, we have

$$\epsilon \neq 0 \wedge st(\epsilon) = 0 \;\rightarrow\; g = st\left(\frac{(x^2 + 2x\epsilon + \epsilon^2) - x^2}{\epsilon}\right), \tag{3.21}$$

where g is the value of the limit in 3.20, as ϵ approaches 0 from any direction. Using the fact that ϵ is infinitesimal, by **TH3**, and by algebra, we may reduce 3.21 to

$$g = st(2x). \tag{3.22}$$

However, we would like for the derivative to be a standard function, not a nonstandard one in terms of the predicate standard, which is implied by the function st in 3.22.

By theorem 3.8.3, if we can show that $st(2x)$ is standard for standard x, then there exists a standard function, say g, such that:

$$\forall_x^{\mathrm{st}} g(x) = st(2x). \tag{3.23}$$

Assuming x is standard, by theorems **TH1** and **TH4**, $st(2x) = 2x$. Since 2 is standard and x is standard then, by **TH6**, $2x$ is standard. This satisfies the requirement of theorem 3.8.3. Furthermore, we may rewrite 3.23:

$$\forall_x^{\mathrm{st}} g(x) = 2x. \tag{3.24}$$

By the transfer principle, we have:

$$\forall_x g(x) = 2x. \tag{3.25}$$

Hence, $g(x) = 2x$ may be regarded as the standard function representing the derivative of x^2.

To demonstrate the ease of use of nonstandard analysis, the reader should compare the formal method used in this nonstandard proof with that based on the formal definition of a limit defined in section 3.7.

3.8.5 Definition of a Continuous Function

Using Nonstandard Analysis, we make precise the definition of a continuous function.

Definition 3.8.1. For a standard real valued function $f : \mathbb{D} \mapsto \mathbb{R}$, where $\mathbb{D} \subseteq \mathbb{R}$, a standard real number $x_1 \in \mathbb{D}$, and any real number $x_2 \in \mathbb{D}$, f is continuous at x_1 iff $x_1 \simeq x_2 \rightarrow f(x_1) \simeq f(x_2)$.

For example, the function $f(x) = x + 1$ is standard. Suppose x_1 is standard, and $x_2 \simeq x_1$. We want to show $f(x_1) \simeq f(x_2)$. This requires we show, by **DEF4** and the definition of f,

$$\mathit{infsml}((x_1 + 1) - (x_2 + 1)),$$

or, by arithmetic, $\mathit{infsml}(x_1 - x_2)$. But this is equivalent to $x_1 \simeq x_2$, which is the hypothesis we started with. Hence, f is continuous.

Using some properties about limited real numbers, we will derive a theorem about continuity for standard functions. First, we need a lemma.

Lemma 3.8.4. *For $x, y \in \mathbb{R}$, $x \simeq y$ iff $st(x) = st(y)$.*

Proof. If $x \simeq y$, then, by **TH8**, $st(x - y) = 0$, by **TH3** and **TH2**, $st(x - y) = st(x) - st(y) = 0$, and so $st(x) = st(y)$. □

Theorem 3.8.5. *For a standard real valued function $f : \mathbb{R} \mapsto \mathbb{R}$, standard real number x_1, and any real number x_2, f is continuous at x_1 iff $x_1 = st(x_2) \rightarrow f(x_1) = st(f(x_2))$.*

Proof. We note that for a standard function f, if x is standard then $f(x)$ is standard. By **TH1**, if x is standard, then $st(f(x)) = f(x)$. Using this observation, and lemma 3.8.4, we may conclude $x_1 = st(x_2) \rightarrow f(x_1) = st(f(x_2))$. □

Theorem 3.8.6. *For a standard real valued function $f : \mathbb{R} \mapsto \mathbb{R}$, f is continuous at x iff $\mathit{limtd}(x) \rightarrow st(f(x)) = f(st(x))$.*

Proof. By **TH11**, the standard part of a limited real number is standard and is in $\mathbb{R}$. For a limited real x, and substituting $st(x)$ for x_1, and x for x_2 in theorem 3.8.5, we have $f(st(x)) = st(f(x))$. □

3.8.6 Definition of Open and Closed Sets

In general topology, sets of real numbers may be classified as open or closed. We will say an internal predicate P is open (closed) if the set $\{x \in \mathbb{R} \mid P(x)\}$ is open (closed).

Definition 3.8.2. For an internal predicate P, whose domain is the set of real numbers $\mathbb{R}$, and x is any real number:

$$\begin{aligned} P \text{ is open} &\overset{\text{def.}}{=} P(st(x)) \rightarrow P(x) \\ P \text{ is closed} &\overset{\text{def.}}{=} P(x) \rightarrow P(st(x)) \end{aligned} \tag{3.26}$$

An example of an open predicate is $x < 5$. We may formally show that it is open using the definition for open in 3.26.

$$\begin{array}{lll} & st(x) < 5 & \\ \text{iff} & st(x) < st(5) & \text{since 5 is standard, } 5 = st(5), \\ \rightarrow & x < 5 & \text{by the contrapositive of } \textbf{TH7}. \end{array}$$

We note that the predicates $true$ and $false$ are each open and closed. If an internal predicate is not identically $true$ or $false$, then it can be open or closed, but not both. It is also possible for an internal predicate to be neither open nor closed, for example: $4 < x \leq 5$.

3.8.7 Extending Definitions to Vectors

Much of the definitions regarding continuity and open and closed sets can be extended to $\mathbb{R}^n$, for a standard integer $n \geq 1$. For a vector x, we define $st(x)$ as the vector whose components are the standard-part of the corresponding components of the vector x:

$$st(x) = st\left(\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} st(x_1) \\ st(x_2) \\ \vdots \\ st(x_n) \end{bmatrix} \tag{3.27}$$

We say a vector x is limited if each of its components is limited.

Definition 3.8.3. For a function $f : \mathbb{R}^n \mapsto \mathbb{R}^n$, f is continuous at $x \in \mathbb{R}^n$ iff *limtd*$(x) \rightarrow st(f(x)) = f(st(x))$.

The definitions for open and closed predicates apply for the vector case, with the assumption that $x \in \mathbb{R}^n$.

3.9 Solutions to Differential Equations

The definition of a solution is critical in the analysis of differential equations. For ordinary differential equations, we seek a function x, where $x : I \mapsto \mathbb{R}^n$. The set I denotes an interval of the real line. We say that the function x is a solution to the differential equation if x is everywhere differentiable over the interval I and its derivative at every point in I satisfies the differential equation. This definition of a solution is referred to as the Newtonian solution. When dealing with differential equations with discontinuous vector fields, it is possible that no Newtonian solution exists for a particular initial condition. The presence of discontinuous vector fields is certainly applicable in the case of analyzing computer controlled systems.

The literature consists of several definitions for differential equation solutions. Many of these definitions *loosen* the requirements of a function required to be considered a solution. In this subsection, we will review some of the more popular definitions. We will discuss the definition of solution based on the semantics of a hybrid automata. We will also discuss definitions of a Carathéodory solution and Filippov solution.

3.9.1 Hybrid Automata

In hybrid automata, a system is defined with a finite graph. Each vertex in the graph is referred to as a *mode*. Associated with each *mode* is a differential equation (or inclusion) and a predicate referred to as the invariant. Associated with each arc leaving the mode is a predicate referred to as a guard and an assignment statement.

Each mode represents, effectively, the mode in which the physical system is governed by some differential equation. It is assumed that the differential equation has a Newtonian solution for the duration in which the value of the solution satisfies the invariant predicate; that is, the solution is continuous and everywhere differentiable with a derivative satisfying the differential equation, and a value satisfying the invariant predicate over some interval $I \subseteq \mathbb{R}$. If the mode consists of a differential inclusion, then the solution, again, is Newtonian, with the requirement that its derivative satisfy the differential inclusion. The guard predicate, informally, is the mechanism whereby switching may be modeled by the automata. If the solution evolves to a state where the guard predicate is satisfied (and the invariant is satisfied), then the system may *jump* out of this mode and enter a different mode. It is crucial to note that, for a particular run, the jump out of a mode is not mandatory as soon as the guard evaluates to true, but may occur at any time that the guard and invariant predicates hold, or may not occur at all. If the Newtonian solution exists only up to a finite time t_f, then a guard must evaluate to true at, and the invariant must hold up to some time $t \leq t_f$, otherwise the present run of the automata is finite and dismissed by the analysis. Similarly, a guard must evaluate to true before the invariant predicate becomes false, otherwise the present run of the automata is considered finite and dismissed. Figure 3.1 shows a diagram of a Hybrid Automata mode, with the key components in the definition of the automata pointed out.

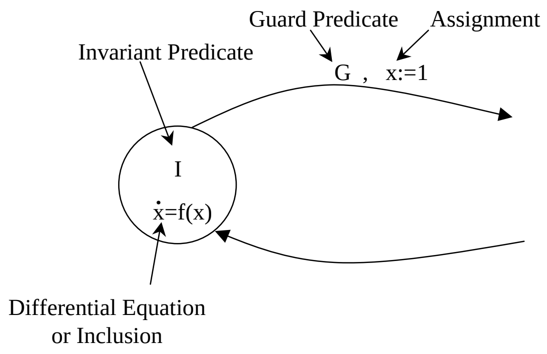

Figure 3.1: Hybrid Automata. Diagram demonstrates key components in the definition of a hybrid automata.

It is possible that two or more guard predicates, associated with the arcs exiting a mode, may evaluate to true. In this case, the system chooses, non-deterministically, a guard predicate and jumps to the corresponding mode. After a jump occurs, it is possible that the state variables may be (instantaneously) assigned arbitrary values, and a (possibly) new mode entered. The idea of an assignment on mode change is crucial in modeling such phenomena as impact, for rigid body dynamics, or digital system output change, for hybrid systems. Clearly, the assignments must be such that the state variables satisfy the invariant predicate of the mode being entered. When a mode is entered, the Newtonian solution associated with that mode evolves based on the new differential equation or inclusion of that mode, starting from an initial condition based on the values of the system variables after assignment. The process repeats as the system jumps from mode to mode. Therefore, the solution of a system defined by a hybrid automata may be regarded as a concatenation of one or more Newtonian solutions. The resulting solution may not be continuous. Since the time at which a jump is taken while a guard evaluates to true is arbitrary, and since a guard is chosen non-deterministically whenever two or more guards simultaneously hold, it is possible that infinitely many solutions exist for a particular hybrid automata. The multiplicity of solutions, however, is of no concern since the hybrid automata is not intended to be used for simulation, but for formal reasoning about reachability and safety properties of the modeled system.

Figure 3.2 shows an example hybrid automata. We note that the guard predicate may hold true at any time that the system variable x is above 4, at which point a jump may be taken, but not required. Figure 3.3 shows a possible solution of the example automata.

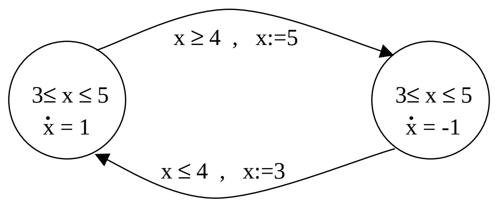

Figure 3.2: An Example Hybrid Automata

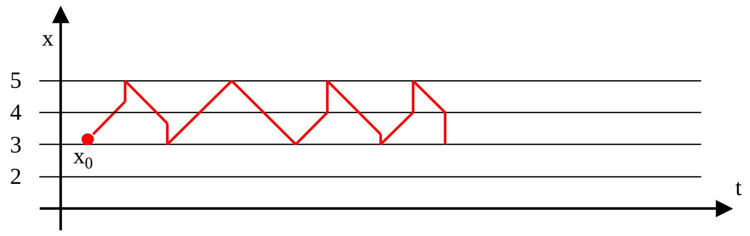

Figure 3.3: A Possible Solution for the Example Hybrid Automata

3.9.2 Carathéodory Solutions

While a Newtonian solution is useful for continuous vector fields, such a solution is strict for discontinuous vector fields. We consider, for example, the following differential equation:

$$\dot{x} = \begin{cases} -1 & , x > 0 \\ 0 & , x = 0 \\ 1 & , x < 0. \end{cases} \tag{3.28}$$

If we assume an initial condition of zero, the above differential equation has the solution $x(t) = 0$ for all $t \geq 0$. If we assume a positive initial condition x_0, the above equation has a Newtonian solution $x(t) = x_0 - t$, for t in the interval $[0, x_0)$. For $t \geq x_0$, the solution x does not satisfy the above equation. We propose a function x over the interval $[0, \infty)$,

$$x(t) = \begin{cases} x_0 - t & , t \in [0, x_0] \\ 0 & , t > x_0. \end{cases} \tag{3.29}$$

However, this function is not a Newtonian solution since it is not differentiable at $t = x_0$.

Unlike the Newtonian solution, a Carathéodory solution requires that the solution satisfy the differential equation at almost every point [27, 68]. Formally, a Carathéodory solution is an absolutely continuous function whose derivative satisfies the differential equation almost everywhere over the specified interval. A Carathéodory solution may also be characterized as follows:

$$x(t) = x(t_0) + \int_{t_0}^{t} f(x(s))ds, \tag{3.30}$$

where the integral is a Lebesgue integral [68]. Therefore, our proposed solution 3.29 for a positive initial condition is a Carathéodory solution of 3.28.

3.9.3 Filippov Solution

While the Carathéodory definition of solution admits some solutions to discontinuous vector fields, there are still some discontinuous vector fields for which no Carathéodory solution exists. Consider the vector field:

$$f(x) = \begin{cases} 3 & , x < 0 \\ 1 & , x = 0 \\ -1 & , x > 0. \end{cases} \tag{3.31}$$

For $x < 0$, the solution has the form $3t + k_1$. For $x > 0$, the solution has the form $-t + k_2$. Hence, in either case, $x(t)$ approaches zero as t increases. Upon approaching zero, the solution remains there, whereby $\dot{x} = 0$. However, at $x = 0$, the differential equation requires $\dot{x} = 1$, a contradiction. Therefore, for an interval $I \subseteq \mathbb{R}$ where $t \in I$ is such that $x(t)$ approaches zero, the differential equation 3.31 does not have a Carathéodory solution.

For a given solution x, suppose that the vector field is discontinuous at $x(t)$. Filippov [27] observes that if the vector field f approaches the discontinuity at $x(t^-)$ and $x(t^+)$, where t^- and t^+ approach t from below and above, respectively, then the solution is not necessarily defined at this discontinuity and may not exist using the classical Carathéodory notion of a solution.

Filippov observed that while a solution may be defined in the classical sense where the vector field is continuous over regions $G_i \subseteq \mathbb{R}^n$ for positive integer i, it is the discontinuity in the vector field at boundaries between regions G_i that resulted in the difficulty in defining a solution. When the vector field experiences a discontinuity along some boundary b, it is possible that the solution itself may take on values in this boundary b for some positive duration of time; that is, $x(t) \in b$ for $t \in [t_1, t_2]$ for reals $t_1 < t_2$. Since the vector field is discontinuous along b, the strict notion of a differential equation solution in the Newtonian or Carathéodory sense may not define a differential equation for which a solution exists in this interval $[t_1, t_2]$. Figure 3.4 shows an example two dimensional space divided into two regions, G_1 and G_2, with a boundary b separating the regions. We assume vector field functions are defined for each region and the boundary. The arrows in figure 3.4 represent the vector field directions in G_1 and G_2. If the solution should reach the boundary, it is not possible for it to move away, since the vector field from either side would *push* it back. This results in the solution residing along boundary points for some positive duration of time. Hence, a solution in the Newtonian or Carathéodory sense is difficult, since there may not exist a function whose derivative matches the vector field for points along the boundary as well as those points in regions G_1 and G_2 close to the boundary.

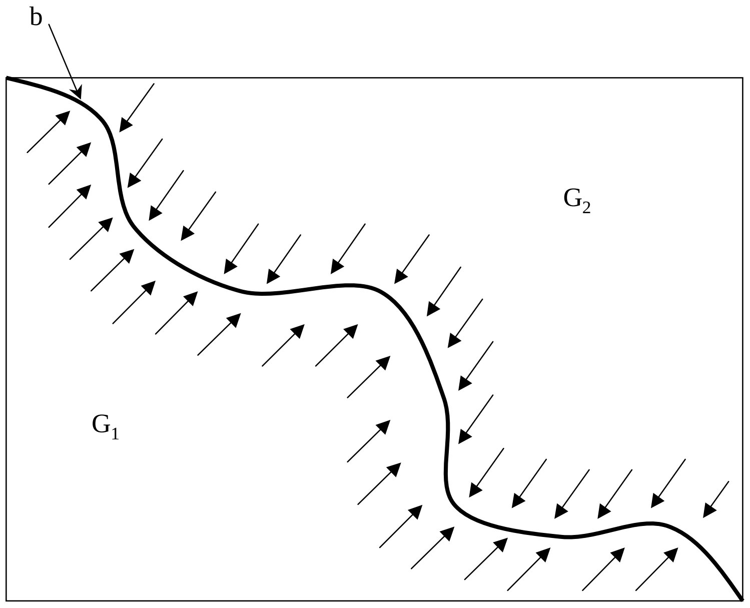

Figure 3.4: An Example Two Dimensional Space Divided Into Two Regions

Rather than using the classical notion of a differential equation, Filippov turned to differential inclusions at those boundaries at which the vector field is discontinuous.

Definition 3.9.1. For a vector field $f : \mathbb{R}^n \mapsto \mathbb{R}^n$ and an interval $[t_0, t_f]$, the Filippov solution $x : [t_0, t_f] \mapsto \mathbb{R}^n$ is an absolutely continuous function which satisfies the following:

$$\dot{x}(t) \in \mathcal{F}f(x(t)) \text{ , for almost all } t \in [t_0, t_f], \tag{3.32}$$

where $\mathcal{F}$ is defined as

$$\mathcal{F}f(x) = \bigcap_{\delta>0} \bigcap_{\mu(x+\delta\, B_1)\neq 0} \overline{co} f(x + \delta\, B_1), \tag{3.33}$$

where B_1 is the unit Ball in $\mathbb{R}^n$ centered at the origin, μ is a Lebesgue volume measure on $\mathbb{R}^n$, and $\overline{co}(Z)$ is the closed convex hull of $Z \subseteq \mathbb{R}^n$ [27, 68].

The operator $\mathcal{F}$ is a set valued function, formally a measure function. Informally, the operator $\mathcal{F}$ returns the smallest convex set which contains the derivative of the solution x when it is near the boundary at which f is discontinuous. For the example 3.31 presented earlier, the boundary over which the vector field is discontinuous is the point $x = 0$. The Filippov solution for this example must satisfy $\dot{x} \in \{3\}$ for $x < 0$, $\dot{x} \in \{-1\}$ for $x > 0$, and $\dot{x} \in [-1, 3]$ for $x = 0$. It can be verified that the following function is a Filippov solution:

$$x(t) = \begin{cases} 3t + k_1 & , x < 0 \\ -t + k_2 & , x > 0 \\ 0 & , x = 0. \end{cases} \tag{3.34}$$

In particular, we note that the proposed solution 3.34 is such that $\dot{x} = 0$ for almost all t where $x(t) = 0$, satisfying the Filippov differential inclusion requirement. It is interesting to note that the Filippov definition of solution reduces to the Newtonian definition for regions over which the vector field is continuous. In this case, the operator $\mathfrak{F}$ returns a singleton set. It is not until a discontinuity is encountered in f that the benefit of the Filippov solution is noted, wherein a non-singleton convex set characterizes the possible values of a solution's derivative at this discontinuity.

Chapter 4

Nonstandard Proof of the Existence and Uniqueness Theorem of Differential Equations

In this chapter, we will use nonstandard methods to prove the existence and uniqueness theorem of ordinary differential equations. The proof demonstrates how nonstandard analysis may be applied to reasoning about differential equations and introduces lemmas that will be used later in reasoning about a nonstandard definition of a solution to a hybrid system model.

Suppose we are given an initial value problem:

$$\dot{x} = f(x), \tag{4.1}$$

where $x(0) = x_0$. The existence and uniqueness theorem states that the solution for this initial value problem exists and is unique if the vector field f is Lipschitz continuous with Lipschitz constant L over some domain $D \subseteq \mathbb{R}^n$, for n a positive integer. It is assumed that L is a standard real constant and f is a standard function. We will focus only on the autonomous vector field case and assume that the domain $D = \mathbb{R}^n$. We note that these assumptions are not severe limitations in exploring existence and uniqueness. For the autonomous assumption, we can model dependence on t by adding a variable z with derivative 1 and initial condition 0. Although, we should point out, the theorem requires continuity of f with respect to t, not Lipschitz continuity. For an arbitrary domain $D \subseteq \mathbb{R}^n$, we require that the vector field is Lipschitz continuous over D, that the initial condition $x_0 \in D$, and that the solution x exists for time t, $0 \leq t < t_f$, where $t_f > 0$ is such that $\forall_{0 \leq t < t_f} x(t) \in D$.

The proof is divided into three parts: in the first, we show existence of a standard function ϕ, whose definition depends on a positive constant infinitesimal ϵ; in the second, we show that, for given x_0 and t, $\phi(x_0, t)$ is independent of which positive infinitesimal ϵ is chosen; in the third, we show that ϕ is a Newtonian solution of the differential equation with initial condition x_0 and is the only solution. Throughout the proof, we will assume that the solution is to be evaluated for non-negative time t.

4.1 Proof of Existence

We assume that the vector field is a standard function; that is, it is a function that is defined without the predicate standard. Given the vector field f, we propose a function ρ^ϵ based on the Euler approximation of the solution:

$$\rho^\epsilon(x,t) = \left\{ \begin{array}{cc} x & t < \epsilon \\ \rho^\epsilon(x + f(x)\epsilon, t - \epsilon) & t \geq \epsilon. \end{array} \right. \tag{4.2}$$

where $\epsilon > 0$.

As an aside on notation, the superscript ϵ of the function symbol ρ^ϵ designates that the function is dependent on ϵ in its definition. We may choose an alternate symbol, say δ substituted for ϵ in the definition, in which case we would denote the function as ρ^δ.

By the standardization principle, if we show that the nonstandard function $st(\rho^\epsilon(x_0, t))$ has a standard value for all standard arguments, then there exists a standard function ϕ which is equal to $st(\rho^\epsilon(x_0, t))$ for all standard arguments. Recall that for a limited real number r, $st(r)$ exists and is standard. Therefore, to show $st(\rho^\epsilon(x_0, t))$ has a standard value for all standard arguments we only need to show that $\rho^\epsilon(x_0, t)$ is limited for standard x_0 and standard t. Before proceeding, we state a lemma which we need in our proof.

Lemma 4.1.1. *For a standard vector field f which is Lipschitz continuous with constant L over some non-empty standard set $D \subseteq \mathbb{R}^n$, $f(x)$ is limited for any limited $x \in D$.*

Proof. Since D is a non-empty standard set, it consists of at least one standard real point z. Since f is standard and defined over all of D, then $f(z)$ is standard, hence limited. By the Lipschitz property of f, for any $x \in D$, $\| f(x) - f(z) \| \leq \; L \; \| x - z \|$. Therefore, since x and z are limited, then $\|x - z\|$ is limited, implying $\| f(x) - f(z)\|$ is limited. Since $f(z)$ is limited, then $f(x)$ is limited. □

We now point out some useful properties about ρ^ϵ. The value at time $t \geq \epsilon$ of the Euler approximation ρ^ϵ may be related with that of the previous time step by the following equation:

$$\rho^\epsilon(x_0, t) = \rho^\epsilon(x_0, t - \epsilon) + f(\rho^\epsilon(x_0, t - \epsilon))\epsilon. \tag{4.3}$$

The Euler approximation ρ^ϵ may be described using the following summation:

$$\rho^\epsilon(x_0, t) = x_0 + \sum_{i=1}^{\lfloor t/\epsilon \rfloor} f(\rho^\epsilon(x_0, t - \epsilon i))\epsilon. \tag{4.4}$$

Both the above properties may be proved by induction. For a given non-negative t, positive constant ϵ, $x_0 \in \mathbb{R}^n$, and function ρ^ϵ, the induction is performed using a well founded structure $\langle S, \prec \rangle$:

$$S = \{t - \epsilon i : i \in \mathbb{N} \wedge 0 \leq i \leq \lfloor t/\epsilon \rfloor\}, \tag{4.5}$$

where, for $t_1, t_2 \in S$, $t_1 \prec t_2$ iff $t_1 < t_2$. Using these properties, we proceed to find an upper bound for $\|f(\rho^\epsilon(x_0, t))\|$. We begin by noting:

$$\|f(\rho^\epsilon(x_0, t))\| = \|f(\rho^\epsilon(x_0, t)) - f(\rho^\epsilon(x_0, t - \epsilon)) + f(\rho^\epsilon(x_0, t - \epsilon))\| . \tag{4.6}$$

By the triangle inequality,

$$\|f(\rho^\epsilon(x_0, t))\| \leq \|f(\rho^\epsilon(x_0, t)) - f(\rho^\epsilon(x_0, t - \epsilon))\| + \|f(\rho^\epsilon(x_0, t - \epsilon))\| . \tag{4.7}$$

By the Lipschitz property of f,

$$\|f(\rho^\epsilon(x_0, t))\| \leq \ L \|\rho^\epsilon(x_0, t)) - \rho^\epsilon(x_0, t - \epsilon)\| + \|f(\rho^\epsilon(x_0, t - \epsilon))\| . \tag{4.8}$$

By the property 4.3,

$$\|f(\rho^\epsilon(x_0, t))\| \leq \ L\epsilon \|f(\rho^\epsilon(x_0, t - \epsilon))\| + \|f(\rho^\epsilon(x_0, t - \epsilon))\| . \tag{4.9}$$

By algebra,

$$\|f(\rho^\epsilon(x_0, t))\| \leq \ (1 + L\epsilon) \|f(\rho^\epsilon(x_0, t - \epsilon))\| . \tag{4.10}$$

Using the summation 4.4 and repeatedly applying the triangle inequality to the summation, we have:

$$\|\rho^{\epsilon}(x_0,t)\| \leq \|x_0\| + \sum_{i=1}^{\lfloor t/\epsilon \rfloor} \|f(\rho^{\epsilon}(x_0,t-\epsilon i))\| \, \epsilon. \tag{4.11}$$

By repeatedly applying the inequality from 4.10, we note that, for non-negative integer $i \leq \lfloor t/\epsilon \rfloor$:

$$\|f(\rho^{\epsilon}(x_0,t-\epsilon i))\| \leq \ (1+\epsilon L)^{(\lfloor t/\epsilon \rfloor - i)} \ \|f(x_0)\| \ . \tag{4.12}$$

This allows us to conclude the following from 4.11:

$$\|\rho^{\epsilon}(x_0,t)\| \leq \|x_0\| + \sum_{i=1}^{\lfloor t/\epsilon \rfloor} (1+\epsilon L)^{(\lfloor t/\epsilon \rfloor - i)} \ \|f(x_0)\| \, \epsilon. \tag{4.13}$$

For positive ϵL, we use the property $(1+\epsilon L)^{i} \leq e^{i\epsilon L}$, which can be verified by the Taylor expansion of $e^{\epsilon L}$. We apply this property to 4.13 to derive the following:

$$\|\rho^{\epsilon}(x_0,t)\| \leq \|x_0\| + \sum_{i=1}^{\lfloor t/\epsilon \rfloor} e^{(\lfloor t/\epsilon \rfloor - i)\epsilon L} \ \|f(x_0)\| \, \epsilon. \tag{4.14}$$

Since, for non-negative i, $e^{(\lfloor t/\epsilon \rfloor - i)\epsilon L} \leq e^{\lfloor t/\epsilon \rfloor \epsilon L}$,

$$\|\rho^{\epsilon}(x_0,t)\| \leq \|x_0\| + \sum_{i=1}^{\lfloor t/\epsilon \rfloor} e^{\lfloor t/\epsilon \rfloor \epsilon L} \ \|f(x_0)\| \, \epsilon. \tag{4.15}$$

By algebra,

$$\|\rho^{\epsilon}(x_0,t)\| \leq \|x_0\| + \|f(x_0)\| \ \ \lfloor t/\epsilon \rfloor \, \epsilon \ e^{\lfloor t/\epsilon \rfloor \epsilon L}. \tag{4.16}$$

Since, for positive ϵ and t, $\lfloor t/\epsilon \rfloor \, \epsilon \leq t$,

$$\|\rho^{\epsilon}(x_0,t)\| \leq \|x_0\| + \|f(x_0)\| \ t \ e^{tL}. \tag{4.17}$$

We should note that the property 4.17 holds for arbitrary $\epsilon > 0$, not necessarily infinitesimal. Hence, by the transfer principle, the property holds for infinitesimal ϵ as well. We note that the right hand side of 4.17, by lemma 4.1.1, is limited for limited t and x_0. Therefore, $\rho^{\epsilon}(x_0, t)$ is limited for limited t and x_0. By the standardization principle, this allows us to define a standard function $\phi(x_0, t)$ whose value is equal to $st(\rho^{\epsilon}(x_0, t))$ for standard x_0 and standard t, where we choose ϵ to be some arbitrary positive infinitesimal constant.

At this point, we have shown that this function ϕ exists, we will show later that this function is indeed the Newtonian solution.

4.2 Proof of Independence of ϕ from ϵ

We note that, for a given x_0 and t, the value of ρ^{ϵ} varies with ϵ. We wish to show that $\phi(x_0, t) \overset{std}{=} st(\rho^{\epsilon}(x_0, t))$ does not depend on the particular positive ϵ chosen; formally, we want to show:

$$st(\rho^{\epsilon_1}(x_0, t)) \;=\; st(\rho^{\epsilon_2}(x_0, t)), \tag{4.18}$$

for positive infinitesimals ϵ_1 and ϵ_2. Before proving the above property, we first prove some properties about ϕ.

Lemma 4.2.1. *For non-negative reals t_1, t_2, with $t_2 \geq t_1$, and $x_0 \in \mathbb{R}^n$, the function ϕ satisfies the following property:*

$$\|\phi(x_0, t_1) - \phi(x_0, t_2)\| \leq \;(t_2 - t_1)\; e^{t_2 L} \;\; \|f(x_0)\|\;. \tag{4.19}$$

Proof. By use of the summation 4.4 and definition of ρ^ϵ, one may show:

$$\rho^\epsilon(x_0,t_2) - \rho^\epsilon(x_0,t_1) = \sum_{i=1}^{\lfloor t_2/\epsilon \rfloor - \lfloor t_1/\epsilon \rfloor} f(\rho^\epsilon(x_0, \epsilon \lfloor t_1/\epsilon \rfloor + \epsilon i - 1))\epsilon, \tag{4.20}$$

for $t_2 \geq t_1 \geq 0$. By use of 4.20, one may prove:

$$\|\rho^\epsilon(x_0,t_2) - \rho^\epsilon(x_0,t_1)\| \leq \ \epsilon(\lfloor t_2/\epsilon \rfloor - \lfloor t_1/\epsilon \rfloor)\ e^{t_2 L}\ \ \|f(x_0)\|, \tag{4.21}$$

for $t_2 \geq t_1 \geq 0$. By **TH7** which states:

$$x \leq y \rightarrow st(x) \leq st(y), \tag{4.22}$$

we may apply standard part to both sides of 4.21:

$$st(\|\rho^\epsilon(x_0,t_1) - \rho^\epsilon(x_0,t_2)\|)\ \leq\ st(\epsilon(\lfloor t_2/\epsilon \rfloor - \lfloor t_1/\epsilon \rfloor)\ e^{t_2 L}\ \ \|f(x_0)\|). \tag{4.23}$$

By properties of norm and standard part, we have:

$$\|st(\rho^\epsilon(x_0,t_1)) - st(\rho^\epsilon(x_0,t_2))\| \leq\ \ st(\epsilon(\lfloor t_2/\epsilon \rfloor - \lfloor t_1/\epsilon \rfloor)\ e^{t_2 L}\ \ \|f(x_0)\|). \tag{4.24}$$

The formula 4.24 holds for its variables taking on both standard and nonstandard values. We will assume, for now, that the variables t_1, t_2, and x_0 in 4.24 are standard and ϵ is infinitesimal. By properties of standard part and floor and 4.24, we have the following:

$$\|st(\rho^\epsilon(x_0,t_1)) - st(\rho^\epsilon(x_0,t_2))\| \leq\ (t_2 - t_1)\ e^{t_2 L}\ \ \|f(x_0)\|\ . \tag{4.25}$$

By the standardization principle, where for standard x_0 and t, we have $\phi(x_0,t) = st(\rho^\epsilon(x_0,t))$, we may conclude from 4.25:

$$\|\phi(x_0,t_1) - \phi(x_0,t_2)\| \leq\ (t_2 - t_1)\ e^{t_2 L}\ \ \|f(x_0)\|\ . \tag{4.26}$$

We have been assuming that t_1, t_2, and x_0 are standard. Since 4.26 is an internal formula, by the transfer principle, it may be concluded that it holds for all real t_1, t_2, and $x_0 \in \mathbb{R}^n$, such that $t_2 \geq t_1$. □

Lemma 4.2.2. *For non-negative real t and $x_0 \in \mathbb{R}^n$, $\phi(x_0, t)$ is continuous with respect to t and x_0.*

Proof. For showing continuity with respect to t, we make use of lemma 4.2.1. We note that as $t_2 - t_1$ becomes infinitesimal, the right hand side of the inequality 4.19 becomes infinitesimal for limited x_0 and standard positive constant L. Hence, by applying standard part to both sides of 4.19, for standard x_0 and t_1, $t_2 \geq t_1$, and $t_2 \simeq t_1$:

$$st(\|\phi(x_0, t_1) - \phi(x_0, t_2)\|) \ \leq \ st((t_2 - t_1)\ e^{t_2 L}\ \ \|f(x_0)\|) = 0, \tag{4.27}$$

since t_2 is limited and x_0 is limited. Therefore, $st(\phi(x_0, t_1)) = st(\phi(x_0, t_2))$, implying that ϕ is continuous with respect to t. A similar proof may be carried out for the case $t_1 \geq t_2$.

For showing continuity with respect to x_0, the following inequality may be used:

$$\|\rho^\epsilon(x_1, t) - \rho^\epsilon(x_2, t)\| \leq \|x_1 - x_2\|\ \ e^{tL}. \tag{4.28}$$

The property 4.28 may be proved by induction using the well founded structure 4.5. A proof similar to that for lemma 4.2.1 may be used to show:

$$\|\phi(x_1, t) - \phi(x_2, t)\| \leq \|x_1 - x_2\|\ \ e^{tL}. \tag{4.29}$$

We note that as $\|x_2 - x_1\|$ becomes infinitesimal, the right hand side of the inequality 4.28 becomes infinitesimal for limited t and standard positive constant L. In a manner similar to that for showing continuity with respect to t, the inequality 4.29 may be used to show ϕ is continuous with respect to x_0. $\square$

Lemma 4.2.3. *For non-negative reals t_1, t_2, and $x_0 \in \mathbb{R}^n$, the function ϕ has the property:*

$$\phi(x_0, t_1 + t_2) = \phi(\phi(x_0, t_1), t_2). \tag{4.30}$$

The above property is referred to as time invariance, since the system solution depends only on the value of the initial condition and not the particular time at which it occurs (a characteristic of autonomous systems [9]).

Proof. To show time invariance of ϕ, we use a property of ρ^ϵ:

$$\rho^\epsilon(x_0, t_1 + t_2) = \rho^\epsilon(\rho^\epsilon(x_0, t_1), t_2), \tag{4.31}$$

for non-negative reals t_1, t_2, and where t_2 is an integer multiple of ϵ. This can be proved by induction, based on a well founded structure similar to 4.5. Using the properties of ρ^ϵ from 4.21 and 4.28, we may then show:

$$st(\rho^\epsilon(x_0, t_1 + t_2)) = st(\rho^\epsilon(st(\rho^\epsilon(x_0, t_1)), t_2)). \tag{4.32}$$

Since $\phi(x_0, t) = st(\rho^\epsilon(x_0, t))$ for standard x_0 and t, by the standardization and transfer principles, we have:

$$\phi(x_0, t_1 + t_2) = \phi(\phi(x_0, t_1), t_2). \tag{4.33}$$

$\square$

Using the time invariance property, we may define $\hat{\phi}$ as:

$$\hat{\phi}(x_0, t) = \begin{cases} x_0 & t < \delta \\ \hat{\phi}(\phi(x_0, \delta), t - \delta) & t \geq \delta. \end{cases} \tag{4.34}$$

where δ is a positive real number. The function $\hat{\phi}(x_0, t)$ takes advantage of time invariance by composing ϕ onto itself for $\lfloor t/\delta \rfloor$ steps, where at each step, ϕ is evaluated for a duration δ. Hence, $\hat{\phi}(x_0, m\ \delta) = \phi(x_0, m\ \delta)$, for non-negative integer m. By use of induction, it can be shown that:

$$\|\rho^{\delta}(x_0, m\delta) - \hat{\phi}(x_0, m\delta)\| \leq \delta^2 mL\ \ \|f(x_0)\|\ \ e^{m\delta L}. \tag{4.35}$$

If we let $m = \lfloor t/\delta \rfloor$, we note that if t is limited, then $m\delta = \lfloor t/\delta \rfloor \delta \leq t$ is limited. Therefore, for limited x_0, limited t, and infinitesimal δ, the right hand side of 4.35 is infinitesimal. From the above, it can be shown that:

$$st(\rho^{\delta}(x_0, \lfloor t/\delta \rfloor \delta)) = st(\hat{\phi}(x_0, \lfloor t/\delta \rfloor \delta)) = st(\phi(x_0, t)), \tag{4.36}$$

for any arbitrary positive infinitesimal δ. It can be shown, for arbitrary positive real δ, that:

$$\rho^{\delta}(x_0, \lfloor t/\delta \rfloor \delta) = \rho^{\delta}(x_0, t). \tag{4.37}$$

By applying standard part to both sides of the equality:

$$st(\rho^{\delta}(x_0, \lfloor t/\delta \rfloor \delta)) = st(\rho^{\delta}(x_0, t)). \tag{4.38}$$

By 4.36 and 4.38, we conclude:

$$st(\rho^{\delta}(x_0, t)) = st(\phi(x_0, t)), \tag{4.39}$$

for any arbitrary positive infinitesimal δ. Using 4.39 and substituting ϵ_1 for δ in one instance, and ϵ_2 for δ in another, we can show:

$$st(\rho^{\epsilon_1}(x_0, t)) = st(\phi(x_0, t)) = st(\rho^{\epsilon_2}(x_0, t)), \tag{4.40}$$

which is what we set out to prove.

We should note that ρ^{ϵ} is defined such that the same value for ϵ is used in each iteration. It is possible to define ρ such that for the i^{th} iteration ϵ_i is chosen as the time step, where $\sum \epsilon_i = t$ for a solution evaluated at t. However, our reasoning is about the bound on the solution which is based on t and the maximum value for $\|f(x)\|$ for the given initial condition and time. Consequently, the bound function would remain in a form similar to what has been demonstrated, with the exception that intermediate steps in our proof would replace the floor function by some integer valued function $g(h(t))$ representing the number of iterations, where $h(t)$ represents the sequence of values to be used for ϵ in each iteration, such that $\sum_{i=1}^{g(h(t))} \epsilon_i = t$. Furthermore, our reasoning about the properties of the solution has been based on its bound, as shown in the continuity, existence, and independence of ϵ proofs. Therefore, as the results would not differ, we assume the same value for ϵ in each iteration of ρ^{ϵ}.

4.3 The Proposed Function ϕ is A Newtonian Solution

At this point, we have shown that ϕ exists and that it is independent of the positive constant ϵ chosen. However, we have not shown that ϕ is in fact a solution. We will suppose an alternative function, say ϕ_2, is a Newtonian solution. For ϕ_2 to be Newtonian solution, its derivative must exist and satisfy $d\phi_2(t)/dt = f(\phi_2(t))$. Using nonstandard notation, we may state this as:

$$st\left(\frac{\phi_2(t+\epsilon)-\phi_2(t)}{\epsilon}\right) = f(\phi_2(t)), \tag{4.41}$$

where t is assumed to be a standard real number, and ϵ is some non-zero infinitesimal. Since ϕ_2 is differentiable, by a vector mean value theorem [28, 65], we have:

$$\phi_2(t+\delta) = \phi_2(t) + f(\phi_2(\zeta))\delta, \tag{4.42}$$

where $t \leq \zeta \leq (t+\delta)$. By rewriting, we have:

$$\frac{\phi_2(t+\delta)-\phi_2(t)}{\delta} = f(\phi_2(\zeta)). \tag{4.43}$$

If we let δ be a non-zero infinitesimal then, using the Lipschitz continuity of f and applying standard part to both sides, we may conclude from 4.43:

$$st\left(\frac{\phi_2(t+\delta)-\phi_2(t)}{\delta}\right) = st\left(f(\phi_2(t))\right), \tag{4.44}$$

for any limited real t, not necessarily standard.

