Combining SMT with Theorem Proving for AMS Verification

Analytically Verifying Global Convergence of a Digital PLL

by

Yan Peng

B.Eng., Zhejiang University, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2015

Abstract

Ubiquitous computer technology is driving increasing integration of digital computing with continuous, physical systems. Examples range from the wireless technology, cameras, motion sensors, and audio IO of mobile devices to sensors and actuators for robots to the analog circuits that regulate the clocks, power supplies, and temperature of CPU chips. While combining analog and digital brings ever increasing functionality, it also creates a design verification challenge: the modeling frameworks for analog and digital design are often quite different, and comprehensive simulations are often impractical. This motivates the use of formal verification: constructing mathematically rigorous proofs that the design has critical properties.

To support such verification, I integrated the Z3 ”satisfiability modulo theories” (SMT) solver into the ACL2 theorem prover. The capabilities of these two tools are largely complementary – Z3 provides fully automated reasoning about boolean formulas, linear and non-linear systems of equalities, and simple data structures such as arrays. ACL2 provides a very flexible framework for induction along with proof structuring facilities to combine simpler results into larger theorems. While both ACL2 and Z3 have been successfully used for large projects, my work is the first to bring them together.

I demonstrate this approach by verifying properties of a clock-generation circuit (called a Phase-Locked Loop or PLL) that is commonly used in CPUs and wireless communication.

Preface

The work presented in this thesis has been published as Yan Peng and Mark Greenstreet (2015). *Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification.* 7th NASA Formal Methods Symposium. April 27-29, 2015, Pasadena, California, USA.

Portions of the text in this thesis are modified with permission from Y. Peng and M. Greenstreet (2015) of which I am one of the authors. I am responsible for designing and constructing all programs and proofs, carrying out performance and result analysis of the research data.

Chapter 2, Program 2.1, Program 2.2, and Program 2.3 are adapted from the online documentation of the open-source theorem prover ACL2. The programs in chapter 2.2.5 are my authentic work and have not been published elsewhere.

I am the lead researcher for the projects located in Chapters 3 and Chapter 4 where I am responsible for all program development, proof construction, data collection and analysis, as well as the majority of manuscript composition. Chapter 4, Figure 4.2 is provided by Yu Ge with permission. Equation 4.1 is modeled and derived by professor Greenstreet. The digital Phase-Locked Loop example originated from my joint work with J. Wei, G. Yu and M. Greenstreet [110].

Table of Contents

Appendices

List of Tables

List of Figures

List of Programs

Acknowledgements

First of all, I wish to express my sincere thanks to Dr. Mark Greenstreet for his delightful guidance in my Master's study. It was Mark who introduced the amazing world of formal verification and analog circuits to me. Dr. Mark Greenstreet's great passion and enthusiasm in research keeps inspiring and stimulating me. Now that I've been prepared with motivation and skill, I think I am ready for continuing my Ph.D. study with Dr. Mark Greenstreet now.

Second, I would like to thank my second reader, professor Ronald Garcia for his insightful comments and valuable suggestions. Discussions with him about programming language theories have given me a new perspective on my thesis work.

Third, I would like to thank professor Shahriar Mirabbasi and his student Yu Ge for instructions on how a Phase-Locked Loop works. Thanks to Yu Ge for helping with the Spectre® simulation.

Fourth, I would like to thank my fellow lab mates: Jijie Wei, Brad Bingham, Shabab Hossain, Sam Bayless, Celina Val, Mike Enescu, Jinzhuo Liu and Long Zhang for the fun time we spent together. Jijie has always been a great friend of mine. I can't imagine my first two years' Master's study without her companion. My limited model checking knowledge are largely due to the stimulating discussions I had with Brad. Now that he has begun his career, I wish him all the best.

Finally, I would like to thank my family for undivided love and support on my foreign study and life. I miss them so much. I would also like to thank my boyfriend, Dongdong Li, for being a man after my own heart.

Chapter 1

Introduction

Ubiquitous computer technology [111] is driving increasing integration of digital computing with continuous, physical systems. Examples range from mobile devices such as cell phones and tablets, through traditional computers such as laptops, desktops, and servers, to embedded systems including toys, kitchen appliances, automobiles, and medical equipments. A common theme in this technology is the integration of digital and analog capabilities. Analog functions include features that are apparent to the user such as wireless networking, cameras, displays, microphones, speakers, and motion sensors, along with system-level infrastructure such as circuits that regulate the clocks, power supplies, and temperature of chips.

Combining analog circuits, digital circuits, and software into systems that interact with physical systems presents many design and verification challenges. First, each of the design domains uses its own models and design methods. For example, analog circuit designers use circuit simulators that are based on numerically integrating non-linear differential equations; digital designers use models based on boolean logic and finite state machines; and software developers use (for example) object-oriented languages with extensive libraries that provide a much higher level of abstraction than those of analog or digital design. Furthermore, each of the relevant time-scales vary widely for each of these design domains: analog designers require simulations with sub-picosecond resolution; digital design works with clock periods a thousand times longer ranging from hundreds of picoseconds to tens of nanoseconds. Software works on time scales another thousand times longer ranging from microseconds to seconds. Finally, the aircraft, hospital patient, or toaster that depends on the computing device may respond in times from seconds to minutes. It is impractical to perform simulations that cover timescales of seconds or minutes with picosecond resolution. Thus, abstractions are essential. Such abstractions are also a favorite hiding place for bugs when an abstraction describes what the designer intended rather than what the implementation actually does.

This thesis focuses on a particular class of designs, *analog/mixed-signal* (AMS) circuits. These circuits combine analog and digital modules to implement functions that would have been purely analog in earlier designs. The designer's motivation for AMS design is that modern fabrication processes for integrated circuits offer the designer billions of transistors that are optimized for digital applications but less-well suited for analog circuits. This reflects the reality that nearly all of the transistors on most chips are used in digital circuits, but the remaining analog functions are critical for the system. Thus, traditional analog functions (e.g. integrators) are replaced by their digital equivalents (e.g. accumulators). The unavoidable analog blocks include circuits such as oscillators, level comparators, voltage regulators, and RF amplifiers. Even these often include digitally controlled configuration settings to compensate for the large, statistical variation between transistors and other circuit components that is inevitable with the very small geometric features of integrated circuits. These AMS circuits are mixed analog and digital systems, typically consisting of multiple analog and digital feedback loops operating at much different time scales. While my focus is on AMS, the issues that I address are common to those in most computing devices.

It is not practical to simulate AMS circuits for all possible device parameters, initial conditions, inputs, and operating conditions. In fact, running just *one* such simulation may require more time than the design schedule. Most AMS circuits are *intended* to be correct for relatively simple reasons – errors occur because the designer's informal reasoning overlooked some critical case or had some simple error. My approach is to verify that the intuitive argument for correctness is indeed correct by reproducing the argument in an automated, interactive theorem prover. The advantage of the theorem prover is soundness and generality: by using a carefully designed and thoroughly tested theorem prover, we have high confidence in the theorems that it establishes. The critical limitation of using a theorem prover is that formulating the proofs can require large amounts of very highly skilled effort. My key contribution is to integrate a "satisfiability modulo theories" (SMT) solver into a theorem prover. This allows many parts of the proof, especially those involving large amounts of tedious algebra, to be performed automatically. As described in Chapter 2, several other research projects have integrated SMT solvers into theorem provers. However, we are not aware of any that have made extensive use of the real-arithmetic capabilities of SMT solvers or that have applied them to realistic problems from AMS design or hybrid systems. My focus on real arithmetic and AMS designs distinguishes the current research from prior work.

Thesis Statement

SMT solvers can be integrated into an interactive theorem prover in a sound and extensible way. This combination provides an effective tool for verifying properties of AMS designs including global convergence.

Contributions

This thesis presents my integration of the Z3 SMT solver [36] into the ACL2 theorem prover [71]. With this approach, the theorem prover provides support for high-level proof structuring and proof techniques such as induction, while the SMT solver discharges many tedious details of the proofs for verifying real-world designs. The implementation presented in this thesis supports booleans, integers, and reals, and my approach can be readily extended to other types including arrays, lists, strings, and more general algebraic data types. For soundness, my implementation replies on ACL2, Z3, and as little other code as possible. To make Z3 easily used from within ACL2, my interface performs many, automatic transformations of ACL2 formulas to convert them into the restricted form required by Z3. I resolve these seemingly conflicting objectives with a software architecture that divides the ACL2-to-Z3 translation process into two phases: most of the transformations are performed in the first phase, and the result is verified by ACL2. The second phase is a very simple direct translation from the s-expressions of ACL2 into their counterparts for Z3's Python API.

I demonstrate this approach by verifying global convergence for an all-digital phase-locked loop (PLL). The PLL is an example of an analog-mixed signal (AMS) design. This thesis considers global convergence: showing that an AMS circuit converges to the intended operating mode from all initial conditions. This requires modeling the large-scale, non-linear behavior of the analog components. If such non-linearities create an unintended basin of attraction, then the AMS circuit may fail to converge to the intended operating point. AMS circuits may make many mode changes per second to minimize power consumption, adapt to changing loads, or changes in operating conditions. Each of these mode changes requires the AMS circuit to converge to a new operating region. Once the AMS circuit is in the small operating region intended by the designer, small-signal analysis based on linear-systems theory is sufficient to show correct operation [73, 76].

The main contributions of this thesis are:

- The first integration of an SMT solver into the ACL2 theorem prover.
- A description of the challenges that arise when integrating a SMT solver with a theorem prover and solutions to these issues with an architecture where the code that must be trusted for soundness is both fairly small and very simple.
- A model for a state-of-the art digital PLL with recurrences using rational functions. This model can be used for evaluating other verification approaches.
- A proof of global convergence of the digital PLL.

Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 surveys prior research related to this thesis, including modern AMS verification techniques, an introduction to theorem proving and SMT techniques, and why combining them is better than using either alone. The chapter also describes prior results for verification of PLLs.

- Chapter 3 explains how to use the **trusted** clause processor construction from ACL2 to integrate Z3 into ACL2. The Chapter describes challenges that come up and presents my solutions to them.

- Chapter 4 describes the proof of global convergence for a state-of-the-art digital PLL using ACL2 with the clause-processor interface to Z3. The digital PLL is modeled with recurrence functions and apply analytical proofs to prove its global convergence. The proof shows the benefits of combining SMT techniques and theorem proving.

- Chapter 5 concludes the thesis and proposes opportunities for further research.

Chapter 2

Related Work and Background

Analog/Mixed-Signal (AMS) circuits are prevalent in integrated circuit designs. Chips require analog functionality for multimedia interfaces, sensors and actuators, and on-chip infrastructure such as power and clock distribution. The AMS approach replaces or augments traditional analog circuits with digital ones. The AMS approach is motivated by several technology trends including:

- Small geometric features lead to greater random variation between circuit components. Intuitively, transistors and wires are now so small that variations of a few atoms impact circuit performance. Many traditional analog circuits rely on having "matched pairs" of transistors, and such matching is no longer practical.

- Small devices and stringent power constraints mandate low operating voltages. Many traditional analog circuits rely on having a "voltage headroom" (the ratio of the power supply voltage to the transistor threshold voltage) that is not available in current processes.

- Digital circuits can exploit transistor scaling more effectively than analog ones. Smaller transistors lead to smaller logic gates and a greater density of digital functions, while the inductors and capacitors of analog design do not shrink by nearly as much. Thus, a designer can save area by replacing analog functions (such as an integrator that requires a large capacitor) with digital ones (such as a 24-bit accumulator).

- Replacing analog functions with digital counterparts makes the AMS circuit more programmable, allowing greater component re-use.

The AMS design approach creates challenges for verification. For example, analog blocks operate on time-scales that require picosecond scale time steps for accurate, transistor-level simulations, whereas digital adaptation loops may require times of microseconds to several milliseconds to converge. Formal approaches can verify circuit properties for a large class of inputs and device parameters and avoid the large compute times and incomplete coverage of simulation based approaches.

In this work, I model AMS circuits as discrete time recurrences of continuous values as proposed in [7]. The differential equation model for the analog circuit is used to determine how the continuous state evolves over a single period of the digital clock.

When verifying a recurrence system, one usually needs to form an induction proof that proves a specific property on each step of a recurrence system starting from any initial state. Specifications can be translated into arithmetic constraints, which can be non-linear. Theorem proving and SMT provide complementary capabilities for reasoning about recurrences. Theorem proving supports induction proofs, and SMT automates reasoning about non-linear equalities and inequalities.

This chapter describes historical and recent works related to my research. Section 2.1 discusses similarity and differences between analog, digital and AMS design verification. Section 2.2 gives an introduction to interactive theorem proving (specifically ACL2) and SMT methods (specifically Z3) and shows the motivation for combining the two. Section 2.3 discusses PLL verification.

2.1 AMS Design and Verification

AMS (Analog/Mixed Signal) design, as indicated by its name, refers to circuit designs that include both digital and analog circuitry. Traditionally, people would think of some circuits as being analog and others being digital.

Nowadays, nearly all circuits that would previously have been pure analog are implemented using mixed signal techniques.

AMS designs bring up new verification challenges. Analog circuits in an AMS design are naturally modeled and specified in terms of continuous behaviors. Thus, it is not possible to enumerate all possible initial states and generate all trajectories. Even if one accepts that full test coverage is unachievable, simulation of AMS designs is difficult because the analog circuits require detailed, short time-step modeling of non-linear circuits. AMS designers face a dilemma of missing deadlines, failing to eliminate corner cases, or using simplified abstractions that may hide real bugs. Digital controls in AMS designs exacerbate this situation. To approximate a smooth, continuous system, AMS designers tend to use digital control circuits that only make small changes to their outputs with each clock step. Thus, convergence can take hundreds to thousands or more clock cycles.

Jang *et al.* [68] discuss this problem in traditional simulation-based methods and propose an event-driven simulation method in SystemVerilog to solve it. They use (nearly) linear models for the analog components in an AMS design and use Laplace transform techniques to find closed-form solutions for the analog behaviours. From these they identify when analog signals cross switching criteria for the digital controller, and use those events to drive an event-driven simulation. Like other simulation based approaches, each simulation run only considers the behaviour from a single initial condition, with a single choice for any input stimulus functions, and a single choice of model parameters. The approach also relies on having accurate, linear models for the analog blocks in the AMS design. The major advantage of their approach is that it is much faster than performing detailed, transistor-level simulation with a simulator such as SPICE [5]. In comparison, formal approaches can reason about the whole system instead of just specific execution traces. This offers both faster verification and more comprehensive coverage than simulation methods.

2.1.1 Circuit Verification

Two main verification problem exist in circuit design: equivalence checking and model checking. This section discusses how these techniques are realized in the digital, analog and AMS domains respectively.

Digital Circuits

Mathematical models that model the dynamics of digital circuits are based on abstractions like the one below:

$$s(i+1) = next(s(i), in(i)) \tag{2.1}$$

where $s(i)$ are state vectors that have only elements 0 and 1; i and $i+1$ are indices for current step and next step; *next* stands for the discrete recurrence formula for calculating next state from current state; *in* represents circuit inputs.

Typically, model checking is performed using a next state *relation* for *next* instead of a function; in other words, a state may have multiple possible successors for the same input. The need for non-determinism arises from several directions including:

- To avoid the state-space explosion problem [96], model checking is usually performed on abstractions. Thus, the actual hardware or software has internal state that is not represented in the abstraction, and the effects of the internal state appear as non-deterministic behaviours.

- The verification may be performed on a high-level design before all of the details have been determined. For example, a router may be described without specifying the exact order in which packets are routed. This gives the designer freedom to optimize the design later when the details are better understood.

- The specification may state assumptions about the allowed inputs. Often, such specifications are complied into state machines for the environment, and model checking algorithm is applied to the product automaton for the system and its environment. The actions of the environment may not be fully determined, and this leads to non-determinism in the product automaton.

- Some physical behaviours such as metastability [85] cannot be captured by deterministic models.

Thus, we will often treat *next* as a relation.

A common approach to hardware design is to describe modules as state machines. Each module has a state that is maintained in *registers* and a *next state function* that describes how the state is updated on each clock cycle according to the current state and external inputs to the module. Such a description is typically written in a hardware description language such as Verilog [4] or VHDL [3] and is called a *register transfer level* (RTL) description. Software refered to as *logic synthesis* [104] converts RTL descriptions into networks of logic gates and flip-flops. Such a network is called a *netlist*. The logic synthesis software can perform very aggressive optimizations. On the other hand, the number of hardware designs that are synthesized and manufactured is much smaller than the number of software programs that are compiled for a mainstream language such as Java or C++. Furthermore, errors in the hardware design are very expensive because of high fabrication costs and the fabricate, test, and revise cycle can take several months. These considerations motivate using formal methods to verify the netlists produced by logic synthesis software. This is the motivating example for the *logic equivalence* problem described below.

There are two problems typically addressed by mature verification methods for digital verification - equivalence checking and model checking. Both of them have been widely adopted by the chip design industry.

- Equivalence checking

  The digital circuit equivalence checking problem is verifying that the next state function described by the netlist is equivalent to the one described by the register-transfer level (RTL). In other words, the equivalence checker shows that $next_{\text{netlist}} \neq next_{\text{RTL}}$ is unsatisfiable. Typically, the RTL description fully specifies the behavior of the device; in this case, *next* is a function. A few examples of approaches to digital equivalence checking include [16, 26, 51, 75].

- Model checking

  Digital circuit model checking asks whether all possible sequences of states arising from the model (see Equation 2.1) have certain desired properties. The kinds of properties include:

  - *Safety*: show that $s(i)$ is never bad; i.e. the state machine model will never go into states that violate certain constraints.
  - *Liveness*: show that $s(i)$ is eventually good; will always go into states that satisfy certain constraints.

  Burch *et al.* [28] propose a symbolic model checking method that uses BDD to represent formulas symbolically and uses μ-calculus algorithm to derive efficient decision procedures for CTL model checking. [30] summarizes major breakthroughs in model checking. Most model checking has been based on BDDs because BDDs provide operations for composing function and relations, and a canonical representation that aids in computing fix points. Recently, IC3 [25] has demonstrated the feasibility of using SAT solvers for model checking by using interpolation [88] and k-induction [103]. The IC3 approach has been very successful on both benchmark problems and real-world examples.

Analog Circuits

Ordinary differential equations, as shown below, are a natural model for the behaviours of analog circuits.

$$\frac{dx}{dt} = f(x, in, u) \tag{2.2}$$

where $x \in \mathbb{R}^N$ represents the state of the analog circuit; $in \in \mathbb{R}^M$ represents external inputs to the circuit; and $u \in \mathbb{R}^K$ represents uncontrollable disturbances. As with models for digital circuits, it can be convenient to use a differential inclusion that accounts for all possible disturbances. Such a differential inclusion can have a form like the one shown below

$$\dot{X} = F(X, In) \tag{2.3}$$

where $X \subseteq \mathbb{R}^N$ is a subset of the state space, and $In \subseteq \mathbb{R}^M$ is a subset of the input space. Likewise, $\dot{X} \subseteq \mathbb{R}^n$ is the set of possible time derivatives for these states and inputs. We note that it is common to have uncertainty in f, the circuit model itself. For example, we may not have an exact model for transistor currents or node capacitances. Such uncertainties can be captured using inclusions. We omit the details of how such inclusions are constructed, and will assume that they are available for the purposes of verification in the remainder of this thesis.

Analog circuit verification problems can be cast as equivalence checking and model checking problems as well.

- Equivalence checking

  Equivalence checking for analog circuits aims at the same target as for digital circuits. But analog circuit equivalence checking is much more tricky. Basically, one might ask the question “how close is close enough” for an implementation and its specification, given the state space is continuous. Different researchers give different answers to this question.

  Hedrich and Barke [63] in 1995 proposed a procedure for calculating a non-linear mapping from one non-linear system to another system. They first compute a linear mapping by doing an eigenvalue analysis then adjust the mapping using quasi-newton optimization on the error of the state derivatives. They argue the two system to be equivalent if for each sampling point, the error of state derivatives and state values are within given ranges.

- Model checking

Being an analogy to digital circuit model checking, the typical model checking problem of an analog design is to look for reachable sets given bounds on inputs. Safety and liveness properties can also be proven by looking at the intersection of reachable sets with bad or good sets.

Formal verification of analog and AMS circuits is an emerging area. I'll describe some other prior work on analog verification and AMS verification together in Section 2.1.1.

AMS Circuits

AMS circuits combine analog and digital circuits. In this work, I model AMS designs using discrete time recurrences with both continuous and discrete valued variables:

$$\begin{aligned} \frac{dx}{dt} &= f_q(x) \\ q(i+1) &= d(q(i), th(x)) \end{aligned} \tag{2.4}$$

where x is a vector of continuous analog states, q is a vector of discrete digital states, t is time, i is the step index, f_q stands for the derivative of x to time given state q, d stands for the next state relation for q and th is a sampling function that samples the continuous states at time points where the discrete steps are.

As with digital and analog circuits, I will roughly categorize prior work as equivalence checking and model checking.

1. Equivalence checking

   For the same reason that equivalence checking of analog circuits is problematic, to what extent can two models be called equivalent is a matter of choice.

2. Model checking

   The model checking problem of an AMS circuit also looks at the problem of whether all trajectories satisfy certain safety or liveness properties. Three approaches exists to do model checking with AMS circuits.

One way involves state space discretization followed by discrete model checking methods. The second way uses a hybrid automata that models the discrete behavior between states and models the continuous behavior within a state. The third way uses the observation that the ODE part of the recurrence model is usually simple. Then one can just solve the linear model and then reason about the recurrences alone.

- Discretization

  The earliest attempt to apply model checking to circuit verification is Kurshan and McMillan's 1991 paper [77]. They partition the range of values for each continuous variable into intervals, and thus discretize the continuous state space as a finite set of hyper-rectangles. They compute bounds on the derivative function and use these to obtain a next state relation. They demonstrate their approach by verifying the asynchronous arbiter circuit from [102] assuming that input transitions are instantaneous. They propose heuristics on how to reduce from a continuous problem to a discrete one by properly choosing granularity of space discretization, time discretization, input value and input function discretization.

  Based on similar idea, Hartong *et al.* [61] proposed a method that automatically subdivides state space into boxes satisfying certain Lipschitz conditions. That way, they can sample points from a given box and argue that the proposed inclusion algorithm over-approximates the reachable states. They also introduced a modified CTL model checking technique for analog verification.

- Hybrid automata and reachability

  Reachability analysis considers the problem of where the trajectories can go given a set of initial state points. It can be distinguished from discretization-based method in that it reasons about the system in the continuous space. Greenstreet [56] presents a method of using Brockett annulus to verify that a toggle circuit modeled by a system of non-linear differential equations satisfies a discrete specification. He uses numerical integration to determine a manifold that contains all feasible trajectories. Coho [57, 112] proposes a method called projectagon that projects high-dimensional objects onto two-dimensional planes. Reachable sets are calculated by integration and linear programming is used to bound reachable trajectories. d/dt [11] uses hybrid automata to model AMS behaviour and uses orthogonal polyhedra to over-approximate reachable sets for proving safety problems. Many other representations exists.

All of these approaches face the challenge that representing arbitrary polyhedra in a high-dimensional space is intractable. Thus, different approaches employ different simplified representations such as orthogonal polyhedra [11], convex polyhedra [44], projection based methods [112], ellipsoids [78], zonotopes [50]. In general, there is a trade-off between the amount of over-approximation incurred by the representation and the time and memory required to perform the analysis.

- Transform to recurrences

  Al-Sammane *et al.* [7] proposes a symbolic method that extracts a mathematical representation of any AMS system in terms of recurrence equations. They build an induction tool in Mathematica to prove correctness using the normalized equations. Note that the model in the example in Chapter 4.2 uses this idea to abstract the continuous dynamics of the phase difference variable. My example shows how their approach can be extended with more powerful analysis tools to verify a state-of-the-art AMS design.

3. Other analytical methods

- Interval based methods

  Tiwary *et al.* [108] proposed a method that starts from the transistor level circuit netlist, using intervals to represent the differential I-V characteristics of transistors. They then model verification problems in mainly linear inequalities and use SMT techniques to solve the linear inequalities. The paper didn't state how the transistor level intervals can be obtained.

- Theorem proving

  Prior work on using theorem proving methods to reason about dynamical systems includes [66] which uses the Isabelle theorem prover to verify bounds on solutions to simple ODEs from a single initial condition. In contrast, I verify properties that hold from all initial conditions. Harutunian [62] present a very general framework for reasoning about hybrid systems using ACL2 and demonstrate the approach with some very simple examples. Here I demonstrate that by discharging arithmetic proof obligations using a SMT solver, it is practical to reason about more realistic designs.

2.1.2 Limitations

The methods described above have several limitations. First, many of the introduced methods require the model of the system to be fixed, meaning that the verification is for a specific choice of values for the circuit parameters. The circuits that are actually fabricated will have different parameters values than those used in the verification.

With continuous state spaces, AMS circuits have an uncountably large number of states. Thus, tools must make approximations. If the approximations are too course, the tools will over-approximate the reachable space and report false-errors. On the other hand, if the approximations are too fine, then the run-time and memory requirements may be completely impractical. Thus, most prior work on AMS verification has been limited to small examples. Furthermore, large amounts of manual effort are often needed, even with “automatic” tools, to tune the circuit models and verification algorithms to a sweet spot that allows the verification to complete. A few larger AMS verification examples have been published in the past few years, all looking at various phase-locked loop designs. I also use a phase-locked loop as the case study in this work. Section 2.3 introduces phase-locked loops and prior verification efforts for such designs.

2.2 Introduction to ACL2 and Z3

This section gives a brief introduction to general theorem proving and SMT techniques. I observe that theorem proving and SMT methods offer complementary capabilities for AMS verification.

2.2.1 Theorem Proving Overview

Theorem proving means using computer program to prove mathematical theorems based upon mathematical logic rules. For the sake of organizing the presentation, this section examines theorem provers in two major groups: those based on first-order logic, and those based on higher-order logic. First-order logic is distinguished from higher-order logic in the sense that first-order logic only quantifies over individuals but higher-order logic can quantify over sets, sets of sets, etc. [9]. Noting that a function can be represented as a set of tuples mapping values in the function's domain to values in its range, it can be observed that first order logic does not admit quantification over functions, but higher order logic allows such quantification. E.g. $\forall P \forall x (\exists y. P(x, y))$ would be a higher-order logic predicate but simply $\forall x (\exists y. P(x, y))$ would be a first-order logic predicate.

Proponents of theorem provers for first-order logic often argue that first order logic is adequate for modeling verification problems [101]. In general, first-order logic is simpler, thus easier to model and manipulate than higher-order logic. Very sophisticated theorems can be built from first-order logic if enough translation and modeling is used.

Conversely, proponents of theorem provers for higher-order logic often argue that modeling problem with higher-order logic is more natural and intuitive. Gordon [55] extensively discusses why higher-order logic should be a good formalism for hardware verification in his early paper. He argues in the paper that higher-order logic are obvious modeling language for hardware verification problems. Furthermore, higher-order logic enables the ability of reasoning about logic within the logic. Because one can position quantifiers ahead of predicates and functions, thus one can naturally prove the correctness of a proof method, or embed semantics for various programming languages within the logic by using higher-order logic.

This section further discusses existing theorem provers in each category. As a representative example of higher-order theorem provers, I will examine the HOL [54] family of theorem provers. Likewise, I will use the Boyer-Moore theorem prover [23] and its decendants, most notably ACL2 [71], as the canonical example of a theorem prover for first-order logic. Many other extensively developed theorem provers include the Coq [18] theorem prover, PVS [94], and nuPRL [67].

The HOL Family

HOL [54] is one of the earliest theorem provers for higher-order logic. HOL means Higher-Order Logic. The HOL family refers to a list of modern theorem provers based on the foundation of HOL [29], including HOL Light [58], HOL4 [106], Isabelle [95] etc. In a HOL-based theorem proving system, all proofs are derived from a small set of HOL axioms. The system supports reasoning about higher-order functions and propositions. The proofs are constructed in the forwards (bottom-up) style.

There are a number of interesting verification results both from industry and academia using HOL family theorem provers. Pusch [98] uses Isabelle to verify soundness of the Java bytecode verifier that checks several security constraints in a Java Virtual Machine (JVM). Harrison [59] uses HOL light for formalization of floating-point arithmetic, and the formal verification of several floating-point algorithms. Many mathematical results have been developed using HOL based theorem provers: [1] is a webpage showing 100 well-known theorems from mathematics that have been formalized using modern theorem provers. Of the 100 theorems, 86 have been formalized and proven using HOL Light, significantly more than any other theorem prover. The QED project [22] aims at building a computer system to represent all important knowledge and techniques in mathematics. These show theorem provers' power in proving classical, mathematical results as well as establishing useful properties of hardware and software designs.

The Boyer-Moore Theorem Prover

The Boyer-Moore theorem prover [23], also known as NQTHM, is a theorem prover for first-order logic based on a dialect of Lisp. The key idea is to develop a version of Lisp with a simple semantics axiomatized in the prover. Users write code in this Lisp dialect both to model and reason about target systems. ACL2 [71] is a direct descendant of the early Boyer-Moore theorem prover. ACL2 is short for A Computational Logic for Applicative Common Lisp.

There are several defining features of ACL2. First, ACL2 reasons about Lisp code within Lisp. Second, it's defined to be both automatic and interactive. It is automatic because its automatic proof search engine is implemented for searching for a proof tree. The underling automatic proof engine follows a procedure called the "waterfall". The waterfall tries to solve each goal by passing it through a series of proof processes [72]. It is interactive because the user needs to follow *The Method* [2] that develops lemmas for unproved goals and iteratively follow this strategy until every lemma is proved. Collections of commonly used lemmas can be collected into *books*. ACL2 *certifies* these books, allowing such lemmas to be used without repeating the proof each time. Many such books have been developed that are carefully crafted to work with the ACL2 waterfall – this allows ACL2 to automatically perform long sequences of common proof steps such as rewriting terms into canonical forms.

The ACL2 community focuses on large verification problems arising from industry. Accordingly, ACL2 has a strong emphasis on speed and automation. There has been a huge number of successful applications of NQTHM and ACL2 to both academic and industrial verification problems. Some examples of proofs performed in ACL2 include:

Concurrent programming:

[91] proves correctness of a system of n processes each running a simple,

non-blocking counter program: if the system runs longer than some given number of steps, then the counter will increase, which guarantees progress.

Microprocessor verification: [41, 92] both apply ACL2 to real, large, industrial examples of processor designs.

Security: [64] considers a security problem of information flow. Given a program that has been annotated with assertions about information flow, their method uses ACL2 and operational semantics to generate and discharge asserted conditions.

Floating point arithmetic: [100] describes a method for translating from a subset of Verilog language into the formal logic of ACL2 and proves correctness of register-transfer level models of floating-point hardware designs at AMD. [65] verifies the floating-point addition/subtraction instructions for the media unit in Centaur's microprocessor.

Numerical algorithms: [99] describes how symbolic differentiation is introduced into ACL2(r) [46]. [47] presents a proof in ACL2(r) on the convergence rate of the sequence of polynomials that approximate arctangent proposed by Medina [89].

Suitability of Theorem Provers for AMS Verification

Theorem provers are suitable for AMS verification because:

- Theorem provers provide extensible capabilities for reasoning about linear and non-linear inequalities.

- Theorem provers are designed to have strong support for reasoning about sequences. This is essential for reasoning about AMS circuits using recurrences as described in 2.1.1. This is due to their powerful induction proof support. For example, in ACL2, every user-defined function must be defined with a proof of termination; in practice, these proofs are often found automatically by ACL2. Once a recursive function is defined, it defines a corresponding induction schema. Thus, introducing new induction schema in ACL2 is straightforward, and inductive reasoning is highly automated.

- Composibility and reusability of verification results are much more obvious in a theorem prover because proved theorems are reusable. Once proved, all theorems will be stored in the system and can be used to prove new results. Often the results of tools such as reachability checkers (see Section 2.1.1) only show one aspect of a complete correctness argument. These lemmas need to be combined to prove the desired claim. Theorem provers provide a natural and comprehensive framework for composing these results.

The common objection to using interactive theorem provers such as ACL2 is that the proofs requires large amounts of manual effort and a level of mathematical sophistication that puts them out of the reach of typical programmers and hardware designers. Much of this is because theorem provers require all claims to be reduced to a small set of axioms. Automatic tools such as SAT and SMT solvers can automate much of this low-level reasoning. Section 2.2.2 examines these solvers.

2.2.2 SMT Solver Overview

Researcher have developed many techniques for solving decision problems that arise in hardware and software verification. Practical decision procedures now exist for many common problem domains. Boolean satisfiability (SAT) problems are the set of problems that ask if there exists a satisfying assignment to a boolean formula. Although SAT problem is in general NP-Complete, modern SAT solvers manage to solve a large portion of the SAT problems that arise in practice quite efficiently by developing efficient search algorithms with useful heuristics. While SAT solvers can answer questions phrased as boolean formulas, other decision procedures have been developed for other domains. For example, the satisfiability of a system of linear equalities and inequalities can be determined using a linear program solver. Solvers exist for classes of non-linear constraints, reads and writes to arrays, and other domains. This motivates devising decision procedures that combine the results of domain specific solvers. When such combined solvers are implemented as a generalized version of a SAT solver, the resulting approach is known as satisfiability modulo theories (SMT). This section describes research in SAT and SMT solvers and some of the modern heuristics used in these solvers.

Booleans and SAT

The SAT problem has been studied since the early days of computer science [34], and is the classical example of a NP-complete problem [32, 70]. From a verification perspective, SAT is interesting because many problems that arise in verification can be naturally expressed as SAT problems. For example, equivalence of an RTL specification and a gate-level netlist can be expressed as a SAT problem [15]. The earliest work proposing an algorithm for solving SAT problems dates back to the 1960s. Davis, Putnam *et al.* [34, 35] developed the earliest Davis-Putnam-Logemann-and-Loveland (DPLL) algorithm framework that remains the foundation for many SAT solvers. The DP (Davis-Putnam) and DPLL algorithms work on formulas written in conjunctive normal form (CNF), i.e., the conjunction of clauses, where each clause is a disjunction of variables or their negations. The basic idea is that if a clause consists of a single variable (or negation of a variable), then that determines the value of the variable in any satisfying assignment. If all such one-literal clauses have been eliminated, then the solver picks a variable and performs case split on the value of that variable and simplifies the resulting formula. If a satisfying assignment is found, it is reported. If a contradiction is found, the solver backtracks. Eventually, either a solution is found or the formula is shown to be unsatisfiable.

Marques-Silva and Sakallah [86] further enhanced the DPLL algorithm by adding a conflict analysis procedure that provides information for more efficient backtracking. Zhang *et al.* [114] survey various conflict driven learning strategies and did a thorough experiment in comparing different learning schemes. Zhang and Malik [113] surveys big breakthroughs in SAT solving including branching heuristics, variation in deduction and conflict learning strategies. Gomes *et al.* [53] summarizes key-features of modern DPLL-based SAT solvers and extended topics on quantified boolean formula (QBF) solving and model counting.

SMT

SMT solvers extend a SAT solver with procedures for solving problems in other domains. Typical domain specific procedures include procedures in integer arithmetic, linear real-arithmetic, non-linear arithmetic and array theory. Closely related to AMS verification are the domain specific solvers for real arithmetic. The first work that gives a decision procedure for "elementary algebra"[1] is by Tarski [107]. In his work, he gives a procedure that proves the decidability of such problem, but the procedure is impractical with a complexity, using Knuth's up-arrow notation [74], of $2 \uparrow\uparrow n$ for a formula of size n. Buchberger developed Gröbner bases [27, 79] which can be used to solve systems of polynomial equalities. The cylindrical algebraic decomposition approach of Collins [31] can find satisfying solutions to systems of polynomial equalities and inequalities, or show that no such solution exist. Both algorithms have doubly-exponential time complexity. Ben-Or *et al.* [17] showed that the decision problem for elementary algebra is exponential-space complete; so, the Collins algorithm is likely to be optimal. Nevertheless, these algorithms have found use in practice, especially when augmented with heuristics to simplify problems before attempting a general solution. Other related work includes Bledsoe *et al.* [21], and Shostak [105].

Research on satisfiability solvers has been complemented by work on combining decision procedures for various domains into a single, unified solver. These solvers go by the name SMT (Satisfiability Modulo Theory) solvers. One of the earliest contributions in this area was the "cooperating decision procedure" approach of Nelson and Oppen [93]. They presented a combination of a theory of linear equalities and inequalities for real numbers, arrays, list structure and uninterpreted functions. They present a unifying

[1] Elementary algebra comes from Tarski's definition in [107].

framework for combining different decision procedures. Their method requires that the separate theories only communicate by equality of terms and it only applies to convex theories. A theory is convex if for each conjunctive formula in the theory, if it implies a finite disjunction of equalities, then it also imply at least one of the equalities. Instead of coordinating two theories, Bozzano *et al.* [24] propose a method called delayed theory combination that first let the SAT solver propose a satisfying assignment for the case splitting on equalities between theories, thus delayed the combination of theories. Their work also works for non-convex theories. Examples of modern SMT solvers include Yices [40], Z3 [36] and CVC4 [14].

There exist other works that focus on various aspects of SMT solving. HySAT [43] uses an algorithm that tightly integrates interval constraint propagation with SAT algorithm to solve large systems of non-linear inequalities. Gao *et al.* [49] formulated a theory of ODEs and proposed an algorithm under the interval constraint propagation (ICP) [52] framework to solve SMT problems with ODE constraints.

Suitability of SMT Solvers for AMS Verification

For the domain of AMS verification, SMT solvers compliment theorem provers for following reasons:

- SMT solvers lack the extensive proof structuring and management of interactive theorem provers. SMT solvers are often used to solve pieces of the verification problem, and a more general framework is needed to make sure that these lemmas are sufficient to prove the desired result.

- SMT solvers are weak at reasoning about infinite structures (i.e. lack of induction). Researchers are aware of it, and there exists preliminary works on extending SMT solver's induction proof abilities.

  For example, Leino [82] proposed a mechanism for translating assertions about recursive functions into the proof obligations for an inductive proof of the claimed property. Leino implemented this approach as an extension to the Dafny [81] program verifier which translates the proof obligations to Boogie 2 [80] which uses the Z3 SMT solver [36]. However, the induction ability such tools can provide is still limited in comparison to a theorem prover.

- SMT solvers are extremely good at solving systems of inequalities with a moderate number of variables. The AMS formula one wants to verify might be too tedious for the user of a theorem prover, thus there's a need for combination of SMT technique into a theorem prover.

- As a fully-automated approach, SMT solvers are vulnerable to the combinatorial explosion problems. By breaking a problem into lemmas in the theorem prover, the SMT solver works on manageable subformulas. It is tempting to write a lemma that "tells the SMT solver everything you know" and then ask it to prove the claim. This often leads to the SMT solver taking more time than the user has patience (typically a few hours, aka, a "time-out" failure) or requiring more memory than available on practical computers (aka a "mem-out" failure). On the other hand, if the user identifies the hypotheses that are likely to be needed and breaks the problem into a few smaller pieces, then the SMT approach succeeds much more often and still spares the user from large amounts of tedious derivation.