For the remainder of this proof, we will make use of a residual function, R, defined as follows:

$$R(t,\epsilon) = \phi_2(t+\epsilon) - \phi_2(t) - f(\phi_2(t))\epsilon. \tag{4.45}$$

The residual may be regarded as the error in approximating $\phi_2(t+\epsilon)$ by $\phi_2(t) + f(\phi_2(t))\epsilon$. For a given non-negative integer m, we define the maximum residual, denoted $R_{max}(m, \epsilon)$, as the maximum of $\|R(i\epsilon, \epsilon)\|$, for i taking on integer values in $[0, m]$. By the definition of R and R_{max}, we have:

$$\begin{aligned} & \|x + f(x)\epsilon - \phi_2(m\epsilon)\| \\ = \; & \|x + f(x)\epsilon - \phi_2(m\epsilon) + \phi_2(m\epsilon - \epsilon) \\ & \quad -\phi_2(m\epsilon - \epsilon) + f(\phi_2(m\epsilon - \epsilon))\epsilon - f(\phi_2(m\epsilon - \epsilon))\epsilon\| \\ = \; & \|x - \phi_2(m\epsilon - \epsilon) \; + \; f(x)\epsilon - f(\phi_2(m\epsilon - \epsilon))\epsilon - R(m\epsilon - \epsilon, \epsilon)\| \\ \leq \; & (1 + \epsilon L) \; \|x - \phi_2(m\epsilon - \epsilon)\| + R_{max}(m, \epsilon). \end{aligned} \tag{4.46}$$

Using this property and induction over m, it can be shown that:

$$\|\rho^\epsilon(\phi_2(0), m\epsilon) - \phi_2(m\epsilon)\| \leq \; mR_{max}(m, \epsilon) \; e^{m\epsilon}. \tag{4.47}$$

By the definition of the residual function and 4.44, we have:

$$\begin{aligned} & st(R(m\epsilon, \epsilon)/\epsilon) \\ = \; & st\left(\frac{\phi_2(m\epsilon + \epsilon) - \phi_2(m\epsilon) - f(\phi_2(m\epsilon))\epsilon}{\epsilon}\right) \\ = \; & st\left(\frac{\phi_2(m\epsilon + \epsilon) - \phi_2(m\epsilon)}{\epsilon}\right) - st\left(f(\phi_2(m\epsilon))\right) \\ = \; & 0. \end{aligned} \tag{4.48}$$

The above property may be used to show that $st(R_{max}(m, \epsilon)/\epsilon) = 0$. Multiplying the right hand side of 4.47 by ϵ/ϵ and taking standard part of both sides, and assuming $m\epsilon$ is limited, one can derive:

$$st(\|\rho^\epsilon(\phi_2(0), m\epsilon) - \phi_2(m\epsilon)\|) \; \leq \; st\left(m\epsilon \; \frac{R_{max}(m, \epsilon)}{\epsilon} \; e^{m\epsilon}\right) = 0. \tag{4.49}$$

By 4.49, the continuity of ρ^ϵ with respect to t, the continuity of ϕ_2 with respect to t, assuming t limited, letting $m = \lfloor t/\epsilon \rfloor$, noting that $\lfloor t/\epsilon \rfloor \, \epsilon$ is limited for t limited, we have:

$$st(\rho^\epsilon(\phi_2(0), t)) = st(\phi_2(t)). \tag{4.50}$$

We recall, by the standardization principle, $\phi(x_0, t)$ has the same value as $st(\rho^\epsilon(x_0, t))$, for standard t and x_0. Therefore, by 4.50, the standardization and transfer principles, we may conclude:

$$\phi(\phi_2(0), t) = \phi_2(t). \tag{4.51}$$

Since ϕ_2 is a solution, then by 4.51, ϕ is a solution for the vector field f, with initial condition $x_0 = \phi_2(0)$. Since we have shown any solution ϕ_2 satisfies 4.51, and that $\phi(x_0, t)$ is independent of the choice of positive infinitesimal ϵ, then the solution ϕ is unique for a particular initial condition.

We should point out that, throughout the proof, we assumed positive infinitesimal ϵ and positive time t. A similar proof may be carried out for the negative ϵ and negative t case.

4.4 Mechanical Proof Using ACL2r

We have proved the theorems and lemmas presented in this chapter using the mechanical theorem prover ACL2r [1, 29] which is an extension of the theorem prover ACL2 [43, 56, 57]. The theorem prover ACL2r is based on internal set theory [59]. In our mechanical proofs, we have assumed a real, scalar value for the physical system variable x.

For brevity, this chapter has defined ρ^ϵ explicitly in terms of t. In defining ρ^ϵ in ACL2r, however, we use a positive integer n. We then substitute $\lfloor t/\epsilon \rfloor$ for n in the proof process to show theorems in terms of t.

The definition of ρ^ϵ as defined in this chapter is shown, followed by its definition in ACL2r.

$$\rho^\epsilon(x_0, t) = \begin{cases} x_0 & t < \epsilon \\ \rho^\epsilon(x_0 + f(x_0)\epsilon, t - \epsilon) & t \geq \epsilon. \end{cases}$$

```
(defun step1 (x eps)
  (+ x (* (f x) eps)))


(defun run (x n eps)
  (cond
   ((zp n) x)
   (t (run (step1 x eps) (- n 1) eps))))
```

The following is a theorem in ACL2r stating that, for standard x, the standard part of the function `run` is standard.

```
(defthm run-standard-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n)
    (i-limited (* eps n))
    (standard-numberp x))
   (standard-numberp (standard-part (run x n eps)))))
```

The following is the ACL2r theorem stating that the function `run` is bounded.

```
(defun run-n-limit (x n eps)
  (+ (abs x)
     (* (eexp (* (L) n eps)) (abs (f x))  n eps)))

(defthm run-limit-eexpt-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (realp x)
    (integerp n)
    (<= 0 n))
   (<= (abs (run x n eps))
       (run-n-limit x n eps))))
```

The following is the application of the standardization principle whereby we show there exists a standard function, `phi`, which is equal to the standard part of `run`, for standard x. The value of ϵ is replaced by the value of an infinitesimal constant represented by `(/ (i-large-integer))`, the reciprocal of a positive integer constant which is not limited. The variable `tm` represents time.

```
(defun-std phi (x tm)
  (cond
   ((not (and
          (realp x)
          (realp tm))) 0)
   (t (standard-part (run x
                          (floor1 (* tm (i-large-integer)))
                          (/ (i-large-integer)))))))
```

The following is a theorem in ACL2r stating that the function ρ^{ϵ}, as represented by `run`, is time invariant.

```
(defthm run-plus-thm
  (implies
   (and
    (integerp m)
    (integerp n)
    (<= 0 m)
    (<= 0 n))
   (equal (run (run x n eps) m eps)
          (run x (+ m n) eps))))
```

The following is a theorem in ACL2r stating that the proposed solution, ϕ, is time invariant.

```
(defthm-std phi-plus-thm
  (implies
   (and
    (realp x)
    (realp tm1)
    (realp tm2)
    (<= 0 tm1)
    (<= 0 tm2))
   (equal (phi (phi x tm1) tm2)
          (phi x (+ tm1 tm2)))))
```

The following is a theorem in ACL2r stating that the standard part of the function `run` is independent of the value of positive `eps` used.

```
(defthm run-any-small-eps-thm
  (implies
   (and
    (realp x)
    (realp tm)
    (standard-numberp x)
    (standard-numberp tm)
    (realp eps)
    (<= 0 tm)
    (< 0 eps)
    (i-small eps))
   (equal (standard-part (run x
                              (floor1 (* tm (i-large-integer)))
                              (/ (i-large-integer))))
          (standard-part (run x
                              (floor1 (/ tm eps))
                              eps)))))
```

The following axiom states that the standard part of the derivative of the function `(phi2 tm)` is equal to the standard part of `(f (phi2 tm))`, for limited `tm`. We state this as an axiom since there is no function `phi2` defined. Rather, we simply add the function signature `(phi2 tm)` to the environment and inform the prover, through this axiom, of the property which this function is to satisfy.

```
(defaxiom phi2-deriv
  (implies
   (and
    (realp tm)
    (i-limited tm)
    (realp eps)
    (not (equal eps 0))
    (i-small eps))
   (equal (standard-part (/ (- (phi2 (+ tm eps))
                               (phi2 tm)) eps))
          (standard-part (f (phi2 tm))))))
```

The following theorem states that the function which we have defined, `phi`, is equal to the function `phi2` under the conditions shown. This shows that `phi` is a function whose derivative is `f`, since it is equal to the function `phi2`, whose derivative is assumed to be `f`.

```
(defthm-std phi2-st-run-eq-std-thm
  (implies
   (and
    (realp tm)
    (<= 0 tm))
   (equal (phi2 tm)
          (phi (phi2 0) tm))))
```

4.5 Summary

We have shown, through the use of a nonstandard proof, that an initial value problem with Lipschitz continuous vector field does have a solution. Furthermore, this solution is unique. In the development of this proof, we have shown that existence of a solution, in nonstandard analysis, reduces to showing that the proposed solution is limited for limited time t and initial condition x_0. We have also shown that the solution is time invariant, as well as continuous with respect to time and the initial condition. We also made extensive use of the standardization and transfer principles in showing properties about the solution ϕ by way of its approximation ρ^{ϵ}. Many of these properties and proof methods will be used again in the proofs of the coming chapters.

Chapter 5

A Hybrid System Model and a Nonstandard Definition of its Solution

In this chapter, we propose a formal model of a hybrid system and a definition of its solution based on nonstandard constructs. As we noted in the case of hybrid automata in section 3.9.1, an essential characteristic of modeling hybrid systems is assignment to the system variables. While Carathéodory and Filippov solutions address the discontinuity in a differential equation, the resulting solution is itself continuous. With the possibility of assignment to system variables in a hybrid system, the solution may not be continuous, but may contain sudden jumps, as presented in the example solution 3.3. We address assignment to system variables as well as discontinuous vector fields in our presentation of a nonstandard definition of a solution to a hybrid system model. The chapter commences by discussing the definition of the hybrid system model which we intend to find a solution for. Thereafter, we present the formal definition for a solution of such a model.

5.1 A Definition of a Hybrid System Model

We assume the differential equation:

$$\dot{x} = F(x), \tag{5.1}$$

where the vector field F may be discontinuous in x. We assume the vector field F has the following form:

$$F(x) = \begin{cases} f_1(x) & x \in G_1 \\ f_2(x) & x \in G_2 \\ \cdots & \\ f_m(x) & x \in G_m, \end{cases} \tag{5.2}$$

where m is a positive standard integer. It is assumed that F satisfies the following:

F1. Each component vector field f_i is Lipschitz continuous, with Lipschitz constant L, over an open neighborhood of the closure of G_i, for integer i, $1 \leq i \leq m$:

$$\|f_i(x_1) - f_i(x_2)\| \leq \; L \, \|x_1 - x_2\|, \tag{5.3}$$

for $x_1, x_2 \in H$, where H is an open neighborhood of the closure of G_i.

F2. Each component vector field f_i is a standard function.

For integer i, $1 \leq i \leq m$, each region G_i is assumed to satisfy the following criteria:

G1. $\emptyset \subset G_i \subseteq \mathbb{R}^n$,

G2. The regions are disjoint; that is, $G_i \cap G_j = \emptyset$, for positive integers $i \neq j$,

G3. $\bigcup_{i \in [1,m]} G_i = \mathbb{R}^n$,

G4. Each region G_i can be described by an internal predicate.

For a particular system, it is possible that the solution resides within some domain D, where $D \subset \mathbb{R}^n$. In this case, we may have $\bigcup_{i \in [1,m-1]} G_i = D$, and for $x \notin D$, $f_m(x) = 0$, which results in a vector field in the form 5.2. A region G_i can be described by an internal predicate if there exists an internal predicate $P(x)$ whose free variables are components of the vector x and $x \in G_i$ iff $P(x)$. Henceforth, we will say that a vector field is in the form of 5.2, with the understanding that F satisfies the criteria F1 and F2 and that the regions G_i satisfy the criteria G1 through G4.

To describe a Hybrid system, we define a vector field F as in 5.2, as well as an assignment function Y and assignment predicate B_Y. The assignment function $Y : \mathbb{R}^n \mapsto \mathbb{R}^n$ maps the values of the system variables to the new values the variables are to be assigned. The assignment is performed when the system reaches a state satisfying the predicate B_Y. The assignment function is assumed to be standard and the assignment predicate is assumed to be internal. For a given vector field F in the form 5.2, the following are the requirements of the assignment function and predicate:

A1. Once the state x satisfies the assignment predicate B_Y, application of the assignment function results in a new state which does not satisfy B_Y:

$$B_Y(x) \rightarrow \neg B_Y(Y(x)). \tag{5.4}$$

A2. For a solution evaluated up to time t, the norm of the difference in variable value between successive assignments, which occur prior to time t, is upper bound by $K(x_0, t)$ times the time duration between successive assignments:

$$\|x_i - x_{i-1}\| \leq K(x_0, t)\, \Delta_i, \tag{5.5}$$

where $K(x_0, t)$ is a positive, monotone increasing, standard function for a solution with initial condition x_0 evaluated up to time t and x_i represents, for positive integer i, the value of x immediately after the i^{th} assignment, with x_0 being the initial value of x. $K(x_0, t)$ should be limited for limited x_0 and t. For $i > 1$, Δ_i is the time duration between the i^{th} assignment and its immediately preceding assignment. The duration Δ_1 is defined to be 1.

We should point out that Δ_1 is not defined as the time at which the first assignment occurs, since, for ϵ infinitesimal, the first assignment may occur at infinitesimal time t.

Definition 5.1.1. A well formed hybrid system $H = (\mathbb{R}^n, F, B_Y, Y, G_1, \ldots, G_m)$, where

- $\mathbb{R}^n$ represents the state space of the system for n a standard positive integer,
- F is a vector field which satisfies the form 5.2 and, for integer i, where $0 < i \leq m$, each component vector field, f_i, satisfies requirements F1 and F2,
- $B_Y : \mathbb{R}^n \mapsto \{true, false\}$ is the assignment predicate and satisfies requirement A1,
- $Y : \mathbb{R}^n \mapsto \mathbb{R}^n$ is the assignment function and satisfies requirement A2, and
- G_i is a region in $\mathbb{R}^n$ such that, for integer i such that $0 < i \leq m$, each region G_i satisfies requirements G1 through G4.

In the case of a computer controlled system, the function Y models the semantics of the computer program, the analog to digital conversion, the real time behavior of the computer, and auxiliary real valued variables that may be required to prove properties about the system. The vector argument to the function Y includes components representing signals which the controller senses from the physical system. The vector value of the function Y includes components representing the output of the controller to the physical system.

It is possible that the computer only senses signals from the physical system and generates no output. In this case, the assignment function is the identity function. One may also assign a vector of system variables $y \in \mathbb{R}^k$, for $0 < k < n$, where y is modified only by the assignment function Y.

5.2 A Sample System: Bouncing Ball

We may use our requirements for a definition of a hybrid system to describe a system consisting of a rigid body, represented by a ball, accelerating due to gravity and impacting the ground. A rigid body is a representation of a physical object which does not bend or deform upon collision and its impact with other rigid bodies is assumed to have an instantaneous duration of time. Actual physical bodies are not perfectly rigid. They do have some degree of deformation, and their impact lasts for some positive duration of time. However, modeling them requires more complex partial differential equations which are difficult to model and analytically represent [69]. Therefore, rigid bodies are used by modelers in such areas as robotics and manufacturing [69].

For our sample system, we will assume a rigid body, a ball, accelerates due to gravity and elastically impacts the ground, rebounding with a coefficient of restitution α, where $0 < \alpha \leq 1$. The coefficient of restitution is a ratio of the velocity of the rigid body after impact to that prior to impact.

The following is the definition for such a system:

$$\begin{aligned}
\dot{v} &= -g \\
\dot{x} &= v \\
B_Y \overset{\text{def.}}{=} & \; v < 0 \wedge x \leq 0 \\
Y(v) &= -\alpha v,
\end{aligned}$$

where g is the acceleration due to gravity, 9.8 m/s, x is the distance between the ball and the ground, and v is the velocity of the ball, where a positive value of v denotes the ball is moving away from the ground. We observe that the component vector fields associated with this system satisfy the vector field requirements. Since the region is $\mathbb{R}^2$, the region requirements are trivially met. Since B_Y is true only if $v < 0$, and the assignment function Y changes the sign of v, it may be verified that the assignment predicate B_Y satisfies A1. In addition, the above equations do satisfy the assignment requirement A2, since we have, for $i > 1$:

$$\|v_{i-1} - v_i\| = \|v_{i-1} - \alpha \; v_{i-1}\| = (1-\alpha) \; \|v_{i-1}\|, \tag{5.6}$$

$$\Delta_i = \frac{2 \; \|v_{i-1}\|}{g}. \tag{5.7}$$

Therefore,

$$\frac{\|v_{i-1} - v_i\|}{\Delta_i} = \frac{(1-\alpha) \; \|v_{i-1}\|}{2 \; \|v_{i-1}\| \; /g} = \frac{g(1-\alpha)}{2}. \tag{5.8}$$

For $i = 1$, $v_1 = -\alpha(v_0 - g \; t)$, then $\|v_1 - v_0\| \leq \|(1+\alpha)v_0 - \alpha \; g \; t\|$, where t is the time at which the first assignment occurs. Therefore, to satisfy A2, we let $K(x_0, T) = 2 \; \|v_0\| + gT + g(1-\alpha)/2$, for the solution evaluated at $t \leq T$.

Figure 5.1 shows a sample graph of the velocity v and position x of the ball over time, for an initial velocity $v_0 = 10$ m/s and position $x_0 = 0$ m with a coefficient of restitution $\alpha = 0.8$. This graph is based on a computer simulation of the system.

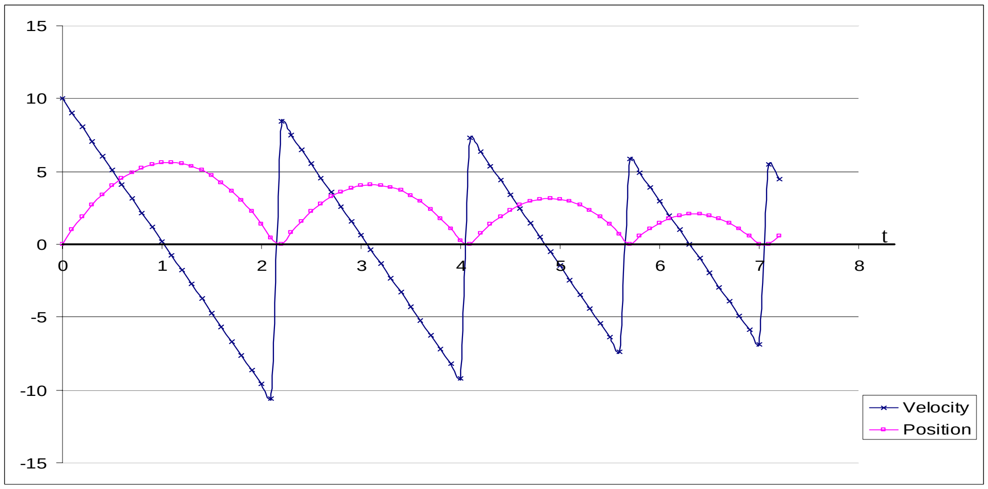

Figure 5.1: Graph of velocity and position trajectories for bouncing ball system with $\alpha = 0.8$

This graph displays the imprecision of computer simulation of physical systems. We note that the graph shows the velocity of the ball initially being 10 m/s. Upon the ball reaching the ground at approximately 2.2 s, the magnitude of the velocity is slightly greater than 10 m/s, which is inaccurate, as the magnitude should be no greater than 10 m/s.

The bouncing ball system has many interesting features. First, the solution for the velocity of the ball is discontinuous in time. We observe from the graph in Figure 5.1 that the velocity instantaneously changes sign at certain points in time; specifically, when B_Y evaluates to true. The solution is not Newtonian, Carathéodory, or Filippov, since the velocity is not continuous. The solution cannot be modeled by a hybrid automata, since the hybrid automata requires the *non-Zeno* condition where the infinite series, represented as the sum of the durations between succeeding impacts, does not uniformly converge to some finite number. We note that for the bouncing ball, the sum of all the durations between succeeding impacts does uniformly converge to a finite number, resulting in infinitely many impacts in finite time.

While the bouncing ball system is not computer controlled, it does have a feature which resembles that of a hybrid system: the instantaneous change in the velocity solution. For this physical system, the instantaneous change is due to impact; for hybrid systems, an instantaneous change may occur due to the discrete system's change in output.

5.3 Nonstandard Definition of Solution

In this section, we will attempt to formally define a solution of a well formed hybrid system. We observe that, since each function f_i in 5.2 is Lipschitz continuous, then a Newtonian solution exists over some time duration for the given initial condition.

For every vector field f_i with initial condition x_0, we define a function $\rho_i^\epsilon(x_0, t)$:

$$\rho_i^\epsilon(x,t) = \begin{cases} x & t < \epsilon \\ \rho_i^\epsilon(x + f_i(x)\epsilon, t - \epsilon) & t \geq \epsilon. \end{cases} \tag{5.9}$$

The function $\rho_i^\epsilon(x_0, t)$ may be regarded as the Euler approximation of the Newtonian solution associated with the vector field f_i with initial condition x_0.

We associate with each region G_i a switch predictor function denoted swp_i^ϵ with mapping:

$$swp_i^\epsilon : \mathbb{R}^n \times \mathbb{R} \mapsto \mathbb{R}. \tag{5.10}$$

Definition 5.3.1. For a well formed hybrid system with vector field F in the form 5.2, assignment predicate B_Y, and ϵ a positive real, the switch predictor function swp_i^ϵ is defined as follows:

$$swp_i^\epsilon(x,t) = \begin{cases} 0 & x \notin G_i \ \vee \ B_Y(x) \ \vee \ t < \epsilon \\ \epsilon + swp_i^\epsilon(x + f_i(x)\epsilon, t - \epsilon) & \text{otherwise.} \end{cases} \tag{5.11}$$

Informally, if a point x_0 is in region G_i, for some positive ϵ, $swp_i^\epsilon(x_0, t)$ evaluates, within ϵ, to the minimum of the following:

1. The least time, if it exists, at which ρ_i^ϵ leaves G_i; that is,

$$\forall_{0 \leq u \leq swp_i^\epsilon(x_0,t) - \epsilon} \ \rho_i^\epsilon(x_0, u) \in G_i \ \wedge \ \rho_i^\epsilon(x_0, swp_i^\epsilon(x_0,t)) \notin G_i, \tag{5.12}$$

2. The least time, if it exists, at which ρ_i^ϵ satisfies B_Y; that is,

$$\forall_{0 \leq u \leq swp_i^\epsilon(x_0,t) - \epsilon} \neg B_Y(\rho_i^\epsilon(x_0, u)) \ \wedge \ B_Y(\rho_i^\epsilon(x_0, swp_i^\epsilon(x_0,t))), \text{ or} \tag{5.13}$$

3. The time t.

Definition 5.3.2. Given a well formed hybrid system with vector field F in the form 5.2, assignment function Y, and assignment predicate B_Y, we define the system solution ϕ at non-negative time t with initial condition x_0 as follows:

$$\phi(x_0, t) \overset{std}{=} st(\gamma^\epsilon(x_0, t)), \text{ where} \tag{5.14}$$

$$\gamma^\epsilon(x,t) = \begin{cases} x & t < \epsilon \\ \gamma^\epsilon(Y(x), t - \epsilon) & B_Y(x) \\ \gamma^\epsilon(\rho_1^\epsilon(x, swp_1^\epsilon(x,t)), t - swp_1^\epsilon(x,t)) & x \in G_1 \\ \gamma^\epsilon(\rho_2^\epsilon(x, swp_2^\epsilon(x,t)), t - swp_2^\epsilon(x,t)) & x \in G_2 \\ \cdots & \cdots \\ \gamma^\epsilon(\rho_m^\epsilon(x, swp_m^\epsilon(x,t)), t - swp_m^\epsilon(x,t)) & x \in G_m, \end{cases} \tag{5.15}$$

where the predicates in the conditional assignments are evaluated from top to bottom. It is also required that $\gamma^\epsilon(x_0, t)$ be limited for limited x_0, limited t, and positive infinitesimal ϵ.

Since we require that $\gamma^\epsilon(x_0, t)$ be limited for limited x_0, limited t, and positive infinitesimal ϵ, and since standard part of a limited point is standard, then for any standard x_0 and t, $st(\gamma^\epsilon(x_0, t))$ is standard. Therefore, by the standardization principle, there exists a standard function ϕ such that $\phi(x_0, t)$ is equal to $st(\gamma^\epsilon(x_0, t))$ for standard x_0 and t. We use the operator $\overset{std}{=}$ to denote that $\phi(x_0, t)$ is equal to $st(\gamma^\epsilon(x_0, t))$ for standard x_0 and t.

5.4 The Switch Predictor Function

The definition 5.3.2 depends on each switch predictor function swp_i^ϵ. Let us assume that swp_i^ϵ evaluates to non-negative standard numbers only. If we assume that swp_i^ϵ evaluates to zero, then the time variable t remains unchanged in each recursive call, whereby the state variable value does not change. If we assume swp_i^ϵ evaluates to a positive standard number, then it may not accurately depict the time required for the solution to leave a closed region.

We demonstrate with an example. Suppose we are given a vector field F defined as follows:

$$F(x) = \begin{cases} -1 & x > 0 \\ 1 & x \leq 0. \end{cases} \tag{5.16}$$

We assume that no assignments exist in the system; that is, B_Y is identically false. If $x = 0$, then the solution moves in the positive direction. If $x > 0$, the solution should move in the negative direction. This results in an oscillation along $x = 0$, as depicted in figure 5.2.

Assuming that $x_0 = 0$, the switch predictor function cannot return zero, as this results in the fixed point discussed earlier. If it returns some positive standard number, this implies the solution is moving with a positive derivative for some positive (standard) duration in the $x > 0$ region, which contradicts the system definition in Equation 5.16.

Therefore, in choosing a standard swp_i^ϵ, the solution may become imprecise. We need some notion of depicting a value of swp_i^ϵ which is greater

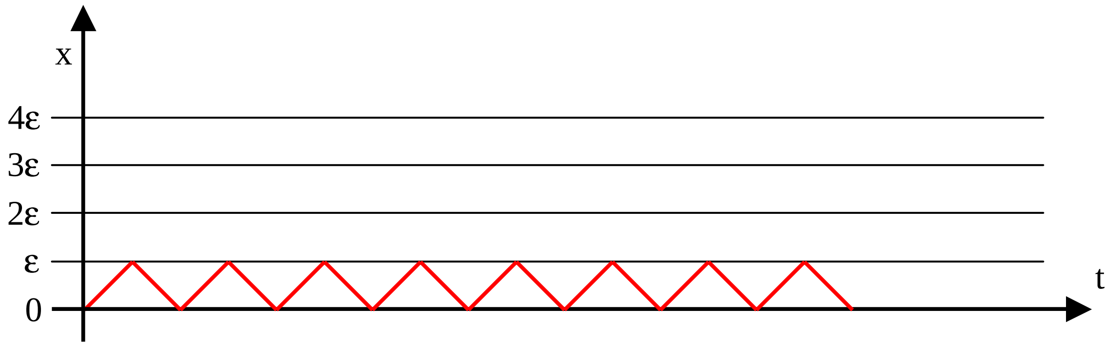

Figure 5.2: Graph of $\gamma^{\epsilon}(0,t)$ for Example System Described by Equation 5.16

than zero, but less than any positive standard number. This is precisely the definition of a positive infinitesimal. By allowing ϵ, and hence swp_i^{ϵ}, to have a positive infinitesimal value, the resulting oscillations depicted in Figure 5.2 would have an infinitesimal magnitude. By the definition of solution 5.3.2, for any given standard t, the solution $\phi(0,t) = st(\epsilon)$, for some positive infinitesimal ϵ. Since the standard part of an infinitesimal is zero, then the solution is $\phi(0,t) = 0$. We note that this is also a Filippov solution for such a system.

5.5 Existence of a Solution

In this section, we will prove the existence of a solution to a well formed hybrid system. We will first show that the recursion γ^{ϵ} does terminate, hence implying that γ^{ϵ} is a total function. We will then show that a solution does exist in the case of a system defined with no assignments. Finally, we will show that a solution exists for a system with assignments.

Lemma 5.5.1. *The definition of $\gamma^{\epsilon}(x,t)$ in 5.15 is that of a total function, assuming $x \in \mathbb{R}^n$, real $t \geq 0$, and ϵ a positive real number.*

Proof. Assuming $\epsilon > 0$, we may show that γ^{ϵ} is a total function by showing that the recursion γ^{ϵ} does terminate for any $x \in \mathbb{R}^n$ and real $t \geq 0$. For $t \geq \epsilon$ and $x_0 \in G_i$, it can be shown that $swp_i^{\epsilon}(x_0,t) \geq \epsilon$. Since $\epsilon > 0$, then for $t \geq \epsilon$, the variable t decreases by at least ϵ in each recursive call. This assures that t eventually decreases to where $t < \epsilon$ and the recursion terminates. We observe that the function results in at most $\lfloor t/\epsilon \rfloor$ recursive calls. $\square$

Lemma 5.5.2. *Given a vector field F in the form 5.2, then for any $x \in \mathbb{R}^n$, $\|F(x)\| \leq L \|x\| + N$, where N is a positive real constant.*

Proof. Since, by criteria G1 each region is not empty and since each region can be described by a standard predicate, there is at least one point, y, in G_i which is standard. Since, by criteria F2, each f_i is standard, then $f(y)$ is standard.

Given a component vector field f_i of F, its norm may be written as follows:

$$\|f_i(x)\| = \|f_i(x) - f_i(y) + f_i(y)\|, \tag{5.17}$$

for $x \in G_i$. By the triangle inequality and since f_i is Lipschitz continuous with constant L, we have:

$$\|f_i(x)\| \leq L \|x - y\| + \|f_i(y)\| . \tag{5.18}$$

Applying the triangle inequality again, we may conclude from 5.18:

$$\|f_i(x)\| \leq L \|x\| + L \|y\| + \|f_i(y)\| . \tag{5.19}$$

Since $\|y\|$ and $\|f(y)\|$ are positive real constants, we let N equal the maximum value of $L \|y\| + \|f_i(y)\|$ ranging over integer $i \in [1, m]$. Therefore, we have:

$$\|f_i(x)\| \leq L \|x\| + N, \tag{5.20}$$

for $x \in G_i$. We extend this result from each component vector field f_i to F:

$$\|F(x)\| \leq L \|x\| + N, \tag{5.21}$$

for $x \in \mathbb{R}^n$. □

Lemma 5.5.3. *For a differential equation with initial condition $x_0 \in \mathbb{R}^n$, vector field in the form 5.2, and assignment predicate B_Y that is identically false, a solution $\phi(x_0, t)$ does exist in the form 5.3.2 for real time $t \geq 0$ and $x_0 \in \mathbb{R}^n$, assuming $\epsilon > 0$.*

Proof. It can be shown that the function γ^ϵ satisfies the following property:

$$\gamma^\epsilon(x_0, t) = \gamma^\epsilon(x_0, t - \epsilon) + F(\gamma^\epsilon(x_0, t - \epsilon))\epsilon. \tag{5.22}$$

Way may apply the norm operator to both sides of the equality in 5.22:

$$\|\gamma^{\epsilon}(x_0,t)\|=\|\gamma^{\epsilon}(x_0,t-\epsilon)+F(\gamma^{\epsilon}(x_0,t-\epsilon))\epsilon\| \ . \tag{5.23}$$

By the triangle inequality,

$$\|\gamma^{\epsilon}(x_0,t)\|\leq\|\gamma^{\epsilon}(x_0,t-\epsilon)\|+\|F(\gamma^{\epsilon}(x_0,t-\epsilon))\epsilon\| \ . \tag{5.24}$$

By the lemma 5.5.2,

$$\|\gamma^{\epsilon}(x_0,t)\|\leq (1+\epsilon L)\ \|\gamma^{\epsilon}(x_0,t-\epsilon)\| + N\epsilon. \tag{5.25}$$

The above relates γ^{ϵ} at time t with that at time $t-\epsilon$. We may use this relation to derive an upper bound on $\gamma^{\epsilon}(x_0,t)$:

$$\|\gamma^{\epsilon}(x_0,t)\|\leq (1+\epsilon L)^{\lfloor t/\epsilon\rfloor}\ \|x_0\| + \sum_{j=0}^{\lfloor t/\epsilon\rfloor-1} (1+\epsilon L)^j N\epsilon. \tag{5.26}$$

We use the property $(1+w)\leq e^w$, for $w\geq 0$, to derive:

$$\|\gamma^{\epsilon}(x_0,t)\|\leq e^{L\epsilon\lfloor t/\epsilon\rfloor}\ \|x_0\| + \sum_{j=0}^{\lfloor t/\epsilon\rfloor-1} e^{L\epsilon j}N\epsilon. \tag{5.27}$$

For non-negative t and positive ϵ, we use the properties $\epsilon\lfloor t/\epsilon\rfloor\leq t$ and, for $0\leq j\leq\lfloor t/\epsilon\rfloor$, $e^{L\epsilon j}\leq e^{Lt}$ to derive:

$$\|\gamma^{\epsilon}(x_0,t)\|\leq (Nt+\|x_0\|)e^{tL}. \tag{5.28}$$

Since N and L are standard positive real constants, if x_0 and t are limited, then the bound on $\|\gamma^{\epsilon}(x_0,t)\|$ is limited, implying that $\gamma^{\epsilon}(x_0,t)$ is limited. Hence, for any standard x_0 and t, and by Lemma 5.5.1, $st(\gamma^{\epsilon}(x_0,t))$ is defined and is standard. By the standardization principle, we may conclude that a standard function ϕ exists for any real $t \geq 0$ and $x_0 \in \mathbb{R}^n$, where $\phi(x_0, t)$ is equal to $st(\gamma^\epsilon(x_0, t))$ for standard x_0 and t. □

We now address the more general case of existence of a solution for a hybrid system definition with assignments.

Theorem 5.5.4. *For a well formed hybrid system with vector field F, assignment function Y, and assignment predicate B_Y, there exists a solution in the form 5.3.2 for standard initial condition $x_0 \in \mathbb{R}^n$ and standard non-negative real time t.*

Proof. By showing that the function $\gamma^\epsilon(x_0, t)$ is limited for positive real ϵ and limited x_0 and t, we may conclude that $st(\gamma^\epsilon(x_0, t))$ is defined. Thereafter, we may use the standardization principle to show that a standard function, ϕ, exists where $\phi(x_0, t) = st(\gamma^\epsilon(x_0, t))$ for standard x_0 and t. Therefore, the existence of the solution ϕ reduces to showing that $\gamma^\epsilon(x_0, t)$ is limited for limited initial condition x_0, limited t, and positive real number ϵ.

For a given t, x_0, and $\epsilon > 0$, there exists some $j \geq 0$ such that $t - \epsilon j$ is the time at which the last assignment occurs up to time t. We let $t_p = t - \epsilon j$. By the requirement of the assignment function, 5.5, it can be shown that γ^ϵ is bounded at time t_p:

$$\|\gamma^\epsilon(x_0, t_p)\| \leq \; t_p K(x_0, t_p). \tag{5.29}$$

Since t_p and $K(x_0, t_p)$ are limited, then $\gamma^\epsilon(x_0, t_p)$ is limited. By the time invariance property of γ^ϵ, which is shown later in lemma 5.5.5:

$$\gamma^\epsilon(x_0, t) = \gamma^\epsilon(\gamma^\epsilon(x_0, t_p), t - t_p), \tag{5.30}$$

where $t - t_p$ is an integer multiple of ϵ. Since there are no assignments after t_p, and since $\gamma^\epsilon(x_0, t_p)$ and $t - t_p$ are limited, then by lemma 5.5.3, $\gamma^\epsilon(x_0, t)$ is limited. Since $\gamma^\epsilon(x_0, t)$ is limited for limited x_0 and t, by the standardization principle, there exists standard function ϕ such that $\phi(x_0, t) = st(\gamma^\epsilon(x_0, t))$ for standard x_0 and t and some infinitesimal positive constant ϵ. $\square$

In the previous proof, we made use of the time invariance property of γ^ϵ. We will formally state this property and give its proof.

Lemma 5.5.5. *For $x_0 \in \mathbb{R}^n$, non-negative reals t_1 and t_2, and t_2 an integer multiple of ϵ, and $\epsilon > 0$:*

$$\gamma^\epsilon(x_0, t_1 + t_2) = \gamma^\epsilon(\gamma^\epsilon(x_0, t_1), t_2). \tag{5.31}$$

Proof. The proof is by induction over the well founded structure $\langle S, \prec \rangle$:

$$S = \{t_2 - j\epsilon : j \in \mathbb{N} \wedge 0 \le j \le \lfloor t_2/\epsilon \rfloor\}, \tag{5.32}$$

where, for $t_a, t_b \in S$, $t_a \prec t_b$ iff $t_a < t_b$. $\square$

5.6 Uniqueness of the Solution

While a solution with the nonstandard definition 5.3.2 does exist, it may not necessarily be unique. We construct a vector field F that shows this:

$$F(x) = \begin{cases} -1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0. \end{cases} \tag{5.33}$$

Suppose the solution has the initial value 5. For an arbitrary positive infinitesimal ϵ:

$$swp^{\epsilon}(5,t) = \begin{cases} 5 & \epsilon \left\lfloor \frac{5}{\epsilon} \right\rfloor = 5 \\ \epsilon \left\lfloor \frac{5}{\epsilon} \right\rfloor + \epsilon & \text{otherwise}, \end{cases} \tag{5.34}$$

for $5 \leq t < 5 + \epsilon$. We choose a nonstandard large, prime, positive integer, Z_∞. If we let $\epsilon = 5/Z_\infty$, then by 5.34, $swp^{\epsilon}(5,t) = 5$. Therefore, with this value of ϵ, the solution reaches zero. However, if we let $\epsilon = 6/Z_\infty$, then:

$$swp^{\epsilon}(5,0) = \frac{6}{Z_\infty} \left\lfloor \frac{5Z_\infty}{6} \right\rfloor + \frac{6}{Z_\infty}. \tag{5.35}$$

We should note that $5Z_\infty/6$ is not an integer, since Z_∞ is prime. Hence, for this value of ϵ, swp^{ϵ} is a number slightly larger than 5, and $\gamma^{\epsilon}(5, swp^{\epsilon}(5,t))$ is slightly less than zero, and its derivate remains -1. Consequently, depending on the value of ϵ chosen, the solution may reach zero and remain there, or continually descend with slope -1.

Therefore, it is possible that multiple solutions exist for a particular system, depending on the value of ϵ chosen. We should also point out, in our definition 5.3.2, the same value of ϵ is assumed in all recursive calls of γ^{ϵ}. We construct an alternative example which may not consist of all expected solutions. Consider the vector field F:

$$F(x) = \begin{cases} \dot{x} = 1 \wedge \dot{y} = 1 & x \neq 0 \wedge y \neq \sqrt{2} \\ \dot{x} = 0 \wedge \dot{y} = 1 & x = 0 \wedge y \neq \sqrt{2} \\ \dot{x} = 1 \wedge \dot{y} = 0 & x \neq 0 \wedge y = \sqrt{2} \\ \dot{x} = 0 \wedge \dot{y} = 0 & x = 0 \wedge y = \sqrt{2}. \end{cases} \tag{5.36}$$

The graph in figure 5.3 shows some Filippov solutions for this system. The solution labeled A cannot be achieved using the definition of solution in 5.3.2. Suppose the initial value is $x = -1$, $y = -1$. For the solution A to exist, we must find an ϵ such that $k_1\epsilon = 1$ and $k_2\epsilon = 1+\sqrt{2}$, for positive integers k_1 and k_2. We rewrite the first equation as $\epsilon = 1/k_1$ and substitute into the second. This results in $k_2/k_1 = 1 + \sqrt{2}$. However, it is impossible to find two integers whose ratio satisfies this equation, since $1 + \sqrt{2}$ is irrational.

One method which overcomes this limitation is the redefinition of the solution 5.3.2 where a new infinitesimal is chosen with each recursive call of γ^ϵ. Another approach is to *widen* some of the regions. We note, for example, the region $x = 0 \wedge y > \sqrt{2}$ requires that the solution x be precisely zero. We may select a positive standard real number α and redefine the region to be $|x| < \alpha \wedge y > \sqrt{2}$. This approach may be acceptable, depending on the concessions the designer wishes to make in the mathematical modeling of the system. Widening these regions also results in one unique solution for this example. This may be the desired affect intended by the system modeler; that is, the modeler may not desire the additional solutions labeled B through D in figure 5.3.

As an aside, it should be noted that nonstandard prime integers do

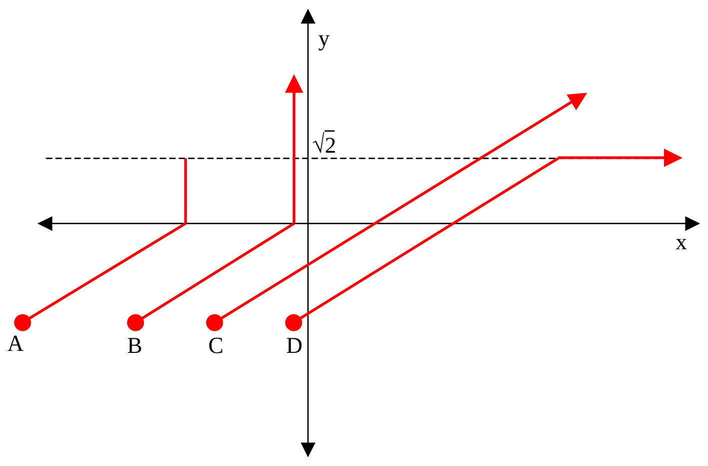

Figure 5.3: Possible trajectories of an example system with a Filippov solution not modeled by the definition 5.3.2. The dot denotes the initial value for each trajectory.

exist. Define an internal predicate $P(x)$ to be:

$$\exists_y (y \geq x \wedge y \in \mathbb{Z}) \rightarrow \text{prime}(y). \tag{5.37}$$

The predicate $P(x)$ is true for all standard real numbers x. By the transfer principle, it is true for all real numbers, including nonstandard ones.

5.7 Continuity of the Solution

In the case where the well formed hybrid system consists of assignments, the solution is discontinuous in t. However, in the case where no assignments occur (the assignment predicate B_Y is identically $false$) the solution is continuous.

Lemma 5.7.1. *For a well formed hybrid system without assignments ($B_Y = false$), limited $x_0 \in \mathbb{R}^n$ and non-negative standard t, the solution $\phi(x_0, t)$, defined in definition 5.3.2, is continuous with respect to t.*

Proof. To show continuity of ϕ with respect to t, for a standard real t_1 and $t_2 \simeq t_1$, we need to show that:

$$st(\phi(x_0, t_1)) = st(\phi(x_0, t_2)). \tag{5.38}$$

For the case in which no assignments occur in the system, we have already shown in the proof of lemma 5.5.3, that γ^ϵ is bounded:

$$\|\gamma^\epsilon(x_0, t)\| \leq (Nt + \|x_0\|)e^{tL}. \tag{5.39}$$

By the bound on F as shown in lemma 5.5.2, and by the bound on γ^ϵ in 5.39, in a proof similar to that of lemma 4.2.1, it may be shown that:

$$\|\phi(x_0, t_1) - \phi(x_0, t_2)\| \leq (t_2 - t_1)(N + NLt_2e^{Lt_2} + Le^{Lt_2} \|x_0\|), \qquad (5.40)$$

for $t_2 \geq t_1$. By **TH7**, we may apply standard part to both sides of 5.40:

$$st(\|\phi(x_0, t_1) - \phi(x_0, t_2)\|) \leq st((t_2 - t_1)(N + NLt_2e^{Lt_2} + Le^{Lt_2} \|x_0\|)), \quad (5.41)$$

Since $t_1 \simeq t_2$, $st(t_2 - t_1) = 0$. Also, since N and L are standard constants, they are limited. Since x_0 and t_2 are limited, the formula 5.41 may be rewritten to:

$$st(\|\phi(x_0, t_1) - \phi(x_0, t_2)\|) \leq 0. \qquad (5.42)$$

By properties of standard part and norm,

$$st(\phi(x_0, t_1)) = st(\phi(x_0, t_2)). \qquad (5.43)$$

A similar proof may be carried out for the case $t_1 \geq t_2$. □

5.8 Solution Time

In the definition of the solution 5.3.2, each recursive call causes the solution time to be decremented by a time duration ϵ. Furthermore, it is possible that the assignment function may be executed for up to half of this solution time duration. If the physical system is defined with some variable *tmr* that has a constant derivative of one, with initial condition of 0, then the variable *tmr* would be approximately half of the solution time. Specifically, if the duration of the solution time is $n\epsilon$, then a time duration up to $\lceil n/2 \rceil \epsilon$ may be due to the assignment function and a duration of $\lfloor n/2 \rfloor \epsilon$ may be due to the updating of physical system variables by the respective vector field function. Therefore, we require that all properties defined for the system, which reason about time, should refer to a physical system variable which tracks time, such as the example variable *tmr* we have described.

It is possible to associate with the assignment function a duration ϵ^2, resulting in the standard part of the solution time being equal to the standard part of the time as tracked by the physical system variable. However, this is an unnecessary complication of the system's formal model, as properties about the system time can be stated by using the physical system variables.

5.9 An Alternative Solution Definition

The definition of solution 5.3.2 requires that the assignment function Y be evaluated at most every other time step. This requirement is instated so as to allow for the evaluation of the vector field functions at least every other time step. An alternative solution definition may be pursued that would remove this requirement. The alternative solution definition is as follows:

$$\phi(x_0, t) \overset{std}{=} st(\gamma^\epsilon(x_0, t)), \text{ where} \tag{5.44}$$

$$\gamma^\epsilon(x,t) = \begin{cases} x & t < \epsilon \\ \gamma^\epsilon(\hat{\rho^\epsilon}_1(x, swp_1^\epsilon(x,t)), t - swp_1^\epsilon(x,t)) & x \in G_1 \\ \gamma^\epsilon(\hat{\rho^\epsilon}_2(x, swp_2^\epsilon(x,t)), t - swp_2^\epsilon(x,t)) & x \in G_2 \\ \dots & \dots \\ \gamma^\epsilon(\hat{\rho^\epsilon}_m(x, swp_m^\epsilon(x,t)), t - swp_m^\epsilon(x,t)) & x \in G_m, \end{cases} \tag{5.45}$$

where each function $\hat{\rho^\epsilon}_i$, is defined as follows:

$$\hat{\rho^\epsilon}_i(x,t) = \begin{cases} x & t < \epsilon \\ \hat{\rho^\epsilon}_i(\hat{Y}(x + f_i(x)\epsilon), t - \epsilon) & t \geq \epsilon, \end{cases} \tag{5.46}$$

where $\hat{Y}$ is defined in terms of the original assignment function Y and assignment predicate B_Y:

$$\hat{Y}(x) = \begin{cases} Y(x) & B_Y(x) \\ x & \text{otherwise.} \end{cases} \tag{5.47}$$

This alternative definition is such that the vector field function is evaluated every time step. However, this alternative definition of solution is not consistent with our original definition. We demonstrate with an example. Suppose we have the following system definition:

$$\begin{aligned} & x_0 = 5, \\ & \frac{dx}{dt} = -1, \\ & B_Y \overset{\text{def.}}{=} x \geq 5, \\ & Y(x) = 8. \end{aligned} \tag{5.48}$$

In our original solution definition 5.3.2, since $x_0 = 5$, then the initial state satisfies the assignment predicate B_Y, resulting in an assignment where the value of x becomes 8.

In the alternative solution 5.45, since a vector field is evaluated prior to performing an assignment, the assignment predicate B_Y is not checked for the initial condition. Rather, the vector field is evaluated, resulting in a new value for the solution: $x = 5 - \epsilon$, for $\epsilon > 0$, which does not satisfy the assignment predicate in this step nor in succeeding steps.

Therefore, while the alternative solution method does eliminate the requirement for checking that an assignment occurs at most every other time step, it introduces the possibility that an assignment is not taken when the solution reaches a state satisfying the assignment predicate.

5.10 Comparing the Nonstandard Definition of Solution with Other Definitions

In dealing with discontinuous vector fields, Filippov presented the definition of a solution whose derivative satisfies a differential inclusion at discontinuities in the vector field, as shown in 3.32. In general, there does not exist, for each point in $\mathcal{F}f(x)$, a corresponding solution with a derivative equal to that point. The solution definition merely states that the solution's derivative, if it exists, satisfies the differential inclusion. This is a definition which admits a greater number of solutions than the Newtonian approach.

For our definition of solution, in the case that the well formed hybrid system is described by a Lipschitz continuous vector field, as shown in chapter 4, the solution is unique and satisfies the Newtonian definition. In the case of a well formed hybrid system without assignments, our system definition reduces to the vector field F shown in 5.2, which is defined in terms of m Lipschitz continuous functions. In the work by Stewart [68], a vector field function F similar to 5.2 is proposed, with the exception that the boundary between regions is piece-wise smooth and each region G_i is assumed to be open, where $\mathbb{R}^n \setminus \bigcup_{i \in [1..m]} G_i$ is a null set, or a set with a Lebesgue measure of zero. For our system, we may assume Stewart's model as a special case of the vector field F in 5.2 where the regions G_i are such that 1) the boundaries between regions are piece-wise smooth and 2) the value of F at each point p on a boundary is such that $F(p) \in co\{f_i(p) : p \in \bar{G}_i\}$, where co is the convex hull, and $\bar{G}_i$ is the closure of G_i. For this special case, the Filippov differential inclusion 3.32 is reduced to the following:

$$\dot{x}(t) \in co\{f_i(x) : i \in I\}, \tag{5.49}$$

where I is the set consisting of the index i of each region G_i near the solution [68]. Stewart also identifies methods whereby one may determine whether such a differential equation has a unique solution for given initial conditions. In general, the solution is not unique.

5.11 Satisfying the Assignment Function Requirements

The assignment function requirement A2 in the definition 5.1.1 requires that the solution satisfy the inequality:

$$\|x_i - x_{i-1}\| \leq K(x_0, t)\, \Delta_i, \tag{5.50}$$

where x_i denotes the value of x immediately after the i^{th} assignment. In general, satisfying this requirement is difficult since it requires knowledge of the system solution. In the special case where the duration between assignments is known to be limited and not infinitesimal, an alternate criteria may be used.

Theorem 5.11.1. *Given an assignment function Y satisfying:*

$$limtd(x) \ \wedge B_y(x) \ \rightarrow \ limtd(Y(x)), \tag{5.51}$$

where $x \in \mathbb{R}^n$ and the duration between assignments Δ is know to be a positive limited real number which is not infinitesimal, then requirement A2 is satisfied.

Proof. We will assume that the value of the state immediately prior to the i^{th} assignment is x_i^-. Since the duration between the assignments Δ_i is known to be not infinitesimal, then A2 may be restated as:

$$\frac{\|Y(x_i^-) - Y(x_{i-1}^-)\|}{\Delta_i} \leq K(x_0, t). \tag{5.52}$$

Since Y is being applied, then it must be the case that x_i^- and x_{i-1}^- each satisfy B_Y. By the assumption 5.51 regarding Y, we may conclude that $Y(x_i^-)$ and $Y(x_{i-1}^-)$ are limited. Since Δ_i is limited and not infintesimal, then the ratio

$$\frac{\|Y(x_i^-) - Y(x_{i-1}^-)\|}{\Delta_i} \tag{5.53}$$

exists and is limited. In particular, for a solution evaluated up to limited t, there exists a limited number of assignments. We let $K(x_0, t)$ be the maximum ratio 5.53 which occurs up to time t, satisfying requirement A2. $\square$

In modeling computer controlled systems, we model the computer as exchanging input/output signals with the physical system at regular intervals, with a cycle time period that is limited and not infinitesimal. Therefore, in such cases, we may apply theorem 5.11.1.

If theorem 5.11.1 is not applicable, another possibility is to assure that Y is continuous. In this case, one can use the continuous time solutions to meet requirement A2. In the example bouncing ball problem in section 5.2, the continuous equations for velocity and position were used.

Theorem 5.11.2. *If the assignment function Y of a proposed hybrid system definition is continuous, where Y satisfies the property*

$$limtd(x) \wedge B_Y(x) \;\rightarrow\; limtd(Y(x)), \tag{5.54}$$

then a standard function K based on the closed form solutions and system time between assignments may be used to meet the requirments of A2.

Proof. In this case, we have:

$$\frac{\|Y(\phi(x_0,t_2)) - Y(\phi(x_0,t_1))\|}{(t_2 - t_1)} \leq K(x_0,t), \tag{5.55}$$

where $x \in \mathbb{R}^n$ and non-negative $t, t_1, t_2 \in \mathbb{R}$, such that $t \geq t_2 > t_1$. The formula is true for all standard and non-standard values. We will assume standard values for x_0, t, t_1, and t_2. Since t_1 and t_2 are standard real numbers and $t_2 > t_1$, then $t_2 - t_1$ is a positive standard real and not infinitesimal. Applying **TH7**:

$$st\left(\frac{\|Y(\phi(x_0,t_2)) - Y(\phi(x_0,t_1))\|}{(t_2 - t_1)}\right) \leq st(K(x_0,t)). \tag{5.56}$$

Since the numerator and denominator are limited, and since Y is continuous and limited:

$$\frac{\|Y(st(\phi(x_0,t_2))) - Y(st(\phi(x_0,t_1)))\|}{(t_2 - t_1)} \leq st(K(x_0,t)). \tag{5.57}$$

By the standardization principle and the definition of ϕ:

$$\frac{\|Y(st(\gamma^\epsilon(x_0,t_2))) - Y(st(\gamma^\epsilon(x_0,t_1)))\|}{(t_2 - t_1)} \leq st(K(x_0,t)). \tag{5.58}$$

By properties of standard part and continuity of Y:

$$st\left(\frac{\|Y(\gamma^\epsilon(x_0,t_2)) - Y(\gamma^\epsilon(x_0,t_1))\|}{(t_2 - t_1)}\right) \leq st(K(x_0,t)). \tag{5.59}$$

By algebra:

$$st\left(\frac{\|Y(\gamma^\epsilon(x_0,t_2)) - Y(\gamma^\epsilon(x_0,t_1))\|}{(t_2 - t_1)}\right) < st(K(x_0,t)+1). \tag{5.60}$$

Finally, applying the contrapositive of **TH7**:

$$\frac{\|Y(\gamma^\epsilon(x_0,t_2)) - Y(\gamma^\epsilon(x_0,t_1))\|}{(t_2 - t_1)} < K(x_0,t)+1. \tag{5.61}$$

□

5.12 Summary

In this chapter, we formally defined a well formed hybrid system and proposed a new, nonstandard definition of a solution for such a system. The system definition admits discontinuous vector fields as well as an assignment function. Both the vector field discontinuity and the assignment function are essential features in modeling a hybrid system due to the possible switching in the vector field, hence discontinuity, and possible change in computer system output, which can be modeled by the assignment function. In the event that the vector field is discontinuous, it is possible for the solution to switch between regions infinitely many times in finite time. Intuitively, the solution resides in a region for an infinitesimal duration of time before switching to a different region, tracing an infinitesimal oscillatory path along the boundaries of these regions.

We also showed that, while the definition of the solution itself is nonstandard due to its dependence on the predicate standard, there does exist a standard function which is equal to the nonstandard definition for standard time t and initial condition x_0. While a solution does exist, it is not guaranteed to be unique. In particular, the solution may depend on the choice of infinitesimal ϵ used in its definition.

Chapter 6

Reasoning About Safety and Progress

Having established requirements for the definition of a Hybrid system and definition of its solution, we would like to reason about correctness properties of such a system. Our objective is to use methods employed in the computer science community in reasoning about correctness of programs, and extending them to reason about correctness of the solutions of a well formed hybrid system. In particular, we would like to reason about two classes of properties: safety and progress. We will formally define these properties and provide proof procedures whereby one may reason about their correctness for a given well formed hybrid system.

6.1 Reasoning About Safety

A safety property holds for the initial state of the system and for all ensuing states. Formally, for a safety property P, we require:

$$P(x_0) \rightarrow \forall_{t \geq 0} \; P(\phi(x_0, t)), \tag{6.1}$$

where P is internal and ϕ is the system solution of a well formed hybrid system, as defined in 5.3.2.

We limit the predicate P to the following grammar:

$$true \mid false \mid (\theta_1 \wedge \theta_2) \mid P_d(x) \mid (\theta_1 \vee \theta_2) \mid g(x) = 0 \mid g(x) \sim 0, \tag{6.2}$$

where θ_1, θ_2 are formula which satisfy the grammar, and g is any real valued, continuous, standard function over $x \in \mathbb{R}^n$. The symbol $\sim$ denotes any inequality operator in the set $\{\leq, \geq\}$.

A predicate P_d is referred to as a discrete predicate and is evaluated on the state space x of the system. The predicate must be internal and must satisfy the following requirement:

$$limtd(x) \rightarrow \ (\, P_d(x) \text{ iff } P_d(st(x))\,). \tag{6.3}$$

The intuition behind the discrete predicate P_d is that it describes some property about the integer valued variables of the computer program of the system. The requirement 6.3 would be difficult to establish for reals in general, since it requires for a predicate to be both open and closed. Alternatively, for a limited integer z, by **TH1** and **TH10**, $st(z) = z$. Therefore, for predicate P_d evaluated over standard integers only, the requirement 6.3 is met.

We shall refer to each formula in 6.2 of the form $true$, $false$, $g(x) = 0$ or $g(x) \sim 0$ as an atomic formula of a predicate P. We shall refer to each formula of the form $g(x) = 0$ or $g(x) \sim 0$ as an inequality of a predicate P.

We note that the grammar 6.2 does not allow strict inequalities. One may conservatively approximate a strict inequality by choosing a positive constant α sufficiently close to zero and using the properties:

$$\begin{aligned} g(x) \leq -\alpha &\quad\rightarrow\quad g(x) < 0 \text{ and} \\ g(x) \geq \alpha &\quad\rightarrow\quad g(x) > 0. \end{aligned} \tag{6.4}$$

For example, to conservatively approximate $x < 5$, one may use $x \leq 5 - 1/100$.

Definition 6.1.1. For a predicate P satisfying grammar 6.2, the ϵ-transform modifies each inequality within P as follows:

$$\begin{aligned} g(x) = 0 &\overset{\epsilon}{\Longrightarrow} |g(x)| \leq h(x,\epsilon), \text{where } h(x,\epsilon) \geq 0, \\ g(x) \leq 0 &\overset{\epsilon}{\Longrightarrow} g(x) \leq h(x,\epsilon), \text{where } h(x,\epsilon) \geq 0, \\ g(x) \geq 0 &\overset{\epsilon}{\Longrightarrow} g(x) \geq h(x,\epsilon), \text{where } h(x,\epsilon) \leq 0. \end{aligned} \tag{6.5}$$

The function $h(x,\epsilon)$ is any standard real valued function which is infinitesimal for ϵ infinitesimal. The transform modifies only the inequalities in P; the logical operators, the predicate P_d, and the constants *true* and *false* in P remain unchanged.

We refer to the transforms in 6.5 as ϵ-transforms. For example, if P is the predicate $x \leq 5$, after applying the ϵ-transform with $h(x,\epsilon) = \epsilon$, the predicate is transformed to $x \leq 5 + \epsilon$. It is permissible to use a different definition of the function h with each application of an ϵ-transform, so long as $h(x,\epsilon)$ satisfies the requirements in 6.5 and is infinitesimal for ϵ infinitesimal. We denote the transformed version of P as P'.

6.2 A Safety Property Proof Procedure

For a predicate satisfying the grammar 6.2, we have developed a proof procedure for showing that P is a safety property of a well formed hybrid system. The procedure proceeds as follows:

1. Transform the predicate P to a new predicate P' under an ϵ-transformation.

2. Assuming the internal predicate $U_\epsilon(x_0, \epsilon)$ holds, where any positive infinitesimal ϵ satisfies the internal predicate $U_\epsilon(x_0, \epsilon)$ for limited x_0, prove the following formula holds:

$$U_\epsilon(x_0, \epsilon) \wedge P'(x_0) \rightarrow \begin{cases} P'(Y(x_0)) & B_Y(x_0) \\ P'(x_0 + f_i(x_0)\epsilon) & x_0 \in G_i \wedge \neg B_Y(x_0), \end{cases} \tag{6.6}$$

If the proof obligation in step 2 is fulfilled, then we may conclude that P is a safety property of the given well formed hybrid system; that is:

$$P(x_0) \rightarrow \forall_{t\geq 0}\ P(\phi(x_0, t)), \tag{6.7}$$

6.3 Soundness of the Safety Property Proof Procedure

In this section, we will show that, for a given well formed hybrid system, if we have relieved the proof obligation in step 2 of the safety proof procedure for a safety property, then the safety property holds.

Lemma 6.3.1. *Assuming the internal predicate $U_\epsilon(x_0, \epsilon)$ holds, where any positive infinitesimal ϵ satisfies the internal predicate $U_\epsilon(x_0, \epsilon)$ for limited x_0, if the formula 6.6 in step 2 of the safety proof procedure is shown to be true, then $\forall_{t\geq 0}\ P'(x_0) \rightarrow P'(\gamma^\epsilon(x_0, t))$, where γ^ϵ is as defined in 5.3.2 for a well formed hybrid system.*

Proof. The proof is by definition of γ^ϵ and induction over the well founded structure $\langle S, \prec \rangle$:

$$S = \{t - j\epsilon : j \in \mathbb{N} \wedge 0 \leq j \leq \lfloor t/\epsilon \rfloor\}, \tag{6.8}$$

where, for $t_a, t_b \in S$, $t_a \prec t_b$ iff $t_a < t_b$. □

We should point out that the proof does not make use of the predicate standard since U_ϵ is an internal predicate.

Theorem 6.3.2. *For a given internal predicate P satisfying grammar 6.2, and P' the ϵ-transform of P, if the proof obligation in step 2 of the safety proof procedure is fulfilled, then $\forall_{t \geq 0}\ P(x_0) \rightarrow P(\phi(x_0, t))$, for any $x_0 \in \mathbb{R}^n$.*

Proof. If $P'(x)$ satisfies 6.6, assuming internal predicate $U_\epsilon(x_0, \epsilon)$ holds, then by lemma 6.3.1, we can show that $\forall_{t \geq 0}\ P'(x_0) \rightarrow P'(\gamma^\epsilon(x_0, t))$. This holds for standard and nonstandard values for x_0 and t. We will assume that x_0 and t take on standard values only. For standard x_0 and t, $\gamma^\epsilon(x_0, t)$ is limited. By letting ϵ be a positive infinitesimal constant, by lemma 6.3.3, we have:

$$P'(x_0) \ \rightarrow \ P'(\gamma^\epsilon(x_0, t)) \ \rightarrow \ P(st(\gamma^\epsilon(x_0, t))) \tag{6.9}$$

By the standardization principle, $st(\gamma^\epsilon(x_0, t))$ is equal to the standard function $\phi(x_0, t)$ for all standard x_0 and t:

$$P'(x_0) \ \rightarrow \ P'(\gamma^\epsilon(x_0, t)) \ \rightarrow \ P(\phi(x_0, t)) \tag{6.10}$$

By lemma 6.3.4, we have $P'(x_0)$ iff $P(x_0)$ since we are assuming standard x_0, allowing us to rewrite 6.10:

$$P(x_0) \ \rightarrow \ P(\phi(x_0, t)). \tag{6.11}$$

Since $U_\epsilon(x_0, \epsilon)$ holds for limited x_0 and positive infinitesimal ϵ, and since ϵ is assumed to be a positive infinitesimal, x_0 is standard, then the assumption U_ϵ may be removed. Since 6.11 is internal, then by the transfer principle, we conclude it holds for all $t \geq 0$ and $x_0 \in \mathbb{R}^n$. We make the universal quantifier over non-negative t explicit:

$$P(x_0) \rightarrow \forall_{t \geq 0} \ P(\phi(x_0, t)). \tag{6.12}$$

□

In the previous proof, we made use of lemmas which we will now formally state and prove.

Lemma 6.3.3. *For a predicate P satisfying grammar 6.2, P' the ϵ-transform of P, limited $x \in \mathbb{R}^n$, and ϵ an infinitesimal, the following holds:*

$$P'(x) \ \rightarrow \ P(st(x)). \tag{6.13}$$

Proof. For a predicate P that satisfies the grammar 6.2, we can model its syntactic structure as a list, where each item in the list is a parenthesis, logical operator, a discrete predicate P_d, or atomic formula which appears in the list in the same order, when read left to right, as it does in P. We will call this the syntax list of a predicate. We observe that the syntax list of P varies form that of P' only in the inequalities; the remaining items are the same. Each inequality in the syntax list of P' is in one of the following forms:

$$\begin{aligned} |g(x)| &\leq h(x,\epsilon), \\ g(x) &\sim h(x,\epsilon). \end{aligned} \tag{6.14}$$

By **TH7**, we may apply standard part to both sides of each inequality in $P'(x)$. After applying standard part, we derive a new formula $P''(x)$, each of whose inequalities is in one of the following forms:

$$\begin{aligned} st(|g(x)|) &\leq st(h(x,\epsilon)), \\ st(g(x)) &\sim st(h(x,\epsilon)). \end{aligned} \tag{6.15}$$

We observe, by propositional logic, that:

$$P'(x) \ \rightarrow \ P''(x). \tag{6.16}$$

Since we are assuming that ϵ is infinitesimal, and since it is required by the ϵ-transform definition 6.1.1 that $h(x,\epsilon)$ be infinitesimal for ϵ infinitesimal, then $st(h(x,\epsilon)) = 0$. We observe that, for any real r,

$$st(|r|) \leq 0 \text{ iff } |st(r)| = 0 \text{ iff } st(r) = 0. \tag{6.17}$$

Using these observations, we may rewrite each inequality 6.15 to the following:

$$\begin{aligned} st(g(x)) &= 0, \\ st(g(x)) &\sim 0. \end{aligned} \tag{6.18}$$

Recall, by the grammar 6.2, each function g is continuous, or $st(g(x)) = g(st(x))$ for x limited. Since we are assuming x is limited, we may rewrite each inequality 6.18 as follows:

$$\begin{aligned} g(st(x)) &= 0, \\ g(st(x)) &\sim 0. \end{aligned} \tag{6.19}$$

By the requirement of the discrete predicate 6.3, and since x is limited, we note that every discrete predicate $P_d(x)$ in P'' can be rewritten to $P_d(st(x))$. We observe that every item in the syntax list of $P^{''}(x)$ is logically equal to the corresponding item in the syntax list of $P(st(x))$, whereby we may conclude:

$$P^{''}(x) \text{ iff } P(st(x)). \tag{6.20}$$

Therefore, by 6.16, we have:

$$P'(x) \;\rightarrow\; P(st(x)). \tag{6.21}$$

□

Lemma 6.3.4. *For an internal predicate P satisfying the grammar 6.2, and P' the ϵ-transform of P, $P(x)$ iff $P'(x)$ for standard x and infinitesimal ϵ.*

Proof. For a transform that yields an inequality of the form:

$$|g(x)| = h(x, \epsilon), \tag{6.22}$$

we have:

	$\lvert g(x)\rvert \leq h(x,\epsilon)$	
$\rightarrow$	$st(\lvert g(x)\rvert) \leq st(h(x,\epsilon))$	By **TH7**.
iff	$st(\lvert g(x)\rvert) \leq 0$	$h(x,\epsilon)$ is infinitesimal for infinitesimal ϵ.
iff	$\lvert st(g(x))\rvert \leq 0$	Property of standard part of absolute value function.
iff	$st(g(x)) = 0$	Property of absolute value function.
iff	$g(x) = 0$	Since x is standard and g is standard.

For the opposite direction, since by the definition of the transforms 6.5, it is required that $h(x, \epsilon) > 0$ for the case $g(x) = 0$, then $|g(x)| \leq h(x, \epsilon)$. Therefore, for standard x, we have $g(x) = 0$ iff $|g(x)| \leq h(x, \epsilon)$.

For a transform that yields an inequality of the form:

$$|g(x)| \sim h(x, \epsilon), \tag{6.23}$$

for a symbol $\sim$ denoting an inequality operator in the set $\{\leq, \geq\}$, we have:

$$\begin{array}{lll}
 & g(x) \sim h(x, \epsilon) & \\
\rightarrow & st(g(x)) \sim st(h(x, \epsilon)) & \text{By } \mathbf{TH7}. \\
\text{iff} & st(g(x)) \sim 0 & h(x, \epsilon) \text{ is infinitesimal} \\
 & & \text{for infinitesimal } \epsilon. \\
\text{iff} & g(x) \sim 0 & \text{Since } x \text{ is standard and } g \text{ is} \\
 & & \text{standard.}
\end{array}$$

For the opposite direction, for the case $g(x) \leq 0$, since by the definition of the transforms 6.5, it is required that $h(x, \epsilon) > 0$, so $g(x) \leq h(x, \epsilon)$. Therefore, we have $g(x) \leq h(x, \epsilon)$ iff $g(x) \leq 0$. By a similar proof for the case $g(x) \geq 0$, we may conclude that $g(x) \geq h(x, \epsilon)$ iff $g(x) \geq 0$.

Since applying the ϵ-transform to P modifies only the inequalities in P, and since, for standard x, each inequality in P is logically equal to its ϵ-transform, then $P(x)$ iff $P'(x)$. □

6.4 Safety Proof Method For Predicates On Non-Time Varying Functions

In this section, we will consider the case where the internal safety predicate P is in terms of some function g:

$$P(g(x)), \tag{6.24}$$

where $g : \mathbb{R}^n \mapsto \mathbb{R}$ is a standard continuous function in x. For this case, an alternative safety proof method may be carried out to that presented in section 6.2. To prove safety in this case, one need only show:

$$st\left(\frac{g(x) - g(\sigma(x,\epsilon))}{\epsilon}\right) = 0, \tag{6.25}$$

where:

$$\sigma(x_0) = \begin{cases} Y(x_0) & B_Y(x_0) \\ x_0 + f_1(x_0)\epsilon & x_0 \in G_1 \\ x_0 + f_2(x_0)\epsilon & x_0 \in G_2 \\ \cdots & \cdots \\ x_0 + f_m(x_0)\epsilon & x_0 \in G_m \end{cases} \tag{6.26}$$

A function satisfying 6.26 does not change with time, and hence called non-time varying.

Theorem 6.4.1. *For a well formed hybrid system and a property of the form $P(g(x))$, where P is an internal predicate, $g : \mathbb{R}^n \mapsto \mathbb{R}$ is a standard function and is continuous in x, the property $P(g(x))$ is a safety property if the formula 6.25 holds for positive infinitesimal ϵ and standard x.*

We should point out that P may be any predicate, not necessarily one satisfying 6.2.

Proof. Suppose we have that:

$$\|g(x) - g(\sigma(x,\epsilon))\| \leq \epsilon\,|\delta|, \tag{6.27}$$

for some positive ϵ and real $|\delta|$. By induction on $\gamma^{\epsilon}(x_0, t)$, it may be shown that:

$$\|g(x_0) - g(\gamma^{\epsilon}(x_0,t))\| \leq \lfloor t/\epsilon \rfloor\,\epsilon\,|\delta|. \tag{6.28}$$

Intuitively, this is the case since a change of $\epsilon\,\delta$ occurs in each step and the function γ^{ϵ} iterates for $\lfloor t/\epsilon \rfloor$ steps. Since $\lfloor t/\epsilon \rfloor\,\epsilon \leq t$, and since we derived this result from 6.27:

$$\|g(x) - g(\sigma(x,\epsilon))\| \leq \epsilon\,|\delta| \;\rightarrow \|g(x_0) - g(\gamma^{\epsilon}(x_0,t))\| \leq t\,|\delta|. \tag{6.29}$$

By the stated assumption, we have that:

$$st\left(\frac{g(x) - g(\sigma(x,\epsilon))}{\epsilon}\right) = 0. \tag{6.30}$$

We may conclude that, for some infinitesimal ϵ_2:

$$\left\|\frac{g(x) - g(\sigma(x,\epsilon))}{\epsilon}\right\| \leq |\epsilon_2|, \tag{6.31}$$

By algebra:

$$\|g(x) - g(\sigma(x,\epsilon))\| \leq \epsilon\,|\epsilon_2|, \tag{6.32}$$

The formula 6.29 holds for standard and nonstandard values. We will assume x_0 and t are standard in 6.29, and substitute ϵ_2 for δ:

$$\|g(x_0) - g(\gamma^{\epsilon}(x_0,t))\| \leq t\,|\epsilon_2|. \tag{6.33}$$

By **TH7**, we apply standard part to both sides of 6.33:

$$st(\|g(x_0) - g(\gamma^{\epsilon}(x_0,t))\|) \leq st(t\,|\epsilon_2|). \tag{6.34}$$

Since t is limited, ϵ_2 is infinitesimal, by the continuity of g, and the standardization principle:

$$\|g(x_0) - g(\phi(x_0,t))\| \leq 0. \tag{6.35}$$

By algebra:

$$g(x_0) = g(\phi(x_0,t)). \tag{6.36}$$

The above is true for standard x_0 and t. By the transfer principle, it is true for all $x_0 \in \mathbb{R}^n$ and real $t \geq 0$. Since $g(x)$ does not change with time, then $g(x) = a$, where a is a real constant for all non-negative time for the given initial condition. Therefore, for any predicate P:

$$P(g(x_0)) \rightarrow \forall_{t\geq 0}\, P(g(\phi(x_0,t))). \tag{6.37}$$

$\square$

The above method is useful in showing that equations with continuous terms, such as energy equations, do not change for a given system. For example, suppose the energy equation is $E(x) = a$, we may apply the proof method outlined in this section to show that it is a safety property.

The above method may be generalized to multiple continuous functions for a predicate Q of the form:

$$Q(g_1(x), g_2(x), \ldots, g_k(x)), \tag{6.38}$$

where each such function g_j, for $j \in [1..k]$, is a standard, continuous function which satisfies the property 6.25, with g_j substituted for g.

6.5 A Safety Proof of a Sample System

We demonstrate the proof of a safety property on a sample system. The vector field of the system is defined as follows:

$$F(x) = \begin{cases} 0 & x \geq 4 \\ 1 & x < 4 \end{cases} \tag{6.39}$$

We assume that the system consists of no assignments; that is, $B_Y = false$. We wish to prove the following safety property:

$$x \leq 4. \tag{6.40}$$

The above predicate does satisfy the grammar 6.2. We choose

$$U_\epsilon(x_0, \epsilon) \overset{\text{def.}}{=} 0 < \epsilon \leq 1. \tag{6.41}$$

We choose an ϵ-transform with $h(x, \epsilon) = \epsilon$ and apply it to the safety property 6.40:

$$x \leq 4 + \epsilon. \tag{6.42}$$

Now, we attempt to prove the formula 6.6:

$$P'(x_0) \to \begin{cases} P'(x_0) & x_0 \geq 4 \\ P'(x_0 + \epsilon) & x_0 < 4, \end{cases} \tag{6.43}$$

for $0 < \epsilon \leq 1$. We substitute the transformed predicate 6.42 for P' in 6.43:

$$x_0 \leq 4 + \epsilon \to \begin{cases} x_0 \ \leq \ 4 + \epsilon & x_0 \geq 4 \\ x_0 + \epsilon \ \leq \ 4 + \epsilon & x_0 < 4 \end{cases} \tag{6.44}$$

This results in two cases, depending on x_0. The case $x_0 \geq 4$ is trivially satisfied. The case $x_0 < 4$, results in the formula:

$$(x_0 < 4) \wedge (x_0 \ \leq \ 4 + \epsilon) \wedge (0 < \epsilon \leq 1) \ \rightarrow x_0 + \epsilon \ \leq \ 4 + \epsilon. \tag{6.45}$$

The above formula is true for all x_0 and ϵ. Therefore, by theorem 6.3.2:

$$\forall_{t \geq 0} \ x_0 \leq 4 \rightarrow \phi(x_0, t) \leq 4, \tag{6.46}$$

where ϕ is the solution defined in accordance to 5.3.2 for the system with vector field 6.39.

We should note that the ϵ-transform was essential in the success of this proof. If we substitute the original predicate P, $x < 4$, for P' in 6.43, we would attain the following case:

$$(x_0 < 4) \wedge (x_0 \ \leq \ 4) \wedge (0 < \epsilon \leq 1) \ \rightarrow x_0 + \epsilon \ \leq \ 4, \tag{6.47}$$

which is not true for all x_0 and ϵ.

6.6 Reasoning About Progress

While a safety property states that some predicate P remains true for all ensuing states of a system, a progress property states that the system will eventually reach a state satisfying some predicate Q. Formally, for a system solution ϕ, we define a progress property as follows:

$$P(x_0) \rightarrow \exists_{t \geq 0} \ Q(\phi(x_0, t)), \tag{6.48}$$

where P and Q are assumed to be internal. The formula 6.48 informally states that, starting from a state satisfying P, there exists a time $t \geq 0$ at which the predicate Q holds for the system solution ϕ. In the case where $P = true$, the formula 6.48 may be simplified to:

$$\exists_{t \geq 0}\, Q(\phi(x_0, t)). \tag{6.49}$$

As we did in the case of reasoning about safety, we will assume that the predicates P and Q satisfy the grammar 6.2.

Definition 6.6.1. For a progress property 6.48, we let P' and Q' be the ϵ-transforms of the predicates P and Q, respectively. For a system solution defined as in 5.3.2, we define a progress counter clk^ϵ using the predicates P' and Q' as follows:

$$clk^\epsilon(x_0) = \begin{cases} 0 & Q'(x_0) \vee \neg P'(x_0) \\ clk^\epsilon(Y(x_0)) & B_Y(x_0) \\ \epsilon + clk^\epsilon(x_0 + f_1(x_0)\epsilon) & x_0 \in G_1 \\ \epsilon + clk^\epsilon(x_0 + f_2(x_0)\epsilon) & x_0 \in G_2 \\ \dots & \dots \\ \epsilon + clk^\epsilon(x_0 + f_m(x_0)\epsilon) & x_0 \in G_m, \end{cases} \tag{6.50}$$

where the conditionals are evaluated from top to bottom.

Informally, for a given positive real ϵ, the progress counter clk^ϵ determines the time, in integer multiples of ϵ, at which γ^ϵ, defined in definition 5.3.2, reaches a state satisfying $Q'(x_0) \vee \neg P'(x_0)$. However, if we show, using the safety property proof procedure, that $P'(\gamma^\epsilon(x_0, t))$ holds for all $t \geq 0$, then clk^ϵ determines the time, in integer multiples of ϵ, at which γ^ϵ reaches a state satisfying $Q'(x_0)$. If γ^ϵ satisfies P' for all states but does not reach a state satisfying Q', then clk^ϵ is undefined.

We define a function σ which relates the current state with the state reached within a time duration ϵ:

$$\sigma(x_0) = \begin{cases} Y(x_0) & B_Y(x_0) \\ x_0 + f_1(x_0)\epsilon & x_0 \in G_1 \\ x_0 + f_2(x_0)\epsilon & x_0 \in G_2 \\ \dots & \dots \\ x_0 + f_m(x_0)\epsilon & x_0 \in G_m, \end{cases} \tag{6.51}$$

with the conditionals being evaluated from top to bottom.