2.2.3 Integrating External Procedures to Theorem Provers

There has been extensive work in the past decade on integrating SAT and SMT solvers into theorem provers including [10, 19, 20, 37, 42, 87, 90]. Many of these papers have followed Harrison and Théry's[60] "skeptical" approach and focused on methods for verifying SMT results within the theorem prover using proof reconstruction, certificates, and similar methods. Several of the papers showed how their methods could be used for the verification of concurrent algorithms such as clock synchronization [42], and the Bakery and Memoir algorithms [90]. While [42] used the CVC-Lite [12] SMT solver to verify properties of simple quadratic inequalites, the use of SMT in theorem provers has generally made light use of the arithmetic capability of such solvers. In fact [20] reported *better* results for SMT for several sets of benchmarks when the arithmetic theory solvers were disabled!

The work that may be the most similar to this work is [37] which presents a translation of Event-B sequents from Rodin [6] to the SMT-LIB format [13]. Like my work, [37] verifies a claim by using a SMT solver to show that its negation is unsatisfiable. They address issues of types and functions. They perform extensive rewriting using Event-B sequents, and then have simple translations of the rewritten form into SMT-LIB. While noting that proof reconstruction is possible in principle, they do not appear to implement such measures. The main focus of [37] is supporting the set-theoretic constructs of Event-B. In contrast, my work shows how the procedures for non-linear arithmetic of a modern SMT solver can be used when reasoning about VLSI circuits.

My work demonstrates the value of theorem proving combined with SMT solvers for verifying properties that are characterized by functions on real numbers and vector fields. Accordingly, the linear- and non-linear arithmetic theory solvers have a central role. As the concern is to bring these techniques to new problem domains, I deliberately take a pragmatic approach to integration, and trust both the theorem prover and the SMT solver.

2.2.4 ACL2 and *The Method*

This section serves as an introduction to how to use the theorem prover ACL2 by following *The Method* [2]. Basically, *The Method* is a depth-first traversal over the derivation tree of the target theorem directed by ACL2. See Figure 2.1.

Given a theorem statement, the user may first write it in ACL2 and check if ACL2 can prove it by automatically applying its proof engine. If proved, then done. If not, the user can look at the checkpoint generated by the proof engine illustrating the point where the proof engine gets stuck. Then the user can come up with a new lemma that should prove the checkpoint theorem statement. Iteratively, the user can run the lemma statement in ACL2 and

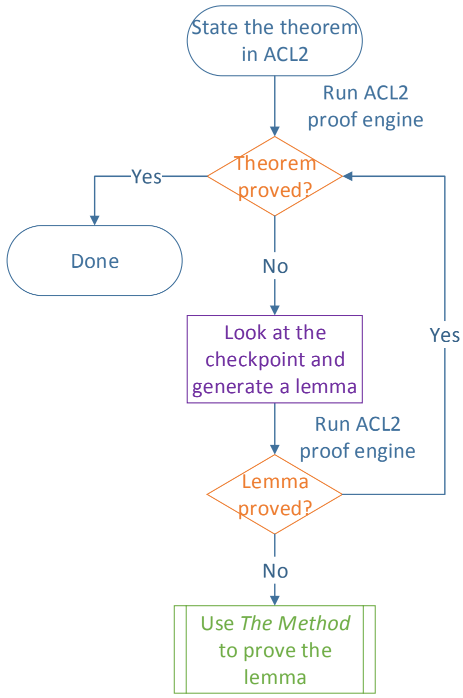


Figure 2.1: *The Method*

check if it's proved. If yes, try proving the original theorem again. If not, apply *The Method* to prove the lemma statement. The process is partially automatic and partially interactive.

I take an example from the ACL2 documentation to show how to apply *The Method*. Suppose one wants to prove Theorem 2.1 below:

Theorem 2.1 (Example theorem)**.**
A list contains no duplicated elements if and only if the reverse of the list contains no duplicated elements.

Suppose we have already define the function for reversing a list and checking for duplicates as in Program 2.1. Program 2.2 shows the theorem statement as written in ACL2.

Program 2.1 Function definitions for rev and dupsp

```
(defun rev (x)
  (if (endp x)
      nil
      (append (rev (cdr x)) (list (car x)))))

(defun dupsp (x)
  (if (endp x)
      nil
      (if (member (car x) (cdr x))
          t
          (dupsp (cdr x)))))
```

Program 2.2 Example theorem statement

```
(defthm dupsp-rev
  (equal (dupsp (rev x)) (dupsp x)))
```

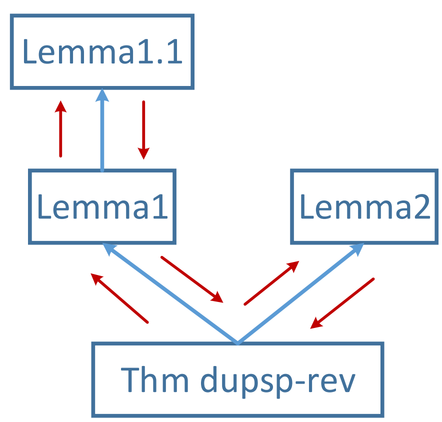


Figure 2.2: The proof tree

Try proving the theorem in ACL2 using *The Method*, one will end up with a proof tree as shown in Figure 2.2, where Lemma1, Lemma1.1 and Lemma2 are shown in Program 2.3. Try proving theorem *dupsp-rev* in ACL2 produces a checkpoint:

```
(IMPLIES
  (AND
    ; X is a non-empty list
    (CONSP X)
    ; the first element of X is not an element of the tail
    (NOT (MEMBER (CAR X) (CDR X)))
    ; the induction hypothesis
    (EQUAL (DUPSP (REV (CDR X)))
                       (DUPSP (CDR X))))
  ; the original claim with REV expanded once
  (EQUAL (DUPSP (APPEND (REV (CDR X)) (LIST (CAR X))))
                (DUPSP (CDR X))))
```

which suggests lemma1. Attempting to prove lemma1, ACL2 produces a checkpoint that contains the term:

```
(MEMBER (CAR X) (APPEND (CDR X) (LIST E)))
```

We see that ACL2 needs to understand how `MEMBER` interacts with `APPEND`, which suggests Lemma 1.1. ACL2 proves Lemma 1.1 without any further assistance. After proving Lemma 1.1, we give Lemma 1 to ACL2, and ACL2 proves Lemma 1 as well. We ask ACL2 to attempt to prove the main theorem, and it fails with a checkpoint of the same form as last one. Through some thinking, one can figure out that ACL2 gets stuck on proving

```
(NOT (MEMBER (CAR X) (REV (CDR X))))
```

even given that it knows

```
(NOT (MEMBER (CAR X) (CDR X))).
```

So we come up with lemma2, which points out that a member of a list is also a member in the reverse of that list. This will lead to ACL2's automatic reasoning for

```
(IMPLIES (NOT (MEMBER (CAR X) (REV (CDR X))))
         (NOT (MEMBER (CAR X) (CDR X))))
```

Finally, ACL2 accepts the initial theorem statement for *dupsp-rev*.

Program 2.3 Lemmas

```
; e is an element of the concatenation of lists a and b
; iff e is an element of a or e is an element of b.
(defthm lemma1.1
   (iff (member e (append a b))
        (or (member e a)
            (member e b))))

; If e is not a member of x,
; then appending e to x does not change whether or not
; x has duplicate elements.
(defthm lemma1
    (implies (not (member e x))
             (equal (dupsp (append x (list e)))
                    (dupsp x))))

; e is an element of the reverse of x
; iff e is a member of x.
(defthm lemma2
   (iff (member e (rev x))
        (member e x)))
```

In summary, using *The Method* to prove a theorem is an automatic and interactive way of building the proof tree in ACL2. ACL2 automatically does the job of decomposing the theorem into subgoals, using rewriting and other techniques on simplifying the main goal and subgoals and so forth. When it gets stuck somewhere in the traversal of the proof tree, user intervention is required to come up with the right lemma to resolve the stuck point. This continues until the original theorem statement is proved. When the proof is complete, the user has an ACL2 script that can be executed to perform the full proof automatically, without user interaction. Note that *The Method* is a guideline for proving theorems in ACL2, but users may at times choose other ways of identifying helpful lemmas and structuring their proofs. For example, there may be a better way of decomposing the initial theorem statement than what is proposed by ACL2. The user can then provide as hint this decomposition to ACL2 so that ACL2 can use this intuitively better proof suggested by the user.

2.2.5 Examples Using ACL2 and Z3

This section presents to examples to illustrate the use of the ACL2 theorem prover, the Z3 SMT solver, and their combination as implemented in this thesis.

I've specifically chosen ACL2 as the theorem prover and Z3 as the SMT solver. The reason for these choices is somewhat coincidental. When I started out, I first tried HOL Light. My first experience with theorem proving got stuck when I tried to figure out how to introduce an external decision procedure. My supervisor mistakenly believed that SMT solvers had been integrated into ACL2 already. So I then tried ACL2. Although no such integration existed at the time, thanks to the comprehensive documentation of ACL2 and constant help from the ACL2 development group, I was able to devise an approach based on how SAT solvers get integrated. The reason I've chosen Z3 is even simpler. First, it is a leading SMT solver. Second, it's very easy to try out given the web-based interactive webpage and the z3py interface. Third, I had used Z3 to prove some simple properties of the digital PLL, i.e. automatically deriving and verifying a ranking function for convergence. This use of Z3 showed both the value of Z3, and the need for a more comprehensive collection of reasoning techniques.

While I have implemented our approach using ACL2 and Z3, the approach is largely independent of the choice of SMT solver and should work equally well with other SMT solvers or even other decision procedures. Likewise, the approach presented Chapter 3 could be used with other theorem provers, but I would not expect as much direct code reuse in that case.

Example: Sum of Geometric Series

The first example demonstrates ACL2's induction power, which is not natively available in Z3. The theorem I want to prove is the geometric sum formula as shown in Theorem 2.2.

Theorem 2.2 (Geometric Sum)**.** *Suppose $r \in \mathbb{R}$, $n \in \mathbb{N}$, $r > 0$ and $r \neq 1$. Then,*

$$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

I proved this theorem using three setups. The first setup uses raw ACL2 without help from any books (see code in Appendix A.1). The second setup uses ACL2's arithmetic book (see code in Appendix A.2). The third setup uses my combination of ACL2 and Z3 (see code in Appendix A.3). Table 2.1 summarizes the effort required for the three approaches. The proof requires induction and thus cannot be completed using Z3 alone.

Setup	LOC	# of theorems	runtime(s)	code time
raw Z3(can't)	-	-	-	-
raw ACL2(proved)	169	19	0.14	2 days
arithmetic-5(proved)	29	1	0.15	10 min
ACL2 & Z3(proved)	72	2	0.06	20 min

Table 2.1: Geometric sum equation proof comparison using different setups

Several observations can be made. First, raw ACL2 is a poor choice for this problem in nearly every sense. It requires one to implement every single lemma. This requires the most lines of code. It requires huge amount of human effort to complete. Of course, this is why users of ACL2 have developed and use the extensive library of "books" (collections of ACL2 theorems) that have been established to avoid this kind of low-level effort. As an example of the efficacy of ACL2's books, Theorem 2.2 is proven with no additional effort by the user when the standard book of arithmetic theorem is included. The combined approach although takes longer code compared to ACL2 with arithmetic book, but relieves the tedium when compared to raw ACL2.

Example: Intersection of 3 Polynomial Inequalities

Reasoning about the recurrence models for AMS circuits (see Eq 2.2) often involves systems of non-linear equalities and inequalities with moderate numbers of variables. To show how Z3 compliments ACL2, we'll consider the problem of showing the unsatisfiability of the conjunction of the three polynomial inequalities given below in Theorem 2.3.

Theorem 2.3 (Polynomial inequality)**.** *Suppose $x \in \mathbb{R}$ and $y \in \mathbb{R}$, then the conjunction of*

$$\begin{aligned} 1.125x^2 + y^2 &\leq 1 \\ x^2 - y^2 &\leq 1 \\ 3(x - 2.125)^2 - 3 &\leq y \end{aligned} \tag{2.5}$$

does not have a solution.

Figure 2.3 depicts this system of inequalities. The green one is the ellipse, the blue one is the hyperbola and the red one is the parabola. The small circles indicates which side of the polynomials the inequalities are referring to. Zooming in at the crossing part, Figure 2.4 clearly shows why these three polynomial inequalities have no solution. The experiment results show the relative power of ACL2 and Z3.

I performed four experiments with four setups. The first setup uses Z3 alone (see code in Appendix A.4). The second and third setups use raw ACL2 and ACL2 with its arithmetic book (see code in Appendix A.5). The fourth setup uses my ACL2 and Z3 combination (see code in Appendix A.6). Table 2.2 shows the results of these experiments.

Setup	**LOC**	**# of theorems**	**runtime(s)**	**code time**
raw Z3(proved)	27	1	0.0004	10 min
raw ACL2(failed)	40	-	-	10 min
arithmetic-5(failed)	41	-	-	10 min
ACL2 & Z3(proved)	59	1	0.02	10 min

Table 2.2: Polynomial inequality proof comparison using different setups

We make three observations. First, Z3 by itself is somewhat faster than

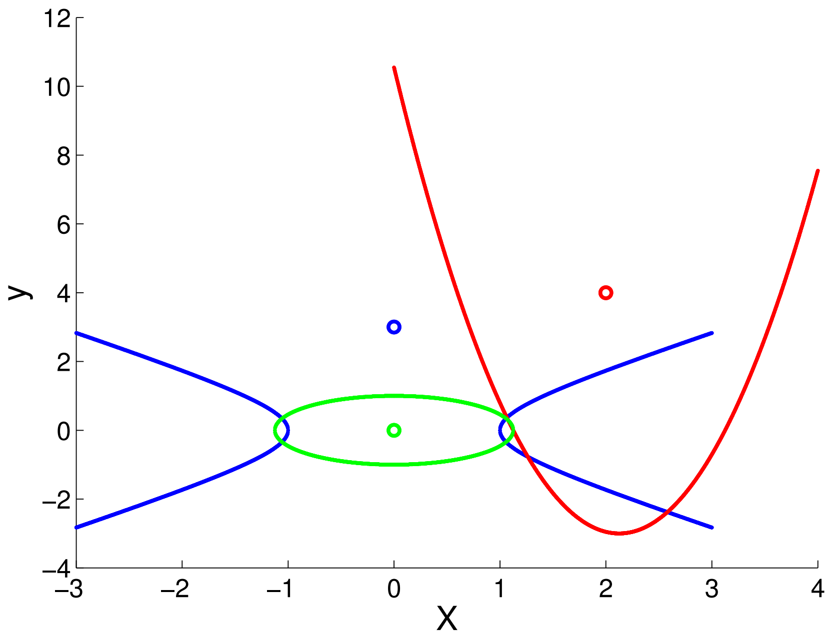


Figure 2.3: Three polynomials

when run within ACL2. This is because the current version of my code creates a new Python process for each clause discharged by Z3. I believe that the time for the proof with the ACL2 and Z3 combination is dominated by the time to create this Python process. Second, ACL2 failed to prove the theorem even with the arithmetic book. Of course, one could, in principle, guide ACL2 through a sequence of theorems to prove the main result, but one can easily imagine how much more time it would take to identify, state, and prove all of the necessary lemmas. This can be shown by examining the global convergence proof in Appendix C.2.1. Part of the proof is using purely ACL2 and the theorems being proved are more complex than the one shown here. Third, the combination does not require significantly more code or time by the user than using Z3 alone. By utilizing the SMT solver, our combination can readily prove the theorem without huge effort in faster speed.

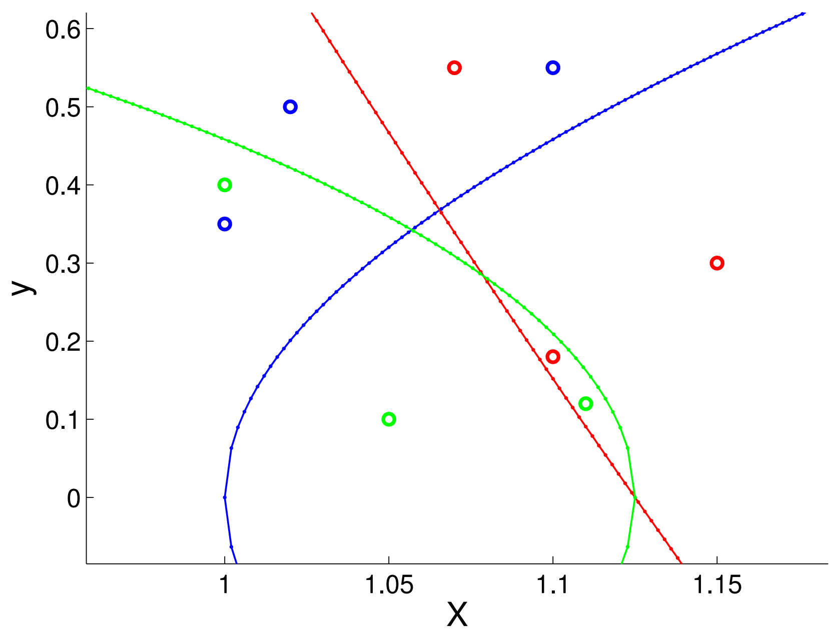


Figure 2.4: Zoom in the crossing part

2.3 PLLs and Verification

There have been several previously published reports of PLL verification using formal methods. The earliest verification that I know of was by Dhingra [38]. Dhingras design uses a fixed-frequency oscillator and adaptively chooses edges to approximate the edges of the lower frequency reference. I am not aware of any such PLLs in use for standard PLL applications such as clock generation, clock-data-recovery, and wireless communication.

PLLs and model checking Dong *et al.* [39] and Wang *et al.* [109] proposed using property checking for AMS verification, including PLLs. Shortly after the work by Dong *et al.*, Jesser and Hedrich [69] described a model-checking result for a simple analog PLL. Althoff *et al.* [8] presented the verification of a charge-pump PLL using an approach that they refer to as continuization. They use a purely linear model for the components of their PLL, and their focus is on the switching activities of the phase-frequency detector, in particular, uncertainties in switching delays. They use zonotopes [50] to conduct reachability analysis and their method works for ranges of parameters. In comparison, my work uses the non-linear recurrence model without any linearizations and reasons directly about this model.

PLLs and SMT More recently, Lin *et al.* [83, 84] developed an approach for verifying a digital PLL using SMT techniques. To the best of my knowledge, they are the first to claim formal verification of a digital PLL. They consider a purely linear, analog model and then reason about the discrepancies between this idealized model and a digital implementation. They use the iSAT [43] SMT solver to verify bounds on this discrepancy. They verify bounds on the lock time of a digitally intensive PLL assuming that most of the digital variables are initialized to fixed values, and that only the oscillator phase is unknown. My work shows initialization for a different PLL design over the complete state space.

PLLs and reachability Using the SpaceEx [44] reachability tool, Wei *et al.* [110] presented a verification of the same digital PLL as described in this thesis. That work made an over-approximation of the reachable space by over-approximating the recurrences of the digital PLL with linear, differential inclusions. As SpaceEx could not verify convergence property for the entire space in a single run, [110] broke the problem into a collection of lemmas that were composed manually. There work demonstrated the need for some kind of theorem proving tool to compose results. Furthermore, they could not show the limit cycles that my proof does; therefore their proof does not provide as tight of bounds on PLL jitter and other properties as can be obtained with my techniques.

Chapter 3

Combining SMT with Theorem Proving

Chapter 2 makes the observation that theorem provers and SMT solvers offer complementary capabilities for verifying AMS circuits. Theorem provers are good at managing structured proofs and SMT solvers are, on the other side, good at automatically solving large non-linear inequalities. Accordingly, I choose to manage the proof in a theorem prover and invoke a SMT solver as directed by the user to discharge clauses that can be expressed in the theories supported by the solver. The theorem prover takes the verification results from the SMT solver and stores the resulting theorem in the current theorem environment as one of the main results or as a lemma for future use.

Figure 3.1 shows an example of how one might use a SMT solver within ACL2 while using *The Method* as described in Chapter 2.2.4. In Figure 3.1, a green theorem means "proved", a yellow theorem means "currently being proven", a gray theorem means "pending for proof" and a red theorem means "proof failed". Suppose in the theorem prover, initially we have proven from Theorem 1 all the way until we reach Theorem smt_problem, which we believe forms a nice SMT problem. The strategy is to take the negation of the claim for Theorem smt_problem and give it to the SMT solver. The SMT solver automatically determines whether the negation is satisfiable or not. If the SMT solver shows the negation of the claim for Theorem smt_problem is UNSAT, this establishes the original theorem. Otherwise, ideally, if the SMT solver returns SAT and gives a satisfiable assignment, we know this is a counter-example to Theorem smt_problem

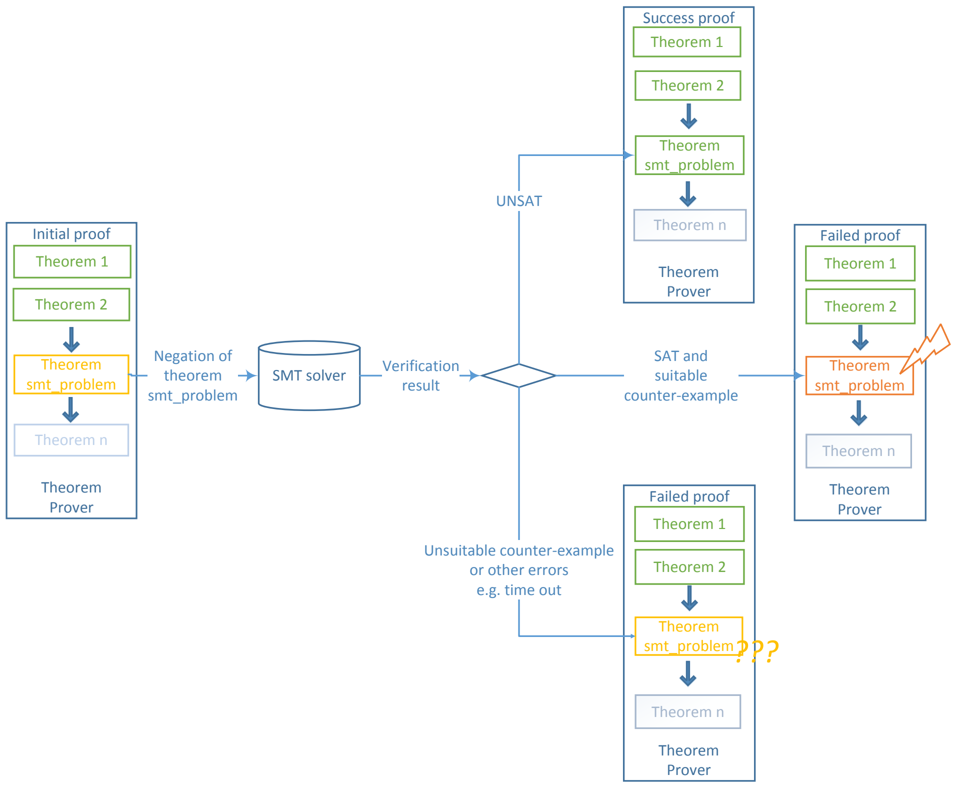


Figure 3.1: Basic framework for the combination.

and this disproves the theorem. If other errors, e.g. a time out happens, we are also given proof failure, but we can't decide the truth value of Theorem smt_problem.

This chapter describes **Smtlink**, my integration of the Z3 SMT solver into the ACL2 theorem prover. As described in Chapter 2, both ACL2 and Z3 have been successfully used for a large variety of research and industrial problems. Implementing the approach sketched above requires addressing numerous issues to link the logics of ACL2 and Z3 while preserving soundness. This chapter presents these issues and describes my solutions.

3.1 Clause Processor Architecture

ACL2 implements a computational logic for an applicative subset of Common Lisp [97]. This computational logic is a set of proof rules including rewriting, induction, and rules for basic Lisp operations such as `cons`, `car`, and `cdr`. These are applied automatically with guidance from the user in the form of prior theorems that are proven or in user provided hints. Typically, the user finds a sequence of simpler theorems that leads ACL2 to a proof of the main result.

Much of the work for the user can be relieved through ACL2's "clause processor" mechanism. A clause processor takes an ACL2 clause (i.e. proposition) as an argument and returns a list of clauses with the interpretation that the conjunction of the result clauses implies the original clause. In particular, if the result clause is empty, then the clause processor is asserting that the original clause is always true. ACL2 supports two types of user-defined clause processors: verified and trusted. A *verified* clause processor is written in the ACL2 subset of Common Lisp and proven correct by ACL2. A *trusted* clause processor does not require a correctness proof; instead, all theorems are tagged to identify the trusted processors that they may depend on. Logically, the tag adds the soundness of the trusted clause processor as a hypothesis to any theorem that depends on the clause processor. In other words, a theorem that depends on a trusted clause processor effectively says:

Program 3.1 trusted tag theorem

```
(defthm trusted-tag-theorem
  (implies (and (the-hypotheses-given-by-the-user)
                (the-clause-processor-is-sound))
           (the-conclusion-holds)))
```

3.1.1 The Top-level Architecture

I incorporate Z3 into ACL2 as a trusted clause processor as shown in Figure 3.2. I call the clause processor `Smtlink`. As directed by a user provided

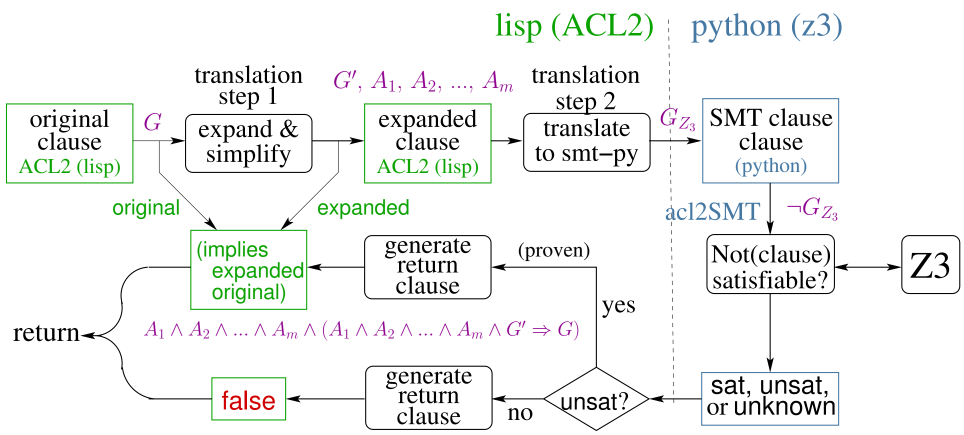


Figure 3.2: Top-level architecture of `Smtlink`

hint, ACL2 can invoke `Smtlink` to discharge a particular goal or subgoal of a proof. Let G denote this goal. As described in the subsequent sections, this formula is first transformed into an equivalent (or stronger) formula, G' and a list of auxiliary claims denoted by $A_1, A_2, ...A_m$. The second phase of translation produces a z3py (Python) representation of G', we'll call this G_{z3}. ACL2 starts a Python process to run a script to test the satisfiability of $\neg G_{z3}$. If Z3 establishes that $\neg G_{z3}$ is unsatisfiable, then G_{z3} and therefore G is a theorem. In this case, `Smtlink` returns the clause $A_1 \wedge A_2 \wedge ... \wedge A_m \wedge (A_1 \wedge A_2 \wedge ... \wedge A_m \wedge G' \Rightarrow G)$ to ACL2. By this mechanism, ACL2 verifies that the transformations performed by the translator were sound. If Z3 finds a satisfying assignment to $\neg G_{z3}$, it is returned as a counter-example. Counter-examples are shown to the user in the printout. More technical issues about how one can make use of the counter-examples are described in Section 3.5. If Z3 fails to determine the satisfiability of $\neg G_{z3}$, `Smtlink` reports that it was unable to make progress. Each of these steps is described in more detail in the remaining sections of chapter.

3.1.2 Ensuring Soundness in `Smtlink`

The soundness (and vulnerabilities) of this approach can be understood from the following logical sequent:

$$
\begin{array}{ll}
(\bigwedge_{i=1}^{m} A_i) & \text{; each } A_i \text{ verified by ACL2} \\
((\bigwedge_{i=1}^{m} A_i) \wedge G') \Rightarrow G & \text{; verified by ACL2} \\
G_{Z_3} \Rightarrow G' & \text{; we trust translation step 2} \\
G_{Z_3} & \text{; verified by Z3}^2 \\
\hline
G &
\end{array}
\tag{3.1}
$$

This is easily shown to be a tautology. Note that the first translation step has *no* impact on soundness; in other words, the sequent above is a tautology for *any* choice of G'. Of course, if the first step is faulty, then it is likely that either it will produce a G' that is too strong and Z3 will be unable to discharge it, or G' will be too weak, and ACL2 will be unable to discharge $((\bigwedge_{i=1}^{m} A_i \wedge G') \Rightarrow G$. Thus, a correct implementation of the first translation step is important for `Smtlink` to be *useful*, but it has no impact on soundness. Accordingly, I organized the code so that most of the complexity would be in the first translation step, and the second step just translates a small number of simple Lisp operations to their equivalent in `acl2SMT.py`, the Python module that I wrote to provide a generic interface between `Smtlink` and SMT solvers.

SMT solvers use heuristics for domain specific problems and it is possible that Z3 may return 'unknown' because of the complexity of the problem. In this case, the SMT solver's response does not help us determine the truth of the original theorem. Furthermore, semantic gaps can exist between the ACL2 formula and formulas that are within the theories supported by the SMT solver. These issues are described in detail in Section 3.2. The translator is written to ensure that the claim, G' that is verified by `Smtlink`

[2] For the purposes of the clause processor, determining the truth of G' also depends on correctly invoking the Python program and properly interpreting the string output by the program to report the outcome from Z3. All of this code is simple, straightforward, and largely based on code for other external clause processors (SAT solvers) that are already in use with ACL2.

Program 3.2 A SMT eligible theorem in ACL2

```
(defun foo (x y) (* x (+ 1 y)))
(defthm test
  (implies (and (and (rationalp x)
                     (integerp y)
                     (integerp z))
                (and (not (<= x 0))
                     (equal z (+ 3/2 4))
                     (or (> x y) (> x (+ y 40/3)))))
           (> (foo x (foo x z)) (foo x y))))
```

is at least as strong as the original claim, G. If G' is stronger than G, then the SMT solver may find a counter-example to G' that does not refute the original claim, G. The goal for `Smtlink` is to ensure soundness, but completeness is not possible, nor is completeness required for `Smtlink` to be useful in practice.

3.2 `Smtlink` Architecture

In ACL2, theorems are written in a comprehensive, applicative subset of Common Lisp. The `Smtlink` translator produces Python programs that use the `acl2SMT` API that I wrote. This API is specifically designed for Z3's Python interface, z3py, but should be suitable for use with other SMT solvers as well. To implement a full translator from ACL2 into Z3 is not possible due to the asymmetry between the two logics. Therefore, `Smtlink` only translates a subset of the ACL2 logic that is practical to express as a SMT problem. With our emphasis on AMS verification, I designed `Smtlink` with an emphasis on using SMT to reason about systems of linear and non-linear equalities and inequalities.

Program 3.2 shows a simplified example of an ACL2 theorem that is suitable for discharging with a SMT solver. Such a theorem consists of four parts: function definitions, type assertions, inequality constraints on the variables and the inequality property to prove. Program 3.3 shows a statement of the same theorem using z3py. In general, the translator needs to

Program 3.3 A SMT theorem in Z3

```
def foo(x,y):
    return x*(1+y)

x=Real("x")
y=Real("y")
z=Real("z")

hypothesis=And(Not(x <= 0), (z == (3/2 + 4)),
               Or((x > y), (x > (y + 40/3))))
conclusion=foo(x,foo(x,z)) > foo(x,y)
Prove(Implies(hypothesis, conclusion))
```

Program 3.4 SMT-eligible ACL2 theorem format

```
(defthm SMT-eligible-theorem
  (implies (and (and list-of-variable-type-assertions)
                (and other-hypothesis))
           conclusion))
```

extract the type assertions for variables, the hypotheses, and the conclusion from the clause given to the clause processor. Furthermore, it should be able to identify and expand calls to user-defined functions.

For simplicity, I require that the ACL2 clause to be proven using the SMT solver has the structure shown in Program 3.4: For example, Program 3.2 has this structure. Each type assertion is of the form `(`*`type-recognizer variable`*`)`, where *type-recognizer* is one of the ACL2 recognizer functions, `booleanp`, `integerp`, or `rationalp`, and *variable* is a symbolic variable appearing in the clause. All variables must be declared in this fashion. The terms *other-hypotheses* and *conclusion* can be any predicates supported by the translator; in particular, these terms are quantifier free. Often, *other-hypotheses* is a conjunction of equality and/or inequality constraints on the variables, and *conclusion* is the equality or inequality to be proven. Requiring this structure simplifies the implementation of the translator and has not been a serious restriction for the examples we have tried (see Chapter 4) as ACL2 theorems about systems of real-valued inequalities are typically written in a form very similar to the one we require.

Several technical issues arose when I implemented the transformation and translation.

- Typed vs. untyped. ACL2 is untyped and Z3 is typed. `Smtlink` requires the user to provide a type assertion for every free variable occurring in the theorem. See Section 3.2.1

- Rational vs. Reals. ACL2 only supports rationals and Z3 supports reals. `Smtlink` strengthens the clause to be proven by replacing rational assertions with real assertions. See Section 3.2.1

- Richer logic in ACL2. ACL2 supports a much richer logic than is supported by Z3. `Smtlink` supports clauses that are boolean combinations of rational function equalities and inequalities. See Section 3.2.2

- Function expansion. For user-defined functions and recursive functions, `Smtlink` expands them into a set of primitive functions. See Section 3.2.2

- Non-polynomial expressions. Z3 only supports theories for polynomial (and rational function) inequalities. `Smtlink` provides a mechanism that allows the user to replace non-polynomial expressions with variables. See Section 3.2.3

- Adding hypotheses. `Smtlink` allows the user to specify additional hypotheses to be added to G_{Z_3} and then verified by ACL2. Typically, these are instances of previously proven theorems, or constraints on variables that the user introduced as replacements for non-polynomial expressions. See Section 3.2.3

- Forwarding hints. Clauses returned to ACL2 are supposed to be "easy" for ACL2 to prove. In fact, the "automatic" aspect of ACL2 requires ACL2 to discharge these clauses without further interaction from the user; otherwise, the proof of the theorem fails. Occasionally, ACL2

Program 3.5 An example showing ACL2's type recognizer

```
(defthm not-really-a-theorem
  (iff (equal x y) (zerop (- x y))) )
```

fails to prove a user added hypothesis. In this case, the user can provide hints. For example, using *The Method* (see Chapter 2.2.4), the user can prove a suitable lemma, and then give a hint that tells ACL2 how to instantiate this lemma to discharge the clause returned by `Smtlink`. See Section 3.2.3

Sections 3.2.1 through 3.2.3 present these challenges and my solutions to each.

3.2.1 Type Assertion

A fundamental difference between the ACL2 and Z3 is that ACL2 uses an untyped logic whereas the logic of Z3 is typed. For example, consider the putative ACL2 theorem 3.5. ACL2 is untyped and requires all functions to be total. Thus, `(- x y)` is defined for all values for `x` and `y`, including non-numeric values. For example, `x` could be a Lisp atom and `y` could be a list. What is `(- 'dog (list "hello", 2, 'world))`? As implemented in ACL2, arithmetic operators treat all non-numeric arguments as if they were zero. Thus,

Expression	Value
`(- 'dog (list "hello", 2, 'world))`	`0`
`(zerop (- 'dog (list "hello", 2, 'world)))`	`t`
`(equal 'dog (list "hello", 2, 'world))`	`nil`
`(iff (equal 'dog (list "hello", 2, 'world))` `(zerop (- 'dog (list "hello", 2, 'world))))`	`nil`

On the other hand, Z3 uses a typed logic, and each variable must have an associated sort. If we treat `x` and `y` as real-valued variables, the z3py equivalent to `not-really-a-theorem` is

Program 3.6 Rewrite the theorem

```
(defthm this-is-a-theorem
  (implies (and (rationalp x) (rationalp y))
           (iff (equal x y) (zerop (- x y))) ))
```

```
>>> x, y = Reals(['x', 'y'])
>>> prove((x == y) == ((x - y) == 0))
    proved
```

In other words, `not-really-a-theorem` as expressed in the untyped logic of ACL2 is not a theorem, but the "best" approximation we can make in the typed logic of Z3 is a theorem.

`Smtlink` employs two methods to address these issues: *type assertions* and *type correspondence*. With *type assertions*, the user indicates the intended type of each free variable in an ACL2 claim. If any variables have values outside of the asserted domains, `Smtlink` states the claim trivially holds. With *type correspondence*, I wrote `Smtlink` to ensure that the Z3 sorts for variables correspond to the types asserted in the ACL2 claim. For booleans and integers, this correspondence is immediate. On the other hand, we represent ACL2 rational numbers using Z3's sort for reals. The remainder of this section describes these design decisions in more detail and presents our justifications for their soundness.

Typed vs. Untyped

Theorems in ACL2 are written as terms in Common Lisp, an untyped language. Variables and expressions in Common Lisp, and hence in ACL2, do not have types. On the other hand, Common Lisp provides type recognizers for values including `integerp` and `rationalp`. We can rewrite the previous using type assertions as Program 3.6. ACL2 proves this theorem automatically. Note that with our previous example, if `x` is the atom `'dog` and `y` is the list `("hello", 2, 'world))`, then `(rationalp x)` and `(rationalp y)` are both false and the theorem holds trivially because the antecedent of the implication is false. In the case that `x` and `y` both have value that are rational numbers, then the theorem states the basic arithmetic result, which was (presumably) the user's intention.