6.7 A Progress Property Proof Procedure

As we did for a safety property, we will outline a proof procedure for showing some progress property is true for a well formed hybrid system. We assume that the property to be shown is in the form 6.48. The steps for the proof procedure are as follows:

1. Apply the ϵ-transform to internal predicates P and Q to yield predicates P' and Q', respectively.

2. Generate an internal predicate R. Apply the ϵ-transform to R to yield R'. Show that the following holds:

$$\exists_{t>0}\forall_{0\leq t'\leq t} R'(x) \rightarrow Q'(\gamma^\epsilon(x,t')), \tag{6.52}$$

   such that $st(t) > 0$.

3. Show that P' is a safety property for the given well formed system.

4. Let S be the set of ordinals less than the ordinal ϵ_0. Assuming an internal predicate $U_\epsilon(x_0, \epsilon)$ holds, where any positive infinitesimal ϵ satisfies the internal predicate $U_\epsilon(x_0, \epsilon)$ for limited x_0, generate a standard measure function $M : \mathbb{R}^n \times \mathbb{R} \mapsto S$ and show that the following are true:

   a. $\neg P'(x_0) \vee R'(x_0) \;\; \rightarrow \;\; M(x_0, \epsilon) = 0$,

   b. For x_0 satisfying $P'(x_0) \wedge \neg R'(x_0)$, the following holds:
   $$M(\sigma(x_0), \epsilon) \;<\; M(x_0, \epsilon). \tag{6.53}$$

   c. The measure M is such that, for limited x_0, and positive infinitesimal ϵ, $M(x_0, \epsilon)\epsilon$ is limited.

If the proof obligations outlined in the proof procedure are fulfilled, then we may conclude:
$$P(x_0) \rightarrow \exists_{t \geq 0}\; Q(\phi(x_0, t)). \tag{6.54}$$

In step 4, the standard measure function $M(x_0, \epsilon)$ evaluates to an ordinal. Since ϵ is a real number, we define what we mean by $M(x_0, \epsilon)\epsilon$ being limited.

Definition 6.7.1. Given an ordinal β less than ϵ_0, we represent β as

$$\beta = (\omega^{\alpha_2} x + \alpha_1). \tag{6.55}$$

We define the dimension of the ordinal recursively as 1 plus the sum of the dimension of α_1 and the dimension of α_2. A finite ordinal, or a natural number, is assumed to have dimension 1.

Definition 6.7.2. Let β be an ordinal where $\beta = M(x, \epsilon)$, for a standard measure function M, $M : \mathbb{R}^n \times \mathbb{R} \mapsto S$ and S the set of ordinals less than ϵ_0. We represent β as:

$$\beta = (\omega^{\alpha_2} x + \alpha_1), \tag{6.56}$$

where α_1 and α_2 are themselves ordinals and x is a finite ordinal. We say that $\beta\,\epsilon$ is limited iff the dimension of β is standard, $x\,\epsilon$ is limited and, recursively, $\alpha_1\,\epsilon$ and $\alpha_2\epsilon$ are limited. If β is a finite ordinal n, then $\beta\epsilon$ is limited iff $n\epsilon$ is limited.

We apply this definition to determine if $M(x_0, \epsilon)\epsilon$ is limited to example definitions of M. Suppose $M(x_0, \epsilon) = \lfloor 5/\epsilon^2 \rfloor$. Then $M(x_0, \epsilon)\epsilon = \lfloor 5/\epsilon^2 \rfloor \epsilon$, which is not limited for ϵ infinitesimal. As another example, consider:

$$M(x_0, \epsilon)\epsilon = (\omega^{\lfloor 3/\epsilon \rfloor} + \lfloor 4/\epsilon \rfloor)\epsilon. \tag{6.57}$$

Since ϵ, $\lfloor 3/\epsilon \rfloor \epsilon$, and $\lfloor 4/\epsilon \rfloor \epsilon$ are limited for positive ϵ, then 6.57 is limited.

6.8 Soundness of the Progress Property Proof Procedure

In proving the soundness of the proof procedure, we will consider to types of well formed hybrid systems 1) a system without assignments (B_Y is identically $false$) and 2) a system with assignments.

For the system without assignments, we will show that the proof procedure may be simplified by choosing a predicate R which is logically equal to Q, thereby eliminating step 2 of the progress proof procedure. Such a simplification is possible since, in this case, the solution is continuous.

For the case with assignments, step 2 is required and a separate proof is shown for soundness of this case.

First, we state a lemma that will be used in the proofs of the progress property proof procedure.

Lemma 6.8.1. *For a progress property:*

$$P(x_0) \to \exists_{t\geq 0}\ Q(\phi(x_0, t)), \tag{6.58}$$

let P' and Q' be the ϵ-transforms of P and Q, respectively. Let S be the set of ordinals less than ϵ_0. Suppose that P' satisfies 6.6. Suppose also there exists an internal predicate $U_\epsilon(x_0, \epsilon)$ that is satisfied for limited x_0 and positive infinitesimal ϵ, and a measure function $M : \mathbb{R}^n \times \mathbb{R} \mapsto S$ such that the following are true:

1. $U_\epsilon(x_0, \epsilon) \wedge (\,\neg P'(x_0) \vee Q'(x_0)\,) \;\to\; M(x_0, \epsilon) = 0,$

2. For x_0 and ϵ satisfying $U_\epsilon(x_0, \epsilon) \wedge P'(x_0) \wedge \neg Q'(x_0)$, the following holds:

$$M(\sigma(x_0), \epsilon) \;<\; M(x_0, \epsilon) \tag{6.59}$$

We may conclude that:

$$P'(x_0) \;\rightarrow\; \exists_t\; Q'(\gamma^\epsilon(x_0, t)). \tag{6.60}$$

Proof. Due to the requirement 6.59, the measure M decreases for each recursive call in clk^ϵ. Also, when $\neg P'(x_0) \vee Q'(x_0)$, the measure M is zero. Therefore, M may be used to show that the recursion defining clk^ϵ terminates. This implies the function clk^ϵ exists and is total. Since clk^ϵ terminates, it can be shown that:

$$P'(x_0) \;\rightarrow\; \neg P'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))) \vee Q'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))). \tag{6.61}$$

For a given positive real ϵ and $x_0 \in \mathbb{R}^n$, the property 6.61 can be shown by the definition of clk^ϵ, γ^ϵ, and induction on a well founded structure $\left\langle S', \prec^\epsilon_\gamma \right\rangle$:

$$S' = \{\gamma^\epsilon(x_0, \epsilon j) : j \in \mathbb{N} \wedge 0 \leq j \leq \lfloor clk^\epsilon(x_0)/\epsilon \rfloor\}, \tag{6.62}$$

where $x_a \prec^\epsilon_\gamma x_b$ iff $M(x_a, \epsilon) < M(x_b, \epsilon)$. Since P' is a safety property, then if the system starts with x_0 satisfying P', for all $t \geq 0$, $P'(\gamma^\epsilon(x_0, t))$. Therefore, 6.61 reduces to:

$$P'(x_0) \;\rightarrow\; Q'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))). \tag{6.63}$$

We can rewrite 6.63 as,

$$P'(x_0) \;\rightarrow\; \exists_t\; Q'(\gamma^\epsilon(x_0, t)), \tag{6.64}$$

since we know 6.64 holds for $t = clk^\epsilon(x_0)$. □

By lemma 6.8.1, we have shown that, if P is a safety property, then for x_0 that satisfies P, there exists a time $clk^\epsilon(x_0)$ at which the solution satisfies Q'. However, this time $clk^\epsilon(x_0)$ is not necessarily standard.

6.8.1 Soundness of Progress Proof Procedure: System Without Assignments

In this section, we will show that, for a given well formed hybrid system, if we have relieved the proof obligations of the the progress proof procedure for a progress property of the form 6.48, then the progress property holds.

Theorem 6.8.2. *Given a well formed hybrid system without assignments (B_Y is identically $false$), for internal predicates P, Q, and R satisfying 6.2, where Q iff R, if we fulfill the proof obligations of steps 3 and 4 of the progress proof procedure, then the progress property 6.48 holds.*

Proof. By applying the ϵ-transform to P and Q, we attain P' and Q', respectively. If we have fulfilled the proof obligation of step 3, then P is a safety property; that is:

$$P(x_0) \rightarrow \forall_t \; P(\phi(x_0, t)). \tag{6.65}$$

The measure M used in step 4 assures that the function clk^ϵ does terminate. In addition, the measure $M(x_0, \epsilon)$ decreases by at least one in each recursive call of clk^ϵ. Therefore, $\lfloor clk^\epsilon(x_0)/\epsilon \rfloor < M(x_0, \epsilon)\epsilon$ or $\lfloor clk^\epsilon(x_0)/\epsilon \rfloor = M(x_0, \epsilon)\epsilon$. Since, by step 4, it is required that $M(x_0, \epsilon)\epsilon$ be limited, then $clk^\epsilon(x_0)$ is limited. Hence, by the standardization principle, there exists a standard function clk such that for standard x_0, $clk(x_0) = st(clk^\epsilon(x_0))$.

If we have fulfilled the proof obligation in step 4, then, by lemma 6.8.1, we have that:

$$P'(x_0) \rightarrow Q'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))). \tag{6.66}$$

The formula 6.66 is true for all values of x_0. We will assume x_0 takes on standard values only. Since ϵ is a positive infinitesimal and P satisfies grammar 6.2, then by lemma 6.3.4, we have that:

$$P'(x_0) \text{ iff } P(x_0). \tag{6.67}$$

By lemma 6.3.3, since Q satisfies grammar 6.2, we have that

$$Q'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))) \rightarrow Q(st(\gamma^\epsilon(x_0, clk^\epsilon(x_0)))) \tag{6.68}$$

Since no assignments occur in the system, the function γ^ϵ is such that:

$$\|\gamma^\epsilon(x_0, t_1) - \gamma^\epsilon(x_0, t_2)\| \leq (t_2 - \epsilon \lfloor t_1/\epsilon \rfloor)(N + NLt_2e^{Lt_2} + Le^{Lt_2}\|x_0\|), \tag{6.69}$$

where $t_2 \geq t_1$ and N, L are standard constants. For t_2 limited, $t_1 \simeq t_2$, and since x_0 is standard, by 6.69 we have that:

$$st(\gamma^\epsilon(x_0, t_1)) = st(\gamma^\epsilon(x_0, t_2)). \tag{6.70}$$

Therefore, by the standardization principle, we may conclude that:

$$st(\gamma^\epsilon(x_0, clk^\epsilon(x_0))) = st(\gamma^\epsilon(x_0, st(clk^\epsilon(x_0)))) = \phi(x_0, clk(x_0)). \tag{6.71}$$

By 6.68, we have:

$$Q'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))) \rightarrow Q(st(\gamma^\epsilon(x_0, clk^\epsilon(x_0)))) = Q(\phi(x_0, clk(x_0))). \tag{6.72}$$

By 6.66, 6.67, and 6.72, we have that:

$$P(x_0) \rightarrow Q(\phi(x_0, clk(x_0))). \tag{6.73}$$

Since $U_\epsilon(x_0, \epsilon)$ holds for limited x_0 and positive infinitesimal ϵ, and since ϵ is assumed to be a positive infinitesimal and x_0 is standard, then the assumption U_ϵ may be removed. The formula 6.73 holds for standard x_0 and, since it is internal, by the transfer principle, holds for all $x_0 \in \mathbb{R}^n$. Since Q holds at $clk(x_0)$, we may write:

$$P(x_0) \rightarrow \exists_t Q(\phi(x_0, t)). \tag{6.74}$$

□

6.8.2 Soundness of Progress Proof Procedure: System With Assignments

In this section, we will consider a well formed hybrid system with assignments (B_Y is not identically $false$). This is similar to the proof presented in the previous subsection, with the exception that we cannot assume that the solution is continuous with respect to time.

Theorem 6.8.3. *Given a well formed hybrid system, for internal predicates P, Q, and R satisfying 6.2, if we fulfill the proof obligations of steps 2, 3, and 4 of the progress proof procedure, then the progress property 6.48 holds for the solution of the well formed hybrid system.*

Proof. By applying the ϵ-transform to P, Q, and R, we attain P', Q', and R', respectively. If we have fulfilled the proof obligation of step 3, then P is a safety property; that is:

$$P(x_0) \to \forall_t \; P(\phi(x_0, t)). \tag{6.75}$$

If we have fulfilled the proof obligation in step 4, then, by lemma 6.8.1, we have that:

$$P'(x_0) \to R'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))). \tag{6.76}$$

The measure M used in step 4 assures that the function clk^ϵ does terminate. In addition, the measure $M(x_0, \epsilon)$ decreases by at least one in each recursive call of clk^ϵ. Therefore, $\lfloor clk^\epsilon(x_0)/\epsilon \rfloor < M(x_0, \epsilon)\epsilon$ or $\lfloor clk^\epsilon(x_0)/\epsilon \rfloor = M(x_0, \epsilon)\epsilon$. Since, by step 4, it is required that $M(x_0, \epsilon)\epsilon$ be limited, then $clk^\epsilon(x_0)$ is limited. By fulfilling the proof obligation of step 2, there must exist some standard time t such that $clk^\epsilon(x_0) \leq t$ and such that:

$$R'(\gamma^\epsilon(x_0, clk^\epsilon(x_0))) \to \exists_t^{st} Q'(\gamma^\epsilon(x_0, t)). \tag{6.77}$$

The formula 6.77 is true for all values of x_0. We will assume x_0 takes on standard values only. Since ϵ is a positive infinitesimal, P satisfies grammar 6.2, then by lemma 6.3.4, we have that:

$$P'(x_0) \text{ iff } P(x_0). \tag{6.78}$$

By lemma 6.3.3, since Q satisfies grammar 6.2, we have that

$$Q'(\gamma^\epsilon(x_0, t)) \to Q(st(\gamma^\epsilon(x_0, t))) \tag{6.79}$$

Since x_0 is standard, by the standardization principle and definition of the function ϕ, we have that:

$$\exists_t^{st} Q(st(\gamma^\epsilon(x_0, t))) = \exists_t^{st} Q(\phi(x_0, t)). \tag{6.80}$$

By 6.76 through 6.80:

$$P(x_0) \rightarrow \exists_t^{st} Q(\phi(x_0, t)). \tag{6.81}$$

Since Q is internal and ϕ is standard, by the dual form of the transfer principle,

$$P(x_0) \rightarrow \exists_t Q(\phi(x_0, t)). \tag{6.82}$$

Since $U_\epsilon(x_0, \epsilon)$ holds for limited x_0 and positive infinitesimal ϵ, and since ϵ is assumed to be a positive infinitesimal and x_0 is standard, then the assumption U_ϵ may be removed. □

6.9 A Progress Proof of a Sample System

We demonstrate the proof of a progress property on a sample system. The vector field of the system is defined as follows:

$$F(x) = \begin{cases} -1 & x > 0 \\ 0 & x = 0 \\ 1 & x < 0 \end{cases} \tag{6.83}$$

We assume that the system consists of no assignments; that is, $B_Y = false$. We wish to prove the following progress property:

$$\exists_t \, x(t) = 0. \tag{6.84}$$

In carrying out the steps of the proof procedure, we will assume that 1) the predicate P for this example is identically *true*, 2) Q iff R, and 3) Q' iff R'. We propose the progress measure:

$$M(x, \epsilon) = \begin{cases} 0 & |x| \leq \epsilon \\ \left\lfloor \dfrac{|x|}{\epsilon} \right\rfloor & \text{otherwise.} \end{cases} \tag{6.85}$$

For step 1 of the progress proof procedure, we take the ϵ-transform of Q to attain Q':

$$|x| \leq \epsilon. \tag{6.86}$$

For step 2 of the progress proof procedure, since Q' iff R', then if we show that Q' is a safety property for γ^ϵ, the proof obligation for this step is satisfied. For step 3 of the progress proof procedure, since P is identically true, then P is a safety property. For step 4, of the proof procedure, we note that, since P is true, and by the definition of Q' and M:

$$|x| \leq \epsilon \quad \rightarrow \quad M(x, \epsilon) = 0. \tag{6.87}$$

Satisfying step 4.a. For step 4.b, we must show that M decreases for $|x| > \epsilon$. We choose $U_\epsilon(x_0, \epsilon) \overset{\text{def.}}{=} 0 < \epsilon \leq 1$.

$$0 < \epsilon \leq 1 \wedge |x| > \epsilon \quad \rightarrow \quad M(\sigma^\epsilon(x), \epsilon) < M(x, \epsilon). \tag{6.88}$$

The proof obligation results in several case splits depending on the value of x and $\sigma^\epsilon(x)$. We show some of these cases here. We consider the case $x > 0$, $|x| > \epsilon$, $|x - \epsilon| > \epsilon$:

$$0 < \epsilon \leq 1 \wedge |x| > \epsilon \quad \rightarrow \quad \left\lfloor \frac{|x - \epsilon|}{\epsilon} \right\rfloor < \left\lfloor \frac{|x|}{\epsilon} \right\rfloor. \tag{6.89}$$

By algebra, this reduces to

$$0 < \epsilon \leq 1 \wedge |x| > \epsilon \quad \rightarrow \quad \left\lfloor \frac{|x|}{\epsilon} - 1 \right\rfloor < \left\lfloor \frac{|x|}{\epsilon} \right\rfloor, \tag{6.90}$$

which is true.

6.10 Asymptotic Stability

A control system is said to be asymptotically stable if, as time t increases, the solution approaches the origin and remains near the origin.

Definition 6.10.1. For a well formed hybrid system, the solution is asymptotically stable if, for any real positive number α, the following criteria are satisfied:

1. $\exists_t \; \|\phi(x_0, t)\| \leq \alpha$,
2. $\|x_0\| \leq \alpha \;\rightarrow\; \forall_t \; \|\phi(x_0, t)\| \leq \alpha$.

Therefore, as can be ascertained by its definition, the proof of asymptotic stability may be decomposed into a progress proof and stability proof. We should point out that, due to the introduction of the variable α, we would add the predicate $\alpha > 0$ conjuncted with zero or more instances of predicates of the form $\alpha > h(x, \epsilon)$, where $h(x, \epsilon)$ is derived from the proof context and is infinitesimal for x limited, and ϵ infinitesimal. This conjunct is added to the hypothesis of every proof obligation in the stability and progress proof procedures.

We note that the properties in definition 6.10.1 are internal and true for any $\alpha > 0$. We choose a positive infinitesimal α. We modify the progress property, applying **TH7** and assuming α is a positive infinitesimal:

$$\exists_t \; st(\|\phi(x_0, t)\|) \leq st(\alpha). \tag{6.91}$$

Hence,

$$\exists_t \; st(\phi(x_0,t)) = \mathbf{0}. \tag{6.92}$$

We note that the new property has a standard part applied to the solution. Hence, it is external. It is possible that the property holds for nonstandard, infinitely large t only. Even then, while the standard part of the solution will reach zero, the solution itself may never reach zero.

We should point out that applying the transfer principle, or the dual form of transfer, would not be useful for the existentially quantified progress formula. If we assume that x_0 and α are universally quantified, then applying the dual form of transfer would require that both α and x_0 be standard, since they both occur in the existentially quantified formula. However, the requirement that α be standard would defeat our effort of assigning it an infinitesimal value.

For the safety property, we will similarly assume that α is positive infinitesimal. The safety property holds for all x and α. We will restrict ourselves to standard x and positive infinitesimal α, and assume the universal quantifier ranges over standard numbers only. Hence, we modify the safety property as follows:

$$\|x_0\| \leq \alpha \;\rightarrow\; \forall_t^{\mathrm{st}} \; \|\phi(x_0,t)\| \leq \alpha. \tag{6.93}$$

For standard y and infinitesimal α, the following property holds:

$$\|y\| \leq \alpha \text{ iff } y = \mathbf{0}. \tag{6.94}$$

Applying the property in 6.94 to 6.93, and noting that ϕ is a standard function:

$$x_0 = \mathbf{0} \;\rightarrow\; \forall_t^{\mathrm{st}} \phi(x_0, t) = \mathbf{0}. \tag{6.95}$$

Applying transfer, as the formula is internal:

$$x_0 = \mathbf{0} \;\rightarrow\; \forall_t \phi(x_0, t) = \mathbf{0}. \tag{6.96}$$

6.11 A Modified Lyapunov Stability Method

The Lyapunov method of determining the asymptotic stability of a Newtonian solution to a differential equation is discussed in section 3.4. We slightly modify the definition of a required Lyapunov function to show stability for a system. Whereas the Lyapunov method shows asymptotic approach, this modified method shows progress, within limited time t, to some state x where $\|x\| \leq \alpha > 0$.

6.11.1 Applying Lyapunov: Progress

For a well formed hybrid system where the vector field is Lipschitz continuous and the assignment predicate B_Y is identically $false$, we may use the modified Lyapunov function in our progress proof procedure to show that the solution reaches, within some limited time t, a state x where $\|x\| \leq \alpha > 0$.

Definition 6.11.1. A modified Lyapunov function V is a standard function which has continuous first partial derivatives and is such that there exist standard real values $\alpha > 0$ and $\gamma > 0$ where:

$$\begin{array}{ll} \gamma < V(x) & \text{for } \|x\| > \alpha, \\ 0 < V(x) \leq \gamma & \text{for } 0 < \|x\| \leq \alpha, \text{ and} \\ V(x) = 0 & \text{otherwise.} \end{array}$$

Based on the definition 6.11.1, every modified Lyapunov function is a Lyapunov function, but the converse is not true. For the modified Lyapunov stability method, we require that the modified Lyapunov function satisfy the following derivative with respect to the defined system, for some standard real $\beta < 0$:

$$\begin{array}{ll} \dot{V}(x) < \beta & \text{for } \|x\| > \alpha, \\ \dot{V}(x) < 0 & \text{for } 0 < \|x\| \leq \alpha, \text{ and} \\ \dot{V}(x) = 0 & \text{otherwise.} \end{array} \tag{6.97}$$

We use the modified Lyapunov function V to formulate a measure M:

$$M(x, \epsilon) = \begin{cases} 0 & \|x\| \leq \alpha \\ \left\lfloor \dfrac{2V(x)}{|\beta|\,\gamma\,\epsilon} \right\rfloor & \text{otherwise.} \end{cases} \tag{6.98}$$

Theorem 6.11.1. *Given a modified Lyapunov function V satisfying definition 6.11.1 and meeting the requirements in 6.97, then the measure M in 6.98 is decreasing for $\|x\| > \alpha$.*

Proof. For this proof, we will assume that the predicate $U_\epsilon(x_0, \epsilon)$ is defined as follows:

$$U_\epsilon(x_0,\epsilon) \overset{\text{def.}}{=} \forall_{x\in\{\gamma^\epsilon(x_0,u):0\leq u\leq \frac{2V(x_0)}{|\beta|\,\gamma}\}}\ \forall_{\zeta_{x_1}} \dots \forall_{\zeta_{x_n}}\ (0<\epsilon<1)\wedge$$
$$(\textstyle\bigwedge_{i\in[1..n]}(\|x-\zeta_{x_i}\|\leq\|f(x)\|\,\epsilon)\wedge\|x\|>\alpha\to\sum_{i=1}^n\frac{\partial V}{\partial x_i}(\zeta_{x_i})\,f_{x_i}(x)\ <\ \frac{\beta}{2})\wedge$$
$$(\textstyle\bigwedge_{i\in[1..n]}(\|x-\zeta_{x_i}\|\leq\|f(x)\|\,\epsilon)\wedge 0\leq\|x\|\leq\alpha\to\sum_{i=1}^n\frac{\partial V}{\partial x_i}(\zeta_{x_i})\,f_{x_i}(x)\ <1). \tag{6.99}$$

We should point out, as required by the proof procedures, $U_\epsilon(x_0, \epsilon)$ must hold for limited x_0 and positive infinitesimal ϵ. This is shown in lemma 6.11.3. We assume that $|\beta| < 1$, and $|\gamma| < 1$. If they are not so, we may adjust their values without violating the required inequalities $\beta < 0$ and $\gamma > 0$. We first consider the case where $\|\sigma(x, \epsilon)\| > \alpha$. In this case, by the definition of M, we have to show that:

$$\left\lfloor\frac{2V(\sigma(x,\epsilon))}{|\beta|\,\gamma\,\epsilon}\right\rfloor<\left\lfloor\frac{2V(x)}{|\beta|\,\gamma\,\epsilon}\right\rfloor. \tag{6.100}$$

For reals a and b, it may be shown that the floor function has the following property:

$$a\leq b-1\ \to\ \lfloor a\rfloor<\lfloor b\rfloor \tag{6.101}$$

Using the property of floor 6.101, we focus on proving:

$$\frac{2V(\sigma(x,\epsilon))}{|\beta|\,\gamma\,\epsilon}\leq\frac{2V(x)}{|\beta|\,\gamma\,\epsilon}-1. \tag{6.102}$$

By the definition of σ, we have:

$$\frac{2V(x+f(x)\epsilon)}{|\beta|\,\gamma\,\epsilon}-\frac{2V(x)}{|\beta|\,\gamma\,\epsilon}\leq-1. \tag{6.103}$$

By algebra:

$$\frac{2V(x+f(x)\epsilon)-2V(x)}{|\beta|\,\gamma\,\epsilon} \le -1. \tag{6.104}$$

By a vector form of the mean value theorem shown later in lemma 6.11.2, we can show that:

$$V(x+f(x)\epsilon)-V(x)=\sum_{i=1}^{n}\frac{\partial V}{\partial x_i}(\zeta_{x_i})f_{x_i}(x)\epsilon, \tag{6.105}$$

where the i^{th} component, c, of each vector ζ_{x_i} satisfies:

$$\begin{aligned} &x_i \le c \le x_i + f_{x_i}(x)\epsilon \quad \text{for } f_{x_i}(x)\epsilon \ge 0, \text{ and} \\ &x_i \ge c \ge x_i + f_{x_i}(x)\epsilon \quad \text{otherwise,} \end{aligned} \tag{6.106}$$

where x_i and f_{x_i} are the i^{th} components of x and f, respectively. Using this mean value theorem result, we rewrite 6.104:

$$\frac{2\sum_{i=1}^{n}\frac{\partial V}{\partial x_i}(\zeta_{x_i})f_{x_i}(x)\epsilon}{|\beta|\,\gamma\,\epsilon} \le -1. \tag{6.107}$$

By 6.106, every vector ζ_{x_i} satisfies $\|x-\zeta_{x_i}\|\le\|f(x)\|\ \epsilon$. By the definition of U_ϵ, and since by the proof procedure we assume that U_ϵ holds, we may conclude that:

$$\sum_{i=1}^{n}\frac{\partial V}{\partial x_i}(\zeta_{x_i})f_{x_i}(x) < \frac{\beta}{2}. \tag{6.108}$$

Since $0<|\gamma|<1$, and $\beta<0$, then, by 6.108, the formula 6.107 holds. This completes the proof for the case $\|\,\sigma(x,\epsilon)\,\| > \alpha$. We now consider the case where $\|\sigma(x,\epsilon)\|\le\alpha$. By the definition of M and a property of floor 6.101, this reduces to:

$$1 \le \frac{2V(x)}{|\beta|\,\gamma\,\epsilon}. \tag{6.109}$$

Since $0<\gamma<V(x)$, $0<|\beta|<1$, and $0<\epsilon<1$, then the above holds. □

6.11.2 Applying Lyapunov: Safety

To show stability, we must show a progress property as well as a safety property. We have demonstrated a decreasing measure to be used for the progress property. Now we state a safety property whereby the solution remains bounded. We require that, upon reaching a state x satisfying $\|x\| \leq \alpha$, all ensuing states must satisfy $\|x\| \leq \alpha$.

By definition 6.11.1 of the modified Lyapunov function, it may be shown that:

$$V(x) \leq \gamma \quad \text{iff} \quad \|x\| \leq \alpha. \tag{6.110}$$

By the safety property proof procedure, we attempt to show that the following holds:

$$V(x) \leq \gamma \quad \rightarrow \quad V(\sigma(x,\epsilon)) \leq \gamma. \tag{6.111}$$

Proof. We reuse the predicate $U_\epsilon(x_0,\epsilon)$ presented in formula 6.99. By the definition of σ and using the vector form of the mean value theorem as shown in lemma 6.11.2, we have:

$$V(\sigma(x,\epsilon)) = V(x) + \sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(\zeta_{x_i})\, f_{x_i}(x)\epsilon. \tag{6.112}$$

We will apply the ϵ-transform and show that $V(x) \leq \gamma + \epsilon$. We will split the proof into two cases based on the value of $V(x)$:

$$0 \leq V(x) \leq \gamma \ \text{ and } \ \gamma < V(x) \leq \gamma + \epsilon.$$

For the case $0 \leq V(x) \leq \gamma$, by the definition of U_ϵ, assuming $0 \leq \|x\| \leq \alpha$, and 6.110, we have that:

$$\sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(\zeta_{x_i})\, f_{x_i}(x) < 1. \tag{6.113}$$

By 6.113, $0 < \epsilon$, the assumption that $V(x) \leq \gamma$ and algebra:

$$V(x) + \sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(\zeta_{x_i})\, f_{x_i}(x)\epsilon \leq \gamma + \epsilon. \tag{6.114}$$

By 6.112 and 6.114:

$$V(\sigma(x, \epsilon)) \leq \gamma + \epsilon. \tag{6.115}$$

For the case $\gamma < V(x) \leq \gamma + \epsilon$, then by 6.110, we have $\alpha < \| x \|$. By the definition of U_ϵ, we have that:

$$\sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(\zeta_{x_i})\, f_{x_i}(x) < \frac{\beta}{2}. \tag{6.116}$$

Since $\beta < 0$,

$$V(x) + \sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(\zeta_{x_i})\, f_{x_i}(x)\epsilon \leq \gamma + \epsilon. \tag{6.117}$$

From which we conclude:

$$V(\sigma(x, \epsilon)) \leq \gamma + \epsilon. \tag{6.118}$$

□

By 6.110, we may conclude that $\|x\| \leq \alpha$ is also a safety property of the system.

6.11.3 Lemmas Needed for Showing Modified Lyapunov Stability

In the proof of the decreasing modified Lyapunov measure theorem 6.11.1, we made use of lemmas which we state and prove here. We begin with the following lemma which extends the mean value theorem to a vector case.

Lemma 6.11.2. *For a function $V : \mathbb{R}^n \mapsto \mathbb{R}$, and for x, δ, and ζ in $\mathbb{R}^n$:*

$$V(x+\delta) - V(x) = \sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(\zeta_{x_i})\delta_i, \tag{6.119}$$

where the i^{th} component, c, of each vector ζ_{x_i} satisfies:

$$\begin{aligned} &x_i \le c \le x_i + \delta_i \quad \textit{for } \delta_i \ge 0, \textit{ and} \\ &x_i \ge c \ge x_i + \delta_i \quad \textit{otherwise}, \end{aligned} \tag{6.120}$$

where x_i and δ_i are the i^{th} components of x and δ, respectively.

Proof. We observe that if we vary only one variable, x_i, in the vector x, the following holds by the single variable version of the mean value theorem:

$$\begin{aligned} &V(x_1, x_2, \ldots, x_i + \delta_i, \ldots, x_{n-1}, x_n) - \\ &\qquad V(x_1, x_2, \ldots, x_i, \ldots, x_{n-1}, x_n) = \tfrac{\partial V}{\partial x_i}(\zeta)\delta_i, \end{aligned} \tag{6.121}$$

where all components of ζ equal to the corresponding components of x, with exception to ζ_i, where $x_i \le \zeta_i \le x_i + \delta_i$ for $\delta_i \ge 0$, and $x_i \ge \zeta_i \ge x_i + \delta_i$ otherwise. We shall denote ζ_{x_i} as the vector yielded by the mean value theorem when varying x_i.

By repeatedly applying 6.121, we have, by adding and subtracting like terms:

$$V(x_1+\delta_1, x_2+\delta_2, \dots, x_n+\delta_n) - V(x_1, x_2, \dots, x_n) =$$

$$\begin{gathered} V(x_1+\delta_1, x_2, \dots, x_n) - V(x_1, x_2, \dots, x_n) + \\ V(x_1+\delta_1, x_2+\delta_2, \dots, x_n) - V(x_1+\delta_1, x_2, \dots, x_n) + \cdots + \\ V(x_1+\delta_1, x_2+\delta_2, \dots, x_n+\delta_n) - V(x_1+\delta_1, x_2+\delta_2, \dots, x_{n-1}+\delta_{n-1}, x_n) = \end{gathered}$$

$$\tfrac{\partial V}{\partial x_1}(\zeta_{x_1})\delta_1 + \tfrac{\partial V}{\partial x_2}(\zeta_{x_2})\delta_2 + \cdots + \tfrac{\partial V}{\partial x_n}(\zeta_{x_n})\delta_n = \textstyle\sum_{i=1}^{n} \tfrac{\partial V}{\partial x_i}(\zeta_{x_i})\delta_i. \tag{6.122}$$

□

The following lemma states that the definition of U_ϵ in 6.99 defined for the modified Lyapunov method is such that U_ϵ holds for limited x_0 and positive infinitesimal ϵ.

Lemma 6.11.3. *For $U_\epsilon(x_0, \epsilon)$ defined in 6.99, and for a modified Lyapunov function V satisfying 6.97:*

$$limtd(x_0) \wedge st(\epsilon) = 0 \wedge \epsilon > 0 \ \rightarrow \ U_\epsilon(x_0, \epsilon). \tag{6.123}$$

Proof. For the case where the implications in the definition of U_ϵ are true, then any infinitesimal ϵ satisfies U_ϵ. Now we show that each implication is true for ϵ infinitesimal and x_0 limited. For $0 \leq u \leq \frac{2V(x_0)}{|\beta|\gamma}\}$, then u is limited, since V is standard and continuous, x_0 is limited, $|\beta|$ and γ are standard and positive. Since $\gamma^\epsilon(x_0, u)$ is limited for x_0 and u limited, and since $x = \gamma^\epsilon(x_0, u)$, then x is limited. For the case where $\|x\| > \alpha$, we note that, for limited $x \in \mathbb{R}^n$,

$y \in \mathbb{R}^n$, and infinitesimal ϵ, we have by **TH7**:

$$\|x - y\| \leq f(x)\epsilon \;\rightarrow\; st(\|x - y\|) \leq st(f(x)\epsilon) \;\text{ iff }\; st(x) = st(y). \tag{6.124}$$

By the definition of V, we have that:

$$\dot{V}(x) < \beta, \tag{6.125}$$

By the chain rule of multivariable calculus:

$$\nabla V(x) \bullet f(x) < \beta. \tag{6.126}$$

By applying **TH7** and by the definition of the gradient:

$$st\left(\sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(x)\, f_{x_i}(x)\right) \leq st(\beta). \tag{6.127}$$

Since x is limited, by the continuity of f and since ∇V is continuous by the requirement that a Lyapunov function V have continuous first partial derivatives:

$$\sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(st(x))\, f_{x_i}(st(x)) \leq st(\beta) < \frac{\beta}{2}. \tag{6.128}$$

Applying the contrapositive of **TH7** to 6.128 and by 6.124, we may conclude that $st(x) = st(\zeta_{x_i})$ for $\|x - \zeta_{x_i}\| \leq \|f(x)\|\, \epsilon$, and since β is standard:

$$\sum_{i=1}^{n} \frac{\partial V}{\partial x_i}(\zeta_{x_i})\, f_{x_i}(x) < \frac{\beta}{2}. \tag{6.129}$$

A similar proof may be carried out for the case $0 \leq \|x\| \leq \alpha$. Therefore, $U_\epsilon(x_0, \epsilon)$ does hold for limited x_0 and infinitesimal ϵ. $\square$

6.12 Showing Asymptotic Approach using Lyapunov

In the previous section, we showed that, using a modified Lyapunov function in the definition of a measure of the progress proof procedure, we may show that the solution does reach a stated x whereby $\|x\| \leq \alpha > 0$. In this section, we will extend the approach by showing asymptotic approach to zero.

We start by formally stating the findings from the previous section regarding the modified Lyapunov method:

$$\forall_x \left(\begin{array}{l} (\gamma < V(x) \;\wedge\; \dot{V}(x) < \beta \wedge\; \|x\| > \alpha) \;\vee \\ (0 < V(x) \leq \gamma \;\wedge\; \dot{V}(x) < 0 \;\wedge\; 0 < \|x\| \leq \alpha) \;\vee \\ (V(x) = 0 \;\wedge\; \dot{V}(x) = 0 \wedge\; \|x\| = \mathbf{0}) \end{array} \right) \;\rightarrow\; \exists_t \; \|\phi(x_0, t)\| \leq \alpha. \tag{6.130}$$

We note that, if x is a standard real vector, y is any real vector, and α, β, and γ are infinitesimal real numbers, we can show:

$$\begin{array}{rcl} V(x) > \gamma & \text{iff} & V(x) > 0, \\ 0 < \|x\| \leq \alpha & \text{iff} & false, \\ \dot{V}(x) < \beta & \text{iff} & \dot{V}(x) < 0, \\ \|x\| > \alpha & \text{iff} & \|x\| > 0 \\ \|y\| \leq \alpha & \rightarrow & st(y) = \mathbf{0}. \end{array} \tag{6.131}$$

By the transfer principle applied only to the hypothesis of 6.130, and applying the properties shown in 6.131 to 6.130, we have, for $x_0 \in \mathbb{R}^n$:

$$\forall_x^{\text{st}} \left(\begin{array}{l} (V(x) > 0 \;\wedge\; \dot{V}(x) < 0 \wedge\; \|x\| > 0) \quad \vee \\ (V(x) = 0 \;\wedge\; \dot{V}(x) = 0 \;\wedge\; x = \mathbf{0}) \end{array} \right) \;\rightarrow\; \exists_t st(\phi(x_0, t)) = \mathbf{0}. \tag{6.132}$$

Again applying the transfer principle only to the hypothesis of 6.132, we have for $x_0 \in \mathbb{R}^n$:

$$\forall_x \left(\begin{array}{ll} (V(x) > 0 \ \wedge \ \dot{V}(x) < 0 \wedge \ \|x\| > 0) & \vee \\ (V(x) = 0 \ \wedge \ \dot{V}(x) = 0 \ \wedge \ x = \mathbf{0}) & \end{array} \right) \quad \rightarrow \quad \exists_t st(\phi(x_0, t)) = \mathbf{0}. \tag{6.133}$$

We note that by the result in 6.133, the solution ϕ may never reach $\mathbf{0}$, but its standard part will, thereby capturing the notion of asymptotic approach. It can also be shown, based on the modified Lyapunov safety property, that once $st(\phi(x_0, t)) = 0$, then for all real $u \geq t$, $st(\phi(x_0, u)) = 0$.

6.13 Lyapunov Method and Nonstandard Analysis

The result of using the Lyapunov function directly to show asymptotic stability and the modified Lyapunov function for reasoning about progress demonstrates how theory already established for internal arithmetic may be used to reason about models defined using nonstandard analysis. In a similar fashion, theory developed for digital systems, for example from model checking or theorem proving approaches, may also be used in establishing the proof obligations of the safety and progress proof procedures. This is a powerful attribute of nonstandard analysis allowing the verifier to reason using those applicable theories, discrete or continuous, in fulfilling the proof obligations.

6.14 Reasoning About Open Predicates

The grammar 6.2 which we have defined disallows the use of open predicates in the definition of safety and progress properties. Intuitively, this is a result of the fact that there does not exist any least or greatest element which satisfies an open predicate. For example, for the open predicate $x > 5$, there is no least number greater than 5. Hence, we require the user to define some notion of "close enough" by converting the predicate into a closed predicate, such as $x \geq 5 + 10^{-10}$. An alternative approach is to use a variable, α, representing some positive real value, as was done in the case of reasoning about stability. This variable α would then be used in converting open predicates into closed predicates. For example, the open predicate $x > 5$ can be converted to the closed predicate $x \geq 5 + \alpha$. The proof procedure would proceed as defined. After the completion of the proof procedure, the resulting property is entirely internal and assumes α is some positive real number. We can apply this property under the special case where α is a positive infinitesimal, while all other variables are standard, and, after some additional proof steps, convert the closed predicate consisting of α back to the initial open predicate.

6.15 Measure Structures

We have observed that ordinals are valuable as well-founded sets in proving program termination. We would like to use such structures in reasoning about stability of hybrid systems. However, real numbers cannot be used directly to show a decreasing measure. Rather, some transformation must be presented whereby the real numbers are mapped to a well founded set, such as the ordinals up to ω^ω.

In this section we define a structure we call the *measure structure*, or *m-structure*. We also define a function *m-floor*, represented as $\lfloor\lfloor x\rfloor\rfloor$, which transforms an m-structure x to an ordinal. The goal is to compare two m-structures x and y using the relational operator $\prec$ such that if we show $x \prec y$, then $\lfloor\lfloor x\rfloor\rfloor < \lfloor\lfloor y\rfloor\rfloor$. The definition of an m-structure follows.

A non-negative real number is a finite m-structure. For a real $x \geq 1$, or $0 \prec x$, we call x a finite, non-zero m-structure. We designate the symbol Γ as an m-structure. If x is a finite, non-zero m-structure, Γx is also an m-structure. Γx is defined to be a transfinite m-structure. For a finite, non-zero m-structure x, Γ^x is an m-structure.

The relation operator $\prec$ compares m-structures. For finite m-structures x and y, $x \prec y$ iff $x \leq y - 1$. For all finite m-structures x, $x \prec \Gamma$. Any m-structure consisting of one or more instances of Γ, with non-zero m-structures for its coefficient and exponent, is a transfinite m-structure. For any transfinite m-structure α and finite m-structure x, $x \prec \alpha$. Assuming finite non-zero m-structures x and y, non-zero m-structures α_2 and β_2, and m-structures α_1 and β_1, the operator $\prec$ is defined over transfinite m-structures as follows:

$$(\Gamma^{\alpha_2}x + \alpha_1) \prec (\Gamma^{\beta_2}y + \beta_1) \quad \overset{\text{def.}}{=} \quad \begin{cases} \alpha_2 \prec \beta_2 \lor \\ \alpha_2 = \beta_2 \land x \leq y - 1 \lor \\ \alpha_2 = \beta_2 \land x = y \land \alpha_1 \prec \beta_1. \end{cases} \tag{6.134}$$

We say that an m-structure is well formed if it is either a finite m-structure or an m-structure of the form $\Gamma^{\alpha_2}x + \alpha_1$, where x is a non-zero finite m-structure,

α_2 is a well formed non-zero m-structure, and α_1 is a well formed m-structure, where $\alpha_1 \prec \Gamma^{\alpha_2}$. Henceforth, we will assume all m-structures are well formed.

The function *m-floor* applied to α, represented as $\lfloor\lfloor\alpha\rfloor\rfloor$, maps an m-structure α to an ordinal. For a finite m-structure x, $\lfloor\lfloor x\rfloor\rfloor = \lfloor x\rfloor$. For non-zero m-structures α_1, α_2, and x, where x is finite, and $\Gamma^{\alpha_2} \prec \alpha_1$:

$$\lfloor\lfloor\Gamma^{\alpha_2} x + \alpha_1\rfloor\rfloor = \omega^{\lfloor\lfloor\alpha_2\rfloor\rfloor}\lfloor x\rfloor + \lfloor\lfloor\alpha_1\rfloor\rfloor. \tag{6.135}$$

Therefore, the m-floor function recurses an m-structure, taking the floor of constituent finite m-structures, and substituting every instance of the symbol Γ with ω.

Theorem 6.15.1. *For m-structures α and β, and letting $<$ represent the usual ordinal relational operator:*

$$\alpha \prec \beta \rightarrow \lfloor\lfloor\alpha\rfloor\rfloor < \lfloor\lfloor\beta\rfloor\rfloor. \tag{6.136}$$

Proof. The proof may be performed by induction over an m-structure, and the fact that every finite m-structure is a non-negative real, and $x \leq y - 1 \rightarrow \lfloor x\rfloor < \lfloor y\rfloor$. $\square$

6.16 Extending Measure Structures with Standard Part

As we did for ordinals, we define the dimension of a measure structure.

Definition 6.16.1. Given an m-structure β , we represent β as

$$\beta = (\Gamma^{\alpha_2} x + \alpha_1). \tag{6.137}$$

We define the dimension of the structure recursively as 1 plus the sum of the dimension of α_1 and the dimension of α_2. A finite m-structure, or a non-negative real number, is assumed to have dimension 1.

We define an equivalence relation using standard part on m-structures with standard dimension. Assuming non-zero finite m-structures x, y, non-zero m-structures α_2, β_2, and for m-structures $\alpha_1 \prec \Gamma^{\alpha_2}$, and $\beta_1 \prec \Gamma^{\beta_2}$, we define the new equivalence relation $=_{st}$ on transfinite m-structures as follows:

$$(\Gamma^{\alpha_2} x + \alpha_1) =_{st} (\Gamma^{\beta_2} y + \beta_1) \quad \overset{\text{def.}}{=} \quad \begin{cases} \alpha_2 =_{st} \beta_2 \wedge \\ st(x - y) = 0 \wedge \\ \alpha_1 =_{st} \beta_1. \end{cases} \tag{6.138}$$

For m-structures x and y which are non-negative real numbers, we have that $x =_{st} y \overset{\text{def.}}{=} st(x - y) = 0$. For finite m-structure x and transfinite m-structure α, then $x =_{st} \alpha$ and $\alpha =_{st} x$ are defined to be $false$.

As an alternative to 6.134, we may define the relational operator on transfinite m-structures of standard dimension with the use of the function standard part and the equivalence relation defined in 6.138:

$$(\Gamma^{\alpha_2} x + \alpha_1) \prec_{st} (\Gamma^{\beta_2} y + \beta_1) \quad \overset{\text{def.}}{=} \quad \begin{cases} \alpha_2 \prec_{st} \beta_2 \vee \\ \alpha_2 =_{st} \beta_2 \wedge st(x-y) \leq -1 \vee \\ \alpha_2 =_{st} \beta_2 \wedge st(x-y) = 0 \wedge \alpha_1 \prec_{st} \beta_1. \end{cases} \tag{6.139}$$

For m-structures x and y which are non-negative real numbers, we have that $x \prec_{st} y \overset{\text{def.}}{=} st(x-y) \leq -1$. For finite m-structure x and transfinite m-structure α, $x \prec_{st} \alpha$ is defined to be $true$ and $\alpha \prec_{st} x$ is defined to be $false$.

6.17 Hybrid Measure Structures

An important result of the modified Lyapunov method is that given some positive valued, differentiable function v, if it can be shown $\dot{v} \leq \beta < 0$, for some standard real constant β, then we can assure that the measure is strictly decreasing, as shown for the ordinal valued measure function 6.98. Rather than producing an ordinal, we will assume that when reasoning about positive valued, continuous measure functions $g(x)$, we require that $\dot{g}(x) \leq \beta < 0$, for some standard real constant β, to assure a decreasing measure. When reasoning about discrete values, we assume the associated measure for showing progress is ordinal valued, and require that the ordinal decrease by at least one in each step.

Consequently, one can construct an m-structure with standard dimension consisting of finite m-structures which are measure functions that are of two forms:

$$\begin{aligned} &\frac{g(x)}{\epsilon}, \quad \text{where } g \text{ is non-negative real and continuous, and} \\ &w(x), \quad \text{where } w \text{ takes on Natural number values.} \end{aligned} \tag{6.140}$$

We assume both g and w are standard functions. We consider such an m-structure a *hybrid* m-structure which is intended to show progress for both the discrete and continuous portions of the system. The ordinal valued functions may be used to show progress for the discrete portion, while the continuous functions are used to show progress for the continuous portion. For example, one may construct the following hybrid m-structure:

$$\Gamma^3\, w_2(x) \,+\, \Gamma^2\, w_1(x) \,+\, \Gamma\, \frac{g_2(x)}{\epsilon} \,+\, \frac{g_1(x)}{\epsilon}. \tag{6.141}$$

In the example 6.141, measure functions w_1 and w_2 can measure the discrete portion of the system, while the measure functions g_1 and g_2 measure the continuous portion of the system.

However, in showing a decreasing measure for this hybrid m-structure, we assume the relational operator $\prec_{st}$ and equivalence relation $=_{st}$, defined in the previous section.

We conjecture that we may show progress toward some predicate Q if we can generate a measure function m whose value is a hybrid m-structure such that:

$$\begin{aligned} \neg Q(x) \;&\rightarrow\; m(x, \sigma(x,\epsilon), \epsilon) \prec_{st} m(x,\epsilon), \text{ and} \\ Q(x) \;&\rightarrow\; m(x,\epsilon) = 0. \end{aligned} \tag{6.142}$$

We note that in the case of a particular continuous measure function g defined as part of a hybrid m-structure, and assuming $\neg Q(x)$, the requirement in 6.142 reduces to showing one of the following for this particular function g:

$$P_1(x,\epsilon) \;\rightarrow\; st\left(\frac{g(\sigma(x,\epsilon))}{\epsilon}\right) - st\left(\frac{g(x)}{\epsilon}\right) \leq -1, \tag{6.143}$$

or

$$P_2(x,\epsilon) \;\rightarrow\; st\left(\frac{g(\sigma(x,\epsilon))}{\epsilon}\right) - st\left(\frac{g(x)}{\epsilon}\right) = 0. \tag{6.144}$$

The predicates P_1 and P_2 represent the conditions under which the inequalities 6.143 and 6.144, respectively, are required to hold to show a decreasing measure. Both of these inequalities may be rewritten, by properties of standard part:

$$P_1(x,\epsilon) \;\rightarrow\; st\left(\frac{g(\sigma(x,\epsilon)) - g(x)}{\epsilon}\right) \leq -1, \tag{6.145}$$

$$P_2(x,\epsilon) \;\rightarrow\; st\left(\frac{g(\sigma(x,\epsilon)) - g(x)}{\epsilon}\right) = 0. \tag{6.146}$$

In a proof similar to that for the modified Lyapunov function, it may be shown that, in some cases, the requirement in 6.145 can be reduced to showing $\dot{g}(x) \leq -1$, for $P_1(x,\epsilon)$. A case where the derivative is not applicable is when $g(\sigma(x,\epsilon)) - g(x)$ is a nonzero standard number. However, in this case, the measure may still be shown to decrease if $g(\sigma(x,\epsilon)) - g(x)$ is negative. It should be noted that if one can generate a function g such that $P_1(x,\epsilon) \rightarrow \dot{g}(x) \leq \beta$, for a constant negative standard real β, then one can generate a function $g_2(x) = g(x)/|\beta|$, where $P_1(x,\epsilon) \rightarrow \dot{g}_2(x) \leq -1$. Similarly, it may be shown that 6.146 can be reduced to $\dot{g}(x) = 0$, for $P_2(x,\epsilon)$.

6.18 Summary

In this chapter we presented formal methods whereby one can show safety and progress properties about the solution of a well formed hybrid model as described in 5.1.1. We also presented soundness proofs for each of the methods. In addition, we presented simple example systems for which we applied the safety and progress proof procedure to show safety and progress properties, respectively. We also demonstrated how a modified form of the Lyapunov function may be used in reasoning about safety and progress using the proof methods we have outlined.

Chapter 7

Formal Reasoning About an Example Hybrid System

We will model a simple closed loop computer controlled positioning system. The computer is to monitor the position of the physical system and adjust this position by sending a signal to the physical system so as to adjust it to a position as entered by the user.

The goal of this chapter is to demonstrate how to model a computer program and the associated physical system. In modeling the physical system the example demonstrates a discontinuous vector field. In modeling the computer system, the example demonstrates modeling of the computer program, modeling of the conversion of the analog signal to a discrete signal, and modeling of the real time execution of the program at discrete intervals. This chapter also demonstrates the use of predicates defined in 6.2 for the generation of safety and progress properties.

The ACL2r theorem prover [1, 29] is used to define the system model. The theorem prover is also used to fulfill the proof obligations of the safety and progress proof procedures.

7.1 System Description

We propose an example computer controlled positioning system consisting of a computer and a simple physical system that has three modes of operation. The physical system is characterized by the real valued variable *pos* which stands for position. The physical system also attains a signal from its environment, the integer variable *posAo*. This variable is assumed to be generated by a discrete device, such as a computer. The modes of the physical system are determined by the relation of the variables *pos* and *posAo*.

The conditional differential equation for the physical system is:

$$\frac{d\,pos}{dt} = \begin{cases} 1 & pos < posAo \\ -1 & pos > posAo \\ 0 & pos = posAo. \end{cases} \tag{7.1}$$

The computer program receives a signal *posAi* representing the position of the physical system. The computer program to be executed by the computer is represented below:

```
IF posAi - posReq < -3 THEN
   posAo := posAo + 5;
ELSE IF  posAi - posReq > 2 THEN
   posAo := posAo - 5;
ELSE
   posAo := posAo;
END IF
```

Since the computer program is assumed to be integer valued, a conversion is required from the real valued position signal *pos* to the integer variable *posAi* used by the computer. To model this, we use the floor function:

$$posAi := \lfloor pos \rfloor . \tag{7.2}$$

We assume the computer executes the program at regular time intervals. To represent time, we add the variable *tmr* to the system. This variable *tmr* is assumed to increase at a constant rate. When $tmr \geq preset$, we assume the computer executes its program. Upon termination of program execution, the computer remains idle until a time duration *preset* has elapsed, at which time it executes its program again, and so on. It is assumed that the computer executes its program instantaneously.

The variable *preset* is assumed to be real valued. Neither the computer nor the physical system modifies the variable *preset*. The variable *posReq* represents the position of the physical system desired by the user. This variable is also not modified by the computer or the physical system. While these variables are not modified by the system, they are still considered as part of the state x. In summary, the system state variables are as follows: *pos*, *posAi*, *posAo*, *posReq*, *tmr*, and *preset*. Since $posAi = \lfloor pos \rfloor$, and *posAi* is not modified by the system, we will drop the variable *posAi* and replace it with $\lfloor pos \rfloor$.

By the requirements of the system model, we define the assignment predicate B_Y:

$$B_Y \overset{\text{def.}}{=} tmr \geq preset \tag{7.3}$$

The assignment function, Y, includes the semantics of the computer program as well as a reset of the variable, *tmr*, to zero. Hence, by the definition of Y and B_Y, the system is such that it executes the program every *preset* time units. The following is the definition of the assignment function Y with respect to the vector component *posAo*, which we denote as $Y_{posAo}(x)$:

$$Y_{posAo}(x) = \begin{cases} posAo + 5 & \lfloor pos \rfloor - posReq < -3 \\ posAo - 5 & \lfloor pos \rfloor - posReq > 2 \\ posAo & -3 \leq (\lfloor pos \rfloor - posReq) \leq 2. \end{cases} \tag{7.4}$$

The following is the definition of the assignment function Y with respect to the vector component *tmr*, which we denote as $Y_{tmr}(x)$:

$$Y_{tmr}(x) = 0. \tag{7.5}$$

It is assumed that Y only modifies *posAo* and *tmr*. To model the physical system as well as the behavior of the variable *tmr*, we define the step function σ as follows:

$$\sigma_{pos}(x, \epsilon) = \begin{cases} pos + \epsilon & pos < posAo \\ pos - \epsilon & pos > posAo \\ pos & pos = posAo. \end{cases} \tag{7.6}$$

$$\sigma_{tmr}(x, \epsilon) = tmr + \epsilon.$$

It is assumed that σ only modifies *pos* and *tmr*.

A representation of the assignment function Y in ACL2r is shown below:

```
(defun Y (X)
  (make-state
   (getPosReq x)
   (getPreset x)
   (getPos x)
   ;;posAo
   (cond
    ((> (- (floor1 (getPos X)) (getposReq X)) 2)
     (- (getposAo X) 5))
    ((< (- (floor1 (getPos X)) (getposReq X)) -3)
     (+ (getposAo X) 5))
    (t (getposAo X)))
   ;;tmr
   0))
```

A representation of the system step function σ in ACL2r is shown below:

```
(defun sigma (X eps)
  (make-state
   (getPosReq x)
   (getPreset x)
   ;;pos
   (cond
      ((> (getPos X) (getPosAo X))
          (- (getpos X) eps))
      ((< (getPos X) (getPosAo X))
          (+ (getpos X) eps))
      (t (getPos X)))
   (getposAo X)
   ;;tmr
   (+ (getTmr X) eps)))
```

The overall system function is shown below:

```
(defun sys-step (X eps)
  (cond
   ((B-Y X) (Y X))
   (t       (sigma X eps))))
```

7.2 Safety and Progress Properties About the Sample System

We commence by demonstrating some safety properties about the system. We would like to show that the variables of the system stay within a certain range and that their types are preserved. For example, the computer variables should remain integer, the variable *tmr* should always satisfy $0 \leq tmr \leq preset$. The ACL2r function definition for a valid state is as follows:

```
(defun valid-state (X eps)
   (and (realp (getPos X))
        (realp (getPreset X))
        (realp (getTmr X))
        (integerp (getPosAo X))
        (integerp (getPosReq X))
        (<= 51/10 (getPreset X))
        (<= 0 (getTmr x))
        (<= (getTmr x) (+ (getpreset x) eps))))
```

The function `valid-state` is a conjunction of several predicates. The inequality predicates have the ϵ-transform applied to them, as can be verified from the inequality `(<= (getTmr x) (+ (getpreset x) eps))` which denotes $tmr \leq preset + \epsilon$. In the above property, the predicate `(integerp (getPosAo X))`, a predicate recognizing integers, is an example of a discrete predicate, as allowed by the grammar 6.2. The predicate `(realp (getPos`

`X))`, a predicate recognizing real numbers, is acceptable since the variables in our model are assumed to be real. The remaining predicates shown are inequalities, which satisfy the requirements of 6.2. By the proof obligation of the safety proof procedure, we must show the property is preserved over a system step function.

This proof obligation is shown in the following ACL2r theorem definition:

```
(defthm valid-state-preserve
 (implies
   (and (valid-state x eps)
        (small-realp eps))
   (valid-state (sys-step x eps) eps)))
```

The predicate `(small-realp eps)` states that `eps` is a real number such that $0 <$ `eps` $\leq 1/100$.

The objective of the computer is to control the physical system until the position is within some range of *posReq*, the position requested by the user. Once *pos* reaches within some range of *posReq*, it should remain within this range. Therefore, we are required to show, starting from a state as defined by `valid-state`, the system will eventually reach a state where *pos* and *posReq* are within some finite range of each other. We attempt to show this with a progress property about the system. As required by the progress proof procedure, we must provide a measure function which is decreasing for every system step, until reaching the state specified by the progress property.

The measure function used is as follows:

$$m_1(x,\epsilon) = \begin{cases} 0 & |posAo - pos| < \\ & \quad pc(preset - tmr) + \epsilon \\ 1 + \frac{|posAo - pos| - (preset - tmr + \epsilon)}{\epsilon} & \text{otherwise} \end{cases}$$

$$m_2(x,\epsilon) = \begin{cases} 0 & |posAo - posReq| \leq 3 \\ |posAo - posReq| & \text{otherwise} \end{cases}$$

$$m_3(x,\epsilon) = \begin{cases} 0 & |pos - posAo| \leq \epsilon \\ \frac{|pos - posAo|}{\epsilon} & \text{otherwise} \end{cases}$$

$$m_4(x,\epsilon) = \begin{cases} 0 & preset < tmr \\ 1 + \frac{preset - tmr}{\epsilon} & \text{otherwise} \end{cases}$$

$$M(x,\epsilon) = \begin{cases} \Gamma^3 m_1(x,\epsilon) + m_4(x,\epsilon) & m_1(x,\epsilon) > 0 \\ \Gamma^2 m_2(x,\epsilon) + \Gamma\, m_2(x,\epsilon) + m_4(x,\epsilon) & m_1(x,\epsilon) = 0 \wedge m_2(x,\epsilon) > 0 \\ \Gamma\, m_3(x,\epsilon) + m_4(x,\epsilon) & m_1(x,\epsilon) = 0 \wedge m_2(x,\epsilon) = 0 \\ & \quad \wedge\, m_3(x,\epsilon) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.7}$$

The function $pc(x)$ returns 0 if $x \leq 0$, and x otherwise. The ACL2r representation of the measure functions is shown below:

```
(defun m1 (X eps)
  (cond
   ((<= (abs (- (getPosAo X) (getPos X)))
   (+ eps (pos-clamp (- (getPreset X)
                        (getTmr X)))))
    0)
   (t (+ 1
          (/ (- (abs (- (getPosAo X) (getPos X)))
                (+ (- (getPreset X) (getTmr X)) eps))
             eps)))))

(defun m2 (X eps)
(declare (ignore eps))
  (if (and
       (<= (- (getPosAo X) (getPosReq X)) 3)
       (>= (- (getPosAo X) (getPosReq X)) -3))
      0
      (abs (- (getPosAo X) (getPosReq X)))))

(defun m3 (X eps)
  (if (<= (abs (- (getPos X) (getPosAo X))) eps)
      0
      (/ (abs (- (getPos X) (getPosAo X))) eps)))
```

```
(defun m4 (X eps)
  (cond
   ((< (getPreset X) (getTmr X)) 0)
   (t (+ 1 (/ (- (getPreset X) (getTmr X)) eps)))))

(defun m (X eps)
  (cond
   ((and
     (< (m1 x eps) 1)
     (< (m2 x eps) 1)) (make-ord 1
                                 (+ 1 (m3 x eps))
                                 (m4 x eps)))
   ((< (m1 x eps) 1) (make-ord 2
                               (+ 1 (m2 x eps))
                               (make-ord 1
                                         (+ 1 (m3 x eps))
                                         (m4 x eps))))
   (t (make-ord 3 (+ 1 (m1 x eps)) (m4 x eps)))))
```

```
(defun m-fix (x eps)
  (cond
   ((not (and (valid-state x eps)
              (small-realp eps)
              (not (and
                     (< (m1 x eps) 1)
                     (< (m2 x eps) 1)
                     (<= (abs (- (getpos x)
                                 (getPosReq x)))
                        (+ 3 (* 2 eps)))))))
    0)
   (t (o-floor1 (m x eps)))))
```

Since the measure functions are real valued, we apply a function `m-fix` to convert the values generated by the measure functions to ordinals, using the `o-floor1` function. The function `o-floor1` is the ACL2r function we have defined to implement the operator $\lfloor\lfloor x \rfloor\rfloor$ discussed in the previous chapter regarding measure structures.

By showing the above measure decreases, we eventually reach a state satisfying the progress property:

$$m_1(x,\epsilon) < 1 \ \wedge \ m_2(x,\epsilon) \ < 1 \ \wedge \ |pos - posReq| \leq 3 + 2\epsilon. \tag{7.8}$$

It can be verified, and has been shown in ACL2r, that 7.8 may be rewritten to:

$$\begin{aligned}&|posAo - pos| \leq pc(preset - tmr) + \epsilon \wedge \\ &|posAo - posReq| \leq 3 \wedge \\ &|pos - posReq| \leq 3 + 2\epsilon.\end{aligned} \tag{7.9}$$

We then formally show that 7.9 is a safety property. This proof obligation for the safety proof procedure is shown below in ACL2r:

```
(defthm safety-property-preserve
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (< (m1 x eps) 1)
        (< (m2 x eps) 1)
        (<= (abs (- (getpos x) (getPosReq x)))
            (+ 3 (* 2 eps))))
   (and
    (valid-state x eps)
    (< (m1 (sys-step x eps) eps) 1)
    (< (m2 (sys-step x eps) eps) 1)
    (<= (abs (- (getpos (sys-step x eps))
                (getposReq (sys-step x eps))))
        (+ 3 (* 2 eps)))))
```

Therefore, we have shown that the system reaches a state satisfying 7.9 and any states thereafter continue to satisfy 7.9. While the property 7.9 is a conjunction of three predicates, the one of interest to us is $|pos - posReq| \leq 3 + 2\epsilon$, which states, neglecting the ϵ-transform, that *pos* and *posReq* are within 3 units of each other.

7.3 Observations about the Measure Function

The measure function 7.7 was derived by observing the system usually approaches stability in two phases:

Ph1. Since the initial state may be any value satisfying `valid-state`, then the computer's analog output *posAo* and the valve position *pos* may not be the same. Therefore, in phase 1, we assume the system attempts to bring the physical system position and the computer output *posAo* within $preset - tmr$ of each. This behavior is captured by measure m_1. Once *posAo* and *pos* are within $preset - tmr$, m_1 becomes zero, and the system moves to phase 2.

Ph2. In phase 2, the system attempts to match *posAo* with *posReq* until they are within 3 of each other. Measure m_2 is used during this phase. It is possible that, within a single step, the *posAo* does not change, and the timer resets. To assure the measure is decreasing even in this scenario, we added measure $m3$, which does decrease while the computer is idle, but the timer *tmr* is resetting.

The measure m_4 is used when the computer is idle (not executing its program). This is important for measuring progress, since it may be, while the computer is idle, the physical system is idle as well, resulting in a measure which is non-decreasing. The use of a timer in the model of the computer system allows the definition of measures which capture progress even when the computer program is not being executed.

7.4 The Measure Function and Control Algorithms

In showing progress, as well as Lyapunov stability, the user is required to define a measure function. Often, when designers produce a control algorithm, they have an intuition of how the controller is to stabilize the system. The progress proof procedure can make this intuition formal by defining a measure function that is suitable for the computer system and physical system under consideration.

There have been attempts to automate the generation of Lyapunov functions for linear systems. However, these methods are predominately effective for linear systems [45]. Such methods can certainly be used to extend our work, as we have shown Lyapunov functions can be used to show stability for a system. In general, there does not exist an automatic method for showing stability of an arbitrary control system, since reachability is undecidable for control systems, even for very simple ones [5, 38].

7.5 Summary

We have demonstrated an example application of the system model defined in Chapter 5, and the safety and progress proof procedures defined in Chapter 6. We showed, using the progress proof procedure, that a measure exists whereby the system reaches a state satisfying a property and, by a safety proof, showed that the system remains in states satisfying this property.

An important observation to be made about the proof of the safety property is that, although our desired objective is to prove some property that involves only the two variables *pos* and *posReq*, it is necessary to state a more complicated property, 7.9. If we attempted to prove the safety property consisting of only *pos* and *posReq*, it would not hold, since the unrestricted variable *posAo* would result in changes to *pos*. Therefore, generating a property requires some degree of consideration of the system behavior and may require inclusion of system variables in addition to those originally intended for the property of interest.

Chapter 8

Summary and Future Work

In this dissertation, we have applied nonstandard analysis to formally model and reason about hybrid systems. The first contribution of this work is the definition of a computer controlled system model which models both the semantics of the computer program as well as the ordinary differential equations governing the behavior of the physical system. The model for the computer program is a computable function, which can model the semantics of finite state machines as well as programs consisting of integer arithmetic. The second contribution of our work is the modeling of physical systems with discontinuous vector fields. The proposed model allows for infinite switches in the system in finite time. The third contribution is the formal definition of a solution to such a model and the proof that such a solution does exist. The fourth contribution is the development of proof procedures whereby one may formally reason about safety and progress properties of the nonstandard model. We have also provided formal proofs of the soundness of the proof procedures.

8.1 Future Work

The model for a physical system which we have presented assumes ordinary differential equations only. Physical systems are modeled with partial differential equations as well. Therefore, one area of extending the system is to allow for the modeling of partial differential equations.

We have developed criteria for the definition of a well formed hybrid system 5.1.1. We have also developed a grammar 6.2 for predicates used in safety and progress properties. It would be useful, in a mechanical theorem proving environment, for the theorem prover to check the definition of a hybrid system, or require the user to satisfy the requirement for a well formed hybrid system, prior to admitting a hybrid system definition. Similarly, the theorem prover can check if the predicates used to reason about safety and progress meet the requirements of 6.2.

Determining the value of $h(x, \epsilon)$ for an ϵ-transformation may not be readily evident. A failed proof attempt may provide the user with some guidance as to the form of $h(x, \epsilon)$ required. Alternatively, a tool may be developed whereby, for some restricted subclass of properties, the required function h may be automatically generated.

In the work presented in Chapter 6 regarding hybrid m-structures, we proposed a method by which to prove progress for such an m-structure. As future work, one may prove the soundness of such an approach. Also, while we have demonstrated proof methods for safety and progress properties, as future work, one may investigate the formulation of a proof method for proving infinitely often properties about the system solution we have defined.

In the example problem presented in Chapter 7, the model for the computer is one in which the program executes instantaneously and at regular intervals. Other models for the computer may be considered. For example, the computer program may still be modeled to execute instantaneously, but using data attained from the physical system during the previous cycle. Also, the model of the computer may also be extended to include parallel computers or distributed computers to model multiple controllers for a system.

In the process of formally demonstrating a property, we have noted that the proof obligations reduce to several cases, where the cases are a composition of the cases of the computer system, the physical system, and the progress or safety property being proved. Rather than formally demonstrating a property about a system, it may be investigated whether one may use the cases resulting from the proof attempt for a property as a guide for generating test sets for the simulation of the system.

Appendices

Appendix A

Mechanical Proof of Existence and Uniqueness

The following sections provide the printout of the definition files submitted to the mechanical theorem prover ACL2r in performing the mechanical proof of the existence and uniqueness theorem for a Lipschitz continuous vector field as defined in Chapter 4. In our proofs, we have assumed a real, scalar value for the physical system variable x.

In Chapter 4, we referred to the Euler approximation as ρ. Here, we refer to the Euler approximation as `run`. Unlike the presentation in Chapter 4, the `run` function is defined in terms of a natural number `n` representing the iteration number in implementing the Euler method. Therefore, when defining ϕ in terms of `run`, we substitute $\lfloor t/\epsilon \rfloor$ for `n`. Throughout all the subsequent file printouts, the variable representing time t is represented as `tm`, since `t` in ACL2r represents the constant *true*.

The printouts shown in the subsequent sections are of the following files:

1. `arith-nsa4.lisp` Definitions of basic arithmetic theorems.
2. `abs.lisp` Definitions of theorems about the absolute value function.

3. `tm-floor.lisp` Definitions of functions and theorems relating time t to $\lfloor t/\epsilon \rfloor \epsilon$, as well as relating t to $n\,\epsilon$, for natural number n.

4. `eexp.lisp` The exponent function, e^x, and some properties about this function used in our proofs.

5. `nsa.lisp` Definitions of basic nonstandard theorems about real numbers.

6. `Computed-hints.lisp` Definitions of computed hint functions. One computed hint function provides automatic hints to ACL2r in opening up function definitions. Another computed hint function applies standard part to both sides of an inequality.

7. `phi-exists.lisp` Definitions to show that the solution function ϕ, as described in Chapter 4, does exist. The file also includes definition of the function `run` which is the ACL2r representation of the function ρ^ϵ described in Chapter 4.

8. `phi-properties.lisp` Definitions to show properties about the solution functions ϕ and ρ^ϵ. These properties include continuity with respect to time t, continuity with respect to x, and time invariance, which are discussed in Chapter 4. This file also includes the definition of the theorem which states that the standard part of ρ^ϵ is identical to the standard part of ρ^δ, for ϵ and δ positive infinitesimals. Hence, demonstrating that ϕ is independent of the choice of positive ϵ.

9. `phi-unique.lisp` Definitions to show that, given an alternative function ϕ_2, denoted `phi2` in ACL2r, which does satisfy the differential equation, then the proposed solution ϕ is equal to ϕ_2. Since ϕ is equal to any function ϕ_2 which satisfies the differential equation, then the solution to the differential equation is unique.

A.1 Arithmetic Theorems

This section presents a printout of file `arith-nsa4.lisp`. This file consists of definitions of some arithmetic theorems we use in our proofs.

```
(in-package "ACL2")

;;===================================
;;arithmetic
(defun coeff-term-order (x y)
(declare (xargs :mode :program))
 (cond
  ((and
    (eq (fn-symb x)
            'binary-*)
    (quotep (fargn x 1))
    (eq (fn-symb y)
            'binary-*)
    (quotep (fargn y 1))) (term-order (fargn x 2)
                                      (fargn y 2)))
  ((and
    (eq (fn-symb x)
            'binary-*)
    (quotep (fargn x 1))) (if (equal (fargn x 2) y)
                              nil
                              (term-order (fargn x 2)
                                          y)))
  ((and
    (eq (fn-symb y)
            'binary-*)
    (quotep (fargn y 1))) (if (equal x (fargn y 2))
                              t
                              (term-order x
                                          (fargn y 2))))
  (t (term-order x y))))

(defthm +-commut-coeff-2way
(implies
 (syntaxp (not (coeff-term-order y x)))
 (equal (+ y x) (+ x y)))
:rule-classes ((:rewrite :loop-stopper nil)))

(defthm +-commut-coeff-3way
(implies
 (syntaxp (not (coeff-term-order y x)))
 (equal (+ y x z) (+ x y z)))
:rule-classes ((:rewrite :loop-stopper nil)))

(defthm *-commut-3way
 (equal (* y x z) (* x y z))
:hints (("Goal" :use ((:instance commutativity-of-* (x y) (y (* x z)))))))

;;disable the usual commuativity of +. If not disabled, looping may occur.
(in-theory (disable commutativity-of-+))

(defthm *-zero
  (equal (* 0 x) 0))

(defthm +-zero
  (equal (+ 0 x) (fix x)))

(defthm uminus-is-*-neg-1
  (equal (- x) (* -1 x)))

(defthm fold-consts-in-+
  (implies (and (syntaxp (quotep x))
                (syntaxp (quotep y)))
           (equal (+ x (+ y z))
                  (+ (+ x y) z))))

(defthm fold-consts-in-*
  (implies (and (syntaxp (quotep x))
                (syntaxp (quotep y)))
           (equal (* x (* y z))
                  (* (* x y) z))))

(defthm combine-terms-+-3way
  (implies (and (syntaxp (quotep a))
                (syntaxp (quotep b)))
           (equal (+ (* a x) (+ (* b x) y))
                  (+ (* (+ a b) x) y))))

(defthm combine-terms-+-2way
  (implies (and (syntaxp (quotep a))
                (syntaxp (quotep b)))
           (equal (+ (* a x) (* b x))
                  (* (+ a b) x))))
```

```
(defthm combine-terms-+-3way-unary
  (implies (syntaxp (quotep a))
           (equal (+ x (+ (* a x) y))
                  (+ (* (+ a 1) x) y))))

(defthm combine-terms-+-2way-unary
  (implies (syntaxp (quotep a))
           (equal (+ x (* a x))
                  (* (+ a 1) x))))

(defthm /-cancellation
  (implies (and (acl2-numberp x)
                (not (equal 0 x)))
           (equal (* x (/ x) y)
                  (fix y)))
  :hints (("Goal" :use ((:instance commutativity-of-*
                             (x y)
                             (y (* x (/ x))))))))

(defthm <-*-right-cancel
  (implies (and (realp x)
                (realp y)
                (realp z))
           (iff (< (* x z) (* y z))
                (cond
                 ((< 0 z)
                  (< x y))
                 ((equal z 0)
                  nil)
                 (t (< y x)))))
  :hints (("Goal" :use
           ((:instance (:theorem
                        (implies (and (realp a)
                                      (< 0 a)
                                      (realp b)
                                      (< 0 b))
                                 (< 0 (* a b))))
                       (a (abs (- y x)))
                       (b (abs z)))))))

(defthm <-*-left-cancel
  (implies (and (realp x)
                (realp y)
                (realp z))
           (iff (< (* z x) (* z y))
                (cond
                 ((< 0 z)
                  (< x y))
                 ((equal z 0)
                  nil)
                 (t (< y x))))))

(defthm distrib-/-over-*
 (equal (/ (* x y)) (* (/ x) (/ y)))
:hints (("Goal" :use ((:instance (:theorem
                                   (implies
                                    (and
                                     (acl2-numberp y)
                                     (acl2-numberp z)
                                     (not (equal y 0))
                                     (not (equal z 0)))
                                    (equal (fix x) (* (/ y) (/ z) y z x))))
                                  (x (/ (* x y))) (y x) (z y))
                       (:instance inverse-of-* (x (* x y)))))
        ("Subgoal 2"
                      :use (:instance (:theorem
                                       (implies
                                        (equal (* x y) 0)
                                        (or (equal (fix x) 0)
                                            (equal (fix y) 0))))))))

;;Generate a list of terms each of which appears as the divisor in
;; a unary-/ term in the given TERM which is assumed to be a product
;; of terms, or itself a unary-/ form.
(defun FIND-Divisors-in-times (TERM)
  (cond
   ((eq (fn-symb term) 'binary-*)
    (append (find-divisors-in-times (fargn term 1))
            (find-divisors-in-times (fargn term 2))))
   ((eq (fn-symb term) 'unary-/)
    (list (fargn term 1)))
   (t nil)))

;;Generate a list of terms each of which appears as the divisor
;; in a unary-/ term in the given TERM which is assumed to be a
;; polynomial in sum of products form.
(DEFUN FIND-DIVISORS-IN-POLY (TERM)
  (IF (EQ (FN-SYMB TERM) 'BINARY-+)
      (append (find-divisors-in-times (FARGN TERM 1))
              (find-divisors-in-poly (FARGN TERM 2)))
      (find-divisors-in-times TERM)))

;;Find a binding to a term which appears as the divisor in a unary-/ term
;; in the given LHS or RHS polynomials.
;; Polynomials are assumed to be in sum of products form.
;; If no such term exists, return nil, if more than one exists,
;; pick the first and bind that term to x.
(DEFUN FIND-DIVISORS-BIND-TERM (LHS RHS)
  (LET ((DIVISOR-LST (APPEND (find-divisors-in-poly LHS)
                             (find-divisors-in-poly RHS))))
       (IF DIVISOR-LST
           (LIST (CONS 'X (CAR DIVISOR-LST)))
           NIL)))

;;If a term appears as a divisor in the LHS or RHS of an equality,
;; where LHS and RHS are assumed to be polynomials,
;; then rewrite the equality where both sides are multiplied by
;; the divisor term, resulting in cancellation of the divisor term
(defthm equal-cancel-divisors
(implies
 (and
  (acl2-numberp LHS)
  (acl2-numberp RHS)
  (BIND-FREE (FIND-divisors-bind-term LHS RHS) (X))
  (acl2-numberp x)
  (not (equal x 0)))
 (equal (equal LHS RHS) (equal (* x LHS) (* x RHS))))
:hints (("Goal" :use ((:instance (:theorem
                                  (implies
                                   (equal a b)
                                   (equal (* x a) (* x b))))
                                 (x (/ x)) (a (* x LHS)) (b (* x RHS)))))))

;;If a term appears as a divisor in the LHS or RHS of an inequality,
;; where LHS and RHS are assumed to be polynomials,
;; then rewrite the inequality where both sides are multiplied by
;; the divisor (with the inequality operand adjusted according
;; to the sign of the divisor term), resulting in cancellation
;; of the divisor term.
(defthm <-cancel-divisors
(implies
 (and
  (realp LHS)
  (realp RHS)
  (BIND-FREE (FIND-divisors-bind-term LHS RHS) (X))
  (realp x)
  (not (equal x 0)))
 (iff (< LHS RHS) (if (< 0 x)
                       (< (* x LHS) (* x RHS))
                       (> (* x LHS) (* x RHS))))))

(defthm /-self-inversion
  (equal (/ (/ x)) (fix x))
:hints (("Goal" :cases ((not (equal x 0))))))

(defthm distributivity-left
  (equal (* (+ x y) z)
         (+ (* x z) (* y z))))

(defthm pos-*-thm
 (implies
  (and
   (realp x)
   (realp y)
   (< 0 x)
   (< 0 y))
  (< 0 (* x y)))
:rule-classes nil)

(defthm pos-factor-<=-thm
  (implies
   (and
    (realp x)
    (realp y)
    (realp a)
    (<= x y)
    (<= 0 a))
   (<= (* a x) (* a y)))
:rule-classes nil
:hints (("Goal" :use ((:instance pos-*-thm (x (- y x)) (y a))))))

(defthm pos-factor-<-thm
  (implies
   (and
    (realp x)
    (realp y)
    (realp a)
    (< x y)
    (< 0 a))
   (< (* a x) (* a y)))
```

```
:rule-classes nil
:hints (("Goal" :use ((:instance pos-*-thm (x (- y x)) (y a))))))

;;end arithmatic
;;==================================================

;;==================================
;; floor1

(defthm floor1-<
(implies
 (and
  (realp x)
  (realp y)
  (<= (+ x 1) y))
 (< (floor1 x) (floor1 y))))

(defthm floor1-limits
(implies
 (and
  (realp x))
 (and
  (<= 0 (- x (floor1 x)))
  (< (- x (floor1 x)) 1)))
:rule-classes nil)

(defthm floor1-+-integer
(implies
 (and
  (integerp i)
  (realp x))
 (equal (floor1 (+ i x)) (+ i (floor1 x)))))

(defthm floor1-pos
(implies
 (and
  (realp x)
  (< 0 x))
 (<= 0 (floor1 x))))

(defthm floor1-neg
(implies
 (and
  (realp x)
  (< x 0))
 (<= (floor1 x) 0)))

(defthm floor1-*-const
(implies
 (and
  (realp x)
  (realp k)
  (syntaxp (quotep k)))
 (equal (< 0 (* k (floor1 x))) (cond
                                 ((< 0 k) (< 0 (floor1 x)))
                                 ((= 0 k) nil)
                                 ((< k 0) (< (floor1 x) 0)))))
:hints (("Goal" :use ((:instance <-*-left-cancel
                           (z k)
                           (x 0)
                           (y (floor1 x)))))))

;;end floor1
;;===================================

;;==================================================
;;non-standard analysis

(defthm standard-part-*-1
  (equal (standard-part (* -1 x))
         (* -1 (standard-part x)))
:hints (("Goal" :use (
                      (:instance standard-part-of-uminus)))))

(defthm standard-part-abs
(implies
 (realp x)
 (equal (standard-part (abs x)) (abs (standard-part x)))))

;;end non-standard analysis
;;==================================================
```