Program 3.4 shows the structure that `Smtlink` requires. In particular, if a goal does not preserve the syntactic format illustrated here, `Smtlink` will produce an error and fail to prove the theorem. `Smtlink` maintains soundness when translating from the untyped logic of ACL2 to the typed logic of SMT solvers by enforcing the syntactic structure described above and by using SMT sorts that can represent all values recognized by the corresponding ACL2 type recognizers. A bit more formally, let U be the set of all values in the ACL2 universe. Then, a theorem like the one depicted in Program 3.4 is equivalent to the logical formula:

$$\forall x_1, x_2, ..., x_m \in U.\ \left(\bigwedge_{i=1}^{m} T_i(x_i) \wedge \bigwedge_{j=1}^{n} h_j(x)\right) \Rightarrow C(x) \tag{3.2}$$

where T_i is the type of the i^{th} variable; h_j is the j^{th} "other" hypothesis; C is the conclusion of the theorem; and the theorem has m free variables and n "other" hypotheses. The corresponding formula to be discharged by the SMT solver is:

$$\forall x_1 \in S_1, x_2 \in S_2, ..., x_m \in S_m.\ \left(\bigwedge_{j=1}^{n} \tilde{h}_j(x)\right) \Rightarrow \tilde{C}(x) \tag{3.3}$$

where S_1, S_2, ..., S_m are the SMT sorts corresponding to the type recognizers T_1, T_2, ..., T_m; $\tilde{h}_j(x)$ is the translation of $h(x)$; and $\tilde{C}(x)$ is the translation of $C(x)$. For soundness, we want to show that if the formula from Equation 3.3 holds, then the formula from Equation 3.2 must hold as well. This correspondence is ensured if:

- $\forall x_i \in U.\ T_i(x_i) \Rightarrow x_i \in S_i$
- $\forall x_1, x_2, ..., x_m \in U.\ (\bigwedge_{i=1}^{m} T_i(x_i)) \Rightarrow (h_j(x) \Rightarrow \tilde{h}_j(x))$
- $\forall x_1, x_2, ..., x_m \in U.\ (\bigwedge_{i=1}^{m} T_i(x_i)) \Rightarrow (\tilde{C}(x) \Rightarrow C(x))$

The first condition requires that every value that satisfies an ACL2 type recognizer must be a value of the corresponding SMT sort. As currently implemented, `Smtlink` supports booleans, integers, and rationals/reals as shown in Table 3.1. Z3's booleans and integers match the same, standard, mathematical definitions as those used in ACL2. On the other hand, ACL2 uses rational numbers where Z3 uses reals – this difference is discussed in more detail below. The last two conditions given above require that `Smtlink` must preserve the meaning of terms. More specifically, the hypotheses as translated by `Smtlink` must be no stronger than those of the ACL2 theorem, and the conclusion must be at least as strong.

As shown in Figure 3.2, `Smtlink` performs translation in two phases, where the first phase is verified by ACL2 and the second is trusted. Type assertion is handled in the second phase. This means that we are trusting `Smtlink` to correctly recognize the syntactic structure depicted in Program 3.4 and to declare SMT variables of the correct sorts corresponding to the ACL2 type recognizers. In both cases, the translation is simple, and the code is easily inspected.

In the current implementation, `Smtlink` does not check to make sure that all free variables have type assertions nor does it check if there are multiple type-assertions for the same variable. If a user omits a type assertion, then the corresponding variable will be undeclared, and this will cause the Python code to report an error. If there are duplicated type assertions for the same variable, ACL2 will take the conjunction of the assertions as hypothesis and Z3 will use the last declaration. Thus, the Z3 hypothesis will be weaker than the ACL2 ones. It will be beneficial to check duplicated variable declaration in future work.

Rationals vs. Reals

Another asymmetry comes from the fact that, due to implementation issues, every number in ACL2 must be either an integer, a rational number, or an integer or rational complex number. In contrast, Z3 provides a sorts for integers and real numbers, but no sort for rational numbers. While we could

Program 3.7 An example showing rational vs. reals problem in ACL2

```
(defthm rational-vs-reals
  (implies (and (and (rationalp x))
                (and))
           (not (equal (* x x) 2))))
```

Program 3.8 An example showing rational vs. reals problem in Z3

```
x = Real("x")
prove(Not(x*x == 2))
```

introduce a user-defined type for rational numbers (i.e. a pair of integers) and define arithmetic and comparison operations on such numbers, doing so would preclude using Z3's decision procedures for non-linear arithmetic, and that is our primary motivation for integrating a SMT solver into ACL2. Z3 uses Gröbner bases combined with rewriting heuristics to reason about systems of polynomial equalities and inequalities. These procedures apply to real-valued variables. Some care is needed to handle this mismatch between real-numbers and rationals. As an example, consider the theorem shown in Program 3.7. This theorem can be proven, albeit with some manual effort, using ACL2 [45]. In English, the theorem states "2 does not have a rational square root". `Smtlink` translates Program 3.7 to the Python code that is roughly equivalent to (but much more verbose than) that shown in Program 3.8.

Because Z3's non-linear arithmetic procedures support real numbers, Z3 finds the counter-example $x = \sqrt{2}$, but this is not a valid counter-example to the original theorem. More generally, `Smtlink` may strengthen a theorem. In this case, the strengthening is because while `(rationalp x)` implies `x` $\in \mathbb{R}$, the converse does not hold. When `Smtlink` discharges a strengthened theorem, the original theorem must hold as well. As currently implemented, `Smtlink` does not provide counter-example generation, and if it refutes a translated theorem, we can make no conclusions about the original version. `Smtlink` prints the counter-example to the ACL2 log for the user to examine, but ACL2 makes no further use of such results. Presumably, one could check to see if a counter-example generated by Z3 only used booleans, integers and rational numbers. If so, then this will be a valid counter-example for the original theorem in ACL2 and could be used as an existential witness. More discussion can be found in Section 3.5.

ACL2	**Z3**
integerp	Int
rationalp	Real
booleanp	Bool

Table 3.1: Type assertion translation

3.2.2 Supported Logic

`Smtlink` minimizes the portion of code that needs to be trusted in translation step 2 (as shown in Figure 3.2). It achieves such goal by defining a small set of primitive functions to be translated in translation step 2. All other functions (including user-defined functions and ACL2's other built-in functions) should be expanded and simplified into the small set of primitive functions. The expansion and simplification happen in translation step 1, which is ensured soundness by `Smtlink`'s software architecture 3.2.

For our intended application, we focus on supporting arithmetic, comparison, and boolean operations from ACL2 and translating these to their SMT equivalents. As shown in Table 3.2,. most of these operators are Lisp macros in ACL2, and our translator sees the macro-expanded form. Accordingly, our translator supports clauses consisting of the Lisp functions appearing in the right column of Table 3.2. Table 3.3 shows how each such Lisp function has a corresponding method in the `acl2SMT` module. Chapter 3.3 discusses the Z3 interface class of `Smtlink`.

[2] Note that macro expansions shown in the table are not exact definitions but example instances. E.g. In ACL2, `+`, `-`, `and` and `or` are actually macro-expanded into a recursive function that takes an uncertain number of inputs.

Before macro expansion	**After macro expansion**
`(+ x y z)`	`(binary-+ x (binary-+ y z))`
`(- x y)`	`(binary-+ x (unary-- y))`
`(* x y z)`	`(binary-* x (binary-* y z))`
`(/ x y)`	`(binary-* x (unary-/ y))`
`(equal x y)`	`(equal x y)`
`(> x y)`	`(> x y)`
`(>= x y)`	`(>= x y)`
`(< x y)`	`(< x y)`
`(<= x y)`	`(<= x y)`
`(and x y z)`	`(if x (if y z nil) nil)`
`(or x y z)`	`(if x t (if y t z))`
`(not x)`	`(not x)`
`(nth listx)`	`(nth listx)`

Table 3.2: ACL2's macro expansions

Function Expansion

For user defined functions or other ACL2 built-in functions, `Smtlink` expands them into the set of primitive functions. This approach has several benefits. First, we won't need to worry about translation of a function definition in ACL2 to Z3, which can be tedious. Second, for recursive functions, it's not even possible to directly translate them into Z3, because recursive definitions can not be symbolically expanded in Z3. As shown in Chapter 3.1.1, clause G' is the result clause after this expansion. Thus the expansion will be ensured correctness when the clause $A_1 \wedge A_2 \wedge ... \wedge A_m \wedge G' \Rightarrow G$ gets returned back for ACL2 to prove.

To see how the function expansion works, for example, given the function definition of `fun-example`:

```
(defun fun-example (a b c) (+ a b c))
```

Suppose in some theorem, we encounter function call `(foo (+ x y) x (/ z x))`. The first phase of translation expands this to:

```
((lambda (VAR1 VAR2 VAR3) (+ VAR1 VAR2 VAR3)) (+ x y) x (/ z x))
```

ACL2 primitives	SMT interface
binary-+	acl2SMT.plus
unary–	acl2SMT.negate
binary-*	acl2SMT.times
unary-/	acl2SMT.reciprocal
equal	acl2SMT.equal
$>$	acl2SMT.gt
$<$	acl2SMT.lt
$\geq$	acl2SMT.ge
$\leq$	acl2SMT.le
if	acl2SMT.ifx
not	acl2SMT.notx
nth	acl2SMT.nth
t	acl2SMT.True
nil	acl2SMT.False

Table 3.3: Z3 interface for each ACL2 primitives

`Smtlink` requires the user to provide a list of functions that should be expanded. For each such function, the user also specifies the maximum depth of the expansion and the return type. The function expander traverses the s-expression and expands along each path for a fixed number of levels for each function provided by the user. In a world without recursive functions, this expansion will be performed only once for any function along a specific path.

With recursive functions, functions along each path will be expanded until user specified levels are reached. Then, `Smtlink` replaces the remaining function calls with newly introduced variables and adds type assertions on the new variables. To ensure correctness, `Smtlink` returns the corresponding type specification theorems back to ACL2 as auxiliary theorems (as mentioned in Chapter 3.1.1). Using this approach, the expansion strengthens the original clause; thus, `Smtlink` asks the SMT solver to prove a stronger theorem. However, this approach can produce an over-strengthening of the original theorem that causes Z3 to fail proving the translated theorem. To solve this issue, the user can provide additional hints to weaken the translated theorem. This is discussed in Section 3.2.3.

3.2.3 Advanced Issues

The previous sections described the top-level structure and basic constructions of `Smtlink`. This section describes other features that makes the integration more flexible and extensible.

User provided substitutions make it possible to substitute part of the clause formula with a new variable. Furthermore, the user can provide hypothesis predicates that constrain those variables or convey other information to the SMT solver that may be "obvious" within ACL2. These hypotheses can make a SMT based proof possible and/or more efficient. `Smtlink` returns the user provided predicates as clauses for ACL2 to prove. Therefore, ACL2 automatically checks if the hypothesis on the substitution is valid. User provided hints help discharge some of the hard (auxiliary) theorems returned back to ACL2. The sections below describe each feature in detail.

User Provided Substitution

To see why we need user provided substitutions, considering the following claim:

$\forall a, b, \gamma \in R, \forall m, n \in Z, 0 < \gamma < 1, 0 < m < n \Rightarrow \gamma^m(a^2 + b^2) \geq \gamma^n(2ab)$.
Program 3.9 is the ACL2 code for this theorem. This theorem seems like a good candidate for proving using SMT methods. Given the function expansion mechanism of `Smtlink`, the exponential functions in this theorem will be expanded to a given level, and the last function call will be replaced with a typed variable. However, this is an over-strengthening of the original theorem and Z3 fails to prove such theorem. Is this theorem impossible to prove using SMT techniques? Taking a closer look, one can see there is a simple reason why the theorem holds. A manual proof would observe that because $0 < gamma < 1$, $0 < m < n$ and $gamma^m > gamma^n > 0$. Furthermore, for any a, $b \in \mathbb{R}$, $a^2 + b^2 >= 2ab$, and $a^2 + b^2 >= 0$. The claim follows directly from these inequalities. All of these are within Z3s theory of non-linear arithmetic except for deduction that $gamma^m > gamma^n > 0$ this step requires a (trivial) proof by induction. Our strategy is to provide

Program 3.9 Why need user provided substitution

```
(defthm substitution
  (implies (and (and (rationalp a)
                     (rationalp b)
                     (rationalp gamma)
                     (integerp m)
                     (integerp n))
                (and (> gamma 0)
                     (< gamma 1)
                     (> m 0)
                     (< m n)))
           (>= (* (expt gamma m)
                  (+ (* a a) (* b b)))
               (* (expt gamma n)
                  (* 2 a b)))))
```

$gamma^m > gamma^n$ and $gamma^n > 0$ to the SMT solver as hints, and let ACL2 discharge those hints using its inductive reasoning capabilities.

I implement this strategy using the mechanisms for substitutions and adding hypotheses described above. User defined substitutions direct `Smtlink` to replace `(expt gamma m)` and `(expt gamma n)` with two newly introduced variables `expt-gamma-m` and `expt-gamma-n`. Then, user provided hypotheses about those two variables are added: `(< expt-gamma-n expt-gamma-m)`, `(> expt-gamma-m 0)` and `(> expt-gamma-n 0)`. `Smtlink` sends the resulting clause G' to Z3. Z3 has no problem discharging this theorem. For each user provided hypothesis, `Smtlink` produces an auxiliary theorem and sends it back to ACL2 to discharge.

This mechanism greatly broadens the set of SMT problems `Smtlink` can handle. Especially when there exists limitations on Z3 that ACL2 can handle, or when we want to build upon known theorems about sub-formulas of the original clause. User provided hypotheses can make it easy to prove theorems that would involve long, tedious derivations if done entirely within ACL2 and that would seem unsuitable for SMT alone given the limitations of the supported theories. Often, a small amount of human reasoning conveyed as simple hints can enable a large degree of proof automation.

User Provided Hints

`Smtlink` allows several supposedly "easy" clauses to be sent back to ACL2. They are in the clause $A_1 \wedge A_2 \wedge ... \wedge A_m \wedge (A_1 \wedge A_2 \wedge ... \wedge A_m \wedge G' \Rightarrow G)$. However, ACL2 may be unable to discharge some of these clauses automatically. Thus hints are needed from human user.

ACL2 has this feature called hints to help guide a proof. Basically, a hint, which is a theorem already proved in the system, is an antecedent added to the intended proof. Consider the case where we want to prove theorem T, and we recognize that T is a simple arithmetic transformation of an existing theorem, H. Adding H as a hint to T is equivalent to constructing a new proof in the form $H \rightarrow T$. Since the user has already proven H somewhere in the ACL2 world, ACL2 knows the theorem T to be true. ACL2 has the hint feature for common theorem statements, but doesn't have this feature for discharging clauses returned by the clause processor. I added this feature to my construction.

3.3 The Low-level Interface

The previous section described the translator part of the clause processor which is composed of clause transformation & simplification and a Lisp-to-Python translator. See `Smtlink` architecture 3.2. This section presents two other parts in the low-level interface: the Z3 interface, and the result interpreter.

3.3.1 Z3 Interface

My current implementation of `Smtlink` uses the Z3 SMT solver. However, the code is written in a way that should make using other SMT solvers straightforward. In particular, all methods of the underlying SMT solver are invoked through methods of an object called `acl2SMT`. For example, `acl2SMT.plus` provides the addition operator; `acl2SMT.True` provides the boolean constant for `True`; etc. I also wrote a module called `acl2_Z3` that provides a class called `to_smt` with a no-arg constructor that returns an object with the methods described above. In this case, this object uses Z3s z3py API to implement these methods.

This mechanism has one significant benefit. The `acl2SMT` interface provides a very flexible interaction between ACL2 and other SMT solvers as shown in figure 3.3. Imagine we want to use another SMT solver, say Yices [40]. The only thing needed to be done is to develop a Yices interface for the same set of primitive functions. It is likely that the functions will be very similar to those for Z3. In a word, `Smtlink` should be easily extended to connect to other SMT solvers.

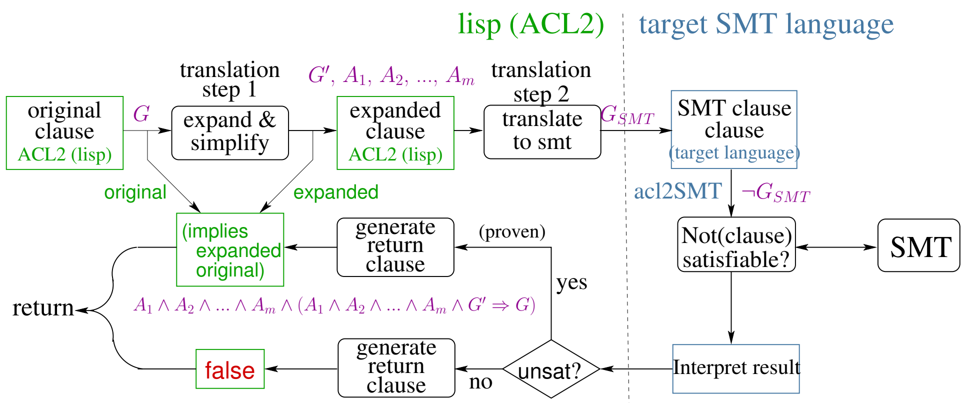


Figure 3.3: Clause processor framework with another SMT.

3.3.2 Interpret the Result

The goal behind interpreting the returned result from SMT solver is to ensure soundness. As shown in figure 3.2, there are three possible outcomes from a SMT solver. When it reports UNSAT, we know the original clause is true. When it reports SAT and provides the counter-example, because `Smtlink` strengthens the theorem when doing clause transformation and translation, we don't know if the counter-example is valid or not. The third case is when the SMT solver reports timeout or other exceptions. We don't know whether the theorem is true or not as well.

To see in detail how this works, ACL2 provides a function called *tshell-call* to call external procedures. This function can only be used when a program is properly tagged as "trusted". This function takes the shell command and returns the output from the command through one of the returned values which is a list of strings. `Smtlink` interprets the result as in one of the three cases discussed above. For example, if Z3 output equals "proved", then one knows the negation is UNSAT and the original theorem is proved. Otherwise, `Smtlink` simply prints the output and lets the user decide what kind of error it is.

3.4 Conclusion: What's Trusted?

This section examines which parts of the code need trust from the user. In other words, these are the assumptions I've made in `Smtlink`.

- First, I assume that the clause processor can correctly recognize the proposed theorem statement structure. In particular, it looks for theorem statement as in Program 3.4. This is a pattern matching that can be easily done in LISP.

- Second, I assume the method of weakening type hypotheses can strengthen the theorem. In particular, I want Z3 sorts to be supersets (or equal to) their ACL2 counterparts: ACL2 `Booleanp` gets translated to Z3 `Bool`, ACL2 `Integerp` gets tranlsated to Z3 `Int` and ACL2 `Rationalp` gets translated to Z3 `Real`, which is a superset of ACL2 `Rationalp`. This is illustrated in Table 3.1.

- Third, I assume the Z3 operators produced by `Smtlink` translator match the semantics of their ACL2 counterparts.

- Fourth, I assume that the code that writes the string generated by the translator to a file, invokes Python, and interprets the result will work correctly. Note that these operations are very simple and straightforward. The high-complexity code only occurs for function expansion, user specified substitutions, and user added hypotheses. However, none of this code requires trust, because this is done in the first step of translation, and the result of that translation is verified by ACL2.

3.5 Future Work

For several design decisions I've made, more general solutions are possible. Due to thesis time limitation, I didn't explore all of them. This section discusses what could be done differently and could potentially give better results.

Guards instead of user-provided types: ACL2 provides a mechanism for restricting inputs and outputs of a function to be in a particular domain. This mechanism is called a "guard". ACL2 users are encouraged to add guards to their modeling functions so that the functions are more well-formed. Given that users of ACL2 might follow this mechanism, type assertions can be retrieved from guards on each function, instead of being retrieved from user provided type assertions. `Smtlink` can provide both methods and let the user decide which mechanism he/she wants to use.

Returning counter-examples: `Smtlink` returns the potential counter-examples to ACL2 and prints them out for the user to check. In principle, `Smtlink` could check to see if the counter-examples from Z3 are meaningful in ACL2 (i.e., all numbers have integer or rational values). Then, `Smtlink` could try that assignment with the original goal. If it satisfies the original goal, then `Smtlink` can report a valid counter-example. If it doesn't, `Smtlink` has an indeterminate result. In principle, one could use this counter-example to refine the definition of G', and try again.

ACL2(r): There is a version of ACL2 called ACL2(r) [46]; the "r" stands for reals. ACL2(r) has support for real number reasoning. When I began developing `Smtlink`, ACL2(r) did not support the full collection of "books" of theorems that the mainstream ACL2 theorem prover did. There is an ongoing effort to unify the two versions that is nearly complete. I have successfully used `Smtlink` with both ACL2 and ACL2(r) for the convergence proof in the next chapter. Note that when ACL2(r) is used, then there is no semantic gap in the number representation between the theorem prover and the SMT solver.

Function expansion with better automation: `Smtlink` can provide better automation to function expansion. `Smtlink` could use a default mode in which functions would always be expanded one level. It could maintain a "do-not-expand" list of functions that should not be expanded. Features can be added to allow a richer set of assertions about the return result for recursive functions. Such assertions could be automatically obtained from guard expressions for the function. Right now, the user can only assert return types, but the user might know more sophisticated properties (e.g. an inductive property) about a recursive function. Adding these features should make `Smtlink` easier to use and enable writing more succinct proofs.

3.6 Summary

This chapter described the implementation of `Smtlink`, my interface between ACL2 and SMT solvers, in particular Z3. A key principle I stick to is that all transformations and translations are only strengthening the theorem; thus soundness is ensured.

`Smtlink` consists of three parts: the translator, the low-level interface and the SMT solver. The translation is performed in two steps. The first step takes an ACL2 theorem as input and transforms and simplifies the theorem into a set of auxiliary theorems and a new goal. This new goal uses only a very small subset of the Lisp functions provided in ACL2. The second step performs a straight forward translation from LISP to z3py on the new goal. The architecture is trustworthy in practice because most complexity falls in clause transformation and simplification code. The result of clause transformation and simplification is returned for ACL2 to check correctness. In principle, one could use proof-reconstruction [60] and/or certificates [19] in which case the user would only need to trust ACL2 itself, but that would be a separate thesis topic. My focus has been on developing a useful tool and demonstrating it on a real example.

The low-level interface makes it possible that one can extend `Smtlink` with different SMT solvers. This chapter also discusses a list of interesting issues that arise in `Smtlink` and proposes my solutions. From the discussion of what is trusted, one can see that `Smtlink` is reliable because it only relies on a limited set of things. Future work could include several improvement directions including better type inference, better counter-examples, using ACL2(r) to get better support for reals and a better function expansion mechanism.

Chapter 4

Verifying Global Convergence of a Digital PLL

This chapter demonstrates the value of `Smtlink` for AMS design by using it to verify the global convergence of a digital phase-locked loop (digital PLL). This experiment shows that one can employ an analytical approach for AMS verification, and how `Smtlink` supports this approach well. The analytical approach allows us to verify properties that cannot be shown by typical reachability methods: in particular it can be shown that a convergence is guaranteed with model parameters in ranges, rather than with specific values.

In this chapter, Section 4.1 introduces phase-locked loops describing both their operation and their applications. The particular digital PLL that I verify is described as well. Section 4.2 develops a mathematical model for the digital PLL that is amenable to formal reasoning, and Section 4.3 presents the proof itself. I note that this proof shows the main result needed to establish convergence, but there are still some details left that would be needed for a complete verification. Section 4.4 summarizes what has been proved in this convergence proof and discusses possible future work.

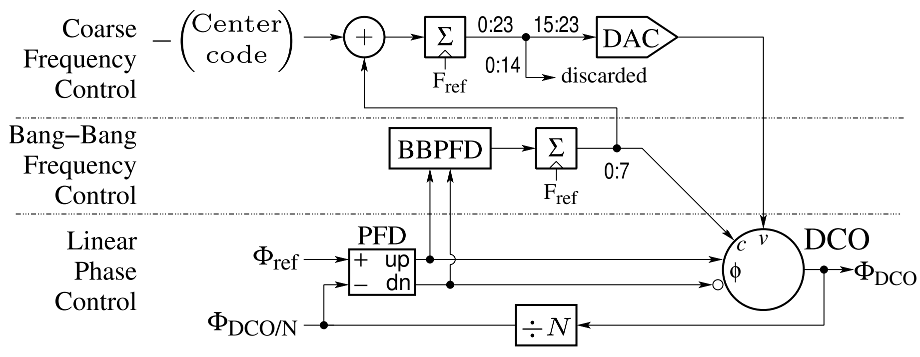
4.1 The Digital PLL

A PLL is a feedback control system that generates an output signal with the same frequency as the input or with a frequency that is some multiple of the input frequency. A PLL also requires that the phase of output signal should match that of the input. To control both the frequency and the phase of the oscillator, a PLL is a second order control system. PLLs are ubiquitous in a wide range of electronic devices. PLLs are used in computers for clock generation and to ensure proper timing of high bandwidth interfaces to DRAM, graphics and network interfaces, etc. For wireless devices such as mobile phones, PLLs are used to generate, modulate, and decode radio signals. These are just a few examples of how PLLs are used.

Traditionally, PLLs have been designed as purely or primarily analog systems. As described in Chapter 2, analog modules are being largely replaced by digital counterparts due to the difficulties of analog design in state-of-the-art fabrication processes and the extra configurability offered by digital designs. These observations apply to PLLs as well, leading to the dominance of “all-digital” or “digitally intensive” PLL design today. Digital blocks in the circuitry change the behavior of a circuit from continuous to partially discrete. The designer’s intuition is that given the original analog circuit is strongly converging, proper discretization shouldn’t drive it too far from converging again. However, discretization can introduce unintended modes of operation. Furthermore, if designers could be confident that their designs did behave as intended, then more aggressive techniques could be used to achieve higher performance, lower power consumption, smaller area, etc. Due to limitations with today’s AMS validation tools, we need formal verification to make sure a PLL is functioning correctly. Section 2.3 gives a discussion on related work of PLL verification research.

Figure 4.1 shows the digital phase-locked-loop (PLL) verified in this thesis; it is a simplified version of the design presented in [33]. The purpose of this PLL is to adjust the digitally-controlled oscillator (DCO) so that its output, Φ_{DCO} has a frequency that is N times that of the reference input, Φ_{ref} and so that their phases match (i.e. each rising edge of Φ_{ref} coincides with a rising edge of Φ_{DCO}). The three control-paths shown in the figure make this a third-order digital control system. By design, the lower two paths dominate the dynamics making the system effectively second-order.

The DCO has three control inputs: ϕ, c, and v. The ϕ input is used by a proportional control path: if Φ_{ref} leads $\Phi_{DCO/N}$ then the PFD will assert `up`, and the DCO will run faster for a time interval corresponding to the phase difference. Conversely, if Φ_{ref} lags $\Phi_{DCO/N}$, the `dn` signal will be



Φ_{ref} is the reference signal whose frequency is denoted by f_{ref}.

Φ_{DCO} is the output of the digitally controlled oscillator whose frequency is denoted by f_{dco}.

Labels of the form lo:hi denote bits lo through hi (inclusive) of a binary value.

Figure 4.1: A Digital Phase-Locked Loop

asserted, and the DCO will run slower for a time interval corresponding to the phase difference. If the frequencies of Φ_{ref} and $\Phi_{DCO/N}$ are not closely matched, then the PFD simply outputs `up` (resp. `dn`) if the frequency of $\Phi_{DCO/N}$ is lower (resp. higher) than that of Φ_{ref}.

The c input of the DCO is used by the integral control path. The DCO in [33] is a ring-oscillator, and the c input controls switched capacitor loads on the oscillator – increasing the capacitive load decreases the oscillator frequency. The bang-bang phase-frequency detector (BBPFD) controls whether this capacitance is increased one step or decreased one step for each cycle of Φ_{ref}. The c input provides a fast tracking loop.

The v input of the DCO is used to re-center c to restore tracking range. This input sets the operating voltage of the oscillator – the oscillator frequency increases with increasing v. The accumulator for this path is driven by the difference between c and its target value c_{center}.

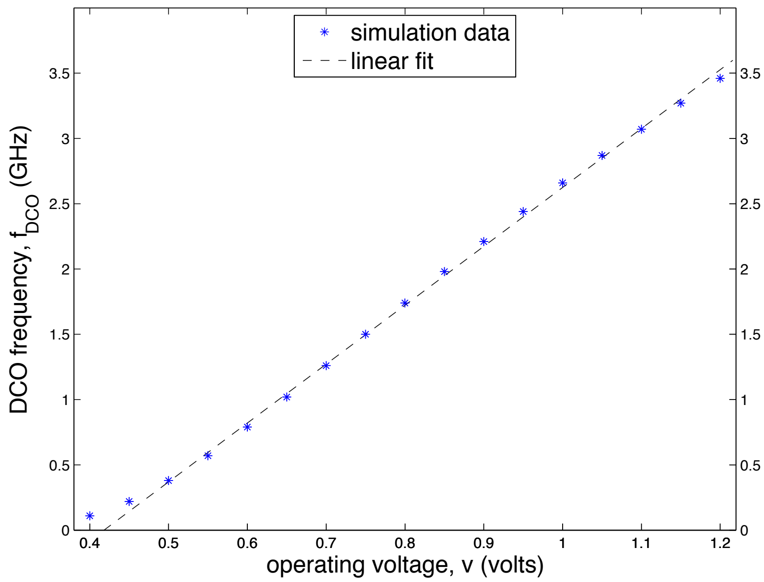
As a control system, the PLL converges to a switching surface where c and ϕ fluctuate near their ideal values. As presented in [33] these limit-cycle variations are designed to be slightly smaller than the unavoidable thermal and shot-noise of the oscillator. Furthermore, the time constants of the three control loops are widely separated. This facilitates intuitive reasoning about the system one loop at a time – it also introduces stiffness into the dynamics that must be considered by any simulation or reachability analysis. These characteristics of convergence to a switching surface and stiffness from multiple control loops with widely separated tracking rates appear to be common in digitally controlled physical systems. This motivates using the digital PLL as a verification example and challenge.

4.2 Modeling the Digital PLL

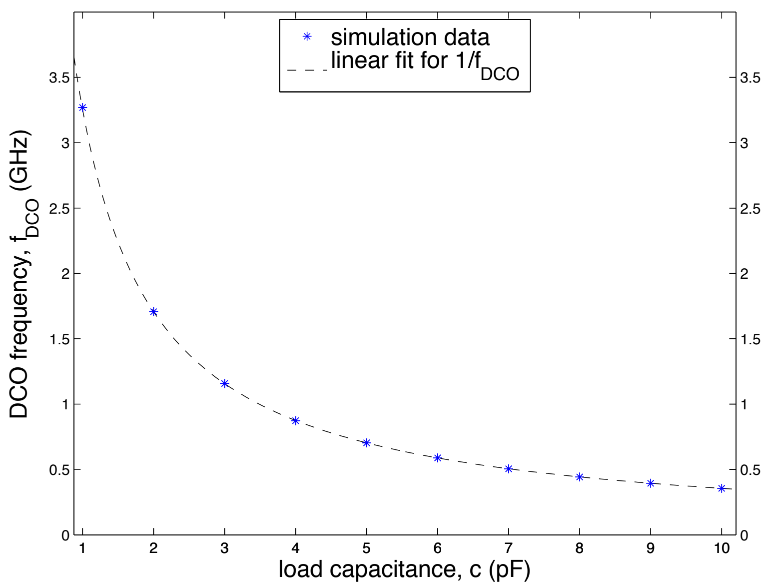
From Spectre simulations (see Figure 4.2), I observe that the oscillator frequency is very nearly linear in v and nearly proportional to the inverse of c for a wide range of each of these parameters. The phase error, ϕ is a continuous quantity, but the values of c and v are determined by the digital accumulators that are updated on each cycle of the reference clock, f_{ref}. This motivates modeling the PLL using a discrete-time recurrence for real-valued variables:

$$
\begin{array}{rcll}
c(i+1) & = & \mathrm{saturate}(c(i) + g_c\,\mathrm{sgn}(\phi), c_{\min}, c_{\max}) & \\
v(i+1) & = & \mathrm{saturate}(v(i) + g_v(c_{center} - c(i)), v_{\min}, v_{\max}) & \\
\phi(i+1) & = & \mathrm{wrap}(\phi(i) + (f_{dco}(c(i), v(i)) - f_{ref}) - g_\phi \phi(i)) & \\
f_{dco}(c, v) & = & \frac{1+\alpha v}{1+\beta c} f_0 & \\
\mathrm{saturate}(x, lo, hi) & = & \min(\max(x, lo), hi) & \\
\mathrm{wrap}(\phi) & = & \mathrm{wrap}(\phi + 1), & \text{if } \phi \leq -1 \\
 & = & \phi, & \text{if } -1 < \phi < 1 \\
 & = & \mathrm{wrap}(\phi - 1), & \text{if } 1 \leq \phi
\end{array}
\tag{4.1}
$$

where g_c, g_v, and g_ϕ are the gain coefficients for the bang-bang frequency control, coarse frequency control, and linear phase paths respectively. The coefficient α is the slope of oscillator frequency with respect to v, and β is the slope of oscillator period with respect to c; both are determined from



(a) frequency vs. operating voltage



(b) frequency vs. load capacitance

Figure 4.2: Ring-oscillator response

simulation data. I measure phase leads or lags in cycles: $\phi = 0.1$ means that $\Phi_{DCO/N}$ leads Φ_{ref} by 10% of the period of Φ_{ref}. We say that c is "saturated" if,

$$c = c_{\min} \ \wedge \ (\phi < 0)$$

or

$$c = c_{\max} \ \wedge \ (\phi > 0)$$

Likewise, v is saturated if,

$$v = v_{\min} \ \wedge \ (c > c_{center})$$

or

$$v = v_{\max} \ \wedge \ (c < c_{center})$$

In this thesis, I scale f_{ref} to 1. With similar scaling, I choose $g_c = 1/3200$, $g_v = -gc/5$, and $g_\phi = 0.8$. I assume bounds for c of $c_{\min} = 0.9$ and $c_{\max} = 1.1$ with $c_{center} = 1$ and bounds for v of $v_{\min} = 0.2$ and $v_{\max} = 2.5$. With these parameters, the PLL is intended to converge to a small neighbourhood of $c = c_{center} = 1$; $v = f_{ref}c_{center} = 1$ and $\phi = 0$.

4.3 Proving Global Convergence

To prove global convergence of this digital PLL, simulations have been conducted in order to get a sense of how things are moving in the state space. From this, I identified key points where I can break the proof up into pieces. By tackling each piece at a time and connecting them together, I form a proof of the global convergence.

4.3.1 Proof in Parts

I formalize the global convergence proof of this digital PLL into four theorems below. Suppose we use B to stand for the blue region, R to stand for the red region, G to stand for the green region and Y to stand for the yellow region in Figure 4.3,

Theorem 4.1. *Global convergence of Digital PLL*

$$\exists \; small \; Y \in B, \forall [c(0), v(0), \phi(0)] \in B, \exists N \geq 0 \; \forall i \geq N,$$
$$s.t. [c(i), v(i), \phi(i)] \in Y$$

Figure 4.3 shows how this Digital PLL converges into a small region in the middle. My verification proceeds in three phases as depicted in the figure. First I show that for all trajectory starting with $c \in [c_{\min}, c_{\max}]$, $v \in [v_{\min}, v_{\max}]$, and $\phi \in [-1, +1]$ (the blue region in Figure 4.3),the trajectory eventually reaches a relatively narrow stripe (the red and green regions) for which $f_{dco} \approx f_{ref}$. To do so, I construct a series of lemmas that form a ranking function. When this PLL is far from lock, its convergence is strong. By proving this I have shown that the non-linearities of the global model do not create unintended stable modes. Theorem 4.1.1 formally states this argument.

Lemma 4.1.1. *Coarse convergence*

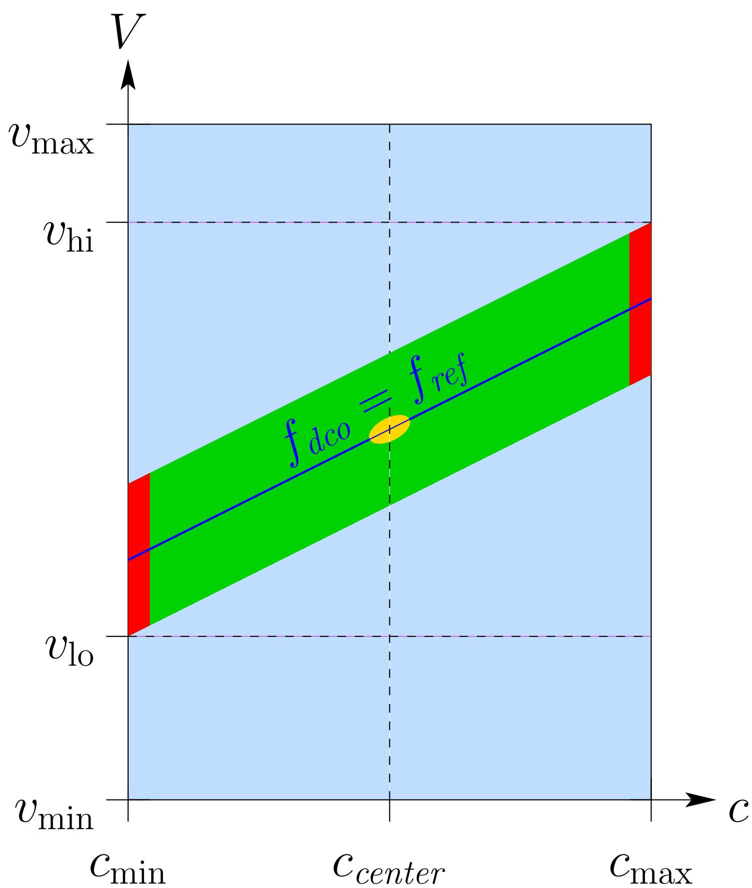
$$\exists \delta > 0 \; and \; N_1, \forall [c(0), v(0), \phi(0)] \in B, and \; |f_{dco}(0) - f_{ref}| > \delta,$$
$$s.t. \forall i \geq N_1, |f_{dco}(i) - f_{ref}| \leq \delta,$$
$$where \; R \cup G = \{[c(i), v(i), \phi(i)] \mid |f_{dco}(i) - f_{ref}| \leq \delta\}$$

Then the second part of the proof pertains to the small, red stripes where $f_{dco} \approx f_{ref}$ but c is close enough to $c_{\min}$ or $c_{\max}$ that saturation remains a concern. Consider the red strip near $c = c_{\min}$. Here, I show that v increases and that c "tracks" v to keep f_{dco} close to f_{ref} and ϕ small. Together, these results show that all trajectories eventually enter the region shown in green in Figure 4.3 in Theorem 4.1.2.

Lemma 4.1.2. *Leaving the saturation*

$$\exists N_2, \forall [c(0), v(0), \phi(0)] \in R, s.t. \forall i \geq N_2, [c(i), v(i), \phi(i)] \in G$$

The final part of the proof shows convergence to the limit cycle region, shown in yellow in Figure 4.3. The key observation here is that ϕ repeatedly



The light blue region denotes the entire space. Using the Lyapunov argument I generated for the blue region, I proved convergence into the middle green region. Then I show convergence into the middle yellow region using another theorem.

Figure 4.3: Global convergence big picture

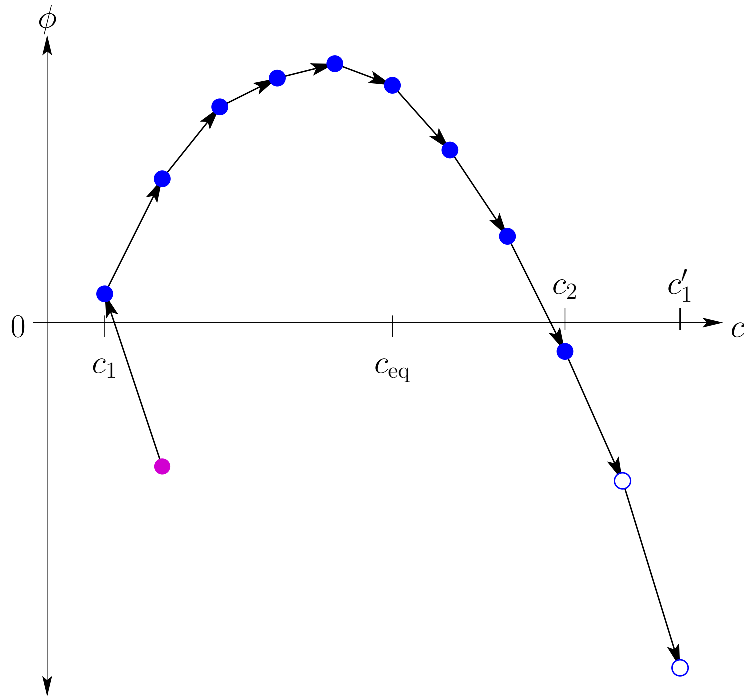


Figure 4.4: Fine convergence

alternates between positive and negative values. For any given value of v, I calculate the value of c for which $f_{dco}(c, v) = f_{ref}$, call this $c_{\text{eq}}(v)$. Figure 4.4 depicts a trajectory from a rising zero-crossing of ϕ to a falling crossing. Let c_1 be the value of c following a rising zero-crossing of ϕ, and let c_2 be the value of c at the subsequent falling crossing. I note that $c_1 < c_{\text{eq}}(v) < c_2$. The fine convergence theorem for the points in the stripe to converge into the middle yellow region has been formally stated in Theorem 4.1.3.

Lemma 4.1.3. *Fine convergence*

$$\exists Y, N_3 \textit{ and } \delta > 0, \forall [c(0), v(0), \phi(0)] \in G, s.t. \forall i \geq N_2, [c(0), v(0), \phi(0)] \in Y$$

The next section discusses the details on how I proved the *fine convergence* proof part using the clause-processor combination. Seeing the proof might give one a better idea of how this proof cannot practically be done by any single tool on itself.

4.3.2 Detailed Proof for Fine Convergence

In the green region of Figure 4.3, it is straightforward to show that c and ϕ settle into an oscillating behavior. The damping term, $-g_\phi \phi(i)$ causes such oscillations to diminish, but they don't die out completely due to the quantization of c. The proof formalizes this intuition.

For any value of v, we can define $c_e q$ so that $f_{dco}(c_{eq}, v) = f_{ref}$. As observed above, the value of c will roughly oscillate around c_{eq} while ϕ oscillates around 0. As shown in Figure 4.4, let c_1 be the value of c when ϕ crosses 0 in a rising direction, and let c_2 be the value of c at the subsequent crossing of 0 by ϕ in the falling direction. Our proof shows that if $|c_1 - c_{eq}|$ is sufficiently large, then $|c_2 - c_{eq}| < |c_1 - c_{eq}| - g_1$. A similar argument applies if we consider a trajectory starting ϕ crossing 0 in the falling direction and going until ϕ crossing 0 in the the rising direction. This shows that if $|c_1 - c_{eq}|$ is sufficiently large, its value will decrease. In our proof, we show this convergence for $|c_1 - c_{eq}| > 3g_1$. This shows that any trajectory in the green region stays in the green region and moves to region very close to the $f_{dco}(c, v) = f_{ref}$ line. Separately, we can show that if such a cycle occurs with $c < c_{center}$ at all points, then v must eventually increase. Likewise, if $c > c_{center}$ at all points of the cycle, the v must eventually decrease. These results together show convergence to the yellow region of Figure 4.3.

The obvious way to show convergence is to show that c_2 is closer to c_{eq} than c_1 is. However, this involves calculating the recurrence step at which ϕ makes its falling crossing of zero, and that involves solving a non-linear system of equations. Although Z3 has a non-linear arithmetic solver, it does not support induction as would be required with an arbitrary choice for c_1.

Instead, I extrapolate the sequence to the last point to the right of c_{eq} that is closer to c_{eq} than c_1 is. I use formula from Eq. 4.1 for computing $c(i+1)$ assuming that $\mathrm{sgn}(\phi) = 1$; either this assumption is valid for the whole sequence, or ϕ had a falling crossing even earlier. Either is sufficient to show convergence.

I stated this theorem in ACL2 and proved it using `Smtlink`. The proof involves solving the recurrence, and rewriting the resulting formula. The key inequality has exponential terms of the form $(1 - g_\phi)^n$ multiplied by rational function terms of the other model parameters. I use the substitution technique from Section 3.2.3 to replace these non-polynomial terms, and add a `:hypothesize` hint that $0 < (1 - g_\phi)^n < 1$. ACL2 readily discharges this added hypothesis using a trivial induction.

The fine convergence proof is based on a 13-page, hand-written proof. The ACL2 version consists of 75 lemmas, 10 of which were discharged using the SMT solver. Of those ten, one was the key, polynomial inequality from the manual proof. The others discharged steps in the manual derivation that were not handled by the standard books of rewrite rules for ACL2. ACL2 completes the proof in a few minutes running on a laptop computer. I found one error in the process of transcribing the hand-written proof to ACL2.

The ACL2 formulation enabled making generalizations that I would not consider making to the manual proof. In particular, the manual proof assumed that $c_{\mathrm{eq}} - c_1$ was an integer multiple of g_1. After verifying the manual proof, I removed this restriction – this took about 12 hours of human time, most of which was to introduce an additional variable $0 \leq d_c < 1$ to account for the non-integer part (see Appendix C.2). I also generalized the proof to allow v to an interval whose width is a small multiple of $|g_2(c_{\max} - c_{\min})|$. This did not require any new operators and took about 3 hours of human time. The interval can be anywhere in $[v_{\mathrm{lo}}, v_{\mathrm{hi}}]$. This shows that the convergence of c and ϕ continues to hold as v progresses toward $f_{ref} c_{center}$. It also sets the foundation for verifying the PLL with a more detailed model including the $\Delta\Sigma$ modulator in the c path, an additional low-pass filter in the v path, and adding error terms in the formula for $f_{dco}(c, v)$.

I completed much of the proof using ACL2 alone while implementing `Smtlink`. I plan to rewrite the proof to take more advantage of the SMT solver and believe that the resulting proof will be simpler, focus more on the high-level issues, and be easer to write and understand. When faced with proving a complicated derivation, one can guide ACL2 through the steps of the derivation, or just check the relationship of the original formula to the final one using the SMT solver. The latter approach allows novice users (including the author of this thesis) to quickly discharge claims that would otherwise take a substantial amount of time even for an expert. As noted before, if Z3 finds a counter-example, the tool does not return it as a witness for ACL2. However, the clause processor prints the counter-example (in its Z3 representation) to the ACL2 proof log. The user can examine this counter-example; in practice, it often points directly to the problem that needs to be addressed.

4.4 Summary and Future Work

In my proof, Lemma 4.1.1 and Lemma 4.1.2 are proven using raw Z3 (see Appendix C.1) and Lemma 4.1.3 is proven using ACL2 with `Smtlink` (see Appendix C.2). `Smtlink` greatly simplified the amount of work in proving inequalities of large arithmetic formulas. I can't imagine proving this lemma in raw ACL2 with reasonable small amount of effort. The analytical approach I'm taking gives the benefit of flexible and extensible proofs when small changes are made to the design. It also shows the limit cycle behavior.

However, future work needs to be done to fulfill the proof. More specifically, following directions are possible directions.

1. A liveness property about Lemma 4.1.2 is left for future proof. To be specific, my proof proves the liveness property that all trajectory on the left wall will finally leave the wall. However, in order to prove the behavior of leaving the saturation, I need to prove all trajectories will eventually left the small saturation region, which means they will never hit the wall again. I can potentially use Z3 as a bounded model checker for fulfilling this lemma.

2. Eventually, I want to translate Z3 proofs for Lemma 4.1.1 and Lemma 4.1.2 into ACL2 and let `Smtlink` call Z3 as a bounded model checker.

3. The main theorem, Theorem 4.1, still needs to be stated in ACL2 and verified. It seems to require reasoning with an existential quantifier as

shown in Theorem 4.1. ACL2 supports proofs with existential quantifiers. Proving this theorem statement will require connecting the three lemmas together.

Chapter 5

Conclusion and Future Work

This thesis demonstrates that SMT techniques and theorem proving provide complementary power for AMS verification problems. It proposes a way of combining a SMT solver and a theorem prover by building an architecture that provides soundness in practice without proof reconstruction. While an error in Z3 or our interface code could, in principle, lead to an unsound theorem, we believe that the likelihood of finding real bugs by applying our tools to real designs is much greater than the risk of an incorrect theorem slipping through our tools. Of course, nothing we have done precludes adding proof reconstruction and/or certificates for those who need that level of soundness. The thesis further applies this combination to proving global convergence for a state-of-the-art digital PLL. Experiment results show how this combination is suitable for AMS design verification.

In this Chapter, Section 5.1 points out the differences between this work and other published AMS verification results. The comparison brings up several strengths that can be provided with this method. Section 5.2 discusses a list of future directions that are enabled by the demonstrated thesis work.

5.1 Conclusions

I presented the integration of the Z3 SMT solver into the ACL2 theorem prover and demonstrated its application for the verification of global convergence for a digital PLL. The proof involves reasoning about systems of polynomial and rational function equalities and inequalities, which is greatly simplified by using Z3's non-linear arithmetic capabilities. ACL2 complements Z3 by providing a versatile induction capability along with a mature environment for proof development and structuring. Chapter 3 described technical issues that must be addressed to ensure the soundness, of the integrated prover, usability issues that are critical for the tool to be practical, and my solutions to these challenges.

Chapter 4 showed how this integrated prover can be used to verify global convergence for a digital phase-locked loop from all initial states to the final limit-cycle behaviours. The analysis of the limit cycle behaviour requires modeling the PLL with recurrences. Such limit cycles are not captured by continuous approximations used in [8, 110]. My approach allowed uncertainty in the model parameters and not just in the signal values. My approach shows strong promise for verification that accounts for device variability and other uncertainties.

Prior work on integrating SMT solvers into theorem provers has focused on using the non-numerical decision procedures of an SMT solver. My work demonstrates the value of bringing an SMT solver into a theorem prover for reasoning about systems where a digital controller interacts with a continuous, analog, physical system. The analysis of such systems often involves long, tedious, and error-prone derivations that primarily use linear algebra and polynomials. I have shown that these are domains where SMT solvers augmented with induction and proof structuring have great promise.

5.2 Future Work

There are three possible directions that are opened up by this work.

5.2.1 Complete the Convergence Proof for the Digital PLL

The digital PLL model is a simplified model. The digital PLL in the original paper [33] also contains a delta-sigma modulator and a low-pass filter. I omitted them because they are not critical components of the PLL in the sense of global convergence.

However, adding those two parts and proving convergence under this new model would still be beneficial. Then I could analyze how this method adapts to new models. By analyzing how much more time and code I devote to proving the expanded version, I'll have more evidence of the scalability of the method.

5.2.2 Build a Better Tool

As discussed in Chapter 3.5, there are several aspects in which I can improve the tool architecture. These include using guards to infer types, providing useful counter-example, using ACL2r, and increasing the automation of function expansion.

There are other things I can implement to extend proof methods and automation. For example, my supervisor is my first "user" for `Smtlink`. We have tried some experiments to automatically identify commonly used substitutions using uninterpreted functions, the syntactic structure of the clauses, and the "fast" theories of Z3 (e.g. linear arithmetic) to identify useful hypotheses. Preliminary results suggest that approaches like this could greatly simplify the proof for the digital PLL and would be useful for other problems as well.

Adding the "hooks" so the user can manipulate the clauses creates new trade-offs between soundness and ease of use. These can be tracked using ACL2s trust-tag mechanism. This opens up the opportunity to try an idea without investing a huge effort to ensure soundness. If it turns out to be useful, then we can go back and progressively remove the need for trust-assumptions.

5.2.3 Other Applications

I am interested in investigating how this combination of theorem proving and SMT solving can be applied to other dynamical physical systems that share common features with AMS designs. Interesting applications include machine learning proofs and other mathematical proofs, medical system and other cyber-physical system verification problems.

Machine learning problems are interesting problems to try because they also make intense use of non-linear arithmetic. In addition to what is already supported by the tool, machine learning problems require more heavy use of linear algebra theories. Other mathematical proofs with similar structure could also benefit from the combination.

Some biomedical devices are naturally modeled as hybrid systems. There's a huge need in medical systems for correctness verification. For example, [48] presents an anaesthetizing system that automatically adjusts the amount of anaesthetic to give to a patient. The system should make sure no overdose or underdose will occur in the feedback control system.

Cyber-physical systems and other physical dynamical systems that demand verification tasks are also interesting problems to try.

I am currently exploring using my methods to verify other AMS designs as well as similar problems that arise in hybrid control systems and machine learning. Trying out these new applications not only helps further justify my belief that this combination is useful, it also gives us a chance to look at what's common in these problems. Those observations might leads to better automation. For example, common modeling blocks could be implemented so that new problems can be composed using these library model blocks. Specification languages could be implemented so that new specifications can be easily expressed and get processed by the tool automatically.

Bibliography

[1] Formalizing 100 theorems. http://www.cs.ru.nl/~freek/100/. Accessed: 2015-01-16. → pages 18

[2] The Method. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____THE-METHOD. Accessed: 2015-02-22. → pages 19, 26

[3] Vhdl analysis and standardization group (vasg). http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome. Accessed: 2015-04-22. → pages 10

[4] 1364-2005 - IEEE standard for Verilog hardware description language. https://standards.ieee.org/findstds/standard/1364-2005.html. Accessed: 2015-04-22. → pages 10

[5] The SPICE page. http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/. Accessed: 2015-04-07. → pages 8

[6] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In *Formal Methods and Software Engineering*, pages 588–605. Springer, 2006. → pages 26

[7] Ghiath Al-Sammane, Mohamed H Zaki, and Sofiène Tahar. A symbolic methodology for the verification of analog and mixed signal designs. In *Proceedings of the conference on Design, automation and test in Europe*, pages 249–254. EDA Consortium, 2007. → pages 7, 15

[8] Matthias Althoff, Akshay Rajhans, Bruce H Krogh, Soner Yaldiz, Xin Li, and Larry Pileggi. Formal verification of phase-locked loops using

reachability analysis and continuization. *Communications of the ACM*, 56(10):97–104, 2013. → pages 35, 75

[9] Peter B Andrews. *An introduction to mathematical logic and type theory*, volume 27. Springer Science & Business Media, 2002. → pages 17

[10] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Certified Programs and Proofs*, pages 135–150. Springer, 2011. → pages 25

[11] Eugene Asarin, Thao Dang, and Oded Maler. The d/dt tool for verification of hybrid systems. In *Computer Aided Verification*, pages 365–370. Springer, 2002. → pages 15

[12] Clark Barrett and Sergey Berezin. CVC Lite: a new implementation of the cooperating validity checker. In *Computer Aided Verification*, pages 515–518. Springer, 2004. → pages 25

[13] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010. → pages 26

[14] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer aided verification*, pages 171–177. Springer, 2011. → pages 24

[15] Jason Baumgartner, Hari Mony, Viresh Paruthi, Robert Kanzelman, and Geert Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 259–266. IEEE, 2007. → pages 22

[16] Bernd Becker and Christoph Scholl. Checking equivalence for partial implementations. In *Equivalence Checking of Digital Circuits*, pages 145–167. Springer, 2004. → pages 11

[17] Michael Ben-Or, Dexter Kozen, and John Reif. The complexity of elementary algebra and geometry. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 457–464. ACM, 1984. → pages 23

[18] Yves Bertot. A short presentation of Coq. In *Theorem Proving in Higher Order Logics*, pages 12–16. Springer, 2008. → pages 18

[19] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs*, pages 48–62. Springer, 2007. → pages 25, 59

[20] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending sledgehammer with SMT solvers. *Journal of automated reasoning*, 51(1):109–128, 2013. → pages 25, 26

[21] Woody W Bledsoe, Robert S Boyer, and William H Henneman. Computer proofs of limit theorems. *Artificial Intelligence*, 3:27–60, 1972. → pages 23

[22] Robert Boyer *et al.* The QED manifesto. *Automated Deduction–CADE*, 12:238–251, 1994. → pages 18

[23] Robert S Boyer and J Strother Moore. A theorem prover for a computational logic. In *10th International Conference on Automated Deduction*, pages 1–15. Springer, 1990. → pages 18, 19

[24] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Silvio Ranise, Peter Van Rossum, and Roberto Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Computer Aided Verification*, pages 335–349. Springer, 2005. → pages 24

[25] Aaron R Bradley. Understanding IC3. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 1–14. Springer, 2012. → pages 11

[26] Randal E Bryant. Symbolic boolean manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys (CSUR)*, 24(3): 293–318, 1992. → pages 11

[27] Bruno Buchberger. An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *Journal of symbolic computation*, 41(3):475–511, 2006. English translation of Bruno Buchberger's 1965 PhD thesis. → pages 23

[28] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE, 1990. → pages 11

[29] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(02):56–68, 1940. → pages 18

[30] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009. → pages 11

[31] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In H. Brakhage, editor, *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 203, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer Berlin Heidelberg, 1975. ISBN 978-3-540-07407-6. doi: 10.1007/3-540-07407-4_17. URL `http://dx.doi.org/10.1007/3-540-07407-4_17`. → pages 23

[32] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971. → pages 22

[33] J. Crossley, E. Naviasky, and E. Alon. An energy-efficient ring-oscillator digital PLL. In *Custom Integrated Circuits Conference (CICC), 2010 IEEE*, pages 1–4, Sept 2010. doi: 10.1109/CICC.2010.5617417. URL `http://dx.doi.org/10.1109/CICC.2010.5617417`. → pages 62, 63, 75

[34] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960. → pages 22

[35] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 113(2): 156–161, 1962. → pages 22

[36] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. → pages 3, 24, 25

[37] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94:130–143, 2014. → pages 25, 26

[38] Inderpreet Singh Dhingra. *Formalising an integrated circuit design style in higher order logic*. PhD thesis, University of Cambridge, 1988. → pages 35

[39] Zhi Jie Dong, Mohamed H Zaki, Ghiath Al Sammane, Sofiene Tahar, and Guy Bois. Checking properties of PLL designs using run-time verification. In *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, pages 125–128. IEEE, 2007. → pages 35

[40] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744. Springer, 2014. → pages 24, 56

[41] Arthur Flatau, Matt Kaufmann, David F Reed, D Russinoff, EW Smith, and Rob Sumners. Formal verification of microproces-

sors at AMD. In *4th International Workshop on Designing Correct Circuits (DCC 2002), Grenoble, France*, 2002. → pages 20

[42] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness+ automation+ soundness: towards combining SMT solvers and interactive proof assistants. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–181. Springer, 2006. → pages 25

[43] Martin Fränzle and Christian Herde. HySAT: an efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007. → pages 24, 36

[44] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011. → pages 15, 36

[45] Ruben Gamboa. Square roots in ACL2: a study in sonata form. Technical report, Citeseer, 1996. → pages 49

[46] Ruben Gamboa and Robert S Boyer. *Mechanically verifying real-valued algorithms in ACL2*. PhD thesis, Citeseer, 1999. → pages 20, 58

[47] Ruben Gamboa and John Cowles. Formal verification of Medina's sequence of polynomials for approximating arctangent. *arXiv preprint arXiv:1406.1561*, 2014. → pages 20

[48] Victor Gan, Guy A Dumont, and Ian M Mitchell. Benchmark problem: a PK/PD model and safety constraints for anesthesia delivery. 2014. → pages 77

[49] Sicun Gao, Soonho Kong, and Edmund M Clarke. Satisfiability modulo odes. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 105–112. IEEE, 2013. → pages 24

[50] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *Hybrid Systems: Computation and Control*, pages 291–305. Springer, 2005. → pages 15, 36

[51] Evgueni Goldberg, Mukul Prasad, and Robert Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the conference on Design, automation and test in Europe*, pages 114–121. IEEE Press, 2001. → pages 11

[52] Alexandre Goldsztejn, Olivier Mullier, Damien Eveillard, and Hiroshi Hosobe. Including ordinary differential equations based constraints in the standard CP framework. In *Principles and Practice of Constraint Programming–CP 2010*, pages 221–235. Springer, 2010. → pages 24

[53] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Handbook of Knowledge Representation*, 3:89–134, 2008. → pages 23

[54] Michael JC Gordon and Tom F Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993. → pages 18

[55] Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, 1986. → pages 17

[56] Mark R Greenstreet. Verifying safety properties of differential equations. In *Computer Aided Verification*, pages 277–287. Springer, 1996. → pages 14

[57] Mark R Greenstreet and Ian Mitchell. Reachability analysis using polygonal projections. In *Hybrid Systems: Computation and Control*, pages 103–116. Springer, 1999. → pages 15

[58] John Harrison. HOL Light: a tutorial introduction. In *Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996. → pages 18

[59] John Harrison. Floating-point verification. In *FM 2005: Formal Methods*, pages 529–532. Springer, 2005. → pages 18

[60] John Harrison and Laurent Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998. → pages 25, 59

[61] Walter Hartong, Lars Hedrich, and Erich Barke. Model checking algorithms for analog verification. In *Proceedings of the 39th annual Design Automation Conference*, pages 542–547. ACM, 2002. → pages 14

[62] Shant Harutunian. *Formal Verification of Computer Controlled Systems*. PhD thesis, University of Texas, Austin, May 2007. URL `https://www.lib.utexas.edu/etd/d/2007/harutunians68792/harutunians68792.pdf`. → pages 16

[63] Lars Hedrich and Erich Barke. A formal approach to nonlinear analog circuit verification. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 123–127. IEEE Computer Society, 1995. → pages 12

[64] Warren A Hunt, Robert Bellarmine Krug, Sandip Ray, and William D Young. Mechanized information flow analysis through inductive assertions. In *Formal Methods in Computer-Aided Design, 2008. FMCAD'08*, pages 1–4. IEEE, 2008. → pages 20

[65] Warren A Hunt Jr and Sol Swords. Centaur technology media unit verification. In *Computer Aided Verification*, pages 353–367. Springer, 2009. → pages 20

[66] Fabian Immler. Formally verified computation of enclosures of solutions of ordinary differential equations. In *NASA Formal Methods*, pages 113–127. Springer, 2014. → pages 16

[67] PB Jackson. The Nuprl proof development system. *Reference manual and Users's Guide (Version 4.1)*, 1994. → pages 18

[68] Ji-Eun Jang, Myeong-Jae Park, Dongyun Lee, and Jaeha Kim. True event-driven simulation of analog/mixed-signal behaviors in SystemVerilog: A decision-feedback equalizing (DFE) receiver example. In *Custom Integrated Circuits Conference (CICC), 2012 IEEE*, pages 1–4. IEEE, 2012. → pages 8

[69] Alexander Jesser and Lars Hedrich. A symbolic approach for mixed-signal model checking. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, pages 404–409. IEEE Computer Society Press, 2008. → pages 35

[70] Richard M Karp. *Reducibility among combinatorial problems.* Springer, 1972. → pages 22

[71] Matt Kaufmann and J Strother Moore. ACL2: an industrial strength version of Nqthm. In *Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*, pages 23–34. IEEE, 1996. → pages 3, 18, 19

[72] Matt Kaufmann and J Strother Moore. An ACL2 tutorial. In *Theorem Proving in Higher Order Logics*, pages 17–21. Springer, 2008. → pages 19

[73] Jaeha Kim, Metha Jeeradit, Byongchan Lim, and Mark A Horowitz. Leveraging designer's intent: a path toward simpler analog CAD tools. In *Custom Integrated Circuits Conference, 2009. CICC'09. IEEE*, pages 613–620. IEEE, 2009. → pages 4

[74] Donald Ervin Knuth. Mathematics and computer science: coping with finiteness. *Science (New York, NY)*, 194(4271):1235–1242, 1976. → pages 23

[75] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual Design Automation Conference*, pages 263–268. ACM, 1997. → pages 11

[76] Kenneth S Kundert. Introduction to RF simulation and its application. *IEEE Journal of Solid-State Circuits*, 34(9):1298–1319, 1999. → pages 4

[77] Robert P Kurshan and Kenneth L McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(11):1356–1371, 1991. → pages 14

[78] Alexander B. Kurzhanski and István Vályi. *Ellipsoidal Calculus for Estimation and Control*. International Institute for Applied Systems Analysis, Austria, 1997. → pages 15

[79] Daniel Lazard. Thirty years of polynomial system solving, and now? *Journal of symbolic computation*, 44(3):222–231, 2009. → pages 23

[80] K Rustan M Leino. This is Boogie 2. *Manuscript KRML*, 178:131, 2008. → pages 25

[81] K Rustan M Leino. Dafny: an automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010. → pages 24

[82] K Rustan M Leino. Automating induction with an SMT solver. In *Verification, Model Checking, and Abstract Interpretation*, pages 315–331. Springer, 2012. → pages 24

[83] Honghuang Lin and Peng Li. Parallel hierarchical reachability analysis for analog verification. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6, 2014. doi: 10.1145/2593069.2593178. URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6881477`. → pages 36

[84] Honghuang Lin, Peng Li, and Chris J Myers. Verification of digitally-intensive analog circuits via kernel ridge regression and hybrid reachability analysis. In *Proceedings of the 50th Annual Design Automation Conference*, page 66. ACM, 2013. → pages 36

[85] Leonard R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, 100(2):107–115, 1981. → pages 10

[86] João P Marques-Silva and Karem A Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. → pages 22

[87] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: a case study combining HOL-Light and CVC Lite. *Electronic Notes in Theoretical Computer Science*, 144(2):43–51, 2006. → pages 25

[88] Kenneth L McMillan. Interpolation and SAT-based model checking. In *Computer Aided Verification*, pages 1–13. Springer, 2003. → pages 11

[89] Herbert A. Medina. A sequence of polynomials for approximating arctangent. *The American Mathematical Monthly*, 113(2):pp. 156–161, 2006. ISSN 00029890. URL `http://www.jstor.org/stable/27641866`. → pages 20

[90] Stephan Merz and Hernán Vanzetto. Automatic verification of TLA+ proof obligations with SMT solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 289–303. Springer, 2012. → pages 25

[91] J Strother Moore. A mechanically checked proof of a multiprocessor result via a uniprocessor view. *Formal Methods in System Design*, 14 (2):213–228, 1999. → pages 19

[92] J. Strother Moore, Thomas W. Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5 K 86 TM floating-point division program. *Computers, IEEE Transactions on*, 47(9):913–926, 1998. → pages 20

[93] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979. → pages 23

[94] Sam Owre, John M Rushby, and Natarajan Shankar. PVS: a prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992. → pages 18

[95] Lawrence C Paulson. *Isabelle: a generic theorem prover*, volume 828. Springer, 1994. → pages 18

[96] Radek Pelánek. Fighting state space explosion: review and evaluation. In *Formal Methods for Industrial Critical Systems*, pages 37–52. Springer, 2009. → pages 9

[97] Kent Pitman and Kathy Chapman. Information Technology–Programming Language–Common Lisp, 1994. → pages 39

[98] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 89–103. Springer, 1999. → pages 18

[99] Peter Reid and Ruben Gamboa. *Automatic differentiation in ACL2*. Springer, 2011. → pages 20

[100] David Russinoff, Matt Kaufmann, Eric Smith, and Robert Sumners. Formal verification of floating-point RTL at AMD using the ACL2 theorem prover. *Proceedings of the 17th IMACS World Congrress on Scientific Computation, Applied Mathematics and Simulation, Paris, France*, 2005. → pages 20

[101] Jun Sawada and Warren A Hunt Jr. Hardware modeling using function encapsulation. In *Formal Methods in Computer-Aided Design*, pages 271–282. Springer, 2000. → pages 17

[102] Charles L Seitz. System timing. *Introduction to VLSI systems*, pages 218–262, 1980. → pages 14

[103] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, pages 127–144. Springer, 2000. → pages 11

[104] Naveed A Sherwani. *Algorithms for VLSI physical design automation.* Kluwer academic publishers, 1995. → pages 10

[105] Robert E Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM (JACM)*, 24(4):529–543, 1977. → pages 23

[106] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008. → pages 18

[107] Alfred Tarski. *A decision method for elementary algebra and geometry.* Springer, 1998. → pages 23

[108] Saurabh K Tiwary, Anubhav Gupta, Joel R Phillips, Claudio Pinello, and Radu Zlatanovici. First steps towards SAT-based formal analog verification. In *Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 1–8. IEEE, 2009. → pages 15

[109] Zhiwei Wang, Naeem Abbasi, Rajeev Narayanan, Mohamed H Zaki, Ghiath Al Sammane, and Sofiene Tahar. Verification of analog and mixed signal designs using online monitoring. In *Mixed-Signals, Sensors, and Systems Test Workshop, 2009. IMS3TW'09. IEEE 15th International*, pages 1–8. IEEE, 2009. → pages 35

[110] Jijie Wei, Yan Peng, Ge Yu, and Mark Greenstreet. Verifying global convergence for a digital phase-locked loop. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 113–120. IEEE, 2013. → pages iii, 36, 75

[111] Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991. → pages 1

[112] Chao Yan and Mark Greenstreet. Faster projection based methods for circuit level verification. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 410–415. IEEE, 2008. → pages 15

[113] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, pages 17–36. Springer, 2002. → pages 22

[114] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001. → pages 22

Appendix A

Example Proofs with ACL2 and Z3

A.1 Geometric Sum Proof with Raw ACL2

This program is successfully proved in 0.14 seconds. Note that in order to make it easier, we made a further constraint that r should be an integer in this proof. To make it work for arbitrary real number, it will take much more theorems and time.

```
;; Basic structure of this file:
;;   To avoid reasoning about division throughout the proof, I first show:
;;      (sum_{i=0}^N r^i)*(r-1) = r^(i+1) - 1
;;   I then divide both sides by r-1 to get the main result.
;;   This file starts with lots of definitions:
;;       (my-expt-pos r n): r to the n^th power, for n >= 0.
;;       (lhs r n):         sum_{i=0}^n r^i
;;       (rhs r n):         (r^(i+1) - 1)/(r-1)
;;   lhs-no-div and rhs-no-div are the lhs*(r-1) and rhs*(r-1) respectively.
;;
;;   The proofs in this file are roughly in the following order:
;;     my-expt-pos and lhs are integer valued.
;;     some trivial algebra results.
;;     a lemma for the induction step for the no-div version of the formula
;;     the no-div version of the formula
;;     the main formula.
```

```
(encapsulate ()
  (defun my-expt-pos (r n)
    (declare (xargs :guard (and (natp n) (integerp r))))
    (if (zp n) 1 (* r (my-expt-pos r (1- n)))))

  (defthm my-expt-pos-integerp
    (implies (and (integerp r) (integerp n) (<= 0 n))
             (integerp (my-expt-pos r n))))

  ;; the summation form for a geometric sum
  (defun lhs (r n)
    (declare (xargs :guard (and (integerp r) (integerp n) (<= 0
   n))))
    (if (and (natp n) (< 0 n))
         (+ (my-expt-pos r n) (lhs r (1- n)))
         1))

  (local (defthm |(integerp (+ a (* b c)))|
    (implies (and (integerp a) (integerp b) (integerp c))
             (integerp (+ a (* b c))))
    :rule-classes nil))

  (encapsulate ()
    (local (defthm lhs-integerp-lemma-1
      (implies (and (integerp r) (integerp n) (< 0 n))
               (integerp (my-expt-pos r (+ -1 n))))
      :rule-classes nil))

    (local (defthm lhs-integerp-lemma-2
      (implies (and (integerp n)
                    (< 0 n)
                    (integerp (lhs r (+ -1 n)))
                    (integerp r))
                   (integerp (+ (lhs r (+ -1 n))
                (* r (my-expt-pos r (+ -1 n))))))
```

```
    :hints (("Goal" :use ( (:instance lhs-integerp-lemma-1 (n n) (r r))
                            (:instance |(integerp (+ a (* b c)))|
                                       (a (lhs r (+ -1 n)))
                                       (b r)
                                       (c (my-expt-pos r (+ -1 n)))))))))

  (defthm lhs-integerp
    (implies (and (integerp r) (integerp n) (<= 0 n))
             (integerp (lhs r n))))
)


(local (defun lhs-no-div (r n)
  (declare (xargs :guard (and (integerp n) (<= 0 n) (integerp r))))
  (* (lhs r n) (1- r))))

(defun rhs-no-div (r n)
  (declare (xargs :guard (and (integerp r) (integerp n) (<= 0 n))))
  (1- (my-expt-pos r (+ n 1))))

(defun rhs (r n)
  (declare (xargs :guard (and (integerp r) (integerp n) (<= 0 n))))
  (if (equal r 1) (1+ n) (/ (rhs-no-div r n) (1- r))))

(defun geo-hyps (r n)
  (and (integerp r) (natp n) (> r 0) (not (equal r 1))))

(local (defthm lemma-plus-minus-one
  (implies (integerp n) (equal (+ 1 -1 n) n))))

(local (defthm lemma-minus-plus-one  ;; simplify (+ -1 1 n)
```

```
  (implies (integerp n) (equal (+ -1 1 n) n))))

(local (defthm unexpand-my-expt-pos  ;; simplify (* r (my-expt r (1- n)))
  (implies (and (geo-hyps r n) (< 0 n))
           (equal (* r (my-expt-pos r (1- n))) (my-expt-pos r n)))
  :rule-classes nil))

(local (defthm expand-lhs
  (implies (and (geo-hyps r n) (< 0 n))
           (equal (lhs r n)
                  (+ (my-expt-pos r n) (lhs r (1- n)))))))

(local (encapsulate ()
  (local (defthm expand-lhs-no-div-lemma-1
    (implies (and (integerp a) (integerp b) (integerp c))
             (equal (* (+ a b) c) (+ (* a c) (* b c)))))))

  (local (defthm expand-lhs-no-div-lemma-2
    (implies (and (integerp b) (integerp r))
              (equal (* (+ -1 r) r b) (+ (* -1 r b) (* r r b)))))))

  (defthm expand-lhs-no-div
    (implies (and (geo-hyps r n) (< 0 n))
             (equal (* (lhs r n) (1- r))
                    (+ (* (1- r) (my-expt-pos r n)) (lhs-no-div r (1- n))))))
))

(local (encapsulate ()
  (local (defthm expand-rhs-no-div-lemma
    (implies (and (integerp b) (integerp c) (integerp r)
                  (equal c (* r b)))
             (equal (+ -1 b (* (+ -1 r) b)) (+ -1 c))
    )
```

```
      :rule-classes nil))

    (defthm expand-rhs-no-div
      (implies (and (geo-hyps r n) (< 0 n))
               (equal (1- (my-expt-pos r (+ n 1)))
                      (+ (* (1- r) (my-expt-pos r n)) (1- (my-expt-pos r n)))))
      :hints (("Goal"
               :do-not-induct t
                 :hands-off (my-expt-pos)
                 :use ( (:instance unexpand-my-expt-pos (n (1+ n)))
                         (:instance lemma-minus-plus-one (n n))
                         (:instance lemma-plus-minus-one (n n))
                         (:instance expand-rhs-no-div-lemma
                                     (b (my-expt-pos r n))
                                     (c (my-expt-pos r (1+ n)))
                                     (r r))
                       )))
      :rule-classes nil)))

  (local (defthm no-div-induct
    (implies (and (geo-hyps r n) (< 0 n) (equal (lhs-no-div r n) (rhs-no-div r n)))
             (equal (* (lhs r n) (1- r))
                    (1- (my-expt-pos r (+ n 1)))
             ))
    :hints (("Goal" :do-not-induct t))))

  (local (defthm geo-lemma-no-div
    (implies (geo-hyps r n)
             (equal (* (lhs r n) (1- r))
                    (1- (my-expt-pos r (+ n 1)))))))

  (local (defthm div-eq-by-eq
    (implies (and (integerp a) (integerp b) (integerp c) (not (equal a 0)) (equal b c))
             (equal (/ b a) (/ c a)))))
```

```

 (local (defthm geo-lemma-1
   (implies (and (integerp r) (integerp n) (<= 0 n))
            (integerp (+ -1 (* r (my-expt-pos r n)))))
   :hints (("Goal"
           :do-not-induct t
            :use ( (:instance my-expt-pos-integerp (r r) (n n))
                   (:instance |(integerp (+ a (* b c)))|
                              (a -1) (b r) (c (my-expt-pos r
  n))))))))

 ;; The main theorem.
 (defthm geo
   (implies (geo-hyps r n) (equal (lhs r n) (rhs r n)))
   :hints (("Goal" :do-not-induct t
         :use ( (:instance div-eq-by-eq
                          (a (1- r))
                          (b (* (lhs r n) (1- r)))
                          (c (1- (my-expt-pos r (+ n 1)))))
                (:instance geo-lemma-no-div (r r) (n n)) ))))
)
```

A.2 Geometric Sum Proof with Arithmetic Book

This program is successfully proved in 0.15 seconds. This is assuming the book has already been loaded.

```
;; This is a program proving the geometric sum equation
;; using arithmetic-5.
;; Arithmetic-5 is tuned for this kind of problem.
;; So it passed easily.
;;
;; by Yan Peng (2015-02-25)
;;

(in-package "ACL2")
```

```
(include-book "arithmetic-5/top" :dir :system)

;; define left hand side
(defun lhs (r n)
  (if (zp n)
      1
      (+ (expt r n) (lhs r (1- n)))))

(defun rhs (r n)
  (/ (- 1 (expt r (+ n 1)))
     (- 1 r)))

(defthm geo
  (implies (and (natp n)
                (rationalp r)
                (> r 0)
                (not (equal r 1))
           )
            (equal (lhs r n)
                   (rhs r n))))
```

A.3 Geometric Sum Proof with Smtlink

This program is successfully proved in 0.06 seconds. This is assuming the book has already been loaded.

```
;;; This is a program proving the geometric sum equation
;; using my clause processor.
;;
;; by Yan Peng (2015-02-25)
;;

(in-package "ACL2")
;; set up directories to clause processor dir
(add-include-book-dir :cp
    "/ubc/cs/home/y/yanpeng/project/ACL2/smtlink")
```

```
(include-book "SMT-connect" :dir :cp)


;; define left hand side
(defun lhs (r n)
  (if (zp n)
      1
      (+ (expt r n) (lhs r (1- n)))))

(defun rhs (r n)
  (/ (- 1 (expt r (+ n 1)))
     (- 1 r)))

(defthm geo-cp-lemma
  (implies (and (and (integerp n)
                     (rationalp r))
                (and (< 0 n)
                     (equal (lhs r (+ -1 n))
                            (+ (/ (+ 1 (- r)))
                               (* (/ (+ 1 (- r)))
                                  (- (* r (expt r (+ -1 n)))))))
                     (<= 0 n)
                     (< 0 r)
                     (not (equal r 1))))
           (equal (+ (lhs r (+ -1 n))
                     (* r (expt r (+ -1 n))))
                  (+ (/ (+ 1 (- r)))
                     (* (/ (+ 1 (- r)))
                        (- (* r r (expt r (+ -1 n))))))))
  :hints (("Goal"
          :clause-processor
          (my-clause-processor clause
                               '((:expand ((:functions ())
                                           (:expansion-level 1)))
                                 (:python-file "geo_cp_2")
                                 (:let ((lhs_n_minus_1 (lhs r (+
    -1 n)) rationalp)
```

```
                                    (expt_r_n_minus_1 (expt r (+ -1 n)) rationalp)))
                                  (:hypothesize ())
                                  (:use ((:let ())
                                         (:hypo (()))
                                         (:main ()))))
                                )))
)

(defthm geo-cp
  (implies (and (natp n)
                (rationalp r)
                (> r 0)
                (not (equal r 1)) )
           (equal (lhs r n)
                  (rhs r n)))
  :hints (("Subgoal *1/1''"
           :clause-processor
           (my-clause-processor clause
                                '((:expand ((:functions ())
                                            (:expansion-level 1)))
                                  (:python-file "geo_cp_1")
                                  (:let ())
                                  (:hypothesize ())
                                  (:use ((:let ())
                                         (:hypo (()))
                                         (:main ()))))
                                ))))
```