A.2 Theorems About Absolute Value

This section presents a printout of file `abs.lisp`. This file consists of definitions of theorems about the absolute value function.

```
(in-package "ACL2")

(include-book "arith-nsa4")

(defthm abs-realp
  (implies
   (realp x)
   (realp (abs x)))
:rule-classes :type-prescription)

(defthm abs-non-neg-thm
  (implies
   (and
    (realp x)
    (<= 0 x))
   (equal (abs x)
          x)))

(defthm abs-neg-thm
  (implies
   (and
    (realp x)
    (< x 0))
   (equal (abs x)
          (- x))))

(defthm abs-pos-*-left-thm
  (implies
   (and
    (realp x)
    (realp y)
    (<= 0 x))
   (equal (abs (* x y))
          (* x (abs y)))))

(defthm abs-pos-*-right-thm
  (implies
   (and
    (realp x)
    (realp y)
    (<= 0 x))
   (equal (abs (* y x))
          (* x (abs y)))))

(defthm abs-neg-*-left-thm
  (implies
   (and
    (realp x)
    (realp y)
    (< x 0))
   (equal (abs (* x y))
          (* (- x) (abs y)))))

(defthm abs-neg-*-right-thm
  (implies
   (and
    (realp x)
    (realp y)
    (< x 0))
   (equal (abs (* y x))
          (* (- x) (abs y)))))

(defthm abs-triangular-inequality-thm
  (implies
   (and
    (realp x)
    (realp y))
   (<= (abs (+ x y))
       (+ (abs x) (abs y))))
:rule-classes :linear)

(defthm abs-triangular-inequality-3way-thm
  (implies
   (and
    (realp x)
    (realp y)
    (realp z))
   (<= (abs (+ x y z))
       (+ (abs x) (abs y) (abs z)))))

(defthm abs-is-non-neg-thm
  (implies
   (realp x)
   (<= 0 (abs x)))
:rule-classes :type-prescription)

(defthm abs0-thm
  (implies
   (equal (abs x) 0)
   (equal x 0))
:rule-classes :forward-chaining)

(defthm abs-*-thm
  (implies
   (and
    (realp x)
    (realp y))
   (equal (abs (* x y)) (* (abs x) (abs y)))))

(defthm abs-<-*-thm
  (Implies
   (and
    (realp x)
    (realp y)
    (realp a)
    (<= (abs x) (abs y)))
   (<= (abs (* a x)) (abs (* a y))))
:rule-classes :linear
:hints (("Goal" :in-theory (disable abs <-*-LEFT-CANCEL)
                :use ((:instance pos-factor-<=-thm (x (abs x))
                                                   (y (abs y))
                                                   (a (abs a)))
                      (:instance abs-*-thm (x a) (y x))
                      (:instance abs-*-thm (x a) (y y))))))

(defthm abs-limited-thm
  (implies
   (and
    (realp x)
    (i-limited x))
   (i-limited (abs x)))
:rule-classes ((:rewrite) (:type-prescription)))


(defthm abs-standard-numberp
  (implies
   (and
    (realp x)
```

```
 (standard-numberp x))
(standard-numberp (abs x))))
```

A.3 Some Theorems About Time

This section presents a printout of file `tm-floor.lisp`. This file consists of definitions which relate time t with $\epsilon \lfloor t/\epsilon \rfloor$ as well as relating t with ϵn, for natural n. Some of the theorems result in rewrite rules which recognize certain terms and replace them with some specific term such as `(tm-eps-fun tm eps)`, which may then trigger a rewrite rule about `(tm-eps-fun tm eps)`. The variable time t is represented as `tm`, since `t` in ACL2r represents the constant *true*.

```
(in-package "ACL2")

(include-book "arith-nsa4")
(include-book "nsa")
(include-book "computed-hints")

(in-theory (disable i-large))
(in-theory (disable standard-part-<=))

(defthm tm-floor-thm-hint-1
  (implies
   (and
    (realp tm)
    (realp eps)
    (< 0 eps)
    (i-small eps))
   (equal (standard-part (* eps (floor1 (/ tm eps))))
          (standard-part tm)))
:rule-classes nil
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable  <-CANCEL-DIVISORS)
         :use ((:instance pos-factor-<-thm (x (- (/ tm eps) 1))
                                           (y (floor1 (/ tm eps))) (a eps))
               (:instance pos-factor-<=-thm (x (floor1 (/ tm eps)))
                                            (y (/ tm eps)) (a eps))))))

(defthm tm-floor-thm
  (implies
   (and
    (realp tm)
    (standard-numberp tm))
   (equal (STANDARD-PART (* (/ (I-LARGE-INTEGER))
                            (FLOOR1 (* (I-LARGE-INTEGER) TM))))
          tm))
:hints (("Goal" :use ((:instance tm-floor-thm-hint-1
                                  (eps (/ (i-large-integer))))))))

(defthm tm-floor-limited-thm
  (implies
   (and
    (realp tm)
    (standard-numberp tm))
   (i-limited (* (/ (I-LARGE-INTEGER))
                            (FLOOR1 (* (I-LARGE-INTEGER) TM)))))
:hints (("Goal" :use ((:instance standard+small->i-limited
                      (x (standard-part
                                 (* (/ (I-LARGE-INTEGER))
                                       (FLOOR1 (* (I-LARGE-INTEGER) TM)))))
                      (eps (- (* (/ (I-LARGE-INTEGER))
                                  (FLOOR1 (* (I-LARGE-INTEGER) TM)))
                              (standard-part
                                     (* (/ (I-LARGE-INTEGER))
                                        (FLOOR1 (* (I-LARGE-INTEGER) TM)))
                              ))))))))

(defthm tm-floor-*a-hint-1
  (implies
   (and
    (realp tm)
    (realp a)
    (i-limited a)
    (standard-numberp tm))
   (equal (STANDARD-PART (* a (/ (I-LARGE-INTEGER))
                            (FLOOR1 (* (I-LARGE-INTEGER) TM))))
          (standard-part (* a tm))))
:rule-classes nil
:hints (("Goal" :in-theory (disable *-COMMUT-3WAY))))

(defthm tm-floor-*a-thm
  (implies
   (and
    (realp tm)
    (realp a)
    (i-limited a)
    (standard-numberp tm))
   (equal (STANDARD-PART (* (/ (I-LARGE-INTEGER)) a
                            (FLOOR1 (* (I-LARGE-INTEGER) TM))))
          (standard-part (* a tm))))
:hints (("Goal" :in-theory (disable i-large)
                :use ((:instance tm-floor-*a-hint-1)))))

(defun tm-fun (tm)
  (* (/ (I-LARGE-INTEGER))
     (FLOOR1 (* (I-LARGE-INTEGER) TM))))

(defthm tm-fun-limited-thm
  (implies
   (and
    (realp a)
    (realp tm)
    (standard-numberp tm))
   (i-limited (tm-fun tm))))

(defthm tm-fun-standard-part-thm
  (implies
   (and
    (realp a)
    (realp tm)
    (standard-numberp tm))
   (equal (standard-part (tm-fun tm))
          tm)))

(defthm tm-fun-rw-1-thm
  (implies
   (and
    (realp tm)
    (standard-numberp tm))
   (equal (* (/ (I-LARGE-INTEGER))
             (FLOOR1 (* (I-LARGE-INTEGER) TM)))
          (tm-fun tm))))

(defthm tm-fun-rw-2-thm
  (implies
   (and
    (realp a)
    (realp tm)
    (standard-numberp tm))
   (equal (* (/ (I-LARGE-INTEGER)) a
             (FLOOR1 (* (I-LARGE-INTEGER) TM)))
          (* (tm-fun tm) a)))
:hints (("Goal" :in-theory (disable tm-fun-rw-1-thm))))

(defthm tm-fun-rw-3-thm
  (implies
   (and
    (realp a)
    (realp tm)
    (standard-numberp tm))
   (equal (* (/ (I-LARGE-INTEGER))
             (FLOOR1 (* (I-LARGE-INTEGER) TM))
             a)
          (* (tm-fun tm) a)))
:hints (("Goal" :in-theory (disable tm-fun-rw-1-thm tm-fun-rw-2-thm))))

(defthm tm-fun-rw-4-thm
  (implies
   (and
    (realp a)
    (realp b)
    (realp tm)
    (standard-numberp tm))
   (equal (* (/ (I-LARGE-INTEGER))
             a
             (FLOOR1 (* (I-LARGE-INTEGER) TM))
             b)
          (* (tm-fun tm) a b)))
:hints (("Goal" :in-theory (disable tm-fun-rw-1-thm
                                    tm-fun-rw-2-thm
                                    tm-fun-rw-3-thm))))


;; tm-fun should be disabled. Enabling it may result in rewrite loops.
(in-theory (disable tm-fun))

(defun eps-n-fun (eps n)
  (* eps n))

(defthm eps-n-fun-type-thm
  (implies
   (and
    (realp eps)
    (realp n))
   (realp (eps-n-fun eps n)))
:rule-classes :type-prescription)
```

```
(defthm eps-n-fun-0-thm
   (equal (eps-n-fun eps 0) 0))

(defthm eps-n-fun-limited-thm
  (implies
   (and
    (realp eps)
    (integerp n)
    (i-limited (* eps n)))
   (i-limited (eps-n-fun eps n))))

(defthm eps-n-fun-rw-1-thm
  (implies
   (and
    (integerp n)
    (realp eps)
    (syntaxp (eq eps 'eps))
    (i-limited (* eps n)))
   (equal (* eps n)
          (eps-n-fun eps n))))

(defthm eps-n-fun-rw-2-thm
  (implies
   (and
    (realp a)
    (syntaxp (eq eps 'eps))
    (integerp n)
    (realp eps)
    (i-limited (* eps n)))
   (equal (* eps a n)
          (* (eps-n-fun eps n) a)))
:hints (("Goal" :in-theory (disable eps-n-fun-rw-1-thm))))

(defthm eps-n-fun-rw-3-thm
  (implies
   (and
    (realp a)
    (syntaxp (eq eps 'eps))
    (integerp n)
    (realp eps)
    (i-limited (* eps n)))
   (equal (* eps n a)
          (* (eps-n-fun eps n) a)))
:hints (("Goal" :in-theory (disable eps-n-fun-rw-1-thm
                                    eps-n-fun-rw-2-thm))))

(defthm eps-n-fun-rw-4-thm
  (implies
   (and
    (realp a)
    (realp b)
    (syntaxp (eq eps 'eps))
    (integerp n)
    (realp eps)
    (i-limited (* eps n)))
   (equal (* eps a n b)
          (* (eps-n-fun eps n) a b)))
:hints (("Goal" :in-theory (disable eps-n-fun-rw-1-thm
                                    eps-n-fun-rw-2-thm
                                    eps-n-fun-rw-3-thm))))

;; eps-n-fun should be disabled. Enabling it may result in rewrite loops.
(in-theory (disable eps-n-fun))

(defun tm-eps-fun (tm eps)
  (* eps
     (FLOOR1 (/ TM eps))))

(defthm tm-eps-type
  (implies
   (realp eps)
   (realp (tm-eps-fun tm eps)))
:rule-classes :type-prescription
:hints (("Goal" :in-theory (enable tm-eps-fun))))

(defthm tm-eps-pos-thm
  (implies
   (and
    (realp tm)
    (realp eps)
    (<= 0 tm)
    (< 0 eps))
   (<= 0 (tm-eps-fun tm eps)))
:rule-classes :type-prescription
:hints (("Goal" :in-theory (enable tm-eps-fun))))

(defthm tm-eps-fun-standard-part-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (i-small eps)
    (realp tm)
    (standard-numberp tm))
   (equal (standard-part (tm-eps-fun tm eps))
          tm))
:hints (("Goal" :use ((:instance tm-floor-thm-hint-1)))))

(defthm tm-eps-fun-limited-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (i-small eps)
    (realp tm)
    (standard-numberp tm))
   (i-limited (tm-eps-fun tm eps)))
:hints (("Goal" :in-theory (disable tm-eps-fun)
                :use ((:instance standard+small->i-limited
                             (x (standard-part (tm-eps-fun tm eps)))
                             (eps (- (tm-eps-fun tm eps)
                                     (standard-part
                                         (tm-eps-fun tm eps)))))))))

(defthm tm-eps-fun-rw-1-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (i-small eps)
    (realp tm)
    (standard-numberp tm))
   (equal (* eps
             (FLOOR1 (* (/ EPS) TM)))
          (tm-eps-fun tm eps))))

(defthm tm-eps-fun-rw-2-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (i-small eps)
    (realp a)
    (realp tm)
    (standard-numberp tm))
   (equal (* eps a
             (FLOOR1 (* (/ EPS) TM)))
          (* (tm-eps-fun tm eps) a)))
:hints (("Goal" :in-theory (disable tm-eps-fun-rw-1-thm))))

(defthm tm-eps-fun-rw-3-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (i-small eps)
    (realp a)
    (realp tm)
    (standard-numberp tm))
   (equal (* eps
             (FLOOR1 (* (/ EPS) TM))
             a)
          (* (tm-eps-fun tm eps) a)))
:hints (("Goal" :in-theory (disable tm-eps-fun-rw-1-thm
                                    tm-eps-fun-rw-2-thm))))

(defthm tm-eps-fun-rw-4-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (i-small eps)
    (realp a)
    (realp b)
    (realp tm)
    (standard-numberp tm))
   (equal (* eps
             a
             (FLOOR1 (* (/ EPS) TM))
             b)
          (* (tm-eps-fun tm eps) a b)))
:hints (("Goal" :in-theory (disable tm-eps-fun-rw-1-thm
                                    tm-eps-fun-rw-2-thm
                                    tm-eps-fun-rw-3-thm))))

;; tm-eps-fun should be disabled. Enabling it may result in rewrite loops.
(in-theory (disable tm-eps-fun))
```

A.4 Exponent Function

This section presents a printout of file `eexp.lisp`. This file consists of definitions for the exponent function, e^x, for real x. Rather than explicitly defining the exponent function as in [29], we add the function symbol `eexp` to the ACL2r environment and define axioms about `eexp` which we use in our proofs.

```
(in-package "ACL2")

(include-book "arith-nsa4")

(defstub eexp (x) t)

(defaxiom eexp-type
  (implies
   (realp x)
   (and
    (<= 0 (eexp x))
    (realp (eexp x))))
:rule-classes :type-prescription)

(defaxiom eexp-standard-thm
  (implies
   (and
    (realp x)
    (standard-numberp x))
   (standard-numberp (eexp x)))
:rule-classes :type-prescription)

(defaxiom eexp-standard-part-thm
  (implies
   (and
    (realp x)
    (i-limited x))
   (equal (standard-part (eexp x))
          (eexp (standard-part x)))))

(defaxiom eexp-i-limited-thm
  (implies
   (and
    (realp x)
    (i-limited x))
   (i-limited (eexp x)))
:rule-classes ((:type-prescription) (:rewrite)))

(defaxiom eexp-0
  (equal (eexp 0)
         1))

(defaxiom eexp-monotone
  (implies
   (and
    (realp x)
    (realp y)
    (< x y))
   (< (eexp x) (eexp y))))

(defaxiom eexp-pos-arg
  (implies
   (and
    (realp x)
    (< 0 x))
   (< 1 (eexp x)))
:rule-classes ((:linear) (:type-prescription)))

(defaxiom eexp-neg-arg
  (implies
   (and
    (realp x)
    (< x 0))
   (< (eexp x) 1))
:rule-classes :linear)

(defaxiom 1+x-<=eexp-thm
  (implies
   (and
    (realp x)
    (<= 0 x))
   (<= (+ 1 x)
       (eexp x)))
:rule-classes :linear)

(defaxiom eexp-plus-thm
  (implies
   (and
    (realp x)
    (realp y))
   (equal (* (eexp x) (eexp y))
          (eexp (+ x y)))))

(defthm eexp-plus-thm-3way
  (implies
   (and
    (realp x)
    (realp y)
    (realp z))
   (equal (* (eexp x) (eexp y) z)
          (* (eexp (+ x y)) z))))
```

A.5 Nonstandard Analysis Theorems

This section presents a printout of file `nsa.lisp`. This file consists of definitions of basic nonstandard theorems about the reals which we use in our proofs.

```
(in-package "ACL2")

(include-book "arith-nsa4")
(include-book "abs")

(deflabel nsa-theory-start)

(defthm standard-part-abs-0-thm
  (iff (= (standard-part (abs x)) 0)
       (= (standard-part x) 0)))

(defthm standard-numberp-times-type-thm
  (implies
   (and
    (standard-numberp x)
    (standard-numberp y))
   (standard-numberp (* x y)))
:rule-classes ((:type-prescription)))

(defthm standard-numberp-plus-type-thm
  (implies
   (and
    (standard-numberp x)
    (standard-numberp y))
   (standard-numberp (+ x y)))
:rule-classes ((:type-prescription)))

(encapsulate ()
 (local (defthm arith-4
          (implies
           (and
            (realp x)
            (realp y)
            (not (equal x 0))
            (not (equal y 0))
            (<= (abs x)
                (abs y)))
          (<= (abs (/ y))
              (abs (/ x))))
         :rule-classes nil))

 (defthm standard-bound-x-implies-limited-x-thm
   (implies
    (and
     (standard-numberp y)
     (realp y)
     (realp x)
     (not (equal y 0))
     (<= (abs x)
         (abs y)))
    (i-limited x))
   :rule-classes nil
   :hints (("Goal" :cases ((equal y 0) (not (equal y 0)))
            :in-theory (disable abs))
           ("Goal'" :use ((:instance arith-4)
                          (:instance standard-part-<= (x (abs (/ y)))
                                     (y (abs (/ x))))))))

 (defthm limited-bound-x-implies-limited-x-thm
   (implies
    (and
     (i-limited y)
     (realp y)
     (realp x)
     (not (equal y 0))
     (<= (abs x)
         (abs y)))
    (i-limited x))
   :rule-classes nil
   :hints (("Goal" :cases ((i-small y) (not (i-small y)))
            :in-theory (disable abs))
           ("Subgoal 2" :use ((:instance arith-4)
                              (:instance standard-part-<= (x (abs (/ y)))
                                         (y (abs (/ x))))))
           ("Subgoal 1" :use ((:instance arith-4)
                              (:instance standard-part-<= (x (abs (/ y)))
                                         (y (abs (/ x)))))))))

(defthm plus-limited
  (implies
   (and
    (realp x)
    (realp y)
    (i-limited x)
    (i-limited y))
   (i-limited (+ x y)))
:rule-classes :rewrite
:hints (("Goal" :use ((:instance standard+small->i-limited
                        (x (standard-part (+ x y)))
                        (eps (- (+ x y) (standard-part (+ x y)))))))))

(defthm times-limited
  (implies
   (and
    (realp x)
    (realp y)
    (i-limited x)
    (i-limited y))
   (i-limited (* x y)))
:rule-classes :rewrite
:hints (("Goal"  :use ((:instance standard+small->i-limited
                            (x (standard-part (* x y)))
                            (eps (- (* x y) (standard-part (* x y)))))))))

(defthm divide-limited
  (implies
   (and
    (realp x)
    (realp y)
    (i-limited x)
    (i-limited y)
    (not (i-small y)))
   (i-limited (/ x y)))
:rule-classes :rewrite
:hints (("Goal"  :use ((:instance standard+small->i-limited
                            (x (standard-part (/ x y)))
                            (eps (- (/ x y) (standard-part (/ x y)))))))))

(defthm btwn-0-and-1-limited-thm
  (implies
   (and
    (realp a)
    (< 0 a)
    (< a 1))
   (i-limited a))
:rule-classes nil
:hints (("Goal"  :cases ((i-small a) (not (i-small a)))))
```

```
        ("Subgoal 1" :use ((:instance (:theorem (implies
                                                  (and
                                                   (realp x)
                                                   (realp y)
                                                   (not (equal x 0))
                                                   (not (equal y 0))
                                                   (< 0 x)
                                                   (< x 1))
                                                  (< 1 (/ x)))))
                           (:instance standard-part-<= (x 1)
                                 (y (/ a)))))))

(defthm sandwich-limited-thm-hint-1
  (implies
   (and
    (realp u)
    (realp v)
    (realp a)
    (< 0 a)
    (< a 1)
    (i-limited u)
    (i-limited v))
   (i-limited (+ v (* a (- u v)))))
:rule-classes nil
:hints (("Goal" :use ((:instance btwn-0-and-1-limited-thm)))))

(defthm sandwich-limited-thm
  (implies
   (and
    (realp u)
    (realp v)
    (realp x)
    (< u x)
    (< x v)
    (i-limited u)
    (i-limited v))
   (i-limited x))
:rule-classes nil
:hints (("Goal" :in-theory (disable i-large
                                    DISTRIBUTIVITY
                                    distributivity-left)
                :use ((:instance  sandwich-limited-thm-hint-1
                           (a (/ (- x v) (- u v))))))))

(defthm /-large-integer-is-ismall-thm
  (i-small (/ (i-large-integer)))
:hints (("Goal" :in-theory (disable i-large-integer-is-large)
                :use ((:instance i-large-integer-is-large)))))

(deftheory nsa-theory
           (set-difference-theories
             (universal-theory :here)
             (universal-theory 'nsa-theory-start)))

(defthm standard-part-limited-thm
 (implies
  (and
   (realp x)
   (i-limited x))
 (i-limited (standard-part x)))
:hints (("Goal" :use ((:instance standards-are-limited
                           (x (standard-part x)))))))


(defthm /-large-integer-standard-part-thm
 (equal (STANDARD-PART (/ (I-LARGE-INTEGER)))
        0)
:hints (("Goal" :in-theory (disable /-LARGE-INTEGER-IS-ISMALL-THM)
                :use ((:instance /-LARGE-INTEGER-IS-ISMALL-THM)))))

(defthm /-large-integer-limited-thm
 (i-limited (/ (I-LARGE-INTEGER)))
:rule-classes ((:type-prescription) (:rewrite))
:hints (("Goal" :in-theory (disable /-LARGE-INTEGER-IS-ISMALL-THM)
                :use ((:instance /-LARGE-INTEGER-IS-ISMALL-THM)))))
```

A.6 Computed Hints

This section presents a printout of file `computed-hints.lisp`. This file consists of computed hint definitions which generate automatic hints for the theorem prover whereby it may open function definitions when the flag `stable-under-simplificationp` is true.

A computed hint is also defined for applying standard part to both sides of an inequality.

```
(in-package "ACL2")

;;Apply the given list of hints in sequence on every instance
;; of stable under simplification of the current clause
;; The hint sequence made available to the subgoals
;; depends on restart-on-new-id and counter.
;; Counter indicates the current position of the
;; hint to be used from the hint-lst.
;; If restart-on-new-id is t, then start from the
;; beginning of the hint-lst for a child goal.
;; If restart-on-new-id is nil, then start from the
;; counter position in the hint-lst for a child goal.
(defun staged-hints (stable-under-simplificationp
                     restart-on-new-id
                     hint-lst
                     id
                     last-id
                     counter)
  (let ((calc-last-id (if (equal id nil) 'id (list 'quote id))))
  (cond ((and stable-under-simplificationp (not (endp hint-lst)))
         (cond
          ((and restart-on-new-id
                (not (equal id last-id)))
           (append `(:computed-hint-replacement
                     ((staged-hints stable-under-simplificationp
                                    ,restart-on-new-id
                                    ,(list 'quote hint-lst)
                                    id ,calc-last-id 1)))
                   (nth 0 hint-lst)))
          ((< counter (len hint-lst))
           (append `(:computed-hint-replacement
                        ((staged-hints stable-under-simplificationp
                                       ,restart-on-new-id
                                       ,(list 'quote hint-lst)
                                       id ,calc-last-id ,(1+ counter))))
                   (nth counter hint-lst)))
          (t nil)))
        (t nil))))

#|
;;example use of staged-hints
(set-default-hints '((staged-hints
                        stable-under-simplificationp
                         nil ;;restart on new id
                         '((:in-theory (enable abs
                                               equal-cancel-divisors
                                               <-cancel-divisors)))
                         nil nil 0)))
|#



;;It is assumed that the symbol hyps represents
;; 1) a list of predicates that are conjoined in a hypothesis of
;; a goal, or 2) a list of only one predicate (not conjoined
;; with any predicate) which is the hypothesis of a goal.
;; The function scans through the list.
;; If the predicate is an inequality (< or <=), then an
;; :instance form to be used in a :use hint
;; is generated, where the :instance form is (:instance
;; standard-part-<= (x a) (y b)), where the symbols a and b
;; represent the first and second arguments, respectively,
;; of the inequality predicate.
;; If the predicate is not an inequality (< or <=), then it is ignored.
;; The function returns a list of :instance forms,
;; where each :instance form
;; corresponds to an inequality predicate in hyps. If no inequality
;; predicates (< or <=) are members of
;; hyps, then the function returns nil.
;;
;; The rewrite rule STANDARD-PART-<= should be disabled
;; so that the rewriter does not rewrite to true
;; the added hypothesis resulting from the :use hint.

(defun make-standard-part-<=-hint-from-hyps (hyps)
  (cond
   ((atom hyps) nil)
   ((or (equal (caar hyps) '<=)
        (equal (caar hyps) '<)) (append '((:instance standard-part-<=
                                              (x ,(nth 1 (car hyps)))
                                              (y ,(nth 2 (car hyps)))))
                            (make-standard-part-<=-hint-from-hyps
                                         (cdr hyps))))
   (t (make-standard-part-<=-hint-from-hyps (cdr hyps)))))

;;Explore the given clause, and attempt to generate a hint from it.
;; If the clause is not an implication, no hint is generated
;; (nil is returned).
;; If the hypothesis is a conjunct of predicates,
;; then a list is generated whose elements
;; are the conjuncts. This list is passed to
;; make-standard-part-<=-hint-from-hyps.
;; If the hypothesis is not a conjunct, then a list is
;; generated consisting only of this hypothesis.
;; This list is then passed to
;; make-standard-part-<=-hint-from-hyps.
;; If the results of the call to
;; make-standard-part-<=-hint-from-hyps is nil, then
;; no hint is generated and nil is returned.
;; If the results of the call to make-standard-part-<=-hint-from-hyps
;; is non-nil, then this result is used to form and return a
;; use hint of the form (:use (t1 ... tn)), where each of
;; t1...tn are of the form
;; (:instance standard-part-<= (x a) (y b)),
;; where the symbols a and b represent the first and second
;; arguments, respectively, of an inequality predicate
;; (either < or <=) which appears in the list
;; passed to make-standard-part-<=-hint-from-hyps, and n
;; represents the number of inequality predicates (either < or <=)
;; which appear in this list. One such ti, 1<=i<=n, is generated for
;; each inequality predicate (either < or <=) which appears in this list.
;; It is assumed that this function is to be called from a
;; computed hint, and that the computed hint fires only when
;; stable-under-simplificationp is true.

(defun make-standard-part-<=-hint (clause)
  (cond
   ((and (equal (car clause) 'implies)
         (equal (caadr clause) 'and))
    (let ((hints (make-standard-part-<=-hint-from-hyps (cdadr clause))))
      (if (not (equal hints nil))
          `(:use ,hints)
        nil)))
   ((and (equal (car clause) 'implies))
    (let ((hints (make-standard-part-<=-hint-from-hyps
                                          (list (cadr clause)))))
      (if (not (equal hints nil))
          `(:use ,hints)
        nil)))
   (t nil)))

(defun standard-part-hint (stable-under-simplificationp clause)
  (declare (xargs :mode :program))
  (cond (stable-under-simplificationp
         (make-standard-part-<=-hint (prettyify-clause clause nil nil)))
        (t nil)))
```

A.7 The Solution ϕ Exists

This section presents a printout of file `phi-exists.lisp`. This file consists of definitions which show that the standard function ϕ exists. In Chapter 4, we referred to the Euler approximation as ρ. Here, we refer to the Euler approximation as `run`. Unlike the presentation in Chapter 4, the `run` function is defined in terms of a natural number `n` representing the iteration number in implementing the Euler method. Therefore, when defining ϕ in terms of `run`, we substitute $\lfloor t/\epsilon \rfloor$ for `n`.

```
(in-package "ACL2")

(include-book "arith-nsa4")
(include-book "nsa")
(include-book "eexp")

(defstub f (x) t)

(defstub L () t)

(defstub h (x1 x2 eps) t)
(defaxiom L-type
  (and
   (realp (L))
   (< 0 (L)))
:rule-classes :type-prescription)

(defaxiom L-standard-thm
  (standard-numberp (L))
:rule-classes :type-prescription)

(defaxiom L-i-limited-thm
  (i-limited (L))
:rule-classes  ((:type-prescription) (:rewrite)))

(defaxiom f-type
  (realp (f x))
:rule-classes :type-prescription)
```

```
(defaxiom f-standard-thm
  (implies
   (and
    (realp x)
    (standard-numberp x))
   (standard-numberp (f x)))
:rule-classes :type-prescription)

(defaxiom f-i-limited-thm
  (implies
   (and
    (realp x)
    (i-limited x))
   (i-limited (f x)))
:rule-classes  ((:type-prescription) (:rewrite)))

(defaxiom f-lim-thm
  (implies
   (and
    (realp x1)
    (realp x2))
   (and
    (<= (abs (- (f x1) (f x2)))
        (* (L) (abs (- x1 x2)))))))

(defun step1 (x eps)
  (+ x (* (f x) eps)))

(defun run (x n eps)
  (cond
   ((zp n) x)
   (t (run (step1 x eps) (- n 1) eps))))

(defun run-limit (delta n eps)
  (cond
   ((zp n) delta)
   (t (* (+ 1 (* eps (L))) (run-limit delta (- n 1) eps)))))

(defthm run-limit-realp
  (implies
   (and
    (realp delta)
    (realp eps))
   (realp (run-limit delta n eps)))
:rule-classes :type-prescription)
```

```
(defthm run-limit-n+1-thm
  (implies
   (and
    (realp eps)
    (realp delta)
    (<= 0 delta)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
  (<= (run-limit delta n eps)
      (run-limit delta (+ n 1) eps)))
:hints (("Goal" :in-theory (disable distributivity-left))))

(defthm abs-step-thm
  (implies
   (and
    (realp x1)
    (realp x2)
    (realp eps)
    (< 0 eps))
   (<= (abs (- (step1 x1 eps)
               (step1 x2 eps)))
          (+ (abs (- x1 x2))
             (abs (* eps (- (f x1) (f x2)))))))
:rule-classes :linear)

(defthm step-1+leps-thm-1
  (implies
   (and
    (realp x1)
    (realp x2)
    (realp eps)
    (< 0 eps))
   (<= (abs (- (step1 x1 (/ eps))
               (step1 x2 (/ eps))))
          (* (+ 1 (* (L) (/ eps))) (abs (- x1 x2)))))
:hints (("Goal"
         :use ((:instance f-lim-thm)
               (:instance abs-step-thm (eps (/ eps))))))
:rule-classes nil)

(defthm step-1+leps-thm
  (implies
   (and
    (realp x1)
    (realp x2)
    (realp eps)
    (< 0 eps))
   (<= (abs (- (step1 x1 eps)
                  (step1 x2 eps)))
          (* (+ 1 (* (L) eps)) (abs (- x1 x2)))))
:hints (("Goal"
         :use ((:instance step-1+leps-thm-1 (eps (/ eps)))))))

(defthm run-realp
  (implies
   (and
    (realp x)
    (realp eps))
   (realp (run x n eps)))
:rule-classes :type-prescription)

(defun n-scheme (n)
  (cond
   ((zp n) 0)
   (t (n-scheme (- n 1)))))

(defthm step-run-thm
  (implies
   (and
    (integerp n)
    (<= 0 n)
    (realp x)
    (realp eps))
   (equal (step1 (run x n eps) eps)
          (run x (+ n 1) eps))))

(defthm run-limit-thm
  (implies
   (and
    (realp x1)
    (realp x2)
    (realp eps)
    (< 0 eps))
   (<= (abs (- (run x1 n eps) (run x2 n eps)))
       (run-limit (abs (- x1 x2)) n eps)))
:hints (("Goal" :do-not '(generalize)
                :induct (n-scheme n)
                :in-theory (disable abs)
```

```
               :nonlinearp t)
        ("Subgoal *1/2" :in-theory (disable abs <-*-LEFT-CANCEL)
                        :use ((:instance run-limit-n+1-thm
                                   (n (- n 1))
                                   (delta (abs (- (step1 x1 eps)
                                                  (step1 x2 eps)))))
                              (:instance step-1+leps-thm
                                   (x1 (run x1 (- n 1) eps))
                                   (x2 (run x2 (- n 1) eps)))
                              (:instance pos-factor-<=-thm
                                   (x (ABS (+ (RUN X1 (+ -1 N) eps)
                                              (* -1 (RUN X2
                                                         (+ -1 N)
                                                         eps)))))
                                   (y (RUN-LIMIT (ABS (+ X1 (* -1 X2)))
                                                 (+ -1 N) eps))
                                   (a (+ 1 (* (L) EPS))))))))

(defthm eexp-1+epsl-thm
  (implies
   (and
    (realp eps)
    (integerp n)
    (< 0 eps)
    (realp delta)
    (<= 0 delta)
    (<= 0 n))
   (<= (* delta (eexp (* (- n 1) eps (L)))
          (+ 1 (* eps (L))))
       (* delta (eexp (* n eps (L))))))
:hints (("Goal" :in-theory (disable 1+x-<=eexp-thm)
                :use ((:instance 1+x-<=eexp-thm (x (* eps (L))))
                      (:instance pos-factor-<=-thm
                           (x (+ 1 (* eps (L))))
                           (y (eexp (* eps (L))))
                           (a (* delta
                                 (eexp (+ (* n eps (L))
                                          (* -1 eps (L))))))))))
:rule-classes nil)

(defthm run-limit-eexp-thm-1
  (implies
   (and
    (realp eps)
    (realp delta)
    (<= 0 delta)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (<= (run-limit delta n eps)
       (* delta (eexp (* eps (L) n)))))
:hints (("Goal" :do-not '(generalize))
        ("Subgoal *1/4" :use ((:instance pos-factor-<=-thm
                                (x (RUN-LIMIT DELTA (+ -1 N) eps))
                                (y (* DELTA
                                   (EEXP (+ (* -1 (L) EPS)
                                            (* (L) EPS N)))))
                                (a (+ 1 (* eps (L)))))
                             (:instance eexp-1+epsl-thm))))
:rule-classes nil)

(defthm run-diff-limit-eexp-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (realp x1)
    (realp x2)
    (integerp n)
    (<= 0 n))
   (<= (abs (- (run x1 n eps) (run x2 n eps)))
       (* (abs (- x1 x2)) (eexp (* eps (L) n)))))
:hints (("Goal" :do-not '(generalize))
        ("Subgoal *1/2" :in-theory (disable abs)
                        :use ((:instance run-limit-thm)
                              (:instance run-limit-eexp-thm-1
                                (delta (abs (- x1 x2)))))))
:rule-classes nil)

(defthm run-plus-thm
  (implies
   (and
    (integerp m)
    (integerp n)
    (<= 0 m)
    (<= 0 n))
   (equal (run (run x n eps) m eps)
          (run x (+ m n) eps))))

(defun run-n-limit (x n eps)
  (+ (abs x)
     (* (eexp (* (L) n eps)) (abs (f x))  n eps)))

(defthm f-step-thm-hint1
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (realp x))
   (<= (abs (f (step1 x eps)))
       (+ (abs (f x)) (abs (- (f (step1 x eps)) (f x))))))
:rule-classes nil)

(defthm f-step-thm-hint2
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (realp x))
   (<= (abs (f (step1 x eps)))
       (* (eexp (* (L) eps)) (abs (f x)))))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs)
                :use ((:instance f-step-thm-hint1)
                      (:instance f-lim-thm (x1 (+ x (* (f x) eps)))
                                           (x2 x))
                      (:instance 1+x-<=eexp-thm (x (* (L) eps)))
                      (:instance pos-factor-<=-thm (x (+ 1 (* eps (L))))
                                         (y (eexp (* eps (L) )))
                                         (a (abs (f x))))))))

(defthm arith-3
  (implies
   (and
    (integerp n)
    (< 0 n))
   (<= 0 (- n 1)))
:rule-classes :type-prescription)

(defthm f-step-thm-hint3
 (implies
  (and
   (integerp n)
   (< 0 n)
   (realp x)
   (realp eps)
   (< 0 eps))
  (<= 0
      (* (EEXP (* -1 EPS (L)))
         (EEXP (* EPS (L) N))
         (+ -1 N)
         EPS)))
:rule-classes nil)

(defthm step1-type-thm
  (implies
   (and
    (realp x)
    (realp eps))
   (realp (step1 x eps)))
:rule-classes :type-prescription)

(defthm abs-step1-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (realp x))
   (<= (abs (step1 x eps))
       (+ (abs x)
          (* (abs (f x)) eps))))
:rule-classes nil)

(defthm run-limit-eexp-step-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (< 0 n)
    (realp x))
   (<= (run-n-limit (step1 x eps)
                  (- n 1)
                  eps)
       (run-n-limit x n eps)))
:rule-classes :linear
:hints (("Goal" :in-theory (disable abs <-*-LEFT-CANCEL)
                :use ((:instance abs-step1-thm)
                      (:instance f-step-thm-hint3)
                      (:instance f-step-thm-hint2)
                      (:instance pos-factor-<=-thm (x 1)
                                       (y (eexp (* eps (L) n)))
                                       (a (* (abs (f x)) eps)))
                      (:instance pos-factor-<=-thm
                           (x (abs (f (step1 x eps))))
                           (y (* (eexp (* (L) eps)) (abs (f x))))
                           (a (* (eexp (* -1 eps (L)))
                                 (eexp (* eps (L) n))
                                 (- n 1) eps)))))))

(defthm run-limit-eexpt-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (realp x)
    (integerp n)
    (<= 0 n))
   (<= (abs (run x n eps))
       (run-n-limit x n eps)))
:rule-classes nil
:hints (("Goal" :in-theory (disable step1 run-n-limit abs))
        ("Subgoal *1/1" :in-theory (e/d (step1 run-n-limit) (abs)))))

(defun run-tm-limit (x tm)
  (+ (abs x)
     (* (eexp (* (L) tm)) (abs (f x))  tm)))

(defthm run-tm-limit-standard-thm
  (implies
   (and
    (realp tm)
    (standard-numberp tm)
    (realp x)
    (standard-numberp x))
   (standard-numberp (run-tm-limit x tm)))
:rule-classes nil)

(defthm run-n-limit-standard-thm
  (implies
   (and
    (realp eps)
    (integerp n)
    (standard-numberp (* eps n))
    (realp x)
    (standard-numberp x))
   (standard-numberp (run-n-limit x n eps)))
:rule-classes nil
:hints (("Goal" :use ((:instance run-tm-limit-standard-thm
                          (tm (* eps n)))))))

(defthm run-tm-limit-limited-thm
  (implies
   (and
    (realp tm)
    (i-limited tm)
    (realp x)
    (i-limited x))
   (i-limited (run-tm-limit x tm)))
:rule-classes nil
:hints (("Goal" :in-theory (disable i-large)
                :use ((:instance standards-are-limited)))))

(defthm run-n-limit-limited-thm
  (implies
   (and
    (realp eps)
    (integerp n)
    (i-limited (* eps n))
    (realp x)
    (i-limited x))
   (i-limited (run-n-limit x n eps)))
:rule-classes :type-prescription
:hints (("Goal" :use ((:instance run-tm-limit-limited-thm
                          (tm (* eps n)))))))

(defthm run-n-limit-type-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (integerp n))
   (realp (run-n-limit x n eps)))
:rule-classes :type-prescription)

(defthm run-standard-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n)
    (i-limited (* eps n))
    (standard-numberp x))
   (standard-numberp (standard-part (run x n eps))))
:hints (("Goal" :in-theory (disable run
                                    run-n-limit
                                    plus-limited
                                    times-limited
                                    divide-limited)
                :use ((:instance run-limit-eexpt-thm)
                      (:instance run-n-limit-limited-thm)
                      (:instance limited-bound-x-implies-limited-x-thm
                           (y (run-n-limit x n eps))
                           (x (run x n eps)))
                      (:instance standardp-standard-part
                           (x (run x n eps)))))))

(defthm floor-limited-thm-hint-1
  (implies (and (i-small (/ (i-large-integer)))
                (realp tm)
                (i-limited tm))
           (i-limited (+ (* -1 (/ (i-large-integer))) tm)))
:rule-classes nil
:hints (("goal" :in-theory (disable i-large
                                    i-small
                                    /-large-integer-is-ismall-thm))))

(defthm floor-limited-thm
  (implies
   (and
    (realp tm)
    (i-limited tm))
   (i-limited (* (/ (i-large-integer)) (floor1 (* (i-large-integer) tm)) )))
:hints (("goal" :in-theory (disable i-large)
                :use ((:instance sandwich-limited-thm
                           (u (/ (- (* tm (i-large-integer)) 1)
                                 (i-large-integer)))
                           (v (/ (* tm (i-large-integer))
                                 (i-large-integer)))
                           (x (* (/ (i-large-integer))
                                 (floor1 (* (i-large-integer) tm)) )))))
        ("subgoal 2"
                     :use ((:instance /-large-integer-is-ismall-thm)
                           (:instance floor-limited-thm-hint-1)))))

(defthm phi-thm
  (implies
    (and (standard-numberp x)
         (standard-numberp tm)
         (realp x)
         (realp tm))
   (standard-numberp (standard-part (run x (floor1 (* (i-large-integer) tm))
                                         (/ (i-large-integer))))))
:hints (("goal" :in-theory (disable i-large)
                :use ((:instance run-standard-thm
                           (n (floor1 (* tm (i-large-integer))))
                           (eps (/ (i-large-integer))))
                      (:instance floor-limited-thm)))))

;The following is the definition of the
; standard function
(defun-std phi (x tm)
  (cond
   ((not (and
          (realp x)
          (realp tm))) 0)
   (t (standard-part (run x
                          (floor1 (* tm (i-large-integer)))
                          (/ (i-large-integer)))))))
```