A.4 Polynomial Inequality Proof with Z3

This program is successfully proved in 0.0004 seconds.

```
# This program check if below theorem can be proven
# by Z3 directly. The theorem basically says a set
# of polynomial inequalities has no solution.
```

```
#
# The three polynomials:
# 1. hyperbola: x*x - y*y <= 1
# 2. parabola: y >= 3*(x - 2.125)*(x-2.125) - 3
# 3. ellipse: 1.125*x*x + y*y <= 1
#
# by Yan Peng (2015-02-25)

# define the hyperbola
def hyperbola(x, y):
    return x*x - y*y <= 1
# define the parabola
def parabola(x, y):
    return y >= 3*(x - 2.125)*(x-2.125) - 3
# define the ellipse
def ellipse(x, y):
    return 1.125*x*x + y*y <= 1

from z3 import *

x = Real("x")
y = Real("y")

print prove(Not(And(hyperbola(x,y), parabola(x,y), ellipse(x,y))))
```

A.5 Polynomial Inequality Proof with ACL2

ACL2 failed to prove this program. It stops at 0.09 seconds.

```
;; This program check if below theorem can be proven
;; by ACL2's arithmetic5 book directly. The theorem
;; basically says a set of polynomial inequalities
;; has no solution.
;;
;; The three polynomials:
;; 1. hyperbola: x*x - y*y <= 1
```

```
;; 2. parabola:  y >= 3*(x - 2.125)*(x - 2.125) - 3
;; 3. ellipse:   1.125*x*x + y*y <= 1
;;
;; by Yan Peng (2015-02-25)

(in-package "ACL2")
(include-book "arithmetic-5/top" :dir :system)

;; define the hyperbola
(defun hyperbola (x y)
  (<= (- (* x x)
    (* y y))
      1))
;; define the parabola
(defun parabola (x y)
  (>= y
      (- (* 3
       (- x 17/8)
       (- x 17/8))
    3)))
;; define the ellipse
(defun ellipse (x y)
  (<= (+ (* 9/8 x x)
    (* y y))
      1))

;; prove the theorem using arithmetic-5
(defthm prove-with-arithmetic-5
  (implies (and (and (realp x) (realp y)))
      (not (and (hyperbola x y)
           (parabola x y)
           (ellipse x y))))
  :rule-classes nil)
```

A.6 Polynomial Inequality Proof with `Smtlink`

This program is successfully proved in 0.02 seconds.

```
;; This program check if below theorem can be proven
;; by our clause processor directly. The theorem
;; basically says a set of polynomial inequalities
;; has no solution.
;;
;; The three polynomials:
;; 1. hyperbola: x*x - y*y <= 1
;; 2. parabola:  y >= 3*(x - 2.125)*(x - 2.125) - 3
;; 3. ellipse:   1.125*x*x + y*y <= 1
;;
;; by Yan Peng (2015-02-26)

(in-package "ACL2")
;; set up directories to clause processor dir
(add-include-book-dir :cp "/ubc/cs/home/y/yanpeng/project/ACL2/smtlink")
(include-book "SMT-connect" :dir :cp)


;; define the hyperbola
(defun hyperbola (x y)
  (<= (- (* x x)
    (* y y))
      1))
;; define the parabola
(defun parabola (x y)
  (>= y
      (- (* 3
       (- x 17/8)
       (- x 17/8))
    3)))
;; define the ellipse
(defun ellipse (x y)
```

```
  (<= (+ (* 9/8 x x)
    (* y y))
      1))

(defthm prove-with-cp
  (implies (and (and (rationalp x) (rationalp y))
      (and ))
      (not (and (hyperbola x y)
           (parabola x y)
           (ellipse x y))))
  :hints (("Goal"
      :do-not '(simplify)
      :clause-processor
      (my-clause-processor clause
                                  '((:expand ((:functions ((hyperbola rationalp)
                                                          (parabola rationalp)
                                                          (ellipse rationalp)))
                                              (:expansion-level 1)))
                                    (:python-file "prove_with_cp")
                                    (:let ())
                                    (:hypothesize ())
                                    (:use ((:let ())
                                           (:hypo (()))
                                           (:main ()))))
                                  )
      ))
  :rule-classes nil)
```

Appendix B

Smtlink Code

B.1 ACL2 Expansion, Translation and Interpretation

```
(in-package "ACL2")

(include-book "config")
(include-book "helper")
(include-book "SMT-extract")
(include-book "SMT-formula")
(include-book "SMT-function")
(include-book "SMT-translator")
(include-book "SMT-interpreter")
(include-book "SMT-run")
(include-book "SMT-z3")
(include-book "SMT-connect")
```

```
;;
;; This file is adapted from :doc define-trusted-clause-processor
;; The dependent files, instead of being in raw Lisp, are in ACL2.
;; That makes me doubt if I really need to do defstub, progn,
;; progn!, and push-untouchable...
;;
;; However, I'm using them right now in case if there are
;; behaviours with those constructs that are not known to me.
;;
(in-package "ACL2")
(include-book "tools/bstar" :dir :system)
(set-state-ok t)
```

```
(defstub acl2-my-prove
  (term fn-lst fn-level fname let-expr new-hypo let-hints hypo-hints main-hints state)
  (mv t nil nil nil nil state))

(program)
(defttag :Smtlink)

(include-book "SMT-z3")
(value-triple (tshell-ensure))

(progn

; We wrap everything here in a single progn, so that the entire form is
; atomic.  That's important because we want the use of push-untouchable to
; prevent anything besides my-clause-processor from calling acl2-my-prove.

  (progn!

   (set-raw-mode-on state) ;; conflict with assoc, should use assoc-equal, not assoc-eq

   (defun acl2-my-prove (term fn-lst fn-level fname let-expr new-hypo let-hints hypo-hints main-hints state)
     (my-prove term fn-lst fn-level fname let-expr new-hypo let-hints hypo-hints main-hints state))
   )

  (defun Smtlink-arguments (hint)
    (b* ((fn-lst (cadr (assoc ':functions
                  (cadr (assoc ':expand hint)))))
    (fn-level (cadr (assoc ':expansion-level
               (cadr (assoc ':expand hint)))))
```

```
    (fname (cadr (assoc ':python-file hint)))
    (let-expr (cadr (assoc ':let hint)))
    (new-hypo (cadr (assoc ':hypothesize hint)))
    (let-hints (cadr (assoc ':type
             (cadr (assoc ':use hint)))))
    (hypo-hints (cadr (assoc ':hypo
              (cadr (assoc ':use hint)))))
    (main-hints (cadr (assoc ':main
              (cadr (assoc ':use hint))))))
   (mv fn-lst fn-level fname let-expr new-hypo let-hints hypo-hints main-hints))
    )

  (defun Smtlink (cl hint state)
    (declare (xargs :guard (pseudo-term-listp cl)
                    :mode :program))
    (prog2$ (cw "Original clause(connect): ~q0" (disjoin cl))
    (b* (((mv fn-lst fn-level fname let-expr new-hypo let-hints hypo-hints main-hints)
     (Smtlink-arguments hint)))
      (mv-let (res expanded-cl type-related-theorem hypo-theorem fn-type-theorem state)
         (acl2-my-prove (disjoin cl) fn-lst fn-level fname let-expr new-hypo let-hints hypo-hints main-hints state)
         (if res
        (let ((res-clause (append (append (append fn-type-theorem type-related-theorem) hypo-theorem)
                    (list (append expanded-cl cl))
                    )))
          (prog2$ (cw "Expanded clause(connect): ~q0 ~% Success!~%" res-clause)
             (mv nil res-clause state)))
        (prog2$ (cw "~|~%NOTE: Unable to prove goal with ~
                                my-clause-processor and indicated hint.~|")
           (mv t (list cl) state)))))))

```

```
  (push-untouchable acl2-my-prove t)
  )

(define-trusted-clause-processor
  Smtlink
  nil
  :ttag Smtlink)
```

```
;; SMT-extract extracts the declarations, hypotheses and conclusion from a SMT formula in ACL2.
;; A typical SMT formula is in below format:
;; (implies (and <decl-list>
;;               <hypo-list>)
;;          <concl-list>)

(in-package "ACL2")
(include-book "./helper")

;; get-orig-param
(defun get-orig-param (decl-list)
  "get-orig-param: given a declaration list of a SMT formula, return a list of variables appearing in the declaration list"
  (if (atom decl-list)
      (cond ((or (equal decl-list 'if)
       (equal decl-list 'nil)
       (equal decl-list 'rationalp)
       (equal decl-list 'integerp)
       (equal decl-list 'quote))
        nil)
       (t decl-list))
      (combine (get-orig-param (car decl-list))
          (get-orig-param (cdr decl-list)))))

;; SMT-extract
(defun SMT-extract (term)
  "extract decl-list, hypo-list and concl from a ACL2 term"
  (let ((decl-list (cadr (cadr term)))
```

```
  (hypo-list (caddr (cadr term)))
  (concl-list (caddr term)))
   (mv decl-list hypo-list concl-list)))
```

```
;; SMT-function
(in-package "ACL2")
(include-book "std/strings/top" :dir :system)
(include-book "./helper")
(include-book "./SMT-extract")
(set-state-ok t)
(set-ignore-ok t)

;; create-name
(defun create-name (num)
  "create-name: creates a name for a new function"
  (let ((index (STR::natstr num)))
    (if (stringp index)
   (mv (intern-in-package-of-symbol
        (concatenate 'string "var" index) 'ACL2)
       (1+ num))
      (prog2$ (cw "Error(function): create name failed: ~q0!" index)
         (mv nil num)))))

;; replace-var
(defun replace-var (body var-pair)
  "replace-var: replace all appearance of a function symbol in the
    body with the var-pair"
  (if (atom body)
      (if (equal body (car var-pair))
     (cadr var-pair)
   body)
    (cons (replace-var (car body) var-pair)
     (replace-var (cdr body) var-pair))))

;; set-fn-body
(defun set-fn-body (body var-list)
  "set-fn-body: set the body for let expression"
```

```
  (if (endp var-list)
      body
    (set-fn-body
     (replace-var body (car var-list))
     (cdr var-list))))

;; make-var-list
(defun make-var-list (formal num)
  "make-var-list: make a list of expressions for replacement"
  (if (endp formal)
      (mv nil num)
    (mv-let (var-name res-num1)
       (create-name num)
       (mv-let (res-expr res-num2)
          (make-var-list (cdr formal) res-num1)
          (mv (cons (list (car formal) var-name) res-expr)
    res-num2)))))

;; assoc-fetch-key
(defun assoc-fetch-key (assoc-list)
  "assoc-fetch-key: fetching keys from an associate list"
  (if (endp assoc-list)
      nil
    (cons (caar assoc-list) (assoc-fetch-key (cdr assoc-list)))))

;; assoc-fetch-value
(defun assoc-fetch-value (assoc-list)
  "assoc-fetch-value: fetching values from an associate list"
  (if (endp assoc-list)
      nil
    (cons (cadr (car assoc-list)) (assoc-fetch-value (cdr
    assoc-list)))))

;; decrease-level-by-1
(defun decrease-level-by-1 (fn fn-level-lst)
  "decrease-level-by-1: decrease a function's expansion level by 1."
  (if (endp fn-level-lst)
```

```
      nil
      (if (equal (car (car fn-level-lst)) fn)
     (cons (list fn (1- (cadr (car fn-level-lst))))
      (cdr fn-level-lst))
     (cons (car fn-level-lst)
      (decrease-level-by-1 fn (cdr fn-level-lst))))))

;; expand-a-fn
;; e.g.(defun double (x y) (+ (* 2 x) y))
;;     (double a b) -> (let ((var1 a) (var2 b)) (+ (* 2 var1) var2))
;;     (double a b) -> ((lambda (var1 var2) (+ (* 2 var1) var2)) a b)
;; 2014-07-01
;; added code for decreasing level for function expanded
(defun expand-a-fn (fn fn-level-lst fn-waiting fn-extended num state)
  "expand-a-fn: expand an expression with a function definition, num should be accumulated by 1. fn should be stored as a symbol"
  (let ((formal (cdr (cadr (meta-extract-formula fn state))))
   ;; the third element is the formalss
   (body (end (meta-extract-formula fn state)))
   ;; the last element is the body
   )
    (if (endp formal)
   (mv body
       (my-delete fn-waiting fn)
       (cons fn fn-extended)
       (decrease-level-by-1 fn fn-level-lst)
       num)
   (mv-let (var-list num1)
      (make-var-list formal num)
      (mv (list 'lambda (assoc-fetch-value var-list)
           (set-fn-body body var-list))
          (my-delete fn-waiting fn)
          (cons fn fn-extended)
          (decrease-level-by-1 fn fn-level-lst)
          num1)))))
```

```

;; lambdap
(defun lambdap (expr)
  "lambdap: check if a formula is a valid lambda expression"
  (if (not (equal (len expr) 3))
      nil
      (let ((lambdax (car expr))
       (formals (cadr expr)))
       ;;(body (caddr expr)))
   (if (and (equal lambdax 'lambda)
       (listp formals)) ;; I can add a check for no
                             ;; occurance of free variable in the future
       t
       nil))))

(skip-proofs
(mutual-recursion
 ;; expand-fn-help-list
 (defun expand-fn-help-list (expr fn-lst fn-level-lst fn-waiting fn-extended num state)
   "expand-fn-help-list"
   (declare (xargs :measure (list (acl2-count (len fn-waiting)) (acl2-count expr))))
   (if (endp expr)
       (mv nil num)
     (mv-let (res-expr1 res-num1)
        (expand-fn-help (car expr) fn-lst fn-level-lst fn-waiting fn-extended num state)
        (mv-let (res-expr2 res-num2)
           (expand-fn-help-list (cdr expr) fn-lst fn-level-lst fn-waiting fn-extended res-num1 state)
           (mv (cons res-expr1 res-expr2) res-num2)))))

 ;; expand-fn-help
 ;; This function should keep three lists of function names.
 ;   First one stores all functions possible for expansion.
```

```
;   Second one is for functions to be expanded
;   and the third one is for functions already expanded.
;; They should be updated accordingly:
;   when one function is expanded along a specific path
;   that function should be deleted from fn-waiting and added
;   into fn-expanded.
;; Resursion detection:
;   When one function call is encountered
;   we want to make sure that function is valid for expansion
;   by looking at fn-lst. Then we expand it, delete it from
;   fn-waiting and add it onto fn-expanded. The we want to make
;   sure that fn-waiting and fn-expaned is changing as we walk
;   through the tree of code.
;; Another way of recursion detection:
;   One might want to use this simpler way of handling recrusion
;   detection. We note the length of fn-lst, then we want to
;   count down the level of expansion. Any path exceeding this
;   length is a sign for recursive call.
(defun expand-fn-help (expr fn-lst fn-level-lst fn-waiting fn-extended num state)
  "expand-fn-help: expand an expression"
  (declare (xargs :measure (list (acl2-count (len fn-waiting)) (acl2-count expr))))
  (cond ((atom expr) ;; base case, when expr is an atom
    (mv expr num))
   ((consp expr)
    (let ((fn0 (car expr)) (params (cdr expr)))
      (cond
        ((and (atom fn0) (exist fn0 fn-lst)) ;; function exists in the list
         (if (> (cadr (assoc fn0 fn-level-lst)) 0) ;; if fn0's level number is still larger than 0
        (mv-let (res fn-w-1 fn-e-1 fn-l-l-1 num2)
           (expand-a-fn fn0 fn-level-lst fn-waiting fn-extended num state) ;; expand a function
           (mv-let (res2 num3)
```

```
            (expand-fn-help res fn-lst fn-l-l-1 fn-w-1 fn-e-1 num2 state)
            (if (endp params)
                (mv res2 num3)
                (mv-let (res3 num4)
                   (expand-fn-help-list params fn-lst fn-level-lst fn-waiting fn-extended num3 state)
                   (mv (cons res2 res3) num4)))))
      (prog2$ (cw "Recursive function expansion level has reached 0: ~q0" fn0)
         (mv expr num))))
      ((atom fn0) ;; when expr is a un-expandable function
       (mv-let (res num2)
          (expand-fn-help-list (cdr expr) fn-lst fn-level-lst fn-waiting fn-extended num state)
          (mv (cons (car expr) res) num2)))
      ((lambdap fn0) ;; function is a lambda expression, expand the body
       (let ((lambdax fn0) (params (cdr expr)))
    (let ((formals (cadr lambdax)) (body (caddr lambdax)))
      (mv-let (res num2)
         (expand-fn-help body fn-lst fn-level-lst fn-waiting fn-extended num state)
         (mv-let (res2 num3)
            (expand-fn-help-list params fn-lst fn-level-lst fn-waiting fn-extended num2 state)
            (mv (cons (list 'lambda formals res) res2) num3))))))
      ((and (not (lambdap fn0)) (consp fn0))
       (mv-let (res num2)
          (expand-fn-help fn0 fn-lst fn-level-lst fn-waiting fn-extended num state)
          (mv-let (res2 num3)
             (expand-fn-help-list params fn-lst fn-level-lst fn-waiting fn-extended num2 state)
             (mv (cons res res2) num3))))
```

```
           (t (prog2$ (cw "Error(function): can not pattern match:
    ~q0" expr)
            (mv expr num)))
           )))
      (t (prog2$ (cw "Error(function): strange expression == ~q0"
    expr)
            (mv expr num)))))
)
)

(mutual-recursion

;; rewrite-formula-params
(defun rewrite-formula-params (expr let-expr)
  "rewrite-formula-params: a helper function for dealing with the
    param list of rewrite-formula function"
  (if (endp expr)
      nil
      (cons (rewrite-formula (car expr) let-expr)
       (rewrite-formula-params (cdr expr) let-expr))))

;; rewrite-formula
;; rewrite the formula according to given hypothesis and
    let-expression
(defun rewrite-formula (expr let-expr)
  "rewrite-formula rewrites an expression by replacing
    corresponding terms in the let expression"
  (cond ((atom expr) ;; if expr is an atom
    (let ((res-pair (assoc-equal expr let-expr)))
      (if (equal res-pair nil)
          expr
          (cadr res-pair))))
      ;; if expr is a consp
   ((consp expr)
    (let ((fn (car expr))
          (params (cdr expr)))
      (if (listp fn)
```

```
          ;; if first elem of expr is a list
          (cond
       ;; if it is a lambda expression
       ((lambdap fn)
        (let ((lambda-params (cadr fn))
         (lambda-body (caddr fn)))
          (let ((res-pair (assoc-lambda
                 lambda-body
                 (create-assoc lambda-params params)
                 let-expr)))
          (if (not (equal res-pair nil))
         (cadr res-pair)
         (cons (list 'lambda lambda-params (rewrite-formula
 lambda-body let-expr))
               (rewrite-formula-params params let-expr))))))
       ;; if it is a only a list, for handling nested list
       (t
        (cons (rewrite-formula fn let-expr)
         (rewrite-formula-params params let-expr))))
          ;; if first elem of expr is an atom
          (let ((res-pair (assoc-equal expr let-expr)))
       (if (not (equal res-pair nil))
           (cadr res-pair)
           (cons fn (rewrite-formula-params params let-expr)))))))
   ;; if expr is nil
   (t (cw "Error(function): nil expression."))))
)

;; extract-orig-param
(defun extract-orig-param (expr)
  (mv-let (decl-list hypo-list concl-list)
     (SMT-extract expr)
     (get-orig-param decl-list)))

;; augment-formula
(defun augment-formula (expr new-decl let-type new-hypo)
```

```
  "augment-formula: for creating a new expression with hypothesis
    augmented with new-hypo, assuming new-hypo only adds to the
    hypo-list"
  (mv-let (decl-list hypo-list concl-list)
     (SMT-extract expr)
     (list 'implies
          (list 'if
           (append-and-decl decl-list new-decl let-type)
           (append-and-hypo hypo-list new-hypo)
           ''nil)
          concl-list
          )))

;; reform-let
(defun reform-let (let-expr)
  "reform-let: reforms a let expression for convenient fetch"
  (let ((inverted-let-expr (invert-assoc let-expr)))
  (if (assoc-no-repeat inverted-let-expr)
      inverted-let-expr
      (cw "Error(function): there's repetition in the associate
    list's values ~q0" let-expr))))

;; initial-level-help
(defun initial-level-help (fn-lst fn-level)
  "initial-level-help: binding a level to each function for
    expansion. fn-lst is a list of functions, fn-level is the
    number of levels we want to expand the functions."
  (if (endp fn-lst)
      nil
      (cons (list (car fn-lst) fn-level)
       (initial-level-help (cdr fn-lst) fn-level))))

;; initial-level
(defun initial-level (fn-lst fn-level)
  "initial-level: binding a level to each function for expansion"
  (if (not (integerp fn-level))
      (initial-level-help fn-lst 1)
```

```
     (initial-level-help fn-lst fn-level)))

;; split-fn-from-type
(defun split-fn-from-type (fn-lst-with-type)
  ""
  (if (endp fn-lst-with-type)
      nil
      (cons (caar fn-lst-with-type)
       (split-fn-from-type (cdr fn-lst-with-type)))))

;; replace-a-rec-fn
(defun replace-a-rec-fn (expr fn-lst-with-type fn-var-decl num)
  ""(mv-let (name res-num)
     (create-name num)
     (prog2$ (cw "~q0" name
            ;;(cons (list name
      ;;   expr
   ;;      (cadr (assoc (car expr) fn-lst-with-type)))
        ;;  fn-var-decl)
        ;;res-num
            )
     (mv name
         (cons (list name
           expr
           (cadr (assoc (car expr) fn-lst-with-type)))
          fn-var-decl)
         res-num))))

(mutual-recursion

 ;; replace-rec-fn-params
(defun replace-rec-fn-params (expr fn-lst-with-type fn-var-decl num)
  ""
  (if (endp expr)
      (mv expr fn-var-decl num)
     (mv-let (res-expr1 res-fn-var-decl1 res-num1)
        (replace-rec-fn (car expr) fn-lst-with-type fn-var-decl num)
```

```
        (mv-let (res-expr2 res-fn-var-decl2 res-num2)
           (replace-rec-fn-params (cdr expr) fn-lst-with-type res-fn-var-decl1 res-num1)
           (mv (cons res-expr1 res-expr2)
          res-fn-var-decl2
          res-num2)))))

;; replace-rec-fn
;; 2014-07-04
;; added function for postorder traversal
(defun replace-rec-fn (expr fn-lst-with-type fn-var-decl num)
  ""
  (cond ((atom expr)
    (mv expr fn-var-decl num))
   ((consp expr)
    (let ((fn0 (car expr)) (params (cdr expr)))
       (cond
         ((and (atom fn0) (not (endp (assoc fn0 fn-lst-with-type)))) ;; function exists in the list
          (prog2$ (cw "fn-lst-with-type: ~q0" fn-lst-with-type)
          (mv-let (res fn-var-decl2 num2)
             (replace-a-rec-fn expr fn-lst-with-type fn-var-decl num)
             (prog2$ (cw "res: ~q0 fn-var-decl2: ~q1, num2: ~q2" res fn-var-decl2 num2)
             (mv res fn-var-decl2 num2)))))
         ((atom fn0) ;; when expr is a un-expandable function
          (mv-let (res fn-var-decl2 num2)
             (replace-rec-fn-params params fn-lst-with-type fn-var-decl num)
             (mv (cons fn0 res) fn-var-decl2 num2)))
         ((lambdap fn0) ;; function is a lambda expression, expand the body
          (let ((lambdax fn0) (params (cdr expr)))
       (let ((formals (cadr lambdax)) (body (caddr lambdax)))
         (mv-let (res fn-var-decl2 num2)
            (replace-rec-fn body fn-lst-with-type fn-var-decl num)
```

```
            (mv-let (res2 fn-var-decl3 num3)
               (replace-rec-fn-params params fn-lst-with-type
   fn-var-decl2 num2)
               (mv (cons (list 'lambda formals res) res2)
                   fn-var-decl3
                   num3))))
      ))
        ((and (not (lambdap fn0)) (consp fn0))
         (mv-let (res fn-var-decl2  num2)
            (replace-rec-fn fn0 fn-lst-with-type fn-var-decl num)
            (mv-let (res2 fn-var-decl3 num3)
               (replace-rec-fn-params params fn-lst-with-type
   fn-var-decl2 num2)
               (mv (cons res res2) fn-var-decl3 num3)))))
        (t (prog2$ (cw "Error(function): Can not pattern match,
   ~q0" expr)
         (mv expr fn-var-decl  num)))
        )))
  (t (prog2$ (cw "Error(function): Strange expr, ~q0" expr)
        (mv expr fn-var-decl  num)))))

)

;; expand-fn
(defun expand-fn (expr fn-lst-with-type fn-level let-expr let-type
   new-hypo state)
  "expand-fn: takes an expr and a list of functions, unroll the
   expression. fn-lst is a list of possible functions for
   unrolling."
  (let ((fn-lst (split-fn-from-type fn-lst-with-type)))
  (let ((reformed-let-expr (reform-let let-expr)))
    (let ((fn-level-lst (initial-level fn-lst fn-level)))
      (mv-let (res-expr1 res-num1)
        (expand-fn-help (rewrite-formula expr reformed-let-expr)
              fn-lst fn-level-lst fn-lst nil 0 state)
        (mv-let (res-expr res-fn-var-decl res-num)
           (replace-rec-fn res-expr1 fn-lst-with-type nil res-num1)
```

```
          (let ((rewritten-expr
            (augment-formula (rewrite-formula res-expr reformed-let-expr)
                   (assoc-get-value reformed-let-expr)
                   let-type
                   new-hypo)))
             (let ((res (rewrite-formula res-expr1 reformed-let-expr)))
          (let ((expr-return ;; (augment-formula res
                 ;;        (assoc-get-value reformed-let-expr)
                 ;;        let-type
                 ;;        new-hypo)
                 res
             )
                (orig-param (extract-orig-param res)))
            (prog2$ (cw "~q0~%~q1~%" rewritten-expr expr-return)
            (mv rewritten-expr expr-return res-num orig-param res-fn-var-decl))))))))))
```

```
;; SMT-formula contains functions for constructing a SMT formula in ACL2
(in-package "ACL2")

;; -------------- SMT-operator  -----------:
(defun operator-list (opr)
  "operator-list: an associate list with possible SMT operators"
  (assoc opr '((binary-+ binary-+ 0)
          (binary-- binary-- 2)
          (binary-* binary-* 0)
          (unary-/ unary-/ 1)
          (unary-- unary-- 1)
          (equal equal 2)
          (> > 2)
          (>= >= 2)
          (< < 2)
          (<= <= 2)
          (if if 3)
```

```
          (not not 1)
          (lambda lambda 2)
          ;; (list list 0)
          ;; (nth nth 2)
          (implies implies 2)
          (integerp integerp 1)
          (rationalp rationalp 1)
          (booleanp booleanp 1)
          (my-floor my-floor 1))))

(defun is-SMT-operator (opr)
  "is-SMT-operator: given an operator in ACL2 format, check if it's
    valid"
    (if (equal (operator-list opr) nil)
   nil
      t))

;; SMT-operator
(defun SMT-operator (opr)
  "SMT-operator: given an operator in ACL2 format, establish its
    ACL2 format by looking up the associated list"
  (if (is-SMT-operator opr)
      (cadr (operator-list opr))
    (prog2$ (cw "Error(formula): Operator ~q0 does not exist!" opr)
       nil)))

;; --------------------- SMT-type -------------------------:

;; is-SMT-type
(defun is-SMT-type (type)
  "SMT-type: given a type in ACL2 format, check if it's valid"
    (if (or (equal type 'RATIONALP)
       (equal type 'INTEGERP)
       (equal type 'BOOLEANP))
   t
      nil))

```

```
;; SMT-type
(defun SMT-type (type)
  "SMT-type: given a type in ACL2 format, establish its ACL2 format
    by looking up the associated list"
  (if (is-SMT-type type)
      type
    (prog2$ (cw "Error(formula): Type ~q0 not supported!" type)
       nil)))

;; --------------------- SMT-number -------------------------:

;; is-SMT-rational
(defun is-SMT-rational (number)
  "is-SMT-rational: Check if this is a SMT rational number"
  (if (and (rationalp number)
      (not (integerp number)))
      t
    nil))

;; is-SMT-integer
(defun is-SMT-integer (number)
  "is-SMT-integer: Check if this is a SMT integer number"
  (if (integerp number)
      t
    nil))

;; is-SMT-number
(defun is-SMT-number (number)
  "is-SMT-number: Check if this is a SMT number"
  (if (or (is-SMT-rational number)
     (is-SMT-integer number))
      t
    nil))

;; SMT-number
(defun SMT-number (number)
  "SMT-number: This is a SMT number"
```

```
  (if (is-SMT-number number)
      number
    (cw "Error(formula): This is not a valid SMT number: ~q0"
   number)))

;; --------------------- SMT-variable -------------------------:
;; Q: I want to add a check on possible SMT-variables.

;; is-SMT-variable
(defun is-SMT-variable (var)
  "is-SMT-variable: check if a variable is a SMT var"
  (if (symbolp var) t nil))

;; SMT-variable
(defun SMT-variable (var)
  "SMT-variable: This is a SMT variable name"
  (if (is-SMT-variable var)
      var
    (cw "Error(formula): This is not a valid SMT variable name:
   ~q0" var)))

;; --------------------- SMT-constant -------------------------:

;; is-SMT-constant-name
(defun is-SMT-constant-name (name)
  "is-SMT-constant-name: Check if this is a SMT constant name"
  (if (symbolp name) t nil))

;; SMT-constant-name
(defun SMT-constant-name (name)
  "SMT-constant-name: This is a SMT constant name"
  (if (is-SMT-constant-name name)
      name
    (cw "Error(formula): This is not a valid SMT constant name:
   ~q0" name)))

;; SMT-constant
```

```
(defun SMT-constant (constant)
  "SMT-constant: This is a SMT constant declaration"
  (if (not (equal (len constant) 2))
      (cw "Error(formula): Wrong number of elements in a constant
    declaration list: ~q0" constant)
    (let ((name (car constant))
     (value (cadr constant)))
      (list (SMT-constant-name name) (SMT-number value)))))

;; SMT-constant-list-help
(defun SMT-constant-list-help (constant-list)
  "SMT-constant-list: This is a list of SMT constant declarations,
    the helper function"
  (if (consp constant-list)
      (cons (SMT-constant (car constant-list))
    (SMT-constant-list-help (cdr constant-list)))
    nil))

;; SMT-constant-list
(defun SMT-constant-list (constant-list)
  "SMT-constant-list: This is a list of SMT constant declarations"
  (if (not (listp constant-list))
      (cw "Error(formula): The SMT constant list is not in the
    right form: ~q0" constant-list)
    (SMT-constant-list-help constant-list)))

;; --------------------- SMT-declaration -------------------------:

;; SMT-declaration
(defun SMT-declaration (decl)
  "SMT-declaration: This is a SMT variable declaration"
  (if (not (equal (len decl) 2))
      (cw "Error(formula): Wrong number of elements in a variable
    declaration list: ~q0" decl)
    (let ((type (car decl))
     (name (cadr decl)))
      (list (SMT-type type) (SMT-variable name)))))
```

```
;; SMT-declaration-list-help
(defun SMT-declaration-list-help (decl-list)
  "SMT-declaration-list-help: This is a list of SMT variable declarations, the helper function"
  (if (consp decl-list)
      (cond ((equal (car decl-list) 'if)
        (cons (SMT-declaration (cadr decl-list))
         (SMT-declaration-list-help (caddr decl-list))))
       (t (cons (SMT-declaration decl-list)
           nil)))
    nil))

;; SMT-declaration-list
(defun SMT-declaration-list (decl-list)
  "SMT-decl-list: This is a list of SMT variable declarations"
  (if (not (listp decl-list))
      (cw "Error(formula): The SMT declaration list is not in the right form: ~q0" decl-list)
    (SMT-declaration-list-help decl-list)))

;; --------------------- SMT-expression -------------------------:

(mutual-recursion

;; SMT-lambda-formal
(defun SMT-lambda-formal (formal)
  "SMT-lambda-formal: check if it's a valid formal list for a lambda expression"
  (if (endp formal)
      nil
    (if (symbolp (car formal))
   (cons (car formal)
         (SMT-lambda-formal (cdr formal)))
      (cw "Error(formula): not a valid symbol in a formal list ~q0" (car formal)))))

```

```
;; SMT-expression-long
(defun SMT-expression-long (expression)
  "SMT-expression-long: recognize a SMT expression, in a SMT
    expression's parameters"
  (if (consp expression)
      (cons (SMT-expression (car expression))
       (SMT-expression-long (cdr expression)))
    nil))

;; SMT-expression
(defun SMT-expression (expression)
  "SMT-expression: a SMT expression in ACL2"
  (if (consp expression)
      (cond ((and (consp (car expression))
        (is-SMT-operator (caar expression))
        (equal (caar expression) 'lambda))
        (cons (list (SMT-operator
           (car (car expression)))
          (SMT-lambda-formal
           (cadr (car expression)))
          (SMT-expression
           (caddr (car expression))))
         (SMT-expression-long (cdr expression))))
       ((is-SMT-operator (car expression))
        (cons (SMT-operator (car expression))
         (SMT-expression-long (cdr expression))))
       ;; for handling a list
       ((equal (car expression) 'QUOTE)
        (if (consp (cadr expression))
       (cons 'list
             (SMT-expression-long (cadr expression)))
       (SMT-expression (cadr expression))))
       (t (cw "Error(formula): This is not a valid operator: ~q0"
    expression)))
    (cond ((is-SMT-number expression) (SMT-number expression))
     ((is-SMT-variable expression) (SMT-variable expression))
```

```
     (t (cw "Error(formula): Invalid number or variable: ~q0"
    expression)))))
)

;; --------------------- SMT-hypothesis -------------------------:

;; SMT-hypothesis-list
(defun SMT-hypothesis-list (hyp-list)
  "SMT-hypothesis-list: This is a SMT hypothesis list"
  (if (not (listp hyp-list))
      (cw "Error(formula): The SMT hypothesis list is not in the
    right form: ~q0" hyp-list)
    (SMT-expression hyp-list)))

;; --------------------- SMT-conclusion -------------------------:

;; SMT-conclusion-list
(defun SMT-conclusion-list (concl-list)
  "SMT-conclusion-list: This is a SMT conclusion list"
  (if (not (listp concl-list))
      (cw "Error(formula): The SMT conclusion list is not in the
    right form: ~q0" concl-list)
    (SMT-expression concl-list)))
;; --------------------- SMT-formula ----------------------------:

;; SMT-formula
(defun SMT-formula (const-list
          decl-list
          hyp-list
          concl-list)
  "SMT-formula: This is a SMT formula"
  (list (SMT-constant-list const-list)
   (SMT-declaration-list decl-list)
   (SMT-hypothesis-list hyp-list)
   (SMT-conclusion-list concl-list))
  )

```

```
;; SMT-formula-top
(defmacro SMT-formula-top (&key const-list
            decl-list
            hyp-list
            concl-list)
  "SMT-formula-top: This is a macro for fetching parameters of a
    SMT formula"
  (list 'quote (SMT-formula const-list
             decl-list
             hyp-list
             concl-list))
  )
```

```
;; translate-SMT-formula translate a SMT formula in ACL2 into Z3
    python code
(in-package "ACL2")
(include-book "SMT-formula")
(include-book "helper")

;; -------------- translate operator  -----------:

;; translate-operator-list
(defun translate-operator-list (opr)
  "translate-operator-list: look up an associate list for the
    translation"
  (assoc opr '((binary-+ "s.plus" 0)
          (binary-- "s.minus" 2)
          (binary-* "s.times" 0)
          (unary-/ "s.reciprocal" 1)
          (unary-- "s.negate" 1)
          (equal "s.equal" 2)
          (> "s.gt" 2)
          (>= "s.ge" 2)
          (< "s.lt" 2)
          (<= "s.le" 2)
          (if "s.ifx" 3)
          (not "s.notx" 1)
```

```
          (lambda "lambda" 2)
          ;; (nth "s.nth" 2)
          ;; (list "s.array" 0)
          (implies "s.implies" 2)
          (integerp "s.integerp" 1)
          (rationalp "s.rationalp" 1)
          (booleanp "s.booleanp" 1)
          (my-floor "s.floor" 1))))

;; translate-operator
(defun translate-operator (opr)
  "translate-operator: given an operator in ACL2 format, translate into its Z3 format by looking up the associated list"
  (let ((result (translate-operator-list opr)))
    (if (equal result nil)
   (prog2$ (cw "Error(translator): Operator ~q0 does not exist!" opr)
      nil)
      (cadr result))))

;; ----------------------- translate-type -----------------------------:

;; translate-type-list
(defun translate-type-list (type)
  "translate-type-list: look up an associate list for the translation"
  (assoc type '((RATIONALP "s.isReal")
      (INTEGERP "s.isReal")
      (BOOLEANP "s.isBool"))))

;; translate-type
(defun translate-type (type)
  "translate-type: translates a type in ACL2 SMT-formula into Z3 type"     ;; all using reals because Z3 is not very good at mixed types
  (let ((result (translate-type-list type)))
```

```
    (if (equal result nil)
   (prog2$ (cw "Error(translator): Type ~q0 does not exist!" type)
     nil)
     (cadr result))))

;; ----------------------- translate-number -----------------------------:

;; translate-number
(defun translate-number (num)
  "translate-number: translates ACL2 SMT-number into a Z3 number"
  (if (is-SMT-rational num)
      (list "Q(" (numerator num) "," (denominator num) ")")
    (if (is-SMT-integer num)
   num
      (cw "Error(translator): Cannot translate an unrecognized number: ~q0" num))))

;; ----------------------- translate-variable ---------------------------:

;; translate-variable
(defun translate-variable (var)
  "translate-variable: transalte a SMT variable into Z3 variable"
  (if (is-SMT-variable var)
      var
    (cw "Error(translator): Cannot translate an unrecognized variable: ~q0" var)))

;; ----------------------- translate-constant ---------------------------:

;; translate-const-name
(defun translate-const-name (const-name)
  "translate-const-name: translate a SMT constant name into Z3"
  (subseq
   (coerce (symbol-name const-name) 'list)
```

```
  1 (1- (len const-name))))

;; translate-constant
(defun translate-constant (const)
  "translate-constant: translate a SMT constant definition into Z3 code"
  (list (translate-const-name (car const)) '= (translate-number (cadr const))))

;; translate-constant-list
(defun translate-constant-list (const-list)
  "translate-constant-list: translate a SMT constant list in ACL2 into a Z3 line of code"
  (if (consp const-list)
      (cons (translate-constant (car const-list))
       (cons #\Newline (translate-constant-list (cdr const-list))))
    nil))

;; ;; check-const
;; (defun check-const (expr)
;;   "check-const: check to see if an expression is a constant"
;;   (if (and (atom expr)
;;       (let ((expr-list (coerce (symbol-name expr) 'list)))
;;         (and (equal #\* (car expr-list))
;;         (equal #\* (nth (1- (len expr-list)) expr-list)))))
;;       t
;;     nil))

;; ;; get-constant-list-help
;; (defun get-constant-list-help (expr const-list)
;;   "get-constant-list-help: check all constants in a clause"
;;   (cond
;;     ( (consp expr)
;;       (let ((const-list-2 (get-constant-list-help (car expr) const-list)))
;;    (get-constant-list-help (cdr expr) const-list-2))
;;     )
```

```
;;      ( (check-const expr)
;;        (mv-let (keyword name value)
;;           (pe expr) ;; pe will not be working for this
;;           (cons (list expr (translate-number value)) const-list))
;;        )
;;      ( (atom expr)
;;        (get-constant-list-help (cdr expr) const-list)
;;        )
;;      ( t
;;        const-list
;;        )
;;      )
;;    )

;; ;; get-constant-list
;; (defun get-constant-list (expr)
;;   "get-constant-list: get the list of constants in an associate
    list"
;;   (get-constant-list-help expr '()))


;; ---------------------- translate-declaration
    --------------------------:

;; translate-declaration
(defun translate-declaration (decl)
  "translate-declaration: translate a declaration in ACL2 SMT
    formula into Z3 declaration"
  (let ((type (car decl))
   (name (cadr decl)))
    (list (translate-variable name) '= (translate-type type) '\(
    '\" (translate-variable name) '\" '\))))

;; translate-declaration-list
(defun translate-declaration-list (decl-list)
 "translate-declaration-list: translate a list of SMT-formula
    declarations into Z3 code"
```

```
  (if (consp decl-list)
      (cons (translate-declaration (car decl-list))
       (cons #\Newline (translate-declaration-list (cdr decl-list))))
    nil))

;; ----------------------- translate-expression --------------------------:

;; make-lambda-list
(defun make-lambda-list (lambda-list)
  "make-lambda-list: translating the binding list of a lambda expression"
  (if (endp (cdr lambda-list))
      (car lambda-list)
    (cons (car lambda-list)
     (cons '\, (make-lambda-list (cdr lambda-list))))))

(skip-proofs
(mutual-recursion

;; translate-expression-long
(defun translate-expression-long (expression)
  "translate-expression-long: translate a SMT expression's parameters in ACL2 into Z3 expression"
  (if (endp (cdr expression))
      (translate-expression (car expression))
    (cons (translate-expression (car expression))
     (cons '\,
      (translate-expression-long
       (cdr expression))))))

;; stuff.let(['x', 2.0], ['y', v('a')*v('b') + v('c')], ['z', ...]).inn(2*v('x') - v('y'))
;; translate-expression
(defun translate-expression (expression)
```

```
"translate-expression: translate a SMT expression in ACL2 to Z3
  expression"
(if (and (not (equal expression nil))
    (consp expression)
    (not (equal expression ''1)))
    (cond ((and (consp (car expression))
      (is-SMT-operator (caar expression))
      ;; special treatment for let expression
      (equal (caar expression) 'lambda))
      (list '\(
       (translate-operator (caar expression))
       #\Space
       (if (endp (cadr (car expression)))
           #\Space
           (make-lambda-list (cadr (car expression))))
       '\:
       (translate-expression (caddr (car expression)))
       '\) '\(
       (if (endp (cdr expression))
           #\Space
           (translate-expression-long (cdr expression)))
       '\)))
     ;; ((and (is-SMT-operator (car expression))
     ;;     (equal (car expression) 'list))
     ;;  (list (translate-operator (car expression))
     ;;      '\( '\[
     ;;      (translate-expression-long (cdr expression))
     ;;      '\] '\)))
     ((is-SMT-operator (car expression))
      (list (translate-operator (car expression))
       '\(
       (translate-expression-long (cdr expression))
       '\)))
     (t (list "s.unknown" '\( (translate-expression-long (cdr
  expression)) '\))))
  (cond ((is-SMT-number expression)
    (translate-number expression))
```

```
    ((equal expression 'nil) "False") ;; what if when 'nil is a list?
    ((equal expression 't) "True")
    ((is-SMT-variable expression)
     (translate-variable expression))
    (t (cw "Error(translator): Invalid number or variable: ~q0" expression)))))
)
)
;; ----------------------- translate-hypothesis --------------------------:

;; translate-hypothesis-list
(defun translate-hypothesis-list (hyp-list)
  "translate-hypothesis-list: translate a SMT-formula hypothesis statement into Z3"
  (list (cons "hypothesis"
        (cons '= (translate-expression hyp-list))) #\Newline))

;; ----------------------- translate-conclusion --------------------------:
;; translate-conclusion-list
(defun translate-conclusion-list (concl-list)
  "translate-conclusion-list: translate a SMT-formula conclusion statement into Z3"
  (list (cons "conclusion"
        (cons '= (translate-expression concl-list))) #\Newline))

;; ----------------------- translate-theorem --------------------------:
;; translate-theorem
(defun translate-theorem ()
  "translate-theorem: construct a theorem statement for Z3"
  (list "s.prove(hypothesis, conclusion)" #\Newline))

;; ----------------------- translate-SMT-formula --------------------------:
```

```
;; translate-SMT-formula
(defun translate-SMT-formula (formula)
  "translate-SMT-formula: translate a SMT formula into its Z3 code"
  (let (;(const-list (car formula))
   (decl-list (cadr formula))
   (hypo-list (caddr formula))
   (concl-list (cadddr formula)))
    (list ;;(translate-constant-list
     ;;  (get-constant-list formula))
     (translate-declaration-list decl-list)
     (translate-hypothesis-list hypo-list)
     (translate-conclusion-list concl-list)
     (translate-theorem))))
```

```
(in-package "ACL2")
(include-book "./helper")
(include-book "./SMT-run")
(include-book "./SMT-interpreter")
(include-book "./SMT-function")
(include-book "./SMT-translator")
(defttag :tshell)
(value-triple (tshell-ensure))
(set-state-ok t)
(set-ignore-ok t)
(program)

(mutual-recursion
;; lisp-code-print-help
(defun lisp-code-print-help (lisp-code-list indent)
  "lisp-code-print-help: make a printable lisp code list"
  (if (endp lisp-code-list)
      nil
    (list #\Space
     (lisp-code-print (car lisp-code-list) indent)
     (lisp-code-print-help (cdr lisp-code-list) indent))))

```

```
;; lisp-code-print: make printable lisp list
(defun lisp-code-print (lisp-code indent)
  "lisp-code-print: make a printable lisp code list"
  (cond ((equal lisp-code 'nil) "nil") ;;
   ((equal lisp-code 'quote) "'") ;; quote
   ((atom lisp-code) lisp-code)
   ((and (equal 2 (length lisp-code))
         (equal (car lisp-code) 'quote))
    (cons "'"
          (lisp-code-print (cadr lisp-code)
            (cons #\Space
                  (cons #\Space indent)))))
   (t
    (list #\Newline indent '\(
          (cons (lisp-code-print (car lisp-code)
                  (cons #\Space
                   (cons #\Space indent)))
           (lisp-code-print-help (cdr lisp-code)
                  (cons #\Space
                   (cons #\Space indent))))
          '\) ))))
)

;; my-prove-SMT-formula
(defun my-prove-SMT-formula (term)
  "my-prove-SMT-formula: check if term is a valid SMT formula"
  (let ((decl-list (cadr (cadr term)))
   (hypo-list (caddr (cadr term)))
   (concl-list (caddr term)))
    (SMT-formula '()
       decl-list
       hypo-list
       concl-list)))

;; my-prove-write-file
(defun my-prove-write-file (term fdir state)
  "my-prove-write-file: write translated term into a file"
```

```
  (write-SMT-file fdir
        (translate-SMT-formula
         (my-prove-SMT-formula term))
        state))

;; my-prove-write-expander-file
(defun my-prove-write-expander-file (expanded-term fdir state)
  "my-prove-write-expander-file: write expanded term into a log file"
  (write-expander-file fdir
             expanded-term
             state))

;; create-level
(defun create-level (level index)
  "create-level: creates a name for a level"
  (intern-in-package-of-symbol
   (concatenate 'string level (str::natstr index)) 'ACL2))

;; my-prove-build-log-file
(defun my-prove-build-log-file (expanded-term-list index)
  "my-prove-build-log-file: write the log file for expanding the functions"
  (if (endp expanded-term-list)
      nil
      (cons (list (create-level "level " index) '\:
        (lisp-code-print
         (car expanded-term-list) '())
        #\Newline #\Newline)
       (my-prove-build-log-file
        (cdr expanded-term-list) (1+ index)))))

;; translate added hypothesis to underling representation
(defun translate-hypo (hypo state)
  "translate-hypo: translate added hypothesis to underling representation"
  (if (endp hypo)
```

```
      (mv nil state)
      (mv-let (res1 state)
         (translate-hypo (cdr hypo) state)
         (mv-let (erp res state)
            (translate (car hypo) t nil t nil (w state) state)
            (if (endp res)
           (mv (cons (car hypo) res1) state)
           (mv (cons res res1) state)))
         )))

;; translate a let binding for added hypothesis
(defun translate-let (let-expr state)
  "translate-let: translate a let binding for added hypo"
  (if (endp let-expr)
      (mv nil state)
      (mv-let (res1 state)
         (translate-let (cdr let-expr) state)
         (mv-let (erp res state)
            (translate (cadar let-expr) t nil t nil (w state) state)
            (if (endp res)
           (mv (cons (list (caar let-expr) (cadar let-expr) (caddar
    let-expr)) res1) state)
           (mv (cons (list (caar let-expr) res (caddar let-expr))
    res1) state)))
         )))

;; get-hint-formula
(defun get-hint-formula (name state)
  "get-hint-formula: get the formula by a hint's name"
  (formula name t (w state)))

;; add-hints
(defun add-hints (hints clause state)
  "add-hints: add a list of hint to a clause, in the form of ((not
    hint3) ((not hint2) ((not hint1) clause)))"
  (if (endp hints)
      clause
```

```
      (add-hints (cdr hints)
       (cons (list 'not (get-hint-formula (car hints) state)) clause)
       state)))

;; construct augmented result
(defun augment-hypothesis-helper (rewritten-term let-expr orig-param main-hints state)
  "augment-hypothesis: augment the returned clause with \
new hypothesis in lambda expression"
  (cond ((and (endp let-expr) (endp main-hints))
    (list (list 'not rewritten-term)))
   ((and (endp main-hints) (not (endp let-expr)))
    (list (list 'not
      (cons (list 'lambda (append (assoc-get-key let-expr) orig-param) rewritten-term)
            (append (assoc-get-value let-expr) orig-param)))))
   ((and (not (endp main-hints)) (endp let-expr))
    (add-hints main-hints (list (list 'not rewritten-term)) state))
   (t
    (add-hints main-hints
          (list (list 'not
            (cons (list 'lambda (append (assoc-get-key let-expr) orig-param) rewritten-term)
                  (append (assoc-get-value let-expr) orig-param))))
          state))
   ))

(defun add-aux (clause aux-thms)
  (if (endp aux-thms)
      clause
      (add-aux (let ((thm (car aux-thms)))
       (cons (list 'not
              (list 'implies (cadar thm) (cadr thm)))
             clause))
           (cdr aux-thms)
           )))
```

```
(defun augment-hypothesis (rewritten-term let-expr orig-param
    main-hints aux-thms state)
  (prog2$ (cw "aux-thms: ~q0~%" aux-thms)
  (let ((res (augment-hypothesis-helper rewritten-term let-expr
    orig-param main-hints state)))
    (add-aux res aux-thms))))

;;separate-type
(defun separate-type (let-expr)
  "separate-type: separate let expression types from let
    expression, I do it in this way for convenience. I might want
    to use them as a whole in the future."
  (if (endp let-expr)
      (mv nil nil)
      (mv-let (res-let-expr res-let-type)
         (separate-type (cdr let-expr))
         (mv (cons (list (caar let-expr) (cadar let-expr))
         res-let-expr)
        (cons (caddar let-expr)
         res-let-type)))))

(defun create-type-theorem-helper-no-hints (decl-hypo-list let-expr
    let-type)
  (if (endp let-expr)
      nil
      (cons (list (list 'not
         (list 'if (cadr decl-hypo-list)
                (append-and-hypo (caddr decl-hypo-list)
                       (list (list 'equal (caar let-expr) (cadar
    let-expr))))
                ''nil))
        (list (car let-type) (caar let-expr)))
       (create-type-theorem-helper-no-hints decl-hypo-list (cdr
    let-expr) (cdr let-type)))))
```

```
(defun create-type-theorem-helper-with-hints (decl-hypo-list
    let-expr let-type let-hints state)
  (if (endp let-expr)
      nil
      (cons (add-hints (car let-hints)
             (list (list 'not
               (list 'if (cadr decl-hypo-list)
                (append-and-hypo (caddr decl-hypo-list)
                       (list (list 'equal (caar let-expr) (cadar
    let-expr))))
                ''nil))
              (list (car let-type) (caar let-expr)))
             state)
       (create-type-theorem-helper-with-hints decl-hypo-list (cdr
    let-expr) (cdr let-type) (cdr let-hints) state))))


;; create-type-theorem
(defun create-type-theorem (decl-hypo-list let-expr let-type
    let-hints state)
  "create-type-theorem"
  (if (endp let-hints)
      (create-type-theorem-helper-no-hints decl-hypo-list let-expr
    let-type)
      (create-type-theorem-helper-with-hints decl-hypo-list
    let-expr let-type let-hints state)))

(defun create-hypo-theorem-helper-no-hints (decl-hypo-list let-expr
    hypo-expr orig-param)
  (if (endp hypo-expr)
      nil
      (cons (list (list 'not decl-hypo-list)
        (cons (list 'lambda (append (assoc-get-key let-expr)
    orig-param) (car hypo-expr))
         (append (assoc-get-value let-expr) orig-param)))
       (create-hypo-theorem-helper-no-hints decl-hypo-list let-expr
    (cdr hypo-expr) orig-param))))
```

```
(defun create-hypo-theorem-helper-with-hints (decl-hypo-list
    let-expr hypo-expr orig-param hypo-hints state)
  (if (endp hypo-expr)
      nil
      (cons (add-hints (car hypo-hints)
        (list (list 'not decl-hypo-list)
         (cons (list 'lambda (append (assoc-get-key let-expr)
    orig-param) (car hypo-expr))
          (append (assoc-get-value let-expr) orig-param)))
        state)
       (create-hypo-theorem-helper-with-hints decl-hypo-list
    let-expr (cdr hypo-expr) orig-param (cdr hypo-hints) state))))

;; create-hypo-theorem
(defun create-hypo-theorem (decl-hypo-list let-expr hypo-expr
    orig-param hypo-hints state)
  "create-hypo-theorem: create a theorem for proving user added
    hypothesis"
  (if (endp hypo-hints)
      (create-hypo-theorem-helper-no-hints decl-hypo-list let-expr
    hypo-expr orig-param)
      (create-hypo-theorem-helper-with-hints decl-hypo-list
    let-expr hypo-expr orig-param hypo-hints state)))

;;create-fn-type-theorem
(defun create-fn-type-theorem (decl-hypo-list fn-var-decl)
  ""
  (if (endp fn-var-decl)
      nil
      (cons (list (list 'not
         (list 'if (cadr decl-hypo-list)
               (append-and-hypo (caddr decl-hypo-list)
                      (list (list 'equal (caar fn-var-decl) (cadar
    fn-var-decl))))
               ''nil))
        (list (caddar fn-var-decl) (caar fn-var-decl)))
```

```
      (create-fn-type-theorem decl-hypo-list (cdr fn-var-decl)))))

;;add-fn-var-decl-helper
(defun add-fn-var-decl-helper (decl-term fn-var-decl)
  ""
  (if (endp fn-var-decl)
      decl-term
      (list 'if
       (list (caddar fn-var-decl) (caar fn-var-decl))
       (add-fn-var-decl-helper decl-term (cdr fn-var-decl))
       ''nil)))

;;add-fn-var-decl
(defun add-fn-var-decl (term fn-var-decl)
  ""
  (list (car term)
   (list (caadr term)
         (add-fn-var-decl-helper (cadadr term) fn-var-decl)
         (caddr (cadr term))
         (cadddr (cadr term)))
   (caddr term)))

;; my-prove
(defun my-prove (term fn-lst fn-level fname let-expr new-hypo
    let-hints hypo-hints main-hints state)
  "my-prove: return the result of calling SMT procedure"
  (let ((file-dir (concatenate 'string
                  *dir-files*
                  "/"
                  fname
                  ".py"))
   (expand-dir (concatenate 'string
               *dir-expanded*
               "/"
               fname
               "\_expand.log")))
    (mv-let (hypo-translated state)
```

```
      (translate-hypo new-hypo state)
   (mv-let (let-expr-translated-with-type state)
      (translate-let let-expr state)
     (mv-let (let-expr-translated let-type)
        (separate-type let-expr-translated-with-type)
        (mv-let (expanded-term-list-1 expanded-term-list-2 num
orig-param fn-var-decl)
           (expand-fn term fn-lst fn-level let-expr-translated
let-type hypo-translated state)
           (let ((expanded-term-list
              (add-fn-var-decl expanded-term-list-1 fn-var-decl)))
              (prog2$ (cw "Expanded(SMT-z3): ~q0 Final index
number: ~q1" expanded-term-list num)
                 (let ((state (my-prove-write-expander-file
                     (my-prove-build-log-file
                      (cons term expanded-term-list) 0)
                     expand-dir
                     state)))
                    (let ((state (my-prove-write-file
                        expanded-term-list
                        file-dir
                        state)))
                 (let ((type-theorem (create-type-theorem (cadr
term)
                                 let-expr-translated
                                 let-type
                                 let-hints
                                 state))
                       (hypo-theorem (create-hypo-theorem (cadr
term)
                                 let-expr-translated
                                 hypo-translated
                                 orig-param
                                 hypo-hints
                                 state))
                       (fn-type-theorem (create-fn-type-theorem
(cadr term)
```

```
                                          fn-var-decl)))
                     (let ((aug-theorem (augment-hypothesis expanded-term-list-2
                                    let-expr-translated
                                    orig-param
                                    main-hints
                                    (append fn-type-theorem
                                      (append hypo-theorem
                                        (append type-theorem)))
                                    state)))
                     (if (car (SMT-interpreter file-dir))
                         (mv t aug-theorem type-theorem hypo-theorem fn-type-theorem state)
                         (mv nil aug-theorem type-theorem hypo-theorem fn-type-theorem state)))))))))))))
```

```
;; SMT-run writes to Z3, invoke Z3 and gets the result
(in-package "ACL2")

(include-book "./config")
(include-book "std/io/top" :dir :system)
(include-book "centaur/misc/tshell" :dir :system)
(defttag :tshell)
(value-triple (tshell-ensure))

;;(set-print-case :downcase state)

(set-state-ok t)
(defttag :writes-okp)

;; princ$-list-of-strings
(defun princ$-list-of-strings (alist channel state)
  "princ$-list-of-strings: the real function to print the Z3 program."
  (if (consp alist)
    (let ((state (princ$-list-of-strings (car alist) channel state)))
```

```
      (princ$-list-of-strings (cdr alist) channel state))
    (if (and (not (equal alist nil))
        (not (acl2-numberp alist)))   ;; if alist is a number, should be treated seperately
      (princ$ (string alist) channel state)
      (if (acl2-numberp alist)
        (princ$ alist channel state)
        state))))

;; coerce a list of strings and characters into a string
(defun coerce-str-and-char-to-str (slist)
  "coerce-str-and-char-to-str: coerce a list of strings and characters into a string"
  (if (endp slist)
      nil
    (cond ((stringp (car slist))
      (concatenate 'string
         (car slist)
         (coerce-str-and-char-to-str (cdr slist))))
     ((characterp (car slist))
      (concatenate 'string
         (coerce (list (car slist)) 'STRING)
         (coerce-str-and-char-to-str (cdr slist))))
     (t (cw "Error(run): Invalid list ~q0." (car slist))))))

;; write-head
(defun write-head ()
  "write-head: writes the head of a z3 file"
  (coerce-str-and-char-to-str
   (list "from sys import path"
    #\Newline
    "path.insert(0,\"" *dir-interface* "\")"
    #\Newline
    "from " *z3-module* " import " *z3-class* ", Q"
    #\Newline
    "s = " *z3-class* "()"
    #\Newline)))
```

```

;; write-SMT-file
(defun write-SMT-file (filename translated-formula state)
  "write-SMT-file: writes the translated formula into a python
    file, it opens and closes the channel and write the including
    of Z3 inteface"
  (mv-let
   (channel state)
   (open-output-channel! filename :character state)
   (let ((state (princ$-list-of-strings
       (write-head) channel state)))
     (let ((state (princ$-list-of-strings translated-formula
    channel state)))
       (close-output-channel channel state)))))

;; write-expander-file
(defun write-expander-file (filename expanded-term state)
  "write-expander-file: write expanded term to a file"
  (mv-let
   (channel state)
   (open-output-channel! filename :character state)
   (let ((state
     (princ$-list-of-strings
      expanded-term channel state)))
     (close-output-channel channel state))))

;; SMT-run
(defun SMT-run (filename)
  "SMT-run: run the external SMT procedure from ACL2"
  (let ((cmd (concatenate 'string *smt-cmd* " " filename)))
    (time$ (tshell-call cmd
                        :print t
                        :save t)
           :msg "; Z3: `~s0`: ~st sec, ~sa bytes~%"
           :args (list cmd))))
```

```
;;SMT-interpreter formats the results
```

```
(in-package "ACL2")
(include-book "SMT-run")
(defttag :tshell)


;; SMT-interpreter
(defun SMT-interpreter (filename)
  "SMT-intepreter: get the result returned from calling SMT
    procedure"
  (mv-let (finishedp exit-status lines)
          (SMT-run filename)
     (cond ((equal finishedp nil)
       (cw "Warning: the command was interrupted."))
      ((not (equal exit-status 0))
       (cw "Z3 failure: ~q0" lines))
      (t (if (equal (car lines) "proved")
             t
           (cw "~q0" lines))))))
```

```
;; This file configs the path to below directories:
;; 1. Z3_interface
;; 2. Z3_files
;; 3. name of z3 class
;; 4. SMT command
(in-package "ACL2")
(defconst *dir-interface*
    "/ubc/cs/home/y/yanpeng/project/ACL2/smtlink/z3\_interface")
(defconst *dir-files* "z3\_files")
(defconst *z3-module* "ACL2\_translator")
(defconst *z3-class* "to_smt")
(defconst *smt-cmd* "python")
(defconst *dir-expanded* "expanded")
```

```
;; helper functions for basic data structure manipulation
(in-package "ACL2")

```

```
;; exist
(defun exist (elem lista)
  "exist: check if an element exist in a list"
  (if (endp lista)
      nil
    (if (equal elem (car lista))
   t
      (exist elem (cdr lista)))))

;; end
(defun end (lista)
  "end: return the last element in a list"
  (if (endp (cdr lista))
      (car lista)
    (end (cdr lista))))

;; my-last
(defun my-last (listx)
 "my-last: fetch the last element from list"
  (car (last listx)))

;; my-delete
(defun my-delete (listx elem)
  "my-delete: delete an element from the list. If there're
    duplicates, this function deletes the first one in the list."
  (if (endp listx) ;; elem does not exist in the list
      listx
      (if (equal (car listx) elem)
     (cdr listx)
     (cons (car listx)
      (my-delete (cdr listx) elem)))))

(defthm delete-must-reduce
    (implies (exist a listx)
        (< (len (my-delete listx a)) (len listx))))

;; dash-to-underscore-char
```

```
(defun dash-to-underscore-char (charx)
  (if (equal charx '-)
      '_
      charx))

;; dash-to-underscore-helper
(defun dash-to-underscore-helper (name-list)
  (if (endp name-list)
      nil
      (cons (dash-to-underscore-char (car name-list))
       (dash-to-underscore-helper (cdr name-list)))))

;; dash-to-underscore
(defun dash-to-underscore (name)
  (intern-in-package-of-symbol
   (coerce
    (dash-to-underscore-helper
     (coerce (symbol-name name)'list))
    'string)
   'ACL2))

;; append-and-decl
(defun append-and-decl (listx listy let-type)
  "append-and-decl: append two and lists together in the underneath
    representation"
  (if (endp listy)
      listx
      (append-and-decl
       (list 'if (list (car let-type) (car listy)) listx ''nil)
       (cdr listy)
       (cdr let-type))))

;; append-and-hypo
(defun append-and-hypo (listx listy)
  "append-and-hypo: append two and lists together in the underneath
    representation"
  (if (endp listy)
```

```
      listx
      (append-and-hypo
       (list 'if (car listy) listx ''nil)
       (cdr listy))))

;; assoc-get-value
(defun assoc-get-value (listx)
  "assoc-get-value: get all values out of an associate list"
  (if (endp listx)
      nil
      (cons (cadar listx)
       (assoc-get-value (cdr listx)))))

;; assoc-get-key
(defun assoc-get-key (listx)
  "assoc-get-key: get all keys out of an associate list"
  (if (endp listx)
      nil
      (cons (caar listx)
       (assoc-get-key (cdr listx)))))

;; assoc-no-repeat
(defun assoc-no-repeat (assoc-list)
  "assoc-no-repeat: check if an associate list has repeated keys"
  (if (endp assoc-list)
      t
      (if (equal (assoc-equal (caar assoc-list) (cdr assoc-list))
    nil)
     (assoc-no-repeat (cdr assoc-list))
     nil)))

;; invert-assoc
(defun invert-assoc (assoc-list)
  "invert-assoc: invert the key and value pairs in an associate
    list"
  (if (endp assoc-list)
      nil
```

```
      (cons (list (cadar assoc-list) (caar assoc-list))
      (invert-assoc (cdr assoc-list)))))

;; create-assoc-helper
 (defun create-assoc-helper (list-keys list-values)
   (if (endp list-keys)
       nil
       (cons (list (car list-keys) (car list-values))
        (create-assoc-helper (cdr list-keys) (cdr list-values)))))

;; create-assoc
(defun create-assoc (list-keys list-values)
  "create-assoc: combines two lists together to form an associate list"
  (if (equal (len list-keys) (len list-values))
      (create-assoc-helper list-keys list-values)
      (cw "Error(helper): list-keys and list-values should be of the same len.")))

;; replace-lambda-params
 (defun replace-lambda-params (expr lambda-params-mapping)
   "replace-lambda-params: replace params in the expression using the mapping"
   (if (atom expr)
       (let ((res (assoc-equal expr lambda-params-mapping)))
    (if (equal res nil)
        expr
        (cadr res)))
       (cons (replace-lambda-params (car expr) lambda-params-mapping)
        (replace-lambda-params (cdr expr) lambda-params-mapping))))

;; assoc-lambda
(defun assoc-lambda (expr lambda-params-mapping assoc-list)
  "assoc-lambda: replacing params in expression using lambda-params-mapping \
```

```
and check if the resulting term exist in assoc-list keys. Return
    the resulting \
pair from assoc-list."
  (let ((new-expr (replace-lambda-params expr
    lambda-params-mapping)))
      (assoc-equal new-expr assoc-list)))

;; combine
(defun combine (lista listb)
  "combine: takes two items, either atoms or lists, then combine
    them together according to some rule. E.g. if either element is
    nil, return the other one; if a is atom and b is list, do cons;
    if both are lists, do append; if a is list and b is atom,
    attach b at the end; if both are atoms, make a list"
  (cond ((and (atom lista) (atom listb) (not (equal lista nil))
    (not (equal listb nil)))
    (list lista listb))
   ((and (atom lista) (listp listb) (not (equal lista nil)))
    (cons lista listb))
   ((and (listp lista) (atom listb) (not (equal listb nil)))
    (append lista (list listb)))
   ((and (listp lista) (listp listb))
    (append lista listb))))
```

B.2 Z3 Interface

```
from z3 import Solver, Bool, Int, Real, BoolSort, IntSort,
    RealSort, And, Or, Not, Implies, sat, unsat, Q, Array, Select,
    Store, ToInt

def sort(x):
    if type(x) == bool:    return BoolSort()
    elif type(x) == int:   return IntSort()
    elif type(x) == float: return RealSort()
    elif hasattr(x, 'sort'):
        if x.sort() == BoolSort(): return BoolSort()
        if x.sort() == IntSort():  return IntSort()
```

```
        if x.sort() == RealSort(): return RealSort()
        else:
            raise Exception('unknown sort for expression')

class to_smt:
    class status:
        def __init__(self, value):
            self.value = value

            def __str__(self):
                if(self.value is True): return 'QED'
                elif(self.value.__class__ == 'msg'.__class__):
    return self.value
                else: raise Exception('unknown status?')

      def isThm(self):
                    return(self.value is True)

    def __init__(self, solver=0):
        if(solver != 0): self.solver = solver
        else: self.solver = Solver()
        self.nameNumber = 0

    def newVar(self):
        varName = '$' + str(self.nameNumber)
        self.nameNumber = self.nameNumber+1
        return varName

    def isBool(self, who):
        return Bool(who)

    def isInt(self, who):
        return Int(who)

    def isReal(self, who):
        return Real(who)
```

```
    def floor(self, x):
        return ToInt(x)

    def plus(self, *args):
        return reduce(lambda x, y: x+y, args)

    def times(self, *args):
        return reduce(lambda x, y: x*y, args)

    def andx(self, *args):
        return reduce(lambda x, y: And(x,y), args)

    def orx(self, *args):
        return reduce(lambda x, y: Or(x,y), args)

    def minus(self, x,y): return x-y

    # special care for reciprocal because
    # in ACL2 3/0 = 0 and in z3 3/0 == 0
    # will return a counter-example
    def reciprocal(self, x):
        if(type(x) is int): return(Q(1,x))
        elif(type(x) is float): return 1.0/x
        elif(x.sort() == IntSort()): return 1/(Q(1,1)*x)
        else: return 1/x

    def negate(self, x): return -x
    def div(self, x, y): return times(self,x,reciprocal(self,y))
    def gt(self, x,y): return x>y
    def lt(self, x,y): return x<y
    def ge(self, x,y): return x>=y
    def le(self, x,y): return x<=y
    def equal(self, x,y): return x==y
    def notx(self, x): return Not(x)

    def implies(self, x, y): return Implies(x,y)
```

```
# type related functions
def integerp(self, x): return x.sort() == IntSort()
def rationalp(self, x): return x.sort() == RealSort()
def booleanp(self, x): return x.sort() == BoolSort()

def ifx(self, condx, thenx, elsex):
    v = 0
    if sort(thenx) == sort(elsex):
        if sort(thenx) == BoolSort(): v = Bool(self.newVar())
        if sort(thenx) == IntSort():  v = Int(self.newVar())
        if sort(thenx) == RealSort(): v = Real(self.newVar())
        if v is 0: raise Exception('mixed type for if-expression')
    self.solver.add(And(Implies(condx, v == thenx), Implies(Not(condx), v == elsex)))
    return(v)

# # array
# def array(self, mylist):
#     if not mylist:
#        raise("Can't determine type of an empty list.")
#     else:
#         ty = sort(mylist[0])
#         a = Array(self.newVar(), IntSort(), ty)
#         n = len(mylist)
#         for i in range(0,n):
#             j = Int(self.newVar())
#             self.solver.add(j == i)
#             self.solver.add(Select(a, j) == mylist[i])
#     return a

# # nth
# def nth(self, i, a):
#     return Select(a, i)

# usage prove(claim) or prove(hypotheses, conclusion)
def prove(self, hypotheses, conclusion=0):
```

```
        if(conclusion is 0): claim = hypotheses
        else: claim = Implies(hypotheses, conclusion)

        self.solver.add(Not(claim))
        res = self.solver.check()

        if res == unsat:
            print "proved"
            return self.status(True)  # It’s a theorem
        elif res == sat:
            print "counterexample"
            m = self.solver.model()
            print m
            # return an counterexample??
            return self.status(False)
        else:
            print "failed to prove"
```

Appendix C

Convergence Proof Code

C.1 Z3 Proof for Coarse Convergence

```
from z3 import *
from DPLL import DPLL_model

def leave(dpll=DPLL_model()):
  c = [Real('c[0]'), Real('c[1]'), Real('c[2]')]
  v = [Real('v[0]'), Real('v[1]'), Real('v[2]')]
  phi = [Real('phi[0]'), Real('phi[1]'), Real('phi[2]')]
  s = Solver()
  s.add(And(initialRegion(dpll, c[0], v[0], phi[0]),
    dpll.next(c[:2], v[:2], phi[:2])))

  # show that the initial region is an invariant
  prove(s, initialRegion(dpll, c[1], v[1], phi[1]), 'initial region
    is invariant')

  # find bound on v when c=c_min and fDCO crosses fref
  s.push()
  s.add(dpll.next(c[1:], v[1:], phi[1:]))
  s.add(And(c[0] == dpll.cmin, dpll.fDCO(c[0], v[0]) < dpll.fref,
    phi[0] == 0, phi[2] >= 0))
  ch = s.check()
  if(ch == sat):
    print 'phi can change sign'
    print str(s.model())
  else:
    print "phi is stuck (how'd that happen?)"
```

```
    print "ch =", str(ch)


def initialRegion(dpll, c, v, phi):
  return And(dpll.cmin <= c, c <= dpll.cmax,
        dpll.vmin <= v, v <= dpll.vmax,
        -1 <= phi,      phi <= +1)

def prove(s, claim, what):
  s.push()
  s.add(Not(claim))
  ch = s.check()
  if(ch == unsat):
    print 'Proven', what
    s.pop()
  else:
    print 'FAILED TO PROVE:', what
    if(ch == sat):
      print "Here's a counter-example:"
      print str(s.model())
    else: print "Z3 couldn't decide"
    s.pop()
    raise Exception('Proof failed');
```

```
from DPLL import *
from z3 import *
import time

def my_prove(what, hyp, concl):
  s = Solver()
  s.add(hyp)
  s.add(Not(concl))
  p = s.check()
  if(p == unsat):
    print 'PROOF! ', what
    return "proved"
  elif(p == sat):
```

```
    print 'Failed to prove: ', what
    print "Here's a counter-example: ", str(s.model())
    print ":("
    return "can't prove"
  else:
    print what + '? -- I dunno'
    return "stuck"

c = Reals(["c", "c'"])
v = Reals(["v", "v'"])
phi = Reals(["phi", "phi'"])
dpll = DPLL_model()
s = Solver()
s.push()
s.add(And(c[0] == 1.05, v[0] == 0.8, phi[0] == 0.25, dpll.next(c,
    v, phi)))
print 'Is the model satisfiable? ', str(s.check())
if(s.check() == sat):
  print "Here's a solution: ", str(s.model())
s.pop()

# All c v phi will stay in valid region
hyp = And(dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = dpll.valid(c[1], v[1], phi[1])
my_prove('invariance of valid states', hyp, concl)

# When f_dco < 0.9*fref, positive phi decreases
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N < 0.9*dpll.fref, \
          0 <= phi[0], \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = phi[1] < phi[0] - dpll.eps
my_prove('Positive phi decreases for f_dco/N < 0.9*f_ref', hyp,
    concl)

# When f_dco < 0.9*fref and phi < 0, phi stays negative
```

```
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N < 0.9*dpll.fref, \
          phi[0] < 0, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = phi[1] < 0
my_prove('invariance of negative phi for f_dco/N < 0.9*f_ref', hyp,
    concl)

# When f_dco < 0.9*fref and phi < 0, c >= cmin+gc, c decreases at
    least for some amount
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N < 0.9*dpll.fref, \
          phi[0] < 0, \
          c[0] >= dpll.cmin + dpll.gc, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = c[1] == c[0] - dpll.gc
my_prove('c decrease by gc for f_dco/N < 0.9*f_fref when phi<0 and
    c >= cmin + gc', hyp, concl)

# When f_dco < 0.9*fref and phi < 0, c < cmin+gc, c collapse to cmin
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N < 0.9*dpll.fref, \
          phi[0] < 0, \
          c[0] < dpll.cmin + dpll.gc, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = c[1] == dpll.cmin
my_prove('c collapses to cmin for f_dco/N < 0.9*f_fref when phi<0
    and c < cmin + gc', hyp, concl)

# How to prove c will crawl up??
# When f_dco < 0.9*fref and phi < 0, c == cmin, v increases
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N < 0.9*dpll.fref, \
          phi[0] < 0, \
          c[0] >= dpll.cmin, \
          c[0] <= dpll.cmin + dpll.gc, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
```

```
concl = v[1] > v[0] + dpll.eps
my_prove('v increases for f_dco/N < 0.9*f_fref when phi<0 and cmin
    + gc >= c >= cmin', hyp, concl)

# How to prove in the middle stripe, when at saturation?
# First prove when f_dco >= 0.9*fref and f_dco <= 1.0*fref
# and phi < 0, c == cmin, v will increase c will stay cmin and phi
    will stay negative
hyp = And(dpll.fDCO(c[0],v[0])/dpll.N <= 1.0*dpll.fref, \
          dpll.fDCO(c[0],v[0])/dpll.N >= 0.9*dpll.fref, \
          phi[0] < 0, \
          c[0] == dpll.cmin, \
          dpll.valid(c[0],v[0],phi[0]), \
          dpll.next(c,v,phi))
concl = And(v[1] > v[0], phi[1] < 0, c[1] == dpll.cmin)
my_prove("v will increase, c and phi will stay when
    0.9*fref<=fdco<=1.0*fref, phi < 0 and c == cmin", hyp, concl)


# Find the next points leave the wall
# v in range [arg_v(fdco/N == fref), arg_v(fdco/N == fref)+gv]
# phi in range [-1,0)
# c = cmin
# ask if after i steps all state will become phi >= 0
def newVar(nameList,indexList):
  res = []
  for j in range(0,len(nameList)):
    arg = nameList[j]+" = Reals(["
    for i in range(0,len(indexList[j])-1):
      arg = arg + "\""+ nameList[j]+"_"+str(indexList[j][i])+"\","
    arg = arg + "\""+ nameList[j]+"_"+str(indexList[j][i+1])+ "\"])"
    res.append(arg)
  return res

def OrPos(argList):
  res = False
  for item in argList:
```

```
    res = Or(res, item > 0)
  return res

def OrNeg(argList):
  res = False
  for item in argList:
    res = Or(res, item > 0)
  return res

def OrEql(argList, v):
  res = False
  for item in argList:
    res = Or(res, item == v)
  return res

def Inc(argList):
  res = True
  for i in range(0,len(argList)-1):
    res = And(res, argList[i]<argList[i+1])
  return res

# All points leave the wall after 7 steps.
start = time.time()
steps = 0
for i in range(2,10):
  decl = newVar(["c","v","phi"], [range(0,i),range(0,i),range(0,i)])
  for stmt in decl:
    exec(stmt)

  tmp = Real("tmp")
  hyp = And(phi[0] < 0, \
            phi[0] >= -1.0, \
            c[0] == dpll.cmin, \
            dpll.fDCO(c[0],tmp)/dpll.N == dpll.fref, \
            v[0] >= tmp, \
            v[0] < tmp - dpll.gv, \
            dpll.valid(c[0],v[0],phi[0]), \
```

```
            dpll.unwind(c,v,phi))
  concl = OrPos(phi)
  if my_prove("All points leave wall after "+str(i-1)+" steps",hyp,concl) == "proved":
    steps = i-1
    break

end = time.time()
print "Time elapsed: " + str(end - start) + "s"

# If can prove for all points leaving the wall, they will go back before
# hitting onto the other wall, then done.


# ======================================================== #
#
# FOR THE UPPER HALF
# When f_dco > 1.1*fref, negative phi increases
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N > 1.1*dpll.fref, \
          0 >= phi[0], \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = phi[1] > phi[0]-dpll.eps
my_prove('Negative phi increases for f_dco/N > 1.1*f_ref', hyp, concl)

# When f_dco > 1.1*fref and phi > 0, phi stays positive
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N > 1.1*dpll.fref, \
          phi[0] > 0, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = phi[1] > 0
my_prove('invariance of positive phi for f_dco/N > 1.1*f_ref', hyp, concl)