A.8 Properties of the Solution

This section presents a printout of file `phi-properties.lisp`. This file consists of definitions which show properties about `run` and the standard solution `phi`. These properties include continuity with respect to time t, continuity with respect to x, and time invariance. This file also includes the definition of the theorem which states that the standard part of ρ^{ϵ} is identical to the standard part of ρ^{δ}, for ϵ and δ positive infinitesimals.

```
(in-package "ACL2")

(include-book "arith-nsa4")
(include-book "abs")
(include-book "eexp")
(include-book "phi-exists")
(include-book "computed-hints")
(include-book "tm-floor")

(in-theory (disable i-large))
(in-theory (disable standard-part-<=))

(defun f-sum (x n eps)
  (cond
   ((zp n) 0)
   (t (+ (* eps (f (run x (- n 1) eps)))
         (f-sum x (- n 1) eps)))))

(defthm run-f-sum-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps))
   (equal (run x n eps)
          (+ x (f-sum x n eps))))
:rule-classes nil
:hints (("Goal" :do-not '(generalize))
        ("Subgoal *1/2" :in-theory (disable step-run-thm)
                         :use ((:instance
                                step-run-thm (n (- n 1)))))))
```

```
(defun resid (x n eps)
  (- (f-sum x n eps) (* n eps (f x))))

(defthm f-sum-type
  (implies
   (and
    (realp x)
    (realp eps))
   (realp (f-sum x n eps)))
:rule-classes :type-prescription)

(defthm run-type
  (implies
   (and
    (realp x)
    (realp eps))
   (realp (run x n eps)))
:rule-classes :type-prescription)

(defthm run-limited-thm
  (implies
   (and
    (realp x)
    (i-limited x)
    (realp eps)
    (integerp n)
    (< 0 eps)
    (<= 0 n)
    (i-limited (* eps n)))
   (i-limited (run x n eps)))
:rule-classes ((:type-prescription) (:rewrite))
:hints (("Goal" :in-theory (disable abs
                                    run
                                    run-n-limit
                                    i-large
                                    plus-limited
                                    times-limited
                                    divide-limited)
                :use ((:instance run-limit-eexpt-thm)
                      (:instance run-n-limit-limited-thm)
                      (:instance limited-bound-x-implies-limited-x-thm
                           (y (run-n-limit x n eps))
                           (x (run x n eps)))))))
```

```
(defthm resid-limited-thm
  (implies
   (and
    (realp x)
    (i-limited x)
    (integerp n)
    (<= 0 n)
    (realp eps)
    (< 0 eps)
    (i-limited (* eps n)))
   (i-limited (resid x n eps)))
:rule-classes :type-prescription
:hints (("Goal" :in-theory (disable i-large)
                :use ((:instance run-f-sum-thm)
                      (:instance run-limited-thm)))
        ("Goal'''" :use ((:instance times-limited
                              (x (* eps n))
                              (y (f x)))
                         (:instance plus-limited
                              (x (+ X (F-SUM X N EPS)))
                              (y (- x)))))))

(defthm resid-standard-thm
  (implies
   (and
    (realp x)
    (i-limited x)
    (integerp n)
    (<= 0 n)
    (realp eps)
    (< 0 eps)
    (i-limited (* eps n)))
   (standard-numberp (standard-part (resid x n eps))))
:hints (("Goal" :in-theory (disable i-large resid)
                :use ((:instance resid-limited-thm)))))


(defthm resid-std-thm
  (IMPLIES
    (AND (STANDARD-NUMBERP X)
         (STANDARD-NUMBERP TM)
         (REALP X)
         (REALP TM)
         (<= 0 TM))
   (STANDARD-NUMBERP (STANDARD-PART
                             (resid X
                                    (FLOOR1 (* (I-LARGE-INTEGER) TM))
                                    (/ (I-LARGE-INTEGER))))))
:hints (("Goal" :in-theory (disable i-large)
                :use ((:instance resid-standard-thm
                           (n (floor1 (* tm (i-large-integer))))
                           (eps (/ (i-large-integer))))))))

(in-theory (disable resid))

(defun-std resid-tm (x tm)
  (cond
   ((not (and (realp x)
              (realp tm)
              (<= 0 tm))) 0)
   (t (standard-part (resid x
                            (floor1 (* tm (i-large-integer)))
                            (/ (i-large-integer)))))))

(in-theory (enable resid))

(defthm f-run-bound-hint-1
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps))
   (equal (f (step1 x eps))
          (+ (f x) (- (f (step1 x eps)) (f x)))))
:rule-classes nil)

(defthm f-run-bound-hint-2
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps))
   (<= (abs (f (step1 x eps)))
       (* (+ 1 (* eps (L))) (abs (f x)))))
:rule-classes nil
:hints (("Goal" :use ((:instance f-run-bound-hint-1)
                      (:instance abs-triangular-inequality-thm
                           (x (f x))
                           (y (- (f (step1 x eps)) (f x))))
                      (:instance f-lim-thm
                         (x1 (step1 x eps))
                         (x2 x))))))

(defthm f-run-bound-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (<= (abs (f (run x n eps)))
       (* (eexp (* n eps (L))) (abs (f x)))))
:hints (("Goal" :in-theory (disable abs step1)
                :do-not '(generalize))
        ("Subgoal *1/5" :use ((:instance pos-factor-<=-thm
                                   (x (abs (f (step1 x eps))))
                                   (y (* (+ 1 (* eps (L))) (abs (f x))))
                                   (a (EEXP (+ (* -1 (L) EPS)
                                               (* (L) EPS N)))))
                              (:instance pos-factor-<=-thm
                                   (x (+ 1 (* eps (L))))
                                   (y (eexp (* eps (L))))
                                   (a (* (EEXP (+ (* -1 (L) EPS)
                                                  (* (L) EPS N)))
                                         (abs (f x)))))
                              (:instance  f-run-bound-hint-2)
                              (:instance  1+x-<=eexp-thm
                                   (x (* eps (L))))))))

(defthm f-sum-exp-bound-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (<= (abs (f-sum x n eps))
       (* n eps (eexp (* eps n (L))) (abs (f x)))))
:rule-classes nil
:hints (("Goal" :induct (f-sum x n eps)
                :do-not-induct t
                :in-theory (disable abs <-*-LEFT-CANCEL)
                :do-not '(generalize))
        ("Subgoal *1/2" :use ((:instance f-run-bound-thm (n (- n 1)))
                              (:instance pos-factor-<=-thm
                                   (x  (abs (f (run x (- n 1) eps))))
                                   (y (* (eexp (* (- n 1) eps (L)))
                                               (abs (f x))))
                                   (a eps))
                              (:instance abs-triangular-inequality-thm
                                   (x (F-SUM X (+ -1 N) EPS))
                                   (y (* EPS (F (RUN X (+ -1 N) EPS)))))
                              (:instance pos-factor-<=-thm
                                   (x 1)
                                   (y (EEXP (* (L) EPS)))
                                   (a (* EPS N (ABS (F X))
                                         (EEXP (+ (* -1 (L) EPS)
                                                  (* (L) EPS N)))))
                               )))))

(defthm f-sum-diff-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (integerp m)
    (<= 0 n)
    (<= n m))
   (equal (- (f-sum x m eps)
             (f-sum x n eps))
          (f-sum (run x n eps) (- m n) eps))))

(defthm f-run-diff-eq-f-sum-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (integerp m)
    (<= 0 n)
    (<= n m))
   (equal (- (run x m eps)
             (run x n eps))
          (f-sum (run x n eps) (- m n) eps)))
:hints (("Goal" :do-not-induct t
                :use ((:instance run-f-sum-thm (n m))
                      (:instance run-f-sum-thm)
                      (:instance f-sum-diff-thm)))))

(defthm pos-*-<=-thm
  (implies
   (and
    (realp a)
    (realp b)
    (realp x)
    (realp y)
    (<= 0 a)
    (<= 0 b)
    (<= a x)
    (<= b y))
   (<= (* a b) (* x y)))
  :hints (("Goal" :use ((:instance pos-factor-<=-thm
                             (x a)
                             (y x)
                             (a b))
                        (:instance pos-factor-<=-thm
                             (x b)
                             (y y)
                             (a x))))))

(defthm  f-run-diff-tm-thm-hint
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp m)
    (integerp n)
    (<= n m))
   (<= 0 (* (- m n) eps (eexp (* (- m n) (L) eps)))))
:rule-classes nil
:hints (("Goal" :in-theory (disable distributivity
                                    distributivity-left)
                :use ((:instance pos-factor-<=-thm
                           (x 0)
                           (y (- m n))
                           (a (* eps
                                 (eexp (* (- m n)
                                          (L)
                                          eps))))
                       )))))
```

```
(defthm f-run-diff-tm-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (integerp m)
    (<= 0 n)
    (<= n m))
   (<= (abs (- (run x m eps)
                  (run x n eps)))
       (* (- m n) eps (eexp (* eps m (L))) (abs (f x)))))
:hints (("Goal" :do-not-induct t
                :in-theory (disable abs)
                :use ((:instance f-run-diff-tm-thm-hint)
                      (:instance f-run-diff-eq-f-sum-thm)
                      (:instance f-sum-exp-bound-thm
                           (x (run x n eps))
                           (n (- m n)))
                      (:instance f-run-bound-thm)
                      (:instance pos-factor-<=-thm
                           (x (abs (f (run x n eps))))
                           (y (* (eexp (* n eps (L))) (abs (f x))))
                           (a (* (- m n)
                                 eps
                                 (eexp (* eps
                                          (- m n)
                                          (L))))))))))

(defthm run-diff-tm-standard-part-thm
  (implies
   (and
    (realp x)
    (standard-numberp x)
    (integerp n)
    (integerp m)
    (realp eps)
    (< 0 eps)
    (<= 0 n)
    (<= 0 m)
    (i-limited (* eps n))
    (i-limited (* eps m))
    (equal (standard-part (* m eps))
          (standard-part (* n eps))))
  (equal (- (standard-part (run x m eps))
            (standard-part (run x n eps)))
         0))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs)
                :cases ((<= n m) (< m n))
                :do-not-induct t)
        ("Subgoal 2" :use ((:instance f-run-diff-tm-thm)))
        ("Subgoal 1" :use ((:instance f-run-diff-tm-thm
                                (m n)
                                (n m))))))

(defthm floor1-large-1<=
  (implies
   (and
    (realp x)
    (< 0 x)
    (standard-numberp x))
  (i-large (* (i-large-integer) x)))
:hints (("Goal" :in-theory (enable i-large))))

(defthm large-gt-1
  (implies
   (and
    (realp x)
    (< 0 x)
    (i-large x))
   (< 1 x))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (enable i-large)
                :use ((:instance standard-part-<=
                           (x 1)
                           (y (/ x)))))))


(defthm standard-diff-*-large-thm
  (implies
   (and
    (realp x)
    (realp y)
    (standard-numberp x)
    (standard-numberp y)
    (< x y))
  (<= (+ (* (i-large-integer) x) 1)
      (* (i-large-integer) y)))
:hints (("Goal" :use ((:instance floor1-large-1<= (x (- y x)))
                      (:instance large-gt-1
                           (x (+ (* -1 (I-LARGE-INTEGER) X)
                                 (* (I-LARGE-INTEGER) Y))))
                      (:instance pos-factor-<-thm
                           (x 0)
                           (y (- y x))
                           (a (i-large-integer)))))))

(defthm abs-standard-numberp
  (implies
   (and
    (realp x)
    (standard-numberp x))
   (standard-numberp (abs x))))

(defthm-std phi-diff-tm-thm
  (implies
   (and
    (realp x)
    (realp tm1)
    (realp tm2)
    (<= 0 tm1)
    (<= tm1 tm2))
   (<= (abs (- (phi x tm2)
                  (phi x tm1)))
          (* (- tm2 tm1) (eexp (* tm2 (L))) (abs (f x)))))
:rule-classes nil
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs <-CANCEL-DIVISORS)
                :use ((:instance f-run-diff-tm-thm
                           (eps (/ (i-large-integer)))
                           (n (floor1 (* tm1 (i-large-integer))))
                           (m (floor1 (* tm2 (i-large-integer)))))
                      (:instance standard-diff-*-large-thm
                           (x TM1)
                           (y TM2))))))
```

```
;;--------------------------------------------
;; The following is the theorem which states
;;  that phi is continuous with respect to time
;;--------------------------------------------
(defthm phi-tm-continuous-thm
  (implies
   (and
    (realp x)
    (standard-numberp x)
    (realp tm1)
    (<= 0 tm1)
    (i-limited tm1))
   (equal (standard-part (phi x tm1))
          (phi x (standard-part tm1))))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs)
                :cases ((<= tm1 (standard-part tm1))
                        (< (standard-part tm1) tm1)))
        ("Subgoal 2" :use ((:instance phi-diff-tm-thm
                                 (tm1 tm1)
                                 (tm2 (standard-part tm1)))))
        ("Subgoal 1" :use ((:instance phi-diff-tm-thm
                                 (tm2 tm1)
                                 (tm1 (standard-part tm1)))))))

(defthm run-plus-thm
  (implies
   (and
    (integerp m)
    (integerp n)
    (<= 0 m)
    (<= 0 n))
   (equal (run (run x n eps) m eps)
          (run x (+ m n) eps))))

(defthm phi-diff-hint-1
  (implies
   (and
    (standard-numberp x1)
    (standard-numberp x2)
    (standard-numberp tm)
    (realp x1)
    (realp x2)
    (realp tm)
    (<= 0 tm))
   (equal (STANDARD-PART (* (EEXP (* (L) (/ (I-LARGE-INTEGER))
                                    (FLOOR1 (* (I-LARGE-INTEGER) TM))))
                            (ABS (+ X1 (* -1 X2)))))
          (* (EEXP (* (L) TM))
             (ABS (+ X1 (* -1 X2))))))
:rule-classes nil
:hints (("Goal" :in-theory (disable *-commut-3way))))

(defthm-std phi-x-diff-thm
  (implies
   (and
    (realp x1)
    (realp x2)
    (realp tm)
    (<= 0 tm))
   (<= (abs (- (phi x1 tm) (phi x2 tm)))
          (* (abs (- x1 x2)) (eexp (* tm (L))))))
:rule-classes nil
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs i-large )
                :use ((:instance run-diff-limit-eexp-thm
                           (n (floor1 (* tm (i-large-integer))))
                           (eps (/ (i-large-integer))))
                      (:instance phi-diff-hint-1)))))

(defthm run-x-continuous-thm
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (realp x)
    (i-limited x)
    (integerp n)
    (i-limited (* eps n))
    (<= 0 n))
   (equal (standard-part (run (standard-part x) n eps))
          (standard-part (run x n eps))))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs)
                :do-not-induct t
                :use ((:instance run-diff-limit-eexp-thm
                           (x1 x)
                           (x2 (standard-part x)))))))
```

```
;;------------------------------------------
;; The following is the theorem which states
;;  that phi is continuous with respect to x
;;------------------------------------------
(defthm phi-x-continuous-thm
  (implies
   (and
    (realp x)
    (i-limited x)
    (realp tm)
    (standard-numberp tm)
    (<= 0 tm))
   (equal (standard-part (phi x tm))
          (phi (standard-part x) tm)))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs)
                :use ((:instance phi-x-diff-thm
                           (x1 x)
                           (x2 (standard-part x)))))))

(defthm phi-plus-hint1
  (implies
   (and
    (realp tm1)
    (realp tm2))
   (equal (standard-part (* (+ (FLOOR1 (* (I-LARGE-INTEGER) TM1))
                             (FLOOR1 (* (I-LARGE-INTEGER) TM2)))
                             (/ (I-LARGE-INTEGER))))
          (standard-part (* (floor1 (* (i-large-integer) (+ tm1 tm2)))
                             (/ (i-large-integer))))))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable <-CANCEL-DIVISORS)
                :use ((:instance pos-factor-<-thm
                           (x (+ (* (I-LARGE-INTEGER) TM1)
                                 (* (I-LARGE-INTEGER) TM2)
                                 -2))
                           (y (+ (FLOOR1 (* (I-LARGE-INTEGER) TM1))
                                 (FLOOR1 (* (I-LARGE-INTEGER) TM2))))
                           (a (/ (i-large-integer))))
                      (:instance pos-factor-<=-thm
                           (x (+ (FLOOR1 (* (I-LARGE-INTEGER) TM1))
                                 (FLOOR1 (* (I-LARGE-INTEGER) TM2))))
                           (y (+ (* (I-LARGE-INTEGER) TM1)
                                 (* (I-LARGE-INTEGER) TM2)))
                           (a (/ (i-large-integer))))
                      (:instance pos-factor-<-thm
                           (x (+ (* (i-large-integer) (+ tm1 tm2)) -1))
                           (y (floor1 (* (i-large-integer) (+ tm1 tm2))))
                           (a (/ (i-large-integer))))
                      (:instance pos-factor-<=-thm
                           (x (floor1 (* (i-large-integer) (+ tm1 tm2))))
                           (y (* (i-large-integer) (+ tm1 tm2)))
                           (a (/ (i-large-integer))))))))

(defthm tm1+tm2-limited-thm
  (implies
   (and
    (standard-numberp tm1)
    (standard-numberp tm2)
    (realp tm1)
    (realp tm2))
   (i-limited (* (/ (i-large-integer))
                 (floor1 (* (i-large-integer) (+ tm1 tm2))))))
:rule-classes nil
:hints (("Goal" :use ((:instance phi-plus-hint1)
                      (:instance standard+small->i-limited
                           (x (standard-part
                                 (* (/ (i-large-integer))
                                    (floor1 (* (i-large-integer)
                                               (+ tm1 tm2))))))
                           (eps (- (* (/ (i-large-integer))
                                      (floor1 (* (i-large-integer)
                                                 (+ tm1 tm2))))
                                   (standard-part
                                      (* (/ (i-large-integer))
                                         (floor1 (* (i-large-integer)
                                                    (+ tm1 tm2)))))))
                       )))))
```

```
;;------------------------------------------
;; The following is the theorem which states
;;  that phi is time invariant
;;------------------------------------------
(defthm-std phi-plus-thm
  (implies
   (and
    (realp x)
    (realp tm1)
    (realp tm2)
    (<= 0 tm1)
    (<= 0 tm2))
   (equal (phi (phi x tm1) tm2)
          (phi x (+ tm1 tm2))))
:hints (("Goal" :in-theory (disable i-large run-plus-thm)
         :use ((:instance phi-plus-hint1)
               (:instance  tm1+tm2-limited-thm)
               (:instance run-plus-thm (n (floor1 (* tm1
                                                     (i-large-integer))))
                                       (m (floor1 (* tm2
                                                     (i-large-integer))))
                                       (eps (/ (i-large-integer))))
               (:instance run-diff-tm-standard-part-thm
                    (eps (/ (i-large-integer)))
                    (m (+ (floor1 (* (i-large-integer) tm1))
                          (floor1 (* (i-large-integer) tm2))))
                    (n (floor1 (* (i-large-integer) (+ tm1 tm2)))))
               (:instance phi-x-continuous-thm
                    (x (RUN X
                           (FLOOR1 (* (I-LARGE-INTEGER) TM1))
                           (/ (I-LARGE-INTEGER))))
                    (tm tm2))))))


;;For some reason, enabling the following rule as a
;; linear lemma may cause ACL2 to stall during simplification
(defthm floor1-1-thm
  (implies
   (and (realp x)
        (<= 1 x))
   (< 0 (floor1 x)))
:rule-classes nil)

;;For some reason, enabling the following rewrite rule
```

```
;; may cause ACL2 to stall during simplification
(defthm floor1-0-thm
  (implies
   (and (realp x)
        (<= 0 x)
        (< x 1))
   (equal (floor1 x) 0))
:rule-classes nil)

(defun tm-m (tm eps)
  (cond
   ((not (and
          (realp eps)
          (< 0 eps)
          (realp tm)
          (<= eps tm))) 0)
   (t (floor1 (/ tm eps)))))

(defun phi-run (x tm eps)
  (declare (xargs :measure (tm-m tm eps)
                  :hints (("Subgoal 1.2"
                           :use ((:instance floor1-1-thm
                                      (x (* (/ eps) tm))))))))
  (cond
   ((not (and (realp eps)
              (< 0 eps)
              (<= eps tm)
              (realp tm))) (phi x tm))
   (t (phi-run (phi x eps) (- tm eps) eps))))

(defthm phi-run-eq-phi-thm
  (implies
   (and
    (realp x)
    (realp tm)
    (<= 0 tm)
    (< 0 eps)
    (realp eps))
   (equal (phi-run x tm eps)
          (phi x tm)))
:rule-classes nil)

(defthm run-equal-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (equal (+ x (* eps n (f x)) (- (f-sum x n eps) (* n eps (f x))))
          (run x n eps)))
:hints (("Goal" :use ((:instance run-f-sum-thm)))))

(defthm-std phi-equal-thm
  (implies
   (and
    (realp x)
    (realp tm)
    (<= 0 tm))
   (equal (+ x (* tm (f x)) (resid-tm x tm))
          (phi x tm)))
:hints (("Goal" :use ((:instance run-f-sum-thm
                           (eps (/ (i-large-integer)))
                           (n (floor1 (* tm (i-large-integer))))
                      )))))

(defthm step-resid-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (equal (+ (resid x n eps)
             (- (* eps (f (run x n eps)))
                (* eps (f x))))
          (resid x (+ n 1) eps))))

(defthm resid-type
  (implies
   (and
    (realp x)
    (realp eps)
    (integerp n))
   (realp (resid x n eps)))
:rule-classes :type-prescription)

(defthm-std resid-tm-type
  (implies
   (and
    (realp x)
    (realp tm))
   (realp (resid-tm x tm)))
:rule-classes :type-prescription)

(defthm step-resid-bound-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (< 0 n))
   (<= (abs (resid x n eps))
       (+ (abs (resid x (- n 1) eps))
          (* eps eps (L) (- n 1)
             (eexp (* eps (- n 1) (L)))
             (abs (f x))))))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs step-resid-thm resid)
                :use ((:instance f-sum-exp-bound-thm (n (- n 1)))
                      (:instance abs-triangular-inequality-thm
                           (x (RESID X (+ -1 N) EPS))
                           (y (- (* eps (f (run x (- n 1) eps)))
                                 (* eps (f x)))))
                      (:instance abs-pos-*-left-thm
                           (x eps)
                           (y (+ (* -1 (F X))
                                 (* (F (RUN X (+ -1 N) EPS))))))
                      (:instance f-lim-thm (x1 (run x (- n 1) eps))
                                           (x2 x))
                      (:instance pos-factor-<=-thm
                           (x (ABS (+ (* -1 (F X))
                                      (* (F (RUN X (+ -1 N) EPS))))))
                           (y (* (L)
                                 (ABS (+ (* -1 X)
                                 (RUN X (+ -1 N) EPS)))))
                           (a eps))
                      (:instance run-f-sum-thm (n (- n 1)))
                      (:instance step-resid-thm (n (- n 1)))
                      (:instance pos-factor-<=-thm
                           (x (abs (f-sum x (- n 1) eps)))
                           (y (* (- n 1)
                                eps
                                (eexp (* eps (- n 1) (L)))
                                (abs (f x))))
                          (a (* eps (L))))))))

(defthm resid-bound-thm
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (<= (abs (resid x n eps))
       (* n n eps eps (L) (eexp (* n eps (L))) (abs (f x)))))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs)
                :do-not '(generalize)
                :do-not-induct t
                :induct (f-sum x n eps))
        ("Subgoal *1/2" :in-theory (disable abs resid
                                            <-*-LEFT-CANCEL)
                        :use ((:instance step-resid-bound-thm)
                              (:instance pos-factor-<=-thm
                                   (x 0)
                                   (y (- n 1))
                                   (a (* (L) eps)))
                              (:instance pos-factor-<=-thm
                                   (x 1)
                                   (y (eexp (* eps (L))))
                                   (a (* (L) EPS EPS N N
                                         (ABS (F X))
                                         (EEXP (+ (* -1 (L) EPS)
                                                  (* (L) EPS N)))))
                              )))))

(defthm tm-fun-rw-5-thm
  (implies
   (and
    (realp a)
    (realp b)
    (realp c)
    (realp tm)
    (standard-numberp tm))
   (equal (* (/ (I-LARGE-INTEGER)) a b
             (FLOOR1 (* (I-LARGE-INTEGER) TM)) c)
          (* (tm-fun tm) a b c)))
:hints (("Goal" :in-theory (e/d (tm-fun) (tm-fun-rw-1-thm
                                          tm-fun-rw-2-thm
                                          tm-fun-rw-3-thm
                                          tm-fun-rw-4-thm)))))

(defthm tm-fun-rw-6-thm
  (implies
   (and
    (realp a)
    (realp b)
    (realp c)
    (realp d)
    (realp tm)
    (standard-numberp tm))
   (equal (* (/ (I-LARGE-INTEGER)) a b c
             (FLOOR1 (* (I-LARGE-INTEGER) TM)) d)
          (* (tm-fun tm) a b c d)))
:hints (("Goal" :in-theory (e/d (tm-fun) (tm-fun-rw-1-thm
                                          tm-fun-rw-2-thm
                                          tm-fun-rw-3-thm
                                          tm-fun-rw-4-thm
                                          tm-fun-rw-5-thm)))))

(defthm-std resid-tm-bound-thm
  (implies
   (and
    (realp x)
    (realp tm)
    (<= 0 tm))
   (<= (abs (resid-tm x tm))
       (* tm tm (L) (eexp (* tm (L))) (abs (f x)))))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs <-CANCEL-DIVISORS)
                :use ((:instance resid-bound-thm
                           (eps (/ (i-large-integer)))
                           (n (floor1 (* tm (i-large-integer)))))))))

(defthm tm-floor-0-thm
  (implies
   (and
    (realp tm)
    (realp eps)
    (<= 0 tm)
    (< 0 eps)
    (< tm eps))
   (equal (FLOOR1 (* (/ EPS) TM))  0))
:hints (("Goal" :use ((:instance floor1-0-thm (x (/ tm eps)))))))

(defthm-std phi-0-thm
  (implies
   (realp x)
   (equal (phi x 0)
          x)))

(defun tm-induct (tm eps)
  (declare (xargs :measure (tm-m tm eps)
                  :hints (("Subgoal 1.2"
                    :use ((:instance floor1-1-thm
                               (x (* (/ eps) tm))))))))
  (cond
   ((not (and
          (realp eps)
          (< 0 eps)
          (realp tm)
          (<= eps tm))) tm)
   (t (tm-induct (- tm eps)
                 eps))))

(defthm phi-eps-step-thm
  (implies
   (and
    (realp x1)
    (realp x2)
    (realp eps)
    (< 0 eps))
   (<= (abs (- (phi x1 eps) (step1 x2 eps)))
       (+ (* (abs (- x1 x2)) (+ 1 (* eps (L))))
          (* eps eps (L) (eexp (* eps (L))) (abs (f x1))))))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs)
                :do-not '(generalize)
                :do-not-induct t
                :use ((:instance phi-equal-thm (x x1) (tm eps))
                      (:instance abs-triangular-inequality-thm
                           (x (+ X1 (* -1 X2)))
                           (y (+(RESID-TM X1 EPS)
                                (* EPS (F X1))
                                (* -1 EPS (F X2)))))
```

```
                     (:instance abs-triangular-inequality-thm
                          (x (RESID-TM X1 EPS))
                          (y (+ (* EPS (F X1))
                                (* -1 EPS (F X2)))))
                     (:instance resid-tm-bound-thm
                          (tm eps)
                          (x x1))
                     (:instance abs-pos-*-left-thm
                          (x eps)
                          (y (- (F X1)
                                (F X2))))
                     (:instance f-lim-thm)
                     (:instance pos-factor-<=-thm
                          (x (abs (- (f x1) (f x2))))
                          (y (* (L) (abs (- x1 x2))))
                          (a eps))))))

(defthm phi-phi-run-thm
  (implies
   (and
    (realp eps)
    (realp x)
    (realp tm)
    (<= 0 tm)
    (<= eps tm)
    (< 0 eps))
   (equal (phi (phi-run x
                       (+ (* -1 EPS)
                          (* EPS
                             (FLOOR1 (* (/ EPS) TM))))
                       eps)
                eps)
          (phi-run x (* eps (floor1 (/ tm eps))) eps)))
:hints (("Goal" :induct (phi-run x tm eps)
                :do-not '(generalize))
        ("Subgoal *1/4" :use ((:instance floor1-1-thm
                                  (x (/ tm eps)))
                              (:instance pos-factor-<=-thm
                                  (x 1)
                                  (y (floor1 (/ tm eps)))
                                  (a eps))))
        ("Subgoal *1/2" :use ((:instance floor1-1-thm (x (/ tm eps)))
                              (:instance pos-factor-<=-thm
                                  (x 1)
                                  (y (floor1 (/ tm eps)))
                                (a eps))))
        ("Subgoal *1/1" :use ((:instance tm-floor-0-thm
                                (tm (- tm eps)))
                             (:instance distributivity
                                (y (FLOOR1 (* (/ EPS) TM)))
                                (z -1)
                                (x eps))))))

(defthm f-standard-part-thm
  (implies
   (and
    (realp x)
    (i-limited x))
   (equal (standard-part (f x))
          (f (standard-part x))))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs STANDARDP-STANDARD-PART)
                :use ((:instance f-lim-thm
                          (x1 (standard-part x))
                          (x2 x))
                      (:instance standardp-standard-part)))))

(defthm-std f-phi-bound-thm
  (implies
   (and
    (realp x)
    (realp tm)
    (<= 0 tm))
   (<= (abs (f (phi x tm)))
       (* (eexp (* tm (L))) (abs (f x)))))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs)
                :use ((:instance f-run-bound-thm
                          (eps (/ (i-large-integer)))
                          (n (floor1 (* tm (i-large-integer))))
                      )))))

(defthm phi-eps-arith-hint
  (implies
   (and
    (realp tm)
    (< 0 eps)
    (realp eps))
   (equal
    (* (EEXP (* (L) EPS))
       (FLOOR1 (* (/ EPS) TM))
       (EEXP (+ (* -1 (L) EPS)
                (* (L) EPS (FLOOR1 (* (/ EPS) TM))))))
    (* (FLOOR1 (* (/ EPS) TM))
       (EEXP (* (L) EPS (FLOOR1 (* (/ EPS) TM)))))))
:hints (("Goal" :in-theory (disable *-commut-3way)
                :use ((:instance *-commut-3way
                           (x (EEXP (* (L) EPS)))
                           (y (FLOOR1 (* (/ EPS) TM)))
                           (z (EEXP (+ (* -1 (L) EPS)
                                       (* (L) EPS
                                          (FLOOR1 (* (/ EPS) TM))))))
                      )))))

(defthm phi-eps-thm
  (implies
   (and
    (realp x)
    (realp tm)
    (realp eps)
    (<= 0 tm)
    (< 0 eps))
   (<= (abs (- (phi-run x
                        (* eps (floor1 (/ tm eps)))
                        eps)
               (run x
                    (floor1 (/ tm eps))
                     eps)))
       (* eps eps
          (floor1 (/ tm eps))
          (L)
          (abs (f x))
          (eexp (* eps (floor1 (/ tm eps)) (L))))))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs)
                :induct (tm-induct tm eps)
                :do-not '(generalize))
        ("Subgoal *1/2" :in-theory (disable abs run step1)
                        :use ((:instance floor1-1-thm (x (/ tm eps)))
                              (:instance pos-factor-<=-thm
                                   (x 1)
                                   (y (floor1 (/ tm eps)))
                                   (a eps))))
        ("Subgoal *1/2.1" :in-theory (disable abs run step1
                                              <-*-left-cancel
                                              <-cancel-divisors)
                :use ((:instance phi-run-eq-phi-thm
                           (tm (+ (* -1 EPS)
                                 (* EPS (FLOOR1 (* (/ EPS) TM))))))
                      (:instance phi-run-eq-phi-thm
                           (tm (* EPS (FLOOR1 (* (/ EPS) TM)))))
                      (:instance 1+x-<=eexp-thm (x (* eps (L))))
                      (:instance phi-eps-arith-hint)
                      (:instance pos-factor-<=-thm
                           (x (+ 1 (* eps (L))))
                           (y (eexp (* eps (L))))
                           (a (ABS (+ (* -1
                                         (RUN X
                                              (+ -1
                                                  (FLOOR1
                                                    (* (/ EPS) TM)))
                                              EPS))
                                      (PHI-RUN X
                                               (+ (* -1 EPS)
                                                  (* EPS
                                                     (FLOOR1
                                                       (* (/ EPS) TM))))
                                               EPS)))))
                      (:instance pos-factor-<=-thm
                           (x (ABS (+ (* -1
                                         (RUN X
                                              (+ -1 (FLOOR1
                                                      (* (/ EPS) TM)))
                                              EPS))
                                      (PHI-RUN X
                                               (+ (* -1 EPS)
                                                  (* EPS
                                                    (FLOOR1
                                                      (* (/ EPS) TM))))
                                               EPS))))
                           (y (+ (* -1 (L)
                                    EPS EPS (ABS (F X))
                                    (EEXP (+ (* -1 (L) EPS)
                                             (* (L) EPS
                                                (FLOOR1
                                                  (* (/ EPS) TM))))))
                                 (* (L)
                                    EPS EPS (ABS (F X))
                                    (FLOOR1 (* (/ EPS) TM))
                                    (EEXP (+ (* -1 (L) EPS)
                                            (* (L)
                                                EPS
                                                (FLOOR1
                                                  (* (/ EPS) TM)))))
                                 )))
                           (a (EEXP (* (L) EPS))))
                      (:instance pos-factor-<=-thm
                           (x (ABS (F (PHI X
                                           (+ (* -1 EPS)
                                              (* EPS
                                                  (FLOOR1
                                                    (* (/ EPS) TM))))
                                       ))))
                           (y (* (ABS (F X))
                                 (EEXP (+ (* -1 (L) EPS)
                                          (* (L) EPS
                                             (FLOOR1
                                               (* (/ EPS) TM))
                                          )))))
                           (a (* (L) EPS EPS (EEXP (* (L) EPS)))))
                      (:instance phi-eps-step-thm
                           (x1 (phi-run x
                                        (* eps
                                           (- (floor1 (/ tm eps)) 1))
                                        eps))
                                 (x2 (run x (- (floor1
                                                  (/ tm eps)) 1) eps)))
                      (:instance f-phi-bound-thm
                           (tm (+ (* -1 EPS)
                                  (* EPS (FLOOR1 (* (/ EPS) TM)))))
                      )))))

   (defthm phi-any-small-eps-thm
     (implies
      (and
       (realp x)
       (realp tm)
       (standard-numberp x)
       (standard-numberp tm)
       (realp eps)
       (<= 0 tm)
       (< 0 eps)
       (i-small eps))
      (equal (standard-part (phi x tm))
             (standard-part (run x (floor1 (/ tm eps)) eps))))
```

```
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs phi)
                :use ((:instance small-are-limited (x eps))
                      (:instance phi-eps-thm)
                      (:instance phi-run-eq-phi-thm
                           (tm (* EPS (FLOOR1 (* (/ EPS) TM)))))))))

;;-----------------------------------------------
;; The following is the theorem which states
;;  that run, hence phi, is independent of the
;;  choice of eps.
;;-----------------------------------------------
(defthm run-any-small-eps-thm
  (implies
   (and
    (realp x)
    (realp tm)
    (standard-numberp x)
    (standard-numberp tm)
    (realp eps)
    (<= 0 tm)
    (< 0 eps)
    (i-small eps))
   (equal (standard-part (run x
                              (floor1 (* tm (i-large-integer)))
                              (/ (i-large-integer))))
          (standard-part (run x
                              (floor1 (/ tm eps))
                              eps))))
:hints (("Goal" :use ((:instance  phi-any-small-eps-thm)))))
```