```

```
# When f_dco > 1.1*fref and phi > 0, c <= cmax-gc, c increases at
    least for some amount
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N > 1.1*dpll.fref, \
          phi[0] > 0, \
          c[0] <= dpll.cmax - dpll.gc, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = c[1] == c[0] + dpll.gc
my_prove('c increase by gc for f_dco/N > 1.1*f_fref when phi>0 and
    c <= cmax - gc', hyp, concl)

# When f_dco > 1.1*fref and phi > 0, c > cmax-gc, c collapse to cmax
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N > 1.1*dpll.fref, \
          phi[0] > 0, \
          c[0] > dpll.cmax - dpll.gc, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = c[1] == dpll.cmax
my_prove('c collapses to cmax for f_dco/N > 1.1*f_fref when phi>0
    and c > cmax - gc', hyp, concl)

# When f_dco > 1.1*fref and phi > 0, c == cmax, v decreases
hyp = And(dpll.fDCO(c[0], v[0])/dpll.N > 1.1*dpll.fref, \
          phi[0] > 0, \
          c[0] <= dpll.cmax, \
          c[0] >= dpll.cmax - dpll.gc, \
          dpll.valid(c[0], v[0], phi[0]), \
          dpll.next(c, v, phi))
concl = v[1] < v[0] - dpll.eps
my_prove('v decreases for f_dco/N > 1.1*f_fref when phi>0 and cmax
    - gc <= c <= cmax', hyp, concl)
```

C.2 ACL2 Proof for Fine Convergence

C.2.1 ACL2 Code

- Definitions:

```
;; There are two files for the proof of recurrence model of the
;; DPLL: global.lisp, DPLL_functions.lisp and
    DPLL_theorems.lisp.
;; global.lisp
;; global.lisp defines global variables that are repeatedly
;; called in a lot of the functions.

(in-package "ACL2")
;; (defconst *g1* 1/3200)
(defconst *g2* (- (/ 1/3200 5)))
(defconst *ccode* 1)
(defconst *Kt* 4/5)
(defconst *N* 1)
(defconst *fref* 1)
(defconst *alpha* 1)
(defconst *beta* 1)
(defconst *f0* 1)
(defconst *vcenter* 1)
;; (defconst *v0* 1)

; Define intermediate variables
(defun equ-c (v0)
  (- (* *f0* (+ 1 (* *alpha* v0)) (/ (* *beta* *N* *fref*)))
     (/ *beta*)))
(defun gamma ()
  (- 1 *Kt*))
;;(defun gamma () (/ 1 2))
(defun mu ()
  (/ *f0* (* *N* *fref*)))
(defun m (n v0 g1)
 (- (/ (equ-c v0) g1) n))
;; (defun m-constraint (n v0 g1)
;;   (and (> m (- (- (/ (equ-c v0) g1) n) 1))
;;         (< m (- (/ (equ-c v0) g1) n))))
(defun dv0 ()
  (* -2 *g2*))
```

- Original proof:

```
(in-package "ACL2")
(include-book "global")

;;(add-include-book-dir :book
    "/ubc/cs/research/isd/users/software/ACL2/acl2-7.0/books")
(deftheory before-arith (current-theory :here))
(include-book "arithmetic/top-with-meta" :dir :system)
(deftheory after-arith (current-theory :here))

(deftheory arithmetic-book-only (set-difference-theories
    (theory 'after-arith) (theory 'before-arith)))

;; for the clause processor to work
(add-include-book-dir :cp
    "/ubc/cs/home/y/yanpeng/project/ACL2/smtlink")
(include-book "top" :dir :cp)
(logic)
:set-state-ok t
:set-ignore-ok t
(tshell-ensure)

;;:start-proof-tree

;; (encapsulate ()

;; (local (include-book "arithmetic-5/top" :dir :system))

;; (defun my-floor (x) (floor (numerator x) (denominator x)))

;; (defthm my-floor-type
;;   (implies (rationalp x)
;;       (integerp (my-floor x)))
;;   :rule-classes :type-prescription)

;; (defthm my-floor-lower-bound
```

```
;;   (implies (rationalp x)
;;       (> (my-floor x) (- x 1)))
;;   :rule-classes :linear)

;; (defthm my-floor-upper-bound
;;   (implies (rationalp x)
;;       (<= (my-floor x) x))
;;   :rule-classes :linear)

;; (defthm my-floor-comparison
;;   (implies (rationalp x)
;;       (< (my-floor (1- x)) (my-floor x)))
;;   :hints (("Goal"
;;       :use ((:instance my-floor-upper-bound (x (1- x)))
;;        (:instance my-floor-lower-bound))))
;;   :rule-classes :linear)
;; )

;; functions
;; n can be a rational value when c starts from non-integer value
(defun fdco (n v0 dv g1)
  (/ (* (mu) (+ 1 (* *alpha* (+ v0 dv)))) (+ 1 (* *beta* n g1))))

(defun B-term-expt (h)
  (expt (gamma) (- h)))

(defun B-term-rest (h v0 dv g1)
  (- (* (mu) (/ (+ 1 (* *alpha* (+ v0 dv))) (+ 1 (* *beta* (+ (* h g1) (equ-c v0)))))) 1))

(defun B-term (h v0 dv g1)
  (* (B-term-expt h) (B-term-rest h v0 dv g1)))

(defun B-sum (h_lo h_hi v0 dv g1)
  (declare (xargs :measure (if (or (not (integerp h_hi)) (not
```

```
    (integerp h_lo)) (< h_hi h_lo))
                 0
                 (1+ (- h_hi h_lo)))))
  (if (or (not (integerp h_hi)) (not (integerp h_lo)) (> h_lo
    h_hi))  0
      (+ (B-term h_hi v0 dv g1) (B-term (- h_hi) v0 dv g1)
    (B-sum h_lo (- h_hi 1) v0 dv g1))))

(defun B-expt (n)
  (expt (gamma) (- n 2)))

(defun B (n v0 dv g1)
  (* (B-expt n)
     (B-sum 1 (- n 2) v0 dv g1)))

;; parameter list functions
(defmacro basic-params-equal (n n-value &optional (v0 'nil)
    (dv 'nil) (g1 'nil) (phi0 'nil) (other 'nil))
  (list 'and
   (append
    (append
     (append
      (append (list 'and
          (list 'integerp n))
         (if (equal g1 'nil) nil (list (list 'rationalp g1))))
      (if (equal v0 'nil) nil (list (list 'rationalp v0))))
     (if (equal phi0 'nil) nil (list (list 'rationalp phi0))))
    (if (equal dv 'nil) nil (list (list 'rationalp dv))))
   (append
    (append
     (append
      (append
       (append
        (append
         (append
          (append
      (list 'and
```

```
            (list 'equal n n-value))
      (if (equal g1 'nil) nil (list (list 'equal g1 '1/3200))))
          (if (equal v0 'nil) nil (list (list '>= v0 '9/10))))
         (if (equal v0 'nil) nil (list (list '<= v0 '11/10))))
        (if (equal dv 'nil) nil (list (list '>= dv (list '- (list 'dv0))))))
       (if (equal dv 'nil) nil (list (list '<= dv (list 'dv0)))))
      (if (equal phi0 'nil) nil (list (list '>= phi0 '0))))
     (if (equal phi0 'nil) nil (list (list '< phi0 (list '- (list 'fdco (list '1+ (list 'm '640 v0 g1)) v0 dv g1) '1)))))
    (if (equal other 'nil) nil (list other)))))

(defmacro basic-params (n nupper &optional (v0 'nil) (dv 'nil) (g1 'nil) (phi0 'nil) (other 'nil))
  (list 'and
   (append
    (append
     (append
      (append (list 'and
          (list 'integerp n))
         (if (equal g1 'nil) nil (list (list 'rationalp g1))))
      (if (equal v0 'nil) nil (list (list 'rationalp v0))))
     (if (equal dv 'nil) nil (list (list 'rationalp dv))))
    (if (equal phi0 'nil) nil (list (list 'rationalp phi0))))
   (append
    (append
    (append
     (append
      (append
        (append
         (append
          (append
      (append (list 'and
              (list '>= n nupper))
         (list (list '<= n '640)))
```

```
            (if (equal g1 'nil) nil (list (list 'equal g1
  '1/3200))))
          (if (equal v0 'nil) nil (list (list '>= v0 '9/10))))
         (if (equal v0 'nil) nil (list (list '<= v0 '11/10))))
        (if (equal dv 'nil) nil (list (list '>= dv (list '-
  (list 'dv0))))))
       (if (equal dv 'nil) nil (list (list '<= dv (list
  'dv0)))))
      (if (equal phi0 'nil) nil (list (list '>= phi0 '0))))
     (if (equal phi0 'nil) nil (list (list '< phi0 (list '-
  (list 'fdco (list '1+ (list 'm '640 v0 g1)) v0 dv g1)
  '1)))))
    (if (equal other 'nil) nil (list other)))))

(encapsulate ()

(local (in-theory (disable arithmetic-book-only)))

(local
(include-book "arithmetic-5/top" :dir :system)
)

(local
(defthm B-term-neg-lemma1
  (implies (basic-params h 1 v0 dv g1)
      (< (+ (* (B-term-expt h) (B-term-rest h v0 dv g1))
          (* (B-term-expt (- h)) (B-term-rest (- h) v0 dv g1)))
         0)
      )
  :hints
  (("Goal"
    :clause-processor
    (Smtlink clause
          '( (:expand ((:functions ((B-term-rest rationalp)
                       (gamma rationalp)
                       (mu rationalp)
                       (equ-c rationalp)
```

```
                    (dv0 rationalp)))
                 (:expansion-level 1)))
            (:python-file "B-term-neg-lemma1") ;;mktemp
            (:let ((expt_gamma_h (B-term-expt h) rationalp)
              (expt_gamma_minus_h (B-term-expt (- h)) rationalp)))
            (:hypothesize ((<= expt_gamma_minus_h (/ 1 5))
                 (> expt_gamma_minus_h 0)
                 (equal (* expt_gamma_minus_h expt_gamma_h) 1)))
           (:use ((:let ())
             (:hypo (()))
             (:main ()))))
         state)
    ))
  )
)

(defthm B-term-neg
  (implies (basic-params h 1 v0 dv g1)
      (< (+ (B-term h v0 dv g1) (B-term (- h) v0 dv g1)) 0))
  :hints (("Goal"
      :use ( (:instance B-term)
       (:instance B-term-neg-lemma1)
        )))
  :rule-classes :linear)
)

(defthm B-sum-neg
  (implies (basic-params n-minus-2 1 v0 dv g1)
      (< (B-sum 1 n-minus-2 v0 dv g1) 0))
  :hints (("Goal"
      :in-theory (disable B-term)
      :induct ())))

(encapsulate ()

```

```
(local ;; B = B-expt*B-sum
 (defthm B-neg-lemma1
   (implies (basic-params n 3 v0 dv g1)
       (equal (B n v0 dv g1)
         (* (B-expt n)
            (B-sum 1 (- n 2) v0 dv g1))))))

(local
 (defthm B-expt->-0
   (implies (basic-params n 3)
       (> (B-expt n) 0))
   :rule-classes :linear))

(local
 (defthm B-neg-lemma2
   (implies (and (rationalp a)
       (rationalp b)
       (> a 0)
       (< b 0))
       (< (* a b) 0))
   :rule-classes :linear))

(local
 (defthm B-neg-type-lemma3
   (implies (and (and (rationalp n-minus-2) (rationalp v0)
    (rationalp g1) (rationalp dv)))
       (rationalp (B-sum 1 n-minus-2 v0 dv g1)))
   :rule-classes :type-prescription))

(local
 (defthm B-neg-type-lemma4
   (implies (basic-params n 3)
       (rationalp (B-expt n)))
   :rule-classes :type-prescription))

(defthm B-neg
  (implies (basic-params n 3 v0 dv g1)
```

```
      (< (B n v0 dv g1) 0))
  :hints (("Goal"
      :do-not-induct t
      :in-theory (disable B-expt B-sum B-sum-neg B-expt->-0)
      :use ((:instance B-sum-neg (n-minus-2 (- n 2)))
       (:instance B-expt->-0)
       (:instance B-neg-type-lemma3 (n-minus-2 (- n 2)))
       (:instance B-neg-type-lemma4)
       (:instance B-neg-lemma2 (a (B-expt n))
                          (b (B-sum 1 (+ -2 n) v0 dv g1)))))))
)

(defun A (n phi0 v0 dv g1)
  (+ (* (expt (gamma) (- (* 2 n) 1)) phi0)
     (* (expt (gamma) (- (* 2 n) 2))
   (- (fdco (m n v0 g1) v0 dv g1) 1))
     (* (expt (gamma) (- (* 2 n) 3))
   (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1))))

(defun phi-2n-1 (n phi0 v0 dv g1)
  (+ (A n phi0 v0 dv g1) (B n v0 dv g1)))

(defun delta (n v0 dv g1)
  (+ (- (* (expt (gamma) (* 2 n))
      (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
   (* (expt (gamma) (* 2 n))
      (- (fdco (m n v0 g1) v0 dv g1) 1)))
     (- (* (expt (gamma) (- (* 2 n) 1))
      (- (fdco (m n v0 g1) v0 dv g1) 1))
   (* (expt (gamma) (- (* 2 n) 1))
      (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))
     (* (expt (gamma) (1- n))
   (+ (* (expt (gamma) (1+ (- n)))
         (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))
       1))
      (* (expt (gamma) (1- n))
```

```
         (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
       1))))))

(defun delta-1 (n v0 dv g1)
  (+ (* (expt (gamma) (* 2 n))
   (- (fdco (1- (m n v0 g1)) v0 dv g1)
      (fdco (m n v0 g1) v0 dv g1)))
     (* (expt (gamma) (- (* 2 n) 1))
   (- (fdco (m n v0 g1) v0 dv g1)
      (fdco (1+ (m n v0 g1)) v0 dv g1)))
     (* (* (expt (gamma) (1- n)) (expt (gamma) (1+ (- n))))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
         (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))) 1))
     (* (* (expt (gamma) (1- n)) (expt (gamma) (1- n)))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
         (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1))))

(defun delta-2 (n v0 dv g1)
  (+ (* (expt (gamma) (* 2 n))
   (- (fdco (1- (m n v0 g1)) v0 dv g1)
      (fdco (m n v0 g1) v0 dv g1)))
     (* (expt (gamma) (- (* 2 n) 1))
   (- (fdco (m n v0 g1) v0 dv g1)
      (fdco (1+ (m n v0 g1)) v0 dv g1)))
     (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
      (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))) 1)
     (* (expt (gamma) (+ -1 n -1 n))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
         (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1))))

(defun delta-3 (n v0 dv g1)
  (* (expt (gamma) (+ -1 n -1 n))
     (+ (* (expt (gamma) 2)
      (- (fdco (1- (m n v0 g1)) v0 dv g1)
         (fdco (m n v0 g1) v0 dv g1)))
   (* (expt (gamma) 1)
```

```
      (- (fdco (m n v0 g1) v0 dv g1)
         (fdco (1+ (m n v0 g1)) v0 dv g1)))
   (* (expt (gamma) (- 2 (* 2 n)))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
       (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))) 1))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
         (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1))))

(defun delta-3-inside (n v0 dv g1)
  (+ (* (expt (gamma) 2)
      (- (fdco (1- (m n v0 g1)) v0 dv g1)
         (fdco (m n v0 g1) v0 dv g1)))
   (* (expt (gamma) 1)
      (- (fdco (m n v0 g1) v0 dv g1)
         (fdco (1+ (m n v0 g1)) v0 dv g1)))
   (* (expt (gamma) (- 2 (* 2 n)))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
       (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))) 1))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
         (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1)))

(defun delta-3-inside-transform (n v0 dv g1)
  (/
   (+ (* (expt (gamma) 2)
    (- (fdco (1- (m n v0 g1)) v0 dv g1)
       (fdco (m n v0 g1) v0 dv g1)))
      (* (expt (gamma) 1)
    (- (fdco (m n v0 g1) v0 dv g1)
       (fdco (1+ (m n v0 g1)) v0 dv g1)))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
       (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1))
   (- 1
      (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
    (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))))))

;; rewrite delta term
(encapsulate ()
```

```
(local
;; considering using smtlink for the proof, probably simpler
(defthm delta-rewrite-1-lemma1
  (implies (basic-params n 3 v0 dv g1)
      (equal (+ (- (* (expt (gamma) (* 2 n))
             (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
          (* (expt (gamma) (* 2 n))
             (- (fdco (m n v0 g1) v0 dv g1) 1)))
            (- (* (expt (gamma) (- (* 2 n) 1))
             (- (fdco (m n v0 g1) v0 dv g1) 1))
          (* (expt (gamma) (- (* 2 n) 1))
             (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))
            (* (expt (gamma) (1- n))
          (+ (* (expt (gamma) (1+ (- n)))
                (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))
              1))
             (* (expt (gamma) (1- n))
                (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
              1)))))
           (+ (* (expt (gamma) (* 2 n))
            (- (fdco (1- (m n v0 g1)) v0 dv g1)
               (fdco (m n v0 g1) v0 dv g1)))
              (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (m n v0 g1) v0 dv g1)
               (fdco (1+ (m n v0 g1)) v0 dv g1)))
              (* (* (expt (gamma) (1- n)) (expt (gamma) (1+ (-
    n))))
            (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
             (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))) 1))
              (* (* (expt (gamma) (1- n)) (expt (gamma) (1- n)))
            (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
             (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
    1)))))
  :hints
```

```
  (("Goal"
    :clause-processor
    (Smtlink clause
          '( (:expand ((:functions ((m integerp)
                      (gamma rationalp)
                      (mu rationalp)
                      (equ-c rationalp)
                      (fdco rationalp)
                      (dv0 rationalp)))
                   (:expansion-level 1)))
              (:python-file "delta-rewrite-1-lemma1") ;;mktemp
              (:let ((expt_gamma_2n
                 (expt (gamma) (* 2 n))
                  rationalp)
                (expt_gamma_2n_minus_1
                 (expt (gamma) (- (* 2 n) 1))
                  rationalp)
                (expt_gamma_n_minus_1
                 (expt (gamma) (1- n))
                  rationalp)
                (expt_gamma_1_minus_n
                 (expt (gamma) (1+ (- n)))
                  rationalp)
                ))
              (:hypothesize ()))
          state)
    )))
)

(local
(defthm delta-rewrite-1
  (implies (basic-params n 3 v0 dv g1)
      (equal (delta n v0 dv g1)
        (delta-1 n v0 dv g1))))
)

(local
```

```
(defthm delta-rewrite-2-lemma1
  (implies (basic-params n 3)
      (equal (* (expt (gamma) (1- n))
             (expt (gamma) (1+ (- n))))
        1))
  :hints (("Goal"
      :use ((:instance expt-minus
                 (r (gamma))
                 (i (- (1+ (- n))))))
      )))
)

(local
(defthm delta-rewrite-2-lemma2
  (implies (basic-params n 3)
      (equal (* (expt (gamma) (1- n))
             (expt (gamma) (1- n)))
        (expt (gamma) (+ -1 n -1 n))))
  :hints (("Goal"
      :do-not-induct t
      :use ((:instance exponents-add-for-nonneg-exponents
                 (i (1- n))
                 (j (1- n))
                 (r (gamma))))
      :in-theory (disable exponents-add-for-nonneg-exponents)
      ))
  )
)

(local
(defthm delta-rewrite-2-lemma3
  (implies (basic-params n 3)
      (equal (+ A
             B
             (* (* (expt (gamma) (1- n))
              (expt (gamma) (1+ (- n))))
         C)
```

```
           (* (* (expt (gamma) (1- n))
            (expt (gamma) (1- n)))
         D))
        (+ A B C
           (* (expt (gamma) (+ -1 n -1 n)) D))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-2-lemma1)
       (:instance delta-rewrite-2-lemma2)))))
)

(local
(defthm delta-rewrite-2
  (implies (basic-params n 3 v0 dv g1)
      (equal (delta-1 n v0 dv g1)
        (delta-2 n v0 dv g1)))
  :hints (("Goal"
      :use ((:instance delta-rewrite-2-lemma3
             (A (* (expt (gamma) (* 2 n))
              (- (fdco (1- (m n v0 g1)) v0 dv g1)
                 (fdco (m n v0 g1) v0 dv g1))))
             (B (* (expt (gamma) (- (* 2 n) 1))
              (- (fdco (m n v0 g1) v0 dv g1)
                 (fdco (1+ (m n v0 g1)) v0 dv g1))))
             (C (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))
    1))
             (D (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c
    v0))))) 1)))))))
)

(local
(defthm delta-rewrite-3-lemma1-lemma1
  (implies (basic-params n 3)
      (equal (expt (gamma) (+ (+ -1 n -1 n) 2))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 2))))
```

```
  :hints (("Goal"
      :use ((:instance exponents-add-for-nonneg-exponents
             (i (+ -1 n -1 n))
             (j 2)
             (r (gamma))))
      :in-theory (disable exponents-add-for-nonneg-exponents
                 delta-rewrite-2-lemma2))))
)

(local
(defthm delta-rewrite-3-lemma1-stupidlemma
  (implies (basic-params n 3)
      (equal (* 2 n) (+ (+ -1 n -1 n) 2))))
)

(local
(defthm delta-rewrite-3-lemma1
  (implies (basic-params n 3)
      (equal (expt (gamma) (* 2 n))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 2))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3-lemma1-lemma1)
       (:instance delta-rewrite-3-lemma1-stupidlemma)))))
)

(local
 (defthm delta-rewrite-3-lemma2-lemma1-lemma1
   (implies (basic-params n 3)
       (>= (+ n n) 2))))

(local
 (defthm delta-rewrite-3-lemma2-lemma1-stupidlemma
   (implies (basic-params n 3)
       (>= (+ -1 n -1 n) 0))
   :hints (("GOal"
       :use ((:instance
```

```
      delta-rewrite-3-lemma2-lemma1-lemma1))))))

(local
 (defthm delta-rewrite-3-lemma2-lemma1-lemma2
   (implies (basic-params n 3)
       (integerp (+ -1 n -1 n)))
   ))

(local
 (defthm delta-rewrite-3-lemma2-lemma1-lemma3
   (implies (basic-params n 3)
       (>= (+ -1 n -1 n) 0))
   :hints (("Goal"
       :use ((:instance
      delta-rewrite-3-lemma2-lemma1-stupidlemma))))))

(local
(defthm delta-rewrite-3-lemma2-lemma1
  (implies (basic-params n 3)
      (equal (expt (gamma) (+ (+ -1 n -1 n) 1))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 1))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3-lemma2-lemma1-lemma2)
       (:instance delta-rewrite-3-lemma2-lemma1-lemma3)
       (:instance exponents-add-for-nonneg-exponents
             (i (+ -1 n -1 n))
             (j 1)
             (r (gamma))))
      )))
)

(local
(defthm delta-rewrite-3-lemma2-stupidlemma
  (implies (basic-params n 3)
      (equal (- (* 2 n) 1)
        (+ (+ -1 n -1 n) 1))))
```

```
)

(local
(defthm delta-rewrite-3-lemma2
  (implies (basic-params n 3)
      (equal (expt (gamma) (- (* 2 n) 1))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 1))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3-lemma2-lemma1)
       (:instance delta-rewrite-3-lemma2-stupidlemma))
      :in-theory (disable delta-rewrite-2-lemma2)))
  )
)

(local
(defthm delta-rewrite-3-lemma3
  (implies (basic-params n 3)
      (equal (* (expt (gamma) (- 2 (* 2 n)))
           (expt (gamma) (+ -1 n -1 n)))
        1))
  :hints (("Goal"
      :use ((:instance expt-minus
            (r (gamma))
            (i (- (- 2 (* 2 n))))))))))
)

(local
(defthm delta-rewrite-3
  (implies (basic-params n 3 v0 dv g1)
      (equal (+ (* (expt (gamma) (* 2 n))
         (- (fdco (1- (m n v0 g1)) v0 dv g1)
            (fdco (m n v0 g1) v0 dv g1)))
           (* (expt (gamma) (- (* 2 n) 1))
         (- (fdco (m n v0 g1) v0 dv g1)
            (fdco (1+ (m n v0 g1)) v0 dv g1)))
           (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
```

```
             (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))) 1)
            (* (expt (gamma) (+ -1 n -1 n))
          (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
  1)))
         (* (expt (gamma) (+ -1 n -1 n))
            (+ (* (expt (gamma) 2)
             (- (fdco (1- (m n v0 g1)) v0 dv g1)
                (fdco (m n v0 g1) v0 dv g1)))
          (* (expt (gamma) 1)
             (- (fdco (m n v0 g1) v0 dv g1)
                (fdco (1+ (m n v0 g1)) v0 dv g1)))
          (* (expt (gamma) (- 2 (* 2 n)))
             (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
              (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0))))) 1))
          (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
  1)))))
:hints
(("Goal"
  :in-theory (disable delta-rewrite-2-lemma1)
  :do-not-induct t
  :clause-processor
  (Smtlink clause
        '( (:expand ((:functions ((m integerp)
                   (gamma rationalp)
                   (mu rationalp)
                   (equ-c rationalp)
                   (fdco rationalp)
                   (dv0 rationalp)))
                (:expansion-level 1)))
           (:python-file "delta-rewrite-3")
           (:let ((expt_gamma_2n
              (expt (gamma) (* 2 n))
               rationalp)
             (expt_gamma_2n_minus_1
              (expt (gamma) (- (* 2 n) 1))
```

```
                 rationalp)
               (expt_gamma_2n_minus_2
                (expt (gamma) (+ -1 n -1 n))
                 rationalp)
               (expt_gamma_2
                (expt (gamma) 2)
                 rationalp)
               (expt_gamma_1
                (expt (gamma) 1)
                 rationalp)
               (expt_gamma_2_minus_2n
                (expt (gamma) (- 2 (* 2 n)))
                 rationalp)
               ))
             (:hypothesize ((equal expt_gamma_2n
                    (* expt_gamma_2n_minus_2 expt_gamma_2))
                  (equal expt_gamma_2n_minus_1
                    (* expt_gamma_2n_minus_2 expt_gamma_1))
                  (equal (* expt_gamma_2_minus_2n
    expt_gamma_2n_minus_2)
                    1)))
             (:use ((:type ())
               (:hypo ((delta-rewrite-3-lemma1)
                  (delta-rewrite-3-lemma2)
                  (delta-rewrite-3-lemma3)))
               (:main ()))))
          state))))
)

(local
(defthm delta-rewrite-4
  (implies (basic-params n 3 v0 dv g1)
      (equal (delta-2 n v0 dv g1)
        (delta-3 n v0 dv g1)))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3)))))
)
```

```

(defthm delta-rewrite-5
  (implies (basic-params n 3 v0 dv g1)
      (equal (delta n v0 dv g1)
        (delta-3 n v0 dv g1)))
  :hints (("Goal"
      :use ((:instance delta-rewrite-1)
       (:instance delta-rewrite-2)
       (:instance delta-rewrite-3)
       (:instance delta-rewrite-4)))))
)

(encapsulate ()

(local
(defthm delta-<-0-lemma1-lemma
  (implies (basic-params n 3 v0 dv g1)
      (implies (< (+ (* (expt (gamma) 2)
              (- (fdco (1- (m n v0 g1)) v0 dv g1)
            (fdco (m n v0 g1) v0 dv g1)))
           (* (expt (gamma) 1)
              (- (fdco (m n v0 g1) v0 dv g1)
            (fdco (1+ (m n v0 g1)) v0 dv g1)))
           (* (expt (gamma) (- 2 (* 2 n)))
              (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
               (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))
    1))
           (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1))
             0)
          (< (* (expt (gamma) (+ -1 n -1 n))
           (+ (* (expt (gamma) 2)
            (- (fdco (1- (m n v0 g1)) v0 dv g1)
               (fdco (m n v0 g1) v0 dv g1)))
              (* (expt (gamma) 1)
            (- (fdco (m n v0 g1) v0 dv g1)
               (fdco (1+ (m n v0 g1)) v0 dv g1)))
```

```
                (* (expt (gamma) (- 2 (* 2 n)))
              (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                    (1+ (* *beta* (+ (* g1 (1- n)) (equ-c
  v0))))) 1))
                (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
  1)))
               0)))
:hints (("Goal"
    :clause-processor
    (Smtlink clause
          '( (:expand ((:functions ((m integerp)
                           (gamma rationalp)
                           (mu rationalp)
                           (equ-c rationalp)
                           (fdco rationalp)
                           (dv0 rationalp)))
                        (:expansion-level 1)))
                (:python-file
  "delta-smaller-than-0-lemma1-lemma")
                (:let ((expt_gamma_2n
                   (expt (gamma) (* 2 n))
                    rationalp)
                  (expt_gamma_2n_minus_1
                   (expt (gamma) (- (* 2 n) 1))
                    rationalp)
                  (expt_gamma_2n_minus_2
                   (expt (gamma) (+ -1 n -1 n))
                    rationalp)
                  (expt_gamma_2
                   (expt (gamma) 2)
                    rationalp)
                  (expt_gamma_1
                   (expt (gamma) 1)
                    rationalp)
                  (expt_gamma_2_minus_2n
                   (expt (gamma) (- 2 (* 2 n)))
```

```
                  rationalp)
                ))
              (:hypothesize ((> expt_gamma_2n_minus_2 0))))
            state))))
)

(local
(defthm delta-<-0-lemma1
  (implies (basic-params n 3 v0 dv g1)
      (implies (< (delta-3-inside n v0 dv g1) 0)
          (< (delta-3 n v0 dv g1) 0))))
)

(local
(defthm delta-<-0-lemma2-lemma
  (implies (basic-params n 3 v0 dv g1)
      (implies (< (/ (+ (* (expt (gamma) 2)
            (- (fdco (1- (m n v0 g1)) v0 dv g1)
               (fdco (m n v0 g1) v0 dv g1)))
              (* (expt (gamma) 1)
            (- (fdco (m n v0 g1) v0 dv g1)
               (fdco (1+ (m n v0 g1)) v0 dv g1)))
              (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
               (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
    1))
           (- 1
              (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))))
             (expt (gamma) (- 2 (* 2 n))))
          (< (+ (* (expt (gamma) 2)
              (- (fdco (1- (m n v0 g1)) v0 dv g1)
            (fdco (m n v0 g1) v0 dv g1)))
           (* (expt (gamma) 1)
              (- (fdco (m n v0 g1) v0 dv g1)
            (fdco (1+ (m n v0 g1)) v0 dv g1)))
           (* (expt (gamma) (- 2 (* 2 n)))
              (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
```

```
              (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))
  1))
          (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
           (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1))
            0)))
 :hints (("Goal"
     :clause-processor
     (Smtlink clause
           '( (:expand ((:functions ((m integerp)
                        (gamma rationalp)
                        (mu rationalp)
                        (equ-c rationalp)
                        (fdco rationalp)
                        (dv0 rationalp)))
                     (:expansion-level 1)))
               (:python-file
  "delta-smaller-than-0-lemma2-lemma")
               (:let ((expt_gamma_2n
                  (expt (gamma) (* 2 n))
                   rationalp)
                 (expt_gamma_2n_minus_1
                  (expt (gamma) (- (* 2 n) 1))
                   rationalp)
                 (expt_gamma_2n_minus_2
                  (expt (gamma) (+ -1 n -1 n))
                   rationalp)
                 (expt_gamma_2
                  (expt (gamma) 2)
                   rationalp)
                 (expt_gamma_1
                  (expt (gamma) 1)
                   rationalp)
                 (expt_gamma_2_minus_2n
                  (expt (gamma) (- 2 (* 2 n)))
                   rationalp)
                 ))
               (:hypothesize ((> expt_gamma_2_minus_2n 0))))
```

```
            state))))
)

(local
(defthm delta-<-0-lemma2
  (implies (basic-params n 3 v0 dv g1)
      (implies (< (delta-3-inside-transform n v0 dv g1)
              (expt (gamma) (- 2 (* 2 n))))
          (< (delta-3-inside n v0 dv g1) 0)))
  :hints (("Goal"
      :use ((:instance delta-<-0-lemma2-lemma)))))
)

(local
;; This is for proving 2n < gamma^(2-2n)
(defthm delta-<-0-lemma3-lemma1
  (implies (and (integerp k)
      (>= k 6))
      (< k (expt (/ (gamma)) (- k 2)))))
)

(local
 (defthm delta-<-0-lemma3-lemma2-stupidlemma
   (implies (basic-params n 3)
       (>= n 3))))

(local
 (defthm delta-<-0-lemma3-lemma2-stupidlemma-omg
   (implies (and (rationalp a) (rationalp b) (>= a b))
       (>= (* 2 a) (* 2 b)))))

(local
 (defthm delta-<-0-lemma3-lemma2-lemma1
   (implies (basic-params n 3)
       (>= (* 2 n) 6))
   :hints (("Goal"
       :use ((:instance delta-<-0-lemma3-lemma2-stupidlemma)
```

```
       (:instance delta-<-0-lemma3-lemma2-stupidlemma-omg
             (a n)
             (b 3))
       ))))
 )

(local
(defthm delta-<-0-lemma3-lemma2
  (implies (basic-params n 3)
      (< (* 2 n)
         (expt (/ (gamma)) (- (* 2 n) 2))))
  :hints (("Goal"
      :use ((:instance delta-<-0-lemma3-lemma1
            (k (* 2 n)))
       (:instance delta-<-0-lemma3-lemma2-lemma1))))
  :rule-classes :linear)
)

(local
(defthm delta-<-0-lemma3-lemma3-stupidlemma
  (equal (expt a n) (expt (/ a) (- n))))
)

(local
(defthm delta-<-0-lemma3-lemma3
  (implies (basic-params n 3)
      (equal (expt (/ (gamma)) (- (* 2 n) 2))
        (expt (gamma) (- 2 (* 2 n)))))
  :hints (("Goal"
      :use ((:instance delta-<-0-lemma3-lemma3-stupidlemma
            (a (/ (gamma)))
            (n (- (* 2 n) 2))))
      :in-theory (disable
    delta-<-0-lemma3-lemma3-stupidlemma))))
)

(local
```

```
(defthm delta-<-0-lemma3-lemma4-stupidlemma
  (implies (and (< a b) (equal b c)) (< a c))
)

(local
(defthm delta-<-0-lemma3-lemma4
  (implies (basic-params n 3)
      (< (* 2 n)
         (expt (gamma) (- 2 (* 2 n)))))
  :hints (("Goal"
      :do-not '(preprocess simplify)
      :use ((:instance delta-<-0-lemma3-lemma2)
       (:instance delta-<-0-lemma3-lemma3)
       (:instance delta-<-0-lemma3-lemma4-stupidlemma
             (a (* 2 n))
             (b (expt (/ (gamma)) (- (* 2 n) 2)))
             (c (expt (gamma) (- 2 (* 2 n))))))
      :in-theory (disable delta-<-0-lemma3-lemma2
                delta-<-0-lemma3-lemma3
                delta-<-0-lemma3-lemma4-stupidlemma)))
  :rule-classes :linear)
)

(local
(defthm delta-<-0-lemma3
  (implies (basic-params n 3 v0 dv g1)
      (implies (< (/ (+ (* (expt (gamma) 2)
            (- (fdco (1- (m n v0 g1)) v0 dv g1)
               (fdco (m n v0 g1) v0 dv g1)))
              (* (expt (gamma) 1)
            (- (fdco (m n v0 g1) v0 dv g1)
               (fdco (1+ (m n v0 g1)) v0 dv g1)))
              (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
               (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
    1))
           (- 1
              (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
```

```
          (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))))
           (* 2 n))
        (< (/ (+ (* (expt (gamma) 2)
          (- (fdco (1- (m n v0 g1)) v0 dv g1)
             (fdco (m n v0 g1) v0 dv g1)))
            (* (expt (gamma) 1)
          (- (fdco (m n v0 g1) v0 dv g1)
             (fdco (1+ (m n v0 g1)) v0 dv g1)))
            (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
             (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
  1))
         (- 1
            (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))))
           (expt (gamma) (- 2 (* 2 n))))))
:hints (("Goal"
    :clause-processor
    (Smtlink clause
          '( (:expand ((:functions ((m integerp)
                     (gamma rationalp)
                     (mu rationalp)
                     (equ-c rationalp)
                     (fdco rationalp)
                     (dv0 rationalp)))
                  (:expansion-level 1)))
            (:python-file "delta-smaller-than-0-lemma3")
            (:let ((expt_gamma_2n
               (expt (gamma) (* 2 n))
                rationalp)
              (expt_gamma_2n_minus_1
               (expt (gamma) (- (* 2 n) 1))
                rationalp)
              (expt_gamma_2n_minus_2
               (expt (gamma) (+ -1 n -1 n))
                rationalp)
              (expt_gamma_2
               (expt (gamma) 2)
```

```
                  rationalp)
                (expt_gamma_1
                 (expt (gamma) 1)
                  rationalp)
                (expt_gamma_2_minus_2n
                 (expt (gamma) (- 2 (* 2 n)))
                  rationalp))
                )
              (:hypothesize ((< (* 2 n) expt_gamma_2_minus_2n)))
              (:use ((:type ())
                   (:hypo ((delta-<-0-lemma3-lemma4)))
                   (:main ())))
              )
            state)
      :in-theory (disable delta-<-0-lemma3-lemma1
                   delta-<-0-lemma3-lemma3-stupidlemma
                   delta-<-0-lemma3-lemma2
                   delta-<-0-lemma3-lemma3
                delta-<-0-lemma3-lemma4-stupidlemma)
      )))
)

(local
(defthm delta-<-0-lemma4
  (implies (basic-params n 3 v0 dv g1)
      (< (/ (+ (* (expt (gamma) 2)
             (- (fdco (1- (m n v0 g1)) v0 dv g1)
           (fdco (m n v0 g1) v0 dv g1)))
          (* (expt (gamma) 1)
             (- (fdco (m n v0 g1) v0 dv g1)
           (fdco (1+ (m n v0 g1)) v0 dv g1)))
          (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
           (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0))))) 1))
       (- 1
          (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
             (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))))
```

```
              (* 2 n)))
       :hints (("Goal"
           :clause-processor
           (Smtlink clause
                 '( (:expand ((:functions ((m integerp)
                                    (gamma rationalp)
                                    (mu rationalp)
                                    (equ-c rationalp)
                                    (fdco rationalp)
                                    (dv0 rationalp)))
                                 (:expansion-level 1)))
                   (:python-file "delta-smaller-than-0-lemma4")
                   (:let ((expt_gamma_2n
                       (expt (gamma) (* 2 n))
                        rationalp)
                     (expt_gamma_2n_minus_1
                       (expt (gamma) (- (* 2 n) 1))
                        rationalp)
                     (expt_gamma_2n_minus_2
                       (expt (gamma) (+ -1 n -1 n))
                        rationalp)
                     (expt_gamma_2
                       (expt (gamma) 2)
                        rationalp)
                     (expt_gamma_1
                       (expt (gamma) 1)
                        rationalp)
                     (expt_gamma_2_minus_2n
                       (expt (gamma) (- 2 (* 2 n)))
                        rationalp))
                     )
                   (:hypothesize ((equal expt_gamma_1 1/5)
                         (equal expt_gamma_2 1/25))))
                 state)
           :in-theory (disable delta-<-0-lemma3-lemma1
                        delta-<-0-lemma3-lemma3-stupidlemma
                        delta-<-0-lemma3-lemma2
```

```
                     delta-<-0-lemma3-lemma3
                  delta-<-0-lemma3-lemma4-stupidlemma
                  delta-<-0-lemma3-lemma4))))
)


(defthm delta-<-0
  (implies (basic-params n 3 v0 dv g1)
      (< (delta n v0 dv g1) 0))
  :hints (("Goal"
      :use ((:instance delta-rewrite-5)
       (:instance delta-<-0-lemma4)
       (:instance delta-<-0-lemma3)
       (:instance delta-<-0-lemma2)
       (:instance delta-<-0-lemma1))
      :in-theory (disable delta-<-0-lemma3-lemma1
                     delta-<-0-lemma3-lemma3-stupidlemma
                     delta-<-0-lemma3-lemma2
                     delta-<-0-lemma3-lemma3
                  delta-<-0-lemma3-lemma4-stupidlemma
                  delta-<-0-lemma3-lemma4)
     )))
) ;; delta < 0 thus is proved

;; prove phi(2n+1) = gamma^2*A+gamma*B+delta
(encapsulate ()

(local
(defthm split-phi-2n+1-lemma1-lemma1
  (implies (basic-params n 3 v0 dv g1 phi0)
      (equal (A (+ n 1) phi0 v0 dv g1)
        (+ (* (expt (gamma) (+ (* 2 n) 1)) phi0)
           (* (expt (gamma) (* 2 n))
         (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
           (* (expt (gamma) (- (* 2 n) 1))
         (- (fdco (m n v0 g1) v0 dv g1) 1))))))
)
```

```

(local
(defthm split-phi-2n+1-lemma1-lemma2
  (implies (basic-params n 3 v0 dv g1 phi0)
      (equal (+ (* (expt (gamma) (+ (* 2 n) 1)) phi0)
           (* (expt (gamma) (* 2 n))
          (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
           (* (expt (gamma) (- (* 2 n) 1))
          (- (fdco (m n v0 g1) v0 dv g1) 1)))
         (+ (* (+ (* (expt (gamma) (- (* 2 n) 1)) phi0)
            (* (expt (gamma) (- (* 2 n) 2))
               (- (fdco (m n v0 g1) v0 dv g1) 1))
            (* (expt (gamma) (- (* 2 n) 3))
               (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))
          (expt (gamma) 2))
           (- (* (expt (gamma) (* 2 n))
            (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
          (* (expt (gamma) (* 2 n))
            (- (fdco (m n v0 g1) v0 dv g1) 1)))
           (- (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (m n v0 g1) v0 dv g1) 1))
          (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1))))))
  )
)

(local
(defthm split-phi-2n+1-lemma1-A
  (implies (basic-params n 3 v0 dv g1 phi0)
      (equal (A (+ n 1) phi0 v0 dv g1)
         (+ (* (A n phi0 v0 dv g1) (gamma) (gamma))
           (- (* (expt (gamma) (* 2 n))
            (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
          (* (expt (gamma) (* 2 n))
            (- (fdco (m n v0 g1) v0 dv g1) 1)))
           (- (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (m n v0 g1) v0 dv g1) 1))
```

```
         (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma1
  (implies (basic-params n 3 v0 dv g1)
      (equal (B (+ n 1) v0 dv g1)
        (* (expt (gamma) (- n 1))
           (B-sum 1 (- n 1) v0 dv g1)))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma2
  (implies (basic-params n 3 v0 dv g1)
      (equal (B (+ n 1) v0 dv g1)
        (* (expt (gamma) (- n 1))
           (+ (B-term (- n 1) v0 dv g1)
         (B-term (- (- n 1)) v0 dv g1)
         (B-sum 1 (- n 2) v0 dv g1))))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma3
  (implies (basic-params n 3 v0 dv g1)
      (equal (B (+ n 1) v0 dv g1)
        (+ (* (expt (gamma) (- n 1))
         (B-sum 1 (- n 2) v0 dv g1))
           (* (expt (gamma) (- n 1))
         (B-term (- n 1) v0 dv g1))
           (* (expt (gamma) (- n 1))
         (B-term (- (- n 1)) v0 dv g1))))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma4
  (implies (basic-params n 3 v0 dv g1)
```

```
      (equal (B (+ n 1) v0 dv g1)
        (+ (* (gamma) (expt (gamma) (- n 2))
         (B-sum 1 (- n 2) v0 dv g1))
           (* (expt (gamma) (- n 1))
         (+ (B-term (- n 1) v0 dv g1)
            (B-term (- (- n 1)) v0 dv g1)))))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma5
  (implies (basic-params n 3 v0 dv g1)
      (equal (B (+ n 1) v0 dv g1)
        (+ (* (gamma) (B n v0 dv g1))
           (* (expt (gamma) (- n 1))
         (+ (B-term (- n 1) v0 dv g1)
            (B-term (- (- n 1)) v0 dv g1)))))))
)

(local
(defthm split-phi-2n+1-lemma2-B
  (implies (basic-params n 3 v0 dv g1)
      (equal (B (+ n 1) v0 dv g1)
        (+ (* (gamma) (B n v0 dv g1))
           (* (expt (gamma) (- n 1))
         (+ (* (expt (gamma) (- (- n 1)))
               (B-term-rest (- n 1) v0 dv g1))
            (* (expt (gamma) (- n 1))
               (B-term-rest (- (- n 1)) v0 dv g1))))))))
)

(local
(defthm split-phi-2n+1-lemma3-delta-stupidlemma
  (implies (basic-params n 3 v0 dv g1)
      (equal (+ (- (* (expt (gamma) (* 2 n))
            (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
         (* (expt (gamma) (* 2 n))
            (- (fdco (m n v0 g1) v0 dv g1) 1)))
```

```
              (- (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (m n v0 g1) v0 dv g1) 1))
            (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))
              (* (expt (gamma) (- n 1))
            (+ (* (expt (gamma) (- (- n 1)))
                  (B-term-rest (- n 1) v0 dv g1))
               (* (expt (gamma) (- n 1))
                  (B-term-rest (- (- n 1)) v0 dv g1)))))
          (+ (- (* (expt (gamma) (* 2 n))
               (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
            (* (expt (gamma) (* 2 n))
               (- (fdco (m n v0 g1) v0 dv g1) 1)))
              (- (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (m n v0 g1) v0 dv g1) 1))
            (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))
              (* (expt (gamma) (1- n))
            (+ (* (expt (gamma) (1+ (- n)))
                  (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                   (1+ (* *beta* (+ (* g1 (1- n)) (equ-c v0)))))
    1))
               (* (expt (gamma) (1- n))
                  (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                   (1+ (* *beta* (+ (* g1 (- 1 n)) (equ-c v0)))))
    1))))))))
)

(local
(defthm split-phi-2n+1-lemma3-delta
  (implies (basic-params n 3 v0 dv g1)
      (equal (+ (- (* (expt (gamma) (* 2 n))
               (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
            (* (expt (gamma) (* 2 n))
               (- (fdco (m n v0 g1) v0 dv g1) 1)))
              (- (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (m n v0 g1) v0 dv g1) 1))
```

```
        (* (expt (gamma) (- (* 2 n) 1))
           (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))
          (* (expt (gamma) (- n 1))
        (+ (* (expt (gamma) (- (- n 1)))
              (B-term-rest (- n 1) v0 dv g1))
           (* (expt (gamma) (- n 1))
              (B-term-rest (- (- n 1)) v0 dv g1)))))
       (delta n v0 dv g1)))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1-lemma3-delta-stupidlemma)
       (:instance delta)))))
)

(local
(defthm split-phi-2n+1-lemma4
  (implies (basic-params n 3 v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1)
        (+ (A (+ n 1) phi0 v0 dv g1)
           (B (+ n 1) v0 dv g1)))))
)

(local
(defthm split-phi-2n+1-lemma5
  (implies (basic-params n 3 v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1)
        (+ (+ (* (A n phi0 v0 dv g1) (gamma) (gamma))
         (- (* (expt (gamma) (* 2 n))
              (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
           (* (expt (gamma) (* 2 n))
              (- (fdco (m n v0 g1) v0 dv g1) 1)))
         (- (* (expt (gamma) (- (* 2 n) 1))
              (- (fdco (m n v0 g1) v0 dv g1) 1))
           (* (expt (gamma) (- (* 2 n) 1))
              (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1))))
          (+ (* (gamma) (B n v0 dv g1))
        (* (expt (gamma) (- n 1))
           (+ (* (expt (gamma) (- (- n 1)))
```

```
             (B-term-rest (- n 1) v0 dv g1))
               (* (expt (gamma) (- n 1))
             (B-term-rest (- (- n 1)) v0 dv g1))))))))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1-lemma1-A)
       (:instance split-phi-2n+1-lemma2-B)))))
)

(local
(defthm split-phi-2n+1-lemma6
  (implies (basic-params n 3 v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1)
        (+ (* (A n phi0 v0 dv g1) (gamma) (gamma))
           (* (gamma) (B n v0 dv g1))
           (+ (- (* (expt (gamma) (* 2 n))
               (- (fdco (1- (m n v0 g1)) v0 dv g1) 1))
            (* (expt (gamma) (* 2 n))
               (- (fdco (m n v0 g1) v0 dv g1) 1)))
         (- (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (m n v0 g1) v0 dv g1) 1))
            (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (1+ (m n v0 g1)) v0 dv g1) 1)))
         (* (expt (gamma) (- n 1))
            (+ (* (expt (gamma) (- (- n 1)))
             (B-term-rest (- n 1) v0 dv g1))
               (* (expt (gamma) (- n 1))
             (B-term-rest (- (- n 1)) v0 dv g1)))))))))
)

(defthm split-phi-2n+1
  (implies (basic-params n 3 v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1)
        (+ (* (gamma) (gamma) (A n phi0 v0 dv g1))
           (* (gamma) (B n v0 dv g1)) (delta n v0 dv g1))))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1-lemma6)
       (:instance split-phi-2n+1-lemma3-delta)))))
```

```
)

;; prove gamma^2*A + gamma*B < 0
(encapsulate ()

(local
(defthm except-for-delta-<-0-lemma1
  (implies (and (and (rationalp c)
                (rationalp a)
                (rationalp b))
      (and (> c 0)
                (< c 1)
                (< (+ A B) 0)
                (< B 0)))
      (< (+ (* c c A) (* c B)) 0))
  :hints (("Goal"
      :clause-processor
      (Smtlink clause
            '( (:expand ((:function ())
                    (:expansion-level 1)))
              (:python-file "except-for-delta-smaller-than-0-lemma1")
              (:let ())
              (:hypothesize ()))
            state)))
  :rule-classes :linear)
)

(defthm except-for-delta-<-0
  (implies (basic-params n 3 v0 dv g1 phi0 (< (phi-2n-1 n phi0 v0 dv g1) 0))
      (< (+ (* (gamma) (gamma) (A n phi0 v0 dv g1))
       (* (gamma) (B n v0 dv g1)))
         0))
  :hints (("Goal"
      :do-not-induct t
```

```
      :use ((:instance except-for-delta-<-0-lemma1
             (c (gamma))
             (A (A n phi0 v0 dv g1))
             (B (B n v0 dv g1)))
       (:instance B-neg)))))
)

;; for induction step
(encapsulate ()

(defthm phi-2n+1-<-0-inductive
  (implies (basic-params n 3 v0 dv g1 phi0 (< (phi-2n-1 n phi0
    v0 dv g1) 0))
      (< (phi-2n-1 (1+ n) phi0 v0 dv g1) 0))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1)
       (:instance delta-<-0)
       (:instance except-for-delta-<-0)))))

(defthm phi-2n+1-<-0-inductive-corollary
  (implies (basic-params (- i 1) 3 v0 dv g1 phi0
          (< (phi-2n-1 (- i 1) phi0 v0 dv g1) 0))
      (< (phi-2n-1 i phi0 v0 dv g1) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-inductive
             (n (- i 1)))))))

(defthm phi-2n+1-<-0-inductive-corollary-2
  (implies (basic-params (- i 1) 3 v0 dv g1 phi0
          (< (phi-2n-1 (- i 1) phi0 v0 dv g1) 0))
      (< (+ (A i phi0 v0 dv g1)
       (* (B-expt i)
          (B-sum 1 (- i 2) v0 dv g1))) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-inductive-corollary)))))

(defthm phi-2n+1-<-0-base
```

```
    (implies (basic-params-equal n 2 v0 dv g1 phi0)
      (< (phi-2n-1 (1+ n) phi0 v0 dv g1) 0))
  :hints (("Goal''"
      :clause-processor
      (Smtlink clause
            '( (:expand ((:function ())
                    (:expansion-level 1)))
              (:python-file "phi-2n+1-smaller-than-0-base")
              (:let ())
              (:hypothesize ()))
            state)))
  )

(defthm phi-2n+1-<-0-base-new
    (implies (basic-params-equal (- i 2) 1 v0 dv g1 phi0)
      (< (phi-2n-1 (- i 1) phi0 v0 dv g1) 0))
  :hints (("Goal''"
      :clause-processor
      (Smtlink clause
            '( (:expand ((:function ())
                    (:expansion-level 1)))
              (:python-file "phi-2n+1-smaller-than-0-base-new")
              (:let ())
              (:hypothesize ()))
            state)))
  )

(defthm phi-2n+1-<-0-base-corollary
  (implies (basic-params-equal (1- i) 2 v0 dv g1 phi0)
      (< (phi-2n-1 i phi0 v0 dv g1) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-base
             (n (- i 1))))))
  )

(defthm phi-2n+1-<-0-base-corollary-2
  (implies (basic-params-equal (1- i) 2 v0 dv g1 phi0)
```

```
      (< (+ (A i phi0 v0 dv g1)
       (* (B-expt i)
          (B-sum 1 (- i 2) v0 dv g1))) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-base-corollary))))
  )

(defthm stupid-proof
  (implies (and (equal a f)
      (equal a i)
      (implies (and m l) l)
      (implies l (and c h))
      (implies (and c h) (and c j))
           (implies (and a b c d) e)
      (implies (and f b c d) g)
      (implies (and f b h d e) g)
      i
      m
      (implies (and a b j d) e)
      f
      b
      l
      d)
      g)
  :rule-classes nil)

(defthm phi-2n+1-<-0-lemma-lemma1
  (implies
 (and
     (implies
          (and (and (integerp (+ -2 i))
                    (rationalp g1)
                    (rationalp v0)
                    (rationalp phi0)
                    (rationalp dv))
               (equal (+ -2 i) 1)
               (equal g1 1/3200)
```

```
           (<= 9/10 v0)
           (<= v0 11/10)
           (<= -1/8000 dv)
           (<= dv 1/8000)
           (<= 0 phi0)
           (< phi0
              (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
                    (/ (+ 1
                          (fix (* (+ 1
                                     (* (+ (fix (* (+ 1
(fix v0)) 1)) -1)
                                        (/ g1))
                                     -640)
                                  g1))))))))
      (< (phi-2n-1 (+ -1 i) phi0 v0 dv g1) 0))
 (implies
      (and (and (integerp (+ -1 i))
                (rationalp g1)
                (rationalp v0)
                (rationalp phi0)
                (rationalp dv))
           (equal (+ -1 i) 2)
           (equal g1 1/3200)
           (<= 9/10 v0)
           (<= v0 11/10)
           (<= -1/8000 dv)
           (<= dv 1/8000)
           (<= 0 phi0)
           (< phi0
              (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
                    (/ (+ 1
                          (fix (* (+ 1
                                     (* (+ (fix (* (+ 1
(fix v0)) 1)) -1)
                                        (/ g1))
```

```
                                     -640)
                                  g1))))))))
      (< (+ (a i phi0 v0 dv g1)
            (* (/ (expt 5 (+ -2 i)))
               (b-sum 1 (+ -2 i) v0 dv g1)))
         0))
 (implies
      (and (and (integerp (+ -1 i))
                (rationalp g1)
                (rationalp v0)
                (rationalp dv)
                (rationalp phi0))
           (<= 3 (+ -1 i))
           (<= (+ -1 i) 640)
           (equal g1 1/3200)
           (<= 9/10 v0)
           (<= v0 11/10)
           (<= -1/8000 dv)
           (<= dv 1/8000)
           (<= 0 phi0)
           (< phi0
              (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
                    (/ (+ 1
                          (fix (* (+ 1
                                     (* (+ (fix (* (+ 1
(fix v0)) 1)) -1)
                                        (/ g1))
                                     -640)
                                  g1)))))))
           (< (phi-2n-1 (+ -1 i) phi0 v0 dv g1) 0))
      (< (+ (a i phi0 v0 dv g1)
            (* (/ (expt 5 (+ -2 i)))
               (b-sum 1 (+ -2 i) v0 dv g1)))
         0))
 (not (or (not (integerp i)) (< i 1)))
 (implies
```

```
          (and (and (integerp (+ -1 -1 i))
                    (rationalp g1)
                    (rationalp v0)
                    (rationalp dv)
                    (rationalp phi0))
               (<= 2 (+ -1 -1 i))
               (<= (+ -1 -1 i) 640)
               (equal g1 1/3200)
               (<= 9/10 v0)
               (<= v0 11/10)
               (<= -1/8000 dv)
               (<= dv 1/8000)
               (<= 0 phi0)
               (< phi0
                  (+ -1
                     (* (fix (+ 1 (fix (+ v0 dv))))
                        (/ (+ 1
                              (fix (* (+ 1
                                         (* (+ (fix (* (+ 1
    (fix v0)) 1)) -1)
                                            (/ g1))
                                         -640)
                                      g1))))))))
          (< (+ (a (+ -1 i) phi0 v0 dv g1)
                (* (/ (expt 5 (+ -2 -1 i)))
                   (b-sum 1 (+ -2 -1 i) v0 dv g1)))
             0))
     (integerp (+ -1 i))
     (rationalp g1)
     (rationalp v0)
     (rationalp dv)
     (rationalp phi0)
     (<= 2 (+ -1 i))
     (<= (+ -1 i) 640)
     (equal g1 1/3200)
     (<= 9/10 v0)
     (<= v0 11/10)
```

```
    (<= -1/8000 dv)
    (<= dv 1/8000)
    (<= 0 phi0)
    (< phi0
       (+ -1
          (* (fix (+ 1 (fix (+ v0 dv))))
             (/ (+ 1
                   (fix (* (+ 1
                              (* (+ (fix (* (+ 1 (fix v0))
   1)) -1)
                                 (/ g1))
                              -640)
                           g1))))))))
(< (+ (a i phi0 v0 dv g1)
      (* (/ (expt 5 (+ -2 i)))
         (b-sum 1 (+ -2 i) v0 dv g1)))
   0))
 :hints (("Goal"
     :use ((:instance stupid-proof
           (a (integerp (+ -1 -1 i)))
           (b (and (rationalp g1)
              (rationalp v0)
              (rationalp dv)
              (rationalp phi0)))
           (c (equal (+ -2 i) 1))
           (d (and (equal g1 1/3200)
              (<= 9/10 v0)
              (<= v0 11/10)
              (<= -1/8000 dv)
              (<= dv 1/8000)
              (<= 0 phi0)
              (< phi0
                 (+ -1
               (* (fix (+ 1 (fix (+ v0 dv))))
                  (/ (+ 1
                   (fix (* (+ 1
                         (* (+ (fix (* (+ 1 (fix v0)) 1)) -1)
```

```
                         (/ g1))
                           -640)
                        g1)))))))))
             (e (< (+ (a (+ -1 i) phi0 v0 dv g1)
                 (* (/ (expt 5 (+ -2 -1 i)))
               (b-sum 1 (+ -2 -1 i) v0 dv g1)))
              0))
             (f (integerp (+ -1 i)))
             (g (< (+ (a i phi0 v0 dv g1)
                 (* (/ (expt 5 (+ -2 i)))
               (b-sum 1 (+ -2 i) v0 dv g1)))
              0))
             (h (and (<= 3 (+ -1 i))
                (<= (+ -1 i) 640)))
             (i (integerp i))
             (j (and (<= 2 (+ -1 -1 i))
                (<= (+ -1 -1 i) 640)))
             (l (and (<= 2 (+ -1 i))
                (<= (+ -1 i) 640)
                ))
             (m (>= i 1)))))))

(defthm phi-2n+1-<-0-lemma-lemma2
  (implies (and (or (not (integerp i)) (< i 1))
              (integerp (+ -1 i))
              (rationalp g1)
              (rationalp v0)
              (rationalp dv)
              (rationalp phi0)
              (<= 2 (+ -1 i))
              (<= (+ -1 i) 640)
              (equal g1 1/3200)
              (<= 9/10 v0)
              (<= v0 11/10)
              (<= -1/8000 dv)
              (<= dv 1/8000)
              (<= 0 phi0)
```

```
               (< phi0
                  (+ -1
                     (* (fix (+ 1 (fix (+ v0 dv))))
                        (/ (+ 1
                              (fix (* (+ 1
                                         (* (+ (fix (* (+ 1 (fix v0)) 1)) -1)
                                            (/ g1))
                                         -640)
                                      g1))))))))
          (< (+ (a i phi0 v0 dv g1)
                (* (/ (expt 5 (+ -2 i)))
                   (b-sum 1 (+ -2 i) v0 dv g1)))
             0))
  :rule-classes nil)

(defthm phi-2n+1-<-0-lemma
  (implies (basic-params (1- i) 2 v0 dv g1 phi0)
      (< (+ (A i phi0 v0 dv g1)
       (* (B-expt i)
          (B-sum 1 (- i 2) v0 dv g1))) 0))
  :hints (("Goal"
      :do-not '(simplify)
      :induct (B-sum 1 i v0 dv g1))
     ("Subgoal *1/2"
     :use ((:instance phi-2n+1-<-0-base-new)
      (:instance phi-2n+1-<-0-base-corollary-2)
      (:instance phi-2n+1-<-0-inductive-corollary-2)
      ))
     ("Subgoal *1/2''"
      :use ((:instance phi-2n+1-<-0-lemma-lemma1)))
     ("Subgoal *1/1'"
      :use ((:instance phi-2n+1-<-0-lemma-lemma2)))
     )
  )

(defthm phi-2n+1-<-0
```

```
  (implies (basic-params (1- i) 2 v0 dv g1 phi0)
      (< (phi-2n-1 i phi0 v0 dv g1) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-lemma))
      ))
  )

(defthm phi-2n-1-<-0
  (implies (basic-params n 3 v0 dv g1 phi0)
      (< (phi-2n-1 n phi0 v0 dv g1) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0
             (i n))))))
)
```