A.9 The Solution ϕ is Unique

This section presents a printout of file `phi-unique.lisp`. This file consists of definitions which show that the standard function ϕ is a solution of the differential equation and is unique. We show uniqueness by demonstrating that, for any function ϕ_2 which satisfies the differential equation, ϕ_2 is equal to ϕ.

```
(in-package "ACL2")

(include-book "arith-nsa4")
(include-book "abs")
(include-book "eexp")
(include-book "phi-exists")
(include-book "phi-properties")
(include-book "computed-hints")
(include-book "tm-floor")

(in-theory (disable i-large))
(in-theory (disable standard-part-<=))

(defstub phi2 (tm) t)

(defaxiom phi2-type
  (implies
   (and
    (realp tm))
   (realp (phi2 tm)))
:rule-classes :type-prescription)

(defaxiom phi2-standard-thm
  (implies
   (and
    (realp tm)
    (standard-numberp tm))
   (standard-numberp (phi2 tm)))
:rule-classes ((:type-prescription) (:rewrite)))

(defun resid2-tm (tm eps)
  (+ (phi2 (+ tm eps))
     (- (phi2 tm))
     (- (* eps (f (phi2 tm))))))

(defthm resid2-tm-type
  (implies
   (and
    (realp tm)
    (realp eps))
   (realp (resid2-tm tm eps)))
:rule-classes :type-prescription)

(defaxiom phi2-deriv
  (implies
   (and
    (realp tm)
    (i-limited tm)
    (realp eps)
    (not (equal eps 0))
    (i-small eps))
   (equal (standard-part (/ (- (phi2 (+ tm eps)) (phi2 tm)) eps))
          (standard-part (f (phi2 tm))))))

(defthm phi2-equal-thm
  (implies
   (and
    (realp tm)
    (realp eps))
   (equal (+ (phi2 tm)
             (* eps (f (phi2 tm)))
             (resid2-tm tm eps))
          (phi2 (+ tm eps)))))

(defthm resid2/eps-small-thm
  (implies
   (and
    (realp tm)
    (i-limited tm)
    (realp eps)
    (not (equal eps 0))
    (i-small eps))
   (equal (standard-part (/ (resid2-tm tm eps) eps))
          0))
:hints (("Goal" :use ((:instance phi2-deriv)))))

(defthm /eps-gt-1-thm
  (implies
   (and
    (realp eps)
    (not (equal eps 0))
    (equal (standard-part eps) 0))
   (< 1 (abs (/ eps))))
:hints (("Goal" :use ((:instance standard-part-<= (y 1) (x (/ (abs eps))))
                      (:instance i-large (x (abs (/ eps))))))))

(defthm resid2-small-thm
  (implies
   (and
    (realp tm)
    (i-limited tm)
    (realp eps)
    (not (equal eps 0))
    (i-small eps))
   (equal (standard-part (resid2-tm tm eps))
          0))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs resid2-tm abs-*-thm)
                :use ((:instance resid2/eps-small-thm)
                      (:instance pos-factor-<-thm
                           (x 1)
                           (y (abs (/ eps)))
                           (a (abs (resid2-tm tm eps))))
                      (:instance abs-*-thm
                           (x (/ eps))
                           (y (resid2-tm tm eps)))))))

(defthm phi2-tm-continuous-thm
  (implies
   (and
    (realp tm)
    (i-limited tm))
   (equal (standard-part (phi2 tm))
          (phi2 (standard-part tm))))
:hints (("Goal" :in-theory (disable resid2-tm standardp-standard-part)
                :cases ((equal tm (standard-part tm))
                        (not (equal tm (standard-part tm)))))
        ("Subgoal 1" :use ((:instance phi2-equal-thm
                                 (eps (- tm (standard-part tm)))
                                 (tm (standard-part tm)))
                           (:instance resid2-small-thm
                                 (eps (- tm (standard-part tm)))
                                 (tm (standard-part tm)))
                           (:instance standardp-standard-part
                                (x tm))
                           (:instance standards-are-limited
                                (x (phi2 (standard-part tm))))))
        ("Subgoal 2" :use ((:instance standardp-standard-part
                                (x tm))))))

(defthm standardp-standard-part-limited
  (implies
   (and
    (acl2-numberp x)
    (standard-numberp (standard-part x)))
   (i-limited x))
:hints (("Goal" :use ((:instance standard+small->i-limited
                           (x (standard-part x))
                           (eps (- x (standard-part x))))))))

(defthm phi2-limited-thm
  (implies
   (and
    (realp tm)
    (i-limited tm))
   (i-limited (phi2 tm)))
:rule-classes ((:type-prescription) (:rewrite))
:hints (("Goal" :use ((:instance standardp-standard-part-limited
                           (x (phi2 tm)))))))

(defthm resid2-tm-limited
  (implies
   (and
    (realp eps)
    (realp tm)
    (i-limited tm)
    (i-small eps))
   (i-limited (resid2-tm tm eps)))
:rule-classes ((:type-prescription) (:rewrite)))

(defun max-abs-resid2-tm (n eps)
  (cond
   ((zp n) (abs (resid2-tm 0 eps)))
   (t (max (abs (resid2-tm (* n eps) eps))
           (max-abs-resid2-tm (- n 1) eps)))))

(defthm max-abs-resid2-is-bound
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (integerp m)
    (<= 0 m)
    (<= m n))
   (<= (abs (resid2-tm (* eps m) eps))
       (max-abs-resid2-tm n eps)))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs))))

(defun find-n (n eps)
  (cond
   ((zp n) 0)
   ((equal (abs (resid2-tm (* n eps) eps))
           (max-abs-resid2-tm n eps)) n)
   (t (find-n (- n 1) eps))))

(defthm find-n-is-max
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (equal (abs (resid2-tm (* eps (find-n n eps)) eps))
          (max-abs-resid2-tm n eps)))
:rule-classes nil)

(defthm find-n-range
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n))
   (and
    (<= 0 (find-n n eps))
    (<= (find-n n eps) n)))
:rule-classes :linear)

(defthm find-n-limited
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n)
    (i-limited (* eps n)))
   (i-limited (* eps (find-n n eps))))
:hints (("Goal" :in-theory (disable abs)
                :do-not-induct t
                :use ((:instance sandwich-limited-thm
                           (u 0)
                           (v (* eps n))
                           (x (* eps (find-n n eps))))
                      (:instance pos-factor-<=-thm
                           (x 0)
                           (y (find-n n eps))
                           (a eps))
                      (:instance pos-factor-<=-thm
                           (x (find-n n eps))
                           (y n)
                           (a eps))
                      (:instance find-n-range)))))

(defthm max-abs-resid2-tm-small
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n)
    (i-small eps)
    (i-limited (* eps n)))
   (equal (standard-part (* (/ eps) (max-abs-resid2-tm n eps)))
          0))
:hints (("Goal" :in-theory (disable abs
                                    resid2-tm
                                    abs-*-thm
                                    ABS-POS-*-LEFT-THM
                                    <-CANCEL-DIVISORS
                                    EQUAL-CANCEL-DIVISORS)
                :do-not-induct t
                :use ((:instance find-n-is-max)
                      (:instance abs-pos-*-left-thm
                           (x (/ eps))
                           (y (RESID2-TM
                                 (EPS-N-FUN EPS (FIND-N N EPS))
```

```
                                EPS)))
                      (:instance resid2/eps-small-thm
                           (tm (* eps (find-n n eps)))
                      )))))

(defthm max-abs-resid2-tm-limited
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n)
    (i-small eps)
    (i-limited (* eps n)))
   (i-limited (* (/ eps) (max-abs-resid2-tm n eps))))
:rule-classes ((:rewrite) (:type-prescription)))

(defthm max-abs-resid2-tm-type
  (implies
   (and
    (realp eps)
    (integerp n))
   (and
    (realp (max-abs-resid2-tm n eps))
    (<= 0 (max-abs-resid2-tm n eps))))
:rule-classes :type-prescription)

(defthm max-abs-resid2-tm-floor-small-hint
  (implies
   (and
    (realp eps)
    (< 0 eps)
    (integerp n)
    (<= 0 n)
    (i-limited tm)
    (i-small eps)
    (i-limited (* eps n)))
   (equal (standard-part (* tm (/ eps)
                            (max-abs-resid2-tm n eps)))
          0))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs *-commut-3way)
                :do-not-induct t
                :use ((:instance max-abs-resid2-tm-small)))))
```

```
(defthm max-abs-resid2-tm-floor-small
  (implies
   (and
    (realp tm)
    (<= 0 tm)
    (i-limited tm))
   (equal (standard-part (* (floor1 (* tm (i-large-integer)))
                            (max-abs-resid2-tm
                               (floor1 (* tm (i-large-integer)))
                               (/ (i-large-integer)))))
          0))
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs resid2-tm)
                :do-not-induct t
                :use ((:instance max-abs-resid2-tm-small
                           (eps (/ (i-large-integer)))
                           (n (floor1 (* tm (i-large-integer)))))
                      (:instance pos-factor-<-thm
                           (x (- (* tm (i-large-integer)) 1))
                           (y (floor1 (* tm (i-large-integer))))
                           (a (max-abs-resid2-tm
                                 (floor1 (* tm (i-large-integer)))
                                 (/ (i-large-integer)))))
                      (:instance pos-factor-<=-thm
                           (x (floor1 (* tm (i-large-integer))))
                           (y (* tm (i-large-integer)))
                           (a (max-abs-resid2-tm
                                 (floor1 (* tm (i-large-integer)))
                                 (/ (i-large-integer)))))))
        ("Subgoal 1" :in-theory (disable abs resid2-tm)
                     :use ((:instance max-abs-resid2-tm-floor-small-hint
                                (n (floor1 (* tm (i-large-integer))))
                                (eps (/ (i-large-integer))))))))

(defun n-induct-scheme (n)
  (cond
   ((zp n) 0)
   (t (n-induct-scheme (- n 1)))))

(defthm phi-run-step-diff
  (implies
   (and
    (realp x)
    (realp eps)
    (< 0 eps)
    (integerp n)
    (< 0 n)
    (integerp m)
    (<= n m))
   (<= (abs (- (step1 x eps)
               (phi2 (* n eps))))
       (+ (* (+ 1 (* eps (L))) (abs (- x (phi2 (* (- n 1) eps)))))
          (max-abs-resid2-tm m eps))))
:hints (("Goal" :in-theory (disable abs resid2-tm)
                :do-not '(generalize)
                :do-not-induct t
                :use ((:instance f-lim-thm
                            (x1 x)
                            (x2 (phi2 (* (- n 1) eps))))
                      (:instance max-abs-resid2-is-bound
                            (m (- n 1)) (n m))
                      (:instance pos-factor-<=-thm
                            (x (abs (- (f x) (f (phi2 (* (- n 1) eps))))))
                            (y (* (L) (abs (- x (phi2 (* (- n 1) eps))))))
                            (a eps))
                      (:instance phi2-equal-thm
                            (tm (* (- n 1) eps)))
                      (:instance abs-pos-*-left-thm
                            (x eps) (y  (+ (F X)
                            (* -1
                               (f (PHI2 (+ (* -1 EPS) (* EPS N))))))))
                      (:instance abs-triangular-inequality-3way-thm
                            (x (- x (phi2 (* (- n 1) eps))))
                            (y (- (* eps (f x))
                                  (* eps (f (phi2 (* (- n 1) eps))))))
                            (z (- (resid2-tm (* (- n 1) eps) eps))))))))

(defthm eexp-unite-exponents-thm
  (implies
   (and
    (realp x)
    (realp y)
    (realp z))
   (equal (* (eexp x) y (eexp z))
          (* y (eexp (+ x z))))))

(defthm phi2-run-eq-thm
  (implies
   (and
    (integerp m)
    (integerp n)
    (<= 0 n)
    (<= n m)
    (< 0 eps)
    (realp eps))
   (<= (abs (- (run (phi2 0) n eps)
               (phi2 (* eps n))))
       (* (eexp (* eps n (L)))
          n
          (max-abs-resid2-tm m eps))))
:rule-classes nil
:hints (("Goal" :do-not '(generalize)
                :induct (n-induct-scheme n)
                :do-not-induct t
                :in-theory (disable abs step1 resid2-tm MAX-ABS-RESID2-TM))
        ("Subgoal *1/2" :use ((:instance phi-run-step-diff
                                   (x (run (phi2 0) (- n 1) eps)))
                              (:instance pos-factor-<=-thm
                                   (x (ABS (+ (RUN (PHI2 0) (+ -1 N) EPS)
                                              (* -1 (PHI2 (+ (* -1 EPS)
                                                             (* EPS N)))))))
                                   (y (+ (* -1 (MAX-ABS-RESID2-TM M EPS)
                                            (EEXP (+ (* -1 (L) EPS)
                                                     (* (L) EPS N))))
                                         (* N (MAX-ABS-RESID2-TM M EPS)
                                            (EEXP (+ (* -1 (L) EPS)
                                                     (* (L) EPS N))))))
                                   (a (eexp (* (L) eps))))
                              (:instance pos-factor-<=-thm
                                   (x 1)
                                   (y (EEXP (* (L) EPS N)))
                                   (a (MAX-ABS-RESID2-TM M EPS)))
                              (:instance 1+x-<=eexp-thm
                                   (x (* eps (L))))
                              (:instance pos-factor-<=-thm
                                   (x (+ 1 (* eps (L))))
                                   (y (eexp (* eps (L))))
                                   (a (ABS (+ (RUN (PHI2 0) (+ -1 N) EPS)
                                              (* -1 (PHI2 (+ (* -1 EPS)
                                                (* EPS N)))))))))))
:otf-flg t)

(defthm phi2-st-run-eq-hint
  (implies
   (and
    (realp tm)
    (i-limited y)
    (<= 0 tm)
    (i-limited tm))
   (equal (standard-part (* y
                            (floor1 (* tm (i-large-integer)))
                            (max-abs-resid2-tm
                                   (floor1 (* tm (i-large-integer)))
                                   (/ (i-large-integer)))))
          0))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs resid2-tm)
                :do-not-induct t
                :use ((:instance max-abs-resid2-tm-floor-small)))))

(defthm phi2-st-run-eq-thm
  (implies
   (and
    (realp tm)
    (standard-numberp tm)
    (<= 0 tm))
   (equal (standard-part (phi2 tm))
          (standard-part (phi (phi2 0) tm))))
:rule-classes nil
:hints ((standard-part-hint stable-under-simplificationp clause)
        ("Goal" :in-theory (disable abs)
                :use ((:instance phi2-run-eq-thm
                           (eps (/ (i-large-integer)))
                           (n (floor1 (* tm (i-large-integer))))
                           (m (floor1 (* tm (i-large-integer)))))
                      (:instance phi2-st-run-eq-hint
                           (y (EEXP (* (L) (TM-FUN TM)))))))))
```

```
;;---------------------------------------------
;; The following is the theorem which states
;;  that, for any phi2 satisfying the
;;  differential equation, phi2 is equal to
;;  phi evaluated at initial condition phi2(0).
;;---------------------------------------------
(defthm-std phi2-st-run-eq-std-thm
  (implies
   (and
    (realp tm)
    (<= 0 tm))
   (equal (phi2 tm)
          (phi (phi2 0) tm)))
:rule-classes nil
:hints (("Goal" :in-theory (disable abs)
                :use ((:instance phi2-st-run-eq-thm)))))
```

Appendix B

Mechanical Proof of Example

The following sections provide the printouts of the definition files submitted to the mechanical theorem prover ACL2r in performing the mechanical proof of the example presented in Chapter 7.

The printouts are of the following files:

1. `o-real-p.lisp`: Definitions regarding the measure structure described in Chapter 6.

2. `example.lisp`: The definitions of the system of the example presented in Chapter 7, along with definitions of theorems for the safety and progress properties of the example.

The file `example.lisp` depends on the files `arith-nsa4`, `abs`, `nsa`, and `computed-hints`, whose printouts are already shown in Appendix A and will not be shown in this appendix.

B.1 Measure Structure Definitions

This section presents a printout of file `o-real-p.lisp`. This file consists of function and theorem definitions of the measure structures described in Section 6.15.

```
(in-package "ACL2")

; definition of the measure structure less than operator
(defun o<-1 (x y)
       (cond ((o-finp x)
              (or (not (o-finp y)) (<= (+ X 1) Y)))
             ((o-finp y) nil)
             ((equal (o-first-expt x)
                     (o-first-expt y))
              (if (equal (o-first-coeff x)
                         (o-first-coeff y))
                  (o<-1 (o-rst x) (o-rst y))
                  (<= (+ (o-first-coeff x) 1)
                      (o-first-coeff y))))
             (t (o<-1 (o-first-expt x)
                      (o-first-expt y)))))

;The following defines a predicate which
; is true if x is a measure structure
(defun o-real-p (x)
       (cond ((o-finp x) (and
                          (realp X)
                          (<= 0 x)))
             ((consp (car x))
              (and (o-real-p (o-first-expt x))
                   (if (acl2-numberp (o-first-expt x))
                       (<= 1 (o-first-expt x))
                       t)
                   (realp (o-first-coeff x))
                   (<= 1 (o-first-coeff x))
                   (o-real-p (o-rst x))
                   (o<-1 (o-first-expt (o-rst x))
                         (o-first-expt x))))
             (t nil)))

;Recursively traverse
```

```
; a measure structure applying floor
; to the real values in the structure
; The intent is to convert a measure structure to
; an ordinal.
(defun o-floor1 (x)
       (cond ((o-finp x) (floor1 x))
             ((consp (car x))
              (make-ord (o-floor1 (o-first-expt x))
                        (floor1 (o-first-coeff x))
                        (o-floor1 (o-rst x))))
             (t nil)))

(defthm o<-1-floor1-neq-thm
(implies
 (and
  (o-real-p x)
  (o-real-p y)
  (o<-1 x y))
 (not (equal (o-floor1 x) (o-floor1 y)))))

;The following theorem states that
; showing a measure structure x is less than y,
; using the o<-1 operator
; implies that the ordinals attained by
; applying o-floor1 to x and y, respectively,
; are less than each other.
; Hence, in our proof obligation, we need only
; show that (o<-1 x y), this theorem may then
; be applied to show the respective ordinals
; attained by applying o-floor1 are less
; than each other.
(defthm o<-1-floor1-o<-thm
(implies
 (and
  (o-real-p x)
  (o-real-p y)
  (o<-1 x y))
 (o< (o-floor1 x) (o-floor1 y))))

(defthm floor1-posp
(implies
 (and
  (realp x)
  (<= 1 x))
 (posp (floor1 x))))
```

```
(defthm o-floor1-non-zero
(implies
 (and
  (o-real-p x)
  (o-real-p y)
  (o<-1 y x))
 (not (equal 0 (o-floor1 x)))))

(defthm o-real-p-caadr
(implies
 (and
  (o-real-p x)
  (consp x)
  (consp (cdr x)))
 (o-real-p (caadr x))))

(defthm o-real-p-caadr-2
(implies
 (and
  (o-real-p x)
  (consp x))
 (equal (caar (o-floor1 x)) (o-floor1 (caar x))))
:hints (("Goal" :do-not-induct t)))

(defthm o-floor1-consp
(implies
 (and
  (consp x)
  (o-real-p x))
 (consp (o-floor1 x))))

(defthm consp-not-zero
(implies
 (consp x)
 (not (equal 0 x))))
```

```
;The following states that if x is a measure
; structure, then (o-floor1 x) is an ordinal.
(defthm o-floor1-thm
(implies
 (o-real-p x)
 (o-p (o-floor1 x)))
:hints (("Goal" :do-not '(generalize)
                :induct (o-real-p x)
                :do-not-induct t)))
```

B.2 Example Problem

This section presents a printout of file `example.lisp`. This file consists of definitions of the example system presented in Chapter 7, as well as definitions of measure functions and theorems for fulfilling proof obligations for showing safety and progress properties for this example, based on the proof methods presented in Chapter 6. The books `arith-nsa4`, `abs`, `nsa`, and `computed-hints` are identical to those shown in Appendix A and are not shown in this appendix.

```
(in-package "ACL2")

(include-book "arith-nsa4")
(include-book "abs")
(include-book "computed-hints")
(include-book "o-real-p")
(include-book "nsa")

;; Enable abs, <-cancel-divisors and divisor cancellation as needed.

(in-theory (disable equal-cancel-divisors <-cancel-divisors))
(set-default-hints '((staged-hints stable-under-simplificationp
                                   nil ;;restart on new id
                                   '((:in-theory (enable abs
                                   equal-cancel-divisors <-cancel-divisors)))
                                   nil nil 0)))

;;Macro defining non-negative real
(defmacro nneg-realp (r)
   `(and (realp ,r)
         (<= 0 ,r)))

;;Macro defining the constraint on the variable eps.
;;   In this case, we require that 0 < eps <= 1/100.
(defmacro small-realp (eps)
   `(and (realp ,eps)
         (<= , eps 1/100)
         (< 0 ,eps)))
```

```
;;Define accessor function for accessing particular variables
;; from the state X
(defun getPosReq (X)
   (nth 0 X))

(defun getPreset (X)
   (nth 1 X))

(defun getPos (X)
   (nth 2 X))

(defun getPosAo (X)
   (nth 3 X))

(defun getTmr (X)
   (nth 4 X))

;;Define a function which creates a system state, consisting
;; of the variables of the system.
(defun make-state (posReq preset pos posAo tmr)
   (list posReq preset pos posAo tmr))

;;Define theorems relating the accessor function and the make
;; state functions.
(defthm state-thm
   (and
      (equal (getPosReq (make-state posReq preset pos posAo tmr)) posReq)
      (equal (getPreset (make-state posReq preset pos posAo tmr)) preset)
      (equal (getPos (make-state posReq preset pos posAo tmr)) pos)
      (equal (getPosAo (make-state posReq preset pos posAo tmr)) posAo)
      (equal (getTmr (make-state posReq preset pos posAo tmr)) tmr)))

;; Disable the accessor functions and make-state function and rely upon the
;;  rewrite rules associated with above theorem.
(in-theory (disable getPosReq getPreset getPos getPosAo getTmr
                    make-state))

;; StateP is a predicate which recognizes whether some variable is a
;;  state variable.
(defun statep (x)
   (equal x
          (make-state (getPosReq x)
                      (getPreset x)
                      (getPos x)
                      (getPosAo x)
                      (getTmr x))))

;; The system assignment function, Y, includes the definition of the
;;  computer program,
;;  the floor function representing the analog to digital conversion,
;;  and the reset of the timer variable.
(defun Y (X)
  (make-state

   (getPosReq x)

   (getPreset x)

   (getPos x)

   ;;posAo
   (cond
    ((> (- (floor1 (getPos X)) (getposReq X)) 2)
     (- (getposAo X) 5))
    ((< (- (floor1 (getPos X)) (getposReq X)) -3)
     (+ (getposAo X) 5))
    (t (getposAo X)))

   ;;tmr
   0))


;; The step definition of the physical system, including timer
(defun sigma (X eps)
  (make-state

   (getPosReq x)

   (getPreset x)

   ;;pos
   (cond
      ((> (getPos X) (getPosAo X))
          (- (getpos X) eps))
      ((< (getPos X) (getPosAo X))
          (+ (getpos X) eps))
      (t (getPos X)))

   (getposAo X)
   ;;tmr
   (+ (getTmr X) eps)))

(defun B-Y (X)
  (>= (getTmr X) (getPreset X)))

;; The system step function, as define by the single step function
;;  sigma, the assignment function Y, and assignment predicate B-Y.
(defun sys-step (X eps)
  (cond
   ((B-Y X) (Y X))
   (t       (sigma X eps))))

;; The positive clamp function "clamps" the given r to a non-negative value.
;; If the value is negative, it returns zero. Otherwise, it returns
;; the given value.
;; It should be noted that the function is continuous in r.
(defun pos-clamp (r)
  (if (<= 0 r)
      r
      0))

;;A component function of the overall measure m.
;;Intuitively, this measure function measures that the difference
;; between pos and posAo decreases over time.
;; This measure is 'active' when the difference between pos and
;; posAo is large.
(defun m1 (X eps)
  (cond
   ((<= (abs (- (getPosAo X) (getPos X)))
        (+ eps (pos-clamp (- (getPreset X) (getTmr X))))) 0)
   (t (+ 1 (/ (- (abs (- (getPosAo X) (getPos X)))
                 (+ (- (getPreset X) (getTmr X)) eps))
              eps)))))

;;A component function of the overall measure m.
;;Intuitively, this measure function measures that the difference
;; between posReq and posAo decreases over time.
(defun m2 (X eps)
(declare (ignore eps))
  (if (and
       (<= (- (getPosAo X) (getPosReq X)) 3)
       (>= (- (getPosAo X) (getPosReq X)) -3))
      0
      (abs (- (getPosAo X) (getPosReq X)))))
```

```
;;A component function of the overall measure m.
;;Intuitively, this measure function measures that the difference
;; between pos and posAo decreases over time.
;; This measure is 'active' when the difference between pos and
;; posAo is small.
(defun m3 (X eps)
  (if (<= (abs (- (getPos X) (getPosAo X))) eps)
      0
      (/ (abs (- (getPos X) (getPosAo X))) eps)))

;;A component function of the overall measure m.
;;Intuitively, this measure function measures that the
;;timer changes in each step. This is useful for showing a
;; decreasing measure while the other system variables are unchanging.
(defun m4 (X eps)
  (cond
   ((< (getPreset X) (getTmr X)) 0)
   (t (+ 1 (/ (- (getPreset X) (getTmr X)) eps)))))

;;The overall measure function
(defun m (X eps)
  (cond
   ((and
     (< (m1 x eps) 1)
     (< (m2 x eps) 1)) (make-ord 1 (+ 1 (m3 x eps))
                                (m4 x eps)))
   ((< (m1 x eps) 1) (make-ord 2 (+ 1 (m2 x eps))
                                (make-ord 1 (+ 1 (m3 x eps))
                                            (m4 x eps))))
   (t (make-ord 3 (+ 1 (m1 x eps)) (m4 x eps)))))

;;definition of the domain of the system variables and constants.
(defun valid-state (X eps)
   (and (realp (getPos X))
        (realp (getPreset X))
        (realp (getTmr X))
        (integerp (getPosAo X))
        (integerp (getPosReq X))
        (<= 51/10 (getPreset X))
        (<= 0 (getTmr x))
        (<= (getTmr x) (+ (getpreset x) eps))))

(set-inhibit-output-lst '(proof-tree prove))
```

```
;;By requirement A1, we must show that if the assignment
;; predicate is satisfied in the current step, it is
;; not satisfied in the next step.
(defthm step-A1-thm
 (implies
   (and
      (valid-state x eps)
      (B-Y x))
   (not (B-Y (Y X))))
:rule-classes nil)

;;Since the computer executes every delta time
;; period which is greater than preset, and since this
;; preset is a positive, standard number, then
;; to satisfy requirement A2, we must show that
;; the assignment function is limited if the
;; state variables are limited and satisfy B-Y.
(defthm step-A2-thm
 (implies
   (and
      (valid-state x eps)
      (B-Y x)
      (i-limited (getPosAo x))
      (i-limited (getTmr x))
      (i-limited (getPos x))
      (i-limited (getPreset x))
      (i-limited (getPosReq x)))
   (and
      (i-limited (getPosAo (Y x)))
      (i-limited (getTmr (Y x)))
      (i-limited (getPos (Y x)))
      (i-limited (getPreset (Y x)))
      (i-limited (getPosReq (Y x)))))
:rule-classes nil
:hints (("Goal" :in-theory (disable i-large))))

;;A theorem that states the formula (< (m1 x eps) 1) is equal
;; to the corresponding safety predicate
;; Hence, we will use the shorter formula
;; (< (m1 x eps) 1) in the remainder of this session.
(defthm m1-lt-1
  (implies
   (and
    (valid-state x eps)
    (small-realp eps))
   (iff (< (m1 x eps) 1)
        (<= (abs (- (getPosAo X) (getPos X)))
                 (+ eps (pos-clamp (- (getPreset X) (getTmr X)))))))
:rule-classes nil)

;;A theorem that states the formula (< (m2 x eps) 1) is equal
;; to the corresponding safety predicate
;; Hence, we will use the shorter formula (< (m2 x eps) 1)
;; in the remainder of this session.
(defthm m2-lt-1
  (implies
   (and
    (valid-state x eps)
    (small-realp eps))
   (iff (< (m2 x eps) 1)
        (and
         (<= (- (getPosAo X) (getPosReq X)) 3)
         (>= (- (getPosAo X) (getPosReq X)) -3))))
:rule-classes nil)

;;check that a valid state is an ordinal real
(defthm ordinal-real-thm
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (not (and
              (< (m1 x eps) 1)
              (< (m2 x eps) 1)
              (<= (abs (- (getpos x) (getPosReq x))) (+ 3 (* 2 eps))))))
   (o-real-p (m x eps))))

;;if we start in valid state, then next state also satisfies valid state
(defthm valid-state-preserve
(implies
   (and (valid-state x eps)
        (small-realp eps))
   (valid-state (sys-step x eps) eps)))

;;the following theorem shows that once m1 < 1, then it remains so
; similarly, once m2 < 1, it remains so. These results
;; are used to show that if m1 < 1 and m2 < 1, then
;; -3-eps <= (abs (- pos PosReq)) <= 3+eps is true
;; for all ensuing states.

(defthm m-1-preserve
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (< (m1 x eps) 1))
   (< (m1 (sys-step x eps) eps) 1))
:rule-classes :linear)

(defthm m-2-preserve
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (< (m1 x eps) 1)
        (< (m2 x eps) 1))
   (< (m2 (sys-step x eps) eps) 1))
:rule-classes :linear)

(defthm pos-posReq-preserve
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (< (m1 x eps) 1)
        (< (m2 x eps) 1)
        (<= (abs (- (getpos x) (getPosReq x))) (+ 3 (* 2 eps))))
   (<= (abs (- (getpos (sys-step x eps))
               (getposReq (sys-step x eps)))) (+ 3 (* 2 eps))))
:rule-classes :linear)

(defthm safety-property-preserve
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (< (m1 x eps) 1)
        (< (m2 x eps) 1)
        (<= (abs (- (getpos x) (getPosReq x))) (+ 3 (* 2 eps))))
   (and
    (valid-state x eps)
    (< (m1 (sys-step x eps) eps) 1)
    (< (m2 (sys-step x eps) eps) 1)
    (<= (abs (- (getpos (sys-step x eps))
                (getposReq (sys-step x eps)))) (+ 3 (* 2 eps)))))
:rule-classes nil
:hints (("Goal" :in-theory (disable sys-step valid-state m1 m2))
        ("Subgoal 2" :use ((:instance pos-posReq-preserve)))))

;;The measure is decreasing on the real ordinals, with
```

```
;; comparison o<-1
(defthm m-1-decreases
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (not (< (m1 x eps) 1)))
   (o<-1 (m (sys-step x eps) eps) (m x eps))))

(defthm m-2-decreases
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (< (m1 x eps) 1)
        (not (< (m2 x eps) 1)))
   (o<-1 (m (sys-step x eps) eps) (m x eps))))

(defthm m-decreases
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (not (and
               (< (m1 x eps) 1)
               (< (m2 x eps) 1)
               (<= (abs (- (getpos x) (getPosReq x))) (+ 3 (* 2 eps))))))
   (o<-1 (m (sys-step x eps) eps) (m x eps))))

(set-inhibit-output-lst '(proof-tree))

(in-theory (disable o<-1 o-floor1 o-real-p sys-step valid-state m m1 m2 m3))

;;fix m so that it always returns an ordinal
(defun m-fix (x eps)
  (cond
   ((not (and (valid-state x eps)
              (small-realp eps)
              (not (and
                    (< (m1 x eps) 1)
                    (< (m2 x eps) 1)
                    (<= (abs (- (getpos x) (getPosReq x)))
                        (+ 3 (* 2 eps)))))))
    0)
   (t (o-floor1 (m x eps)))))

;;m-fix is an ordinal
(defthm m-fix-o-p
(o-p (m-fix x eps)))

;;sys-step decreases, using measure m-fix
(defthm m-fix-decreases
(implies
   (and (valid-state x eps)
        (small-realp eps)
        (not (and
              (< (m1 x eps) 1)
              (< (m2 x eps) 1)
              (<= (abs (- (getpos x) (getPosReq x)))
                  (+ 3 (* 2 eps))))))
   (o< (m-fix (sys-step x eps) eps) (m-fix x eps))))
```

Bibliography

[1] ACL2 home page. http://www.cs.utexas.edu/users/moore/acl2/.

[2] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001. Springer Verlag.

[3] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[4] R. Alur, C. Courcoubetis, T. A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[5] R. Alur, T. Dang, and F. Ivancic. Reachability analysis of hybrid systems via predicate abstraction. In *HSCC*, pages 35–48, 2002.

[6] J. J. Beaman and H. M. Paynter. *Modeling of Physical Systems.* Class Notes. University of Texas at Austin, 1993.

[7] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[8] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC'99)*, 1999.

[9] W. E. Boyce and R. C. DiPrima. *Elementary Differential Equations and Boundary Value Problems*. John Wiley and Sons, Inc., New York, fourth edition, 1986.

[10] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, 2004.

[11] R.S. Boyer, M.W. Green, and J S. Moore. The use of a formal simulator to verify a simple real time control program. In *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 54–66. Springer Verlag Texts and Monographs in Computer Science, 1990.

[12] M. Branicky. Multiple lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Transactions on Automatic Control*, 43:475–482, 1998.

[13] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[14] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[15] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[16] S. Campos. Symbolic model checking in practice. In *XII Symposium on Integrated Circuits and System Design*, 2000.

[17] S. Campos, E. Clarke, and M. Minea. The verus tool: A quantitative approach to the formal verification of real-time systems. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 452–455. Springer Verlag, 1997.

[18] S. Campos, M. Teixeira, M. Minea, A. Kuehlmann, and E. Clarke. Model checking semi-continuous time models using BDDs. In Alessandro Cimatti and Orna Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 23. Elsevier, 2001.

[19] S. V. A. Campos and E. Clarke. The Verus language: representing time efficiently with BDDs. *Theoretical Computer Science*, 253(1):95–118, 2001.

[20] S. V. A. Campos, E. M. Clarke, W. R. Marrero, and M. Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Work-*

shop on Languages, Compilers, & Tools for Real-Time Systems, pages 70–78, 1995.

[21] A. Chutinan and B. H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control, Second International Workshop, HSCC'99*, volume 1569 of *LNCS*, pages 76–90, Berg en Dal, The Netherlands, March 1999. Springer.

[22] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[23] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifcations. In *ACM Transactions on Programming Languages and Systems*, volume 8(2), pages 244–263, 1986.

[24] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, October 1995. Springer.

[25] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In P. Wolper, editor, *7th International Conference On Computer Aided Verification*, volume 939, pages 409–422, Liege, Belgium, 1995. Springer Verlag.

[26] E. A. Emerson and R. Trefler. Generalized quantitative temporal reasoning: An automata-theoretic approach. In *TAPSOFT: 7th International Joint Conference on Theory and Practice of Software Development*, 1997.

[27] A. F. Filippov. *Differential Equations with Discontinuous Righthand Sides.* Kluwer Academic Publishers, Norwell, MA, 1988.

[28] M. Furi and M. Martelli. A multidimensional version of Rolle's Theorem. *The American Mathematical Monthly*, 102(3):243–249, 1995.

[29] R. A. Gamboa. *Mechanically Verified Real-Valued Algorithms in ACL2.* PhD thesis, University of Texas at Austin, 1999.

[30] N. Halbwachs, Y. E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *International Static Analysis Symposium, SAS'94*, Namur (Belgium), September 1994.

[31] P. R. Halmos. *Naive Set Theory.* Van Nostrand, 1960.

[32] J. Harrison and L. Thery. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780, pages 174–185, Vancouver, Canada, 1993. Springer Verlag, published 1994.

[33] J.M. Henle and E.M. Kleinberg. *Infinitesimal Calculus.* Dover Publications, Inc., Mineola, New York, 2003.

[34] T. A. Henzinger. The theory of hybrid automata. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[35] T. A. Henzinger, P. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[36] T. A. Henzinger and P. H. Ho. Algorithmic analysis of nonlinear hybrid systems. In P. Wolper, editor, *7th International Conference On Computer Aided Verification*, volume 939, pages 225–238, Liege, Belgium, 1995. Springer Verlag.

[37] T. A. Henzinger and P. W. Kopke. State equivalences for rectangular hybrid automata. In *International Conference on Concurrency Theory*, pages 530–545, 1996.

[38] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Proc. 27th Annual ACM Symp. on Theory of Computing (STOC)*, pages 373–382, 1995.

[39] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

[40] T. A. Henzinger and H. Wong-Toi. Linear phase-portrait approximations for nonlinear hybrid systems. In *Hybrid Systems*, pages 377–388, 1995.

[41] T. A. Henzinger and H. Wong-Toi. Using hytech to synthesize control parameters for a steam boiler. In J.-R. Abrial, E. Borger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 265–282. Springer Verlag, 1996.

[42] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Society Press.

[43] W. A. Hunt, R. B. Krug, and J Moore. Linear and nonlinear arithmetic in ACL2. In D. Geist, editor, *CHARME 2003*, volume 2860 of *LNCS*, pages 319–333. Springer Verlag, 2003.

[44] Y. Iwasaki, A. Farquhar, V. A. Saraswat, D. G. Bobrow, and V. Gupta. Modeling time in hybrid systems: How fast is "Instantaneous"? In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1773–1781, 1995.

[45] M. Johansson and A. Rantzer. Computation of piecewise quadratic lyapunov functions for hybrid systems. *Dept. Automatic Control, Lund Institute of Technology*, Tech Rep. TFRT-7549, June 1996.

[46] D. Krob and S. Bliudze. Towards a functional formalism for modeling complex industrial systems. In P. Bourgine, F. Kps, and M. Schoenauer, editors, *European Conference on Complex Systems (ECCS05)*, page (article 193) 20 pages, 2005.

[47] B. C. Kuo. *Automatic Control Systems*. Prentice Hall, Englewood Cliffs, New York, 1991.

[48] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[49] M. Lawford and H. Wu. Verification of real-time control software using PVS. In P. Ramadge and S. Verdu, editors, *2000 Conference on Information Sciences and Systems*, volume 2, pages TP1–13–TP1–17, Princeton, NJ, March 2000. Dept. of Electrical Engineering, Princeton University.

[50] A. Mader, E. Brinksma, H. Wupper, and N. Bauer. Design of a PLC program for VHS case study 1. *European Journal of Control*, 7(4):416–439, 2001.

[51] P. Manolios, M. Kaufmann, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[52] J. Misra. A logic for concurrent programming: Progress. *J. Computer and Software Eng.*, 3(2):273–300, 1995.

[53] J. Misra. A logic for concurrent programming: Safety. *J. Computer and Software Eng.*, 3(2):239–272, 1995.

[54] J. Mller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology, Technical University of Denmark, February 1999.

[55] M. Oliver Möller, H. Rueß, and M. Sorea. Predicate abstraction for dense real-time systems. *Electronic Notes in Theoretical Computer Science*, 65(6), 2002.

[56] J Moore and R.S. Boyer. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence 11*, pages 83–124. Oxford University Press, 1988.

[57] J S. Moore. Proving theorems about java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, volume 1710 of *LNCS*, pages 139–162, 1999.

[58] K. Nakamura and A. Fusaoka. On transfinite hybrid automata. In *Lecture Notes in Computer Science*, volume 3414, pages 495–510. Springer, 2005.

[59] E. Nelson. *Internal Set Theory.*
On Line Book: http://www.math.princeton.edu/˜nelson/books.html.

[60] L. T. Pillage, R. A. Rohrer, and C. Visweswariah. *Electronic circuit and system simulation methods.* McGraw-Hill, New York, 1995.

[61] A. Robinson. *Non-Standard Analysis.* Princeton University Press, 1996.

[62] H. Rust. A non-standard approach to operational semantics for timed systems. In *Abstract State Machines*, pages 423–424, 2003.

[63] H. Rust. Modeling discretely timed systems using different magnitudes of non-standard reals. In *Abstract State Machines*, pages 218–233, 2004.

[64] H. Rust. *Operational Semantics for Timed Systems: A Non-standard Approach to Uniform Modeling of Timed and Hybrid Systems*, volume 3456 of *Lecture Notes in Computer Science*. Springer, 2005.

[65] D. E. Sanderson. A versatile vector mean value theorem. *The American Mathematical Monthly*, 79(4):381–383, 1972.

[66] N. Shankar. Verification of real-time systems using PVS. In Costas Courcoubetis, editor, *Computer Aided Verification, CAV '93*, volume 697, pages 280–291, Elounda, Greece, June 1993. Springer Verlag.

[67] T. Stauner, O. Mueller, and M. Fuchs. Using HYTECH to verify an automotive control system. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *LNCS*, pages 139–153, Grenoble, France, 1997. Springer Verlag.

[68] D. E. Stewart. A high accuracy method for solving ODE's with discontinuous right-hand side. *Numer. Mathem.*, 58:299–328, 1990.

[69] D. E. Stewart. Rigid-body dynamics with friction and impact. *SIAM Review*, 42(1):3–39, 2000.

[70] A. Tiwari and G. Khanna. Nonlinear systems: Approximating reach sets. In R. Alur and G. Pappas, editors, *Hybrid Systems: Computation and Control HSCC*, LNCS. Springer, March 2004.

[71] VHS project homepage. http://www-verimag.imag.fr/VHS/main.html.

Vita

Shant Harutunian received the Bachelor of Science degree on May 1994 and the Master of Science degree on May 1996. Both degrees were attained in Electrical Engineering from the University of Texas at Austin. Shant Harutunian is currently employed by Harutunian Engineering, Inc. and resides in Austin, Texas.

Permanent address: P. O. Box W
Austin, Texas 78713

This dissertation was typeset with $\LaTeX$[†] by the author.

[†] $\LaTeX$ is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's $\TeX$ Program.