- Augmented proof with arbitrary c:

```
(in-package "ACL2")
(include-book "global")

(deftheory before-arith (current-theory :here))
(include-book "arithmetic/top-with-meta" :dir :system)
(deftheory after-arith (current-theory :here))

(deftheory arithmetic-book-only (set-difference-theories
    (theory 'after-arith) (theory 'before-arith)))

;; for the clause processor to work
(add-include-book-dir :cp
    "/ubc/cs/home/y/yanpeng/project/ACL2/smtlink")
(include-book "top" :dir :cp)
(logic)
:set-state-ok t
:set-ignore-ok t
(tshell-ensure)

;;:start-proof-tree
```

```
;; (encapsulate ()

;; (local (include-book "arithmetic-5/top" :dir :system))

;; (defun my-floor (x) (floor (numerator x) (denominator x)))

;; (defthm my-floor-type
;;   (implies (rationalp x)
;;       (integerp (my-floor x)))
;;   :rule-classes :type-prescription)

;; (defthm my-floor-lower-bound
;;   (implies (rationalp x)
;;       (> (my-floor x) (- x 1)))
;;   :rule-classes :linear)

;; (defthm my-floor-upper-bound
;;   (implies (rationalp x)
;;       (<= (my-floor x) x))
;;   :rule-classes :linear)

;; (defthm my-floor-comparison
;;   (implies (rationalp x)
;;       (< (my-floor (1- x)) (my-floor x)))
;;   :hints (("Goal"
;;       :use ((:instance my-floor-upper-bound (x (1- x)))
;;        (:instance my-floor-lower-bound))))
;;   :rule-classes :linear)
;; )

;; functions
;; n can be a rational value when c starts from non-integer
   value
(defun fdco (n v0 dv g1 dc)
  (/ (* (mu) (+ 1 (* *alpha* (+ v0 dv)))) (+ 1 (* *beta* (+ n
   dc) g1))))
```

```
(defun B-term-expt (h)
  (expt (gamma) (- h)))

(defun B-term-rest (h v0 dv g1 dc)
  (- (* (mu) (/ (+ 1 (* *alpha* (+ v0 dv))) (+ 1 (* *beta* (+
    (* (+ h dc) g1) (equ-c v0)))))) 1))

(defun B-term (h v0 dv g1 dc)
  (* (B-term-expt h) (B-term-rest h v0 dv g1 dc)))

(defun B-sum (h_lo h_hi v0 dv g1 dc)
  (declare (xargs :measure (if (or (not (integerp h_hi)) (not
    (integerp h_lo)) (< h_hi h_lo))
                0
                (1+ (- h_hi h_lo)))))
  (if (or (not (integerp h_hi)) (not (integerp h_lo)) (> h_lo
    h_hi))  0
      (+ (B-term h_hi v0 dv g1 dc) (B-term (- h_hi) v0 dv g1
    dc) (B-sum h_lo (- h_hi 1) v0 dv g1 dc))))

(defun B-expt (n)
  (expt (gamma) (- n 2)))

(defun B (n v0 dv g1 dc)
  (* (B-expt n)
     (B-sum 1 (- n 2) v0 dv g1 dc)))

;; parameter list functions
(defmacro basic-params-equal (n n-value &optional (dc 'nil)
    (v0 'nil) (dv 'nil) (g1 'nil) (phi0 'nil) (other 'nil))
  (list 'and
   (append
    (append
     (append
      (append
      (append (list 'and
```

```
          (list 'integerp n))
         (if (equal dc 'nil) nil (list (list 'rationalp dc))))
      (if (equal g1 'nil) nil (list (list 'rationalp g1))))
      (if (equal v0 'nil) nil (list (list 'rationalp v0))))
     (if (equal phi0 'nil) nil (list (list 'rationalp phi0))))
    (if (equal dv 'nil) nil (list (list 'rationalp dv))))
   (append
    (append
     (append
      (append
       (append
        (append
         (append
          (append
      (append
          (append
      (list 'and
            (list 'equal n n-value))
      (if (equal dc 'nil) nil (list (list '>= dc '0))))
          (if (equal dc 'nil) nil (list (list '< dc '1))))
      (if (equal g1 'nil) nil (list (list 'equal g1 '1/3200))))
          (if (equal v0 'nil) nil (list (list '>= v0 '9/10))))
         (if (equal v0 'nil) nil (list (list '<= v0 '11/10))))
        (if (equal dv 'nil) nil (list (list '>= dv (list '-
    (list 'dv0))))))
       (if (equal dv 'nil) nil (list (list '<= dv (list
    'dv0)))))
      (if (equal phi0 'nil) nil (list (list '>= phi0 '0))))
     (if (equal phi0 'nil) nil (list (list '< phi0 (list '-
    (list 'fdco (list '1+ (list 'm '640 v0 g1)) v0 dv g1 dc)
    '1)))))
    (if (equal other 'nil) nil (list other)))))

(defmacro basic-params (n nupper &optional (dc 'nil) (v0 'nil)
    (dv 'nil) (g1 'nil) (phi0 'nil) (other 'nil))
  (list 'and
   (append
```

```
 (append
  (append
   (append
   (append (list 'and
       (list 'integerp n))
      (if (equal dc 'nil) nil (list (list 'rationalp dc))))
   (if (equal g1 'nil) nil (list (list 'rationalp g1))))
   (if (equal v0 'nil) nil (list (list 'rationalp v0))))
  (if (equal dv 'nil) nil (list (list 'rationalp dv))))
 (if (equal phi0 'nil) nil (list (list 'rationalp phi0))))
(append
 (append
 (append
  (append
   (append
     (append
      (append
       (append
   (append
   (append
     (append (list 'and
         (list '>= n nupper))
        (list (list '<= n '640)))
     (if (equal dc 'nil) nil (list (list '>= dc '0))))
   (if (equal dc 'nil) nil (list (list '< dc '1))))
   (if (equal g1 'nil) nil (list (list 'equal g1 '1/3200))))
       (if (equal v0 'nil) nil (list (list '>= v0 '9/10))))
      (if (equal v0 'nil) nil (list (list '<= v0 '11/10))))
     (if (equal dv 'nil) nil (list (list '>= dv (list '-
(list 'dv0))))))
    (if (equal dv 'nil) nil (list (list '<= dv (list
'dv0)))))
   (if (equal phi0 'nil) nil (list (list '>= phi0 '0))))
  (if (equal phi0 'nil) nil (list (list '< phi0 (list '-
(list 'fdco (list '1+ (list 'm '640 v0 g1)) v0 dv g1 dc)
'1)))))
 (if (equal other 'nil) nil (list other)))))
```

```
(encapsulate ()

(local (in-theory (disable arithmetic-book-only)))

(local
(include-book "arithmetic-5/top" :dir :system)
)

(local
(defthm B-term-neg-lemma1
  (implies (basic-params h 1 dc v0 dv g1)
      (< (+ (* (B-term-expt h) (B-term-rest h v0 dv g1 dc))
          (* (B-term-expt (- h)) (B-term-rest (- h) v0 dv g1 dc)))
         0)
      )
  :hints
  (("Goal"
    :clause-processor
    (Smtlink clause
          '( (:expand ((:functions ((B-term-rest rationalp)
                      (gamma rationalp)
                      (mu rationalp)
                      (equ-c rationalp)
                      (dv0 rationalp)))
                   (:expansion-level 1)))
              (:python-file "B-term-neg-lemma1") ;;mktemp
              (:let ((expt_gamma_h (B-term-expt h) rationalp)
                (expt_gamma_minus_h (B-term-expt (- h)) rationalp)))
              (:hypothesize ((<= expt_gamma_minus_h (/ 1 5))
                   (> expt_gamma_minus_h 0)
                   (equal (* expt_gamma_minus_h expt_gamma_h) 1)))
             (:use ((:let ())
               (:hypo (()))
```

```
               (:main ()))))
           state)
     ))
   )
)

(defthm B-term-neg
  (implies (basic-params h 1 dc v0 dv g1)
      (< (+ (B-term h v0 dv g1 dc) (B-term (- h) v0 dv g1 dc))
    0))
  :hints (("Goal"
      :use ( (:instance B-term)
       (:instance B-term-neg-lemma1)
        )))
  :rule-classes :linear)
)

(defthm B-sum-neg
  (implies (basic-params n-minus-2 1 dc v0 dv g1)
      (< (B-sum 1 n-minus-2 v0 dv g1 dc) 0))
  :hints (("Goal"
      :in-theory (disable B-term)
      :induct ())))

(encapsulate ()

(local ;; B = B-expt*B-sum
 (defthm B-neg-lemma1
   (implies (basic-params n 3 dc v0 dv g1)
       (equal (B n v0 dv g1 dc)
         (* (B-expt n)
            (B-sum 1 (- n 2) v0 dv g1 dc))))))

(local
 (defthm B-expt->-0
   (implies (basic-params n 3)
       (> (B-expt n) 0))
```

```
   :rule-classes :linear))

(local
 (defthm B-neg-lemma2
   (implies (and (rationalp a)
       (rationalp b)
       (> a 0)
       (< b 0))
       (< (* a b) 0))
   :rule-classes :linear))

(local
 (defthm B-neg-type-lemma3
   (implies (and (and (rationalp n-minus-2) (rationalp v0) (rationalp g1) (rationalp dv) (rationalp dc)))
       (rationalp (B-sum 1 n-minus-2 v0 dv g1 dc)))
   :rule-classes :type-prescription))

(local
 (defthm B-neg-type-lemma4
   (implies (basic-params n 3)
       (rationalp (B-expt n)))
   :rule-classes :type-prescription))

(defthm B-neg
  (implies (basic-params n 3 dc v0 dv g1)
      (< (B n v0 dv g1 dc) 0))
  :hints (("Goal"
      :do-not-induct t
      :in-theory (disable B-expt B-sum B-sum-neg B-expt->-0)
      :use ((:instance B-sum-neg (n-minus-2 (- n 2)))
       (:instance B-expt->-0)
       (:instance B-neg-type-lemma3 (n-minus-2 (- n 2)))
       (:instance B-neg-type-lemma4)
       (:instance B-neg-lemma2 (a (B-expt n))
                         (b (B-sum 1 (+ -2 n) v0 dv g1 dc)))))))
```

```
)

(defun A (n phi0 v0 dv g1 dc)
  (+ (* (expt (gamma) (- (* 2 n) 1)) phi0)
     (* (expt (gamma) (- (* 2 n) 2))
   (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
     (* (expt (gamma) (- (* 2 n) 3))
   (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1))))

(defun phi-2n-1 (n phi0 v0 dv g1 dc)
  (+ (A n phi0 v0 dv g1 dc) (B n v0 dv g1 dc)))

(defun delta (n v0 dv g1 dc)
  (+ (- (* (expt (gamma) (* 2 n))
      (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
   (* (expt (gamma) (* 2 n))
      (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
     (- (* (expt (gamma) (- (* 2 n) 1))
      (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
   (* (expt (gamma) (- (* 2 n) 1))
      (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))
     (* (expt (gamma) (1- n))
   (+ (* (expt (gamma) (1+ (- n)))
         (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0)))))
       1))
      (* (expt (gamma) (1- n))
         (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0)))))
       1))))))

(defun delta-1 (n v0 dv g1 dc)
  (+ (* (expt (gamma) (* 2 n))
   (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
      (fdco (m n v0 g1) v0 dv g1 dc)))
     (* (expt (gamma) (- (* 2 n) 1))
   (- (fdco (m n v0 g1) v0 dv g1 dc)
```

```
      (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
     (* (* (expt (gamma) (1- n)) (expt (gamma) (1+ (- n))))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0)))))
    1))
     (* (* (expt (gamma) (1- n)) (expt (gamma) (1- n)))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0)))))
    1))))

(defun delta-2 (n v0 dv g1 dc)
  (+ (* (expt (gamma) (* 2 n))
   (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
      (fdco (m n v0 g1) v0 dv g1 dc)))
     (* (expt (gamma) (- (* 2 n) 1))
   (- (fdco (m n v0 g1) v0 dv g1 dc)
      (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
     (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
      (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))) 1)
     (* (expt (gamma) (+ -1 n -1 n))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0)))))
    1))))

(defun delta-3 (n v0 dv g1 dc)
  (* (expt (gamma) (+ -1 n -1 n))
     (+ (* (expt (gamma) 2)
      (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
         (fdco (m n v0 g1) v0 dv g1 dc)))
   (* (expt (gamma) 1)
      (- (fdco (m n v0 g1) v0 dv g1 dc)
         (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
   (* (expt (gamma) (- 2 (* 2 n)))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
       (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))) 1))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0)))))
```

```
    1))))

(defun delta-3-inside (n v0 dv g1 dc)
  (+ (* (expt (gamma) 2)
      (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
         (fdco (m n v0 g1) v0 dv g1 dc)))
   (* (expt (gamma) 1)
      (- (fdco (m n v0 g1) v0 dv g1 dc)
         (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
   (* (expt (gamma) (- 2 (* 2 n)))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
       (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))) 1))
   (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
         (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0)))))
    1)))

(defun delta-3-inside-transform (n v0 dv g1 dc)
  (/
   (+ (* (expt (gamma) 2)
    (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
       (fdco (m n v0 g1) v0 dv g1 dc)))
      (* (expt (gamma) 1)
    (- (fdco (m n v0 g1) v0 dv g1 dc)
       (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
       (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0)))))
    1))
   (- 1
      (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
    (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))))))

;; rewrite delta term
(encapsulate ()

(local
;; considering using smtlink for the proof, probably simpler
(defthm delta-rewrite-1-lemma1
```

```
(implies (basic-params n 3 dc v0 dv g1)
    (equal (+ (- (* (expt (gamma) (* 2 n))
          (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
       (* (expt (gamma) (* 2 n))
          (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
         (- (* (expt (gamma) (- (* 2 n) 1))
          (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
       (* (expt (gamma) (- (* 2 n) 1))
          (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))
         (* (expt (gamma) (1- n))
       (+ (* (expt (gamma) (1+ (- n)))
             (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
              (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
 v0)))))
           1))
          (* (expt (gamma) (1- n))
             (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
              (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
 v0)))))
           1)))))
        (+ (* (expt (gamma) (* 2 n))
         (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
            (fdco (m n v0 g1) v0 dv g1 dc)))
           (* (expt (gamma) (- (* 2 n) 1))
         (- (fdco (m n v0 g1) v0 dv g1 dc)
            (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
           (* (* (expt (gamma) (1- n)) (expt (gamma) (1+ (-
 n))))
         (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
 v0))))) 1))
           (* (* (expt (gamma) (1- n)) (expt (gamma) (1- n)))
         (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
 v0))))) 1)))))
:hints
(("Goal"
```

```
    :clause-processor
    (Smtlink clause
          '( (:expand ((:functions ((m integerp)
                          (gamma rationalp)
                          (mu rationalp)
                          (equ-c rationalp)
                          (fdco rationalp)
                          (dv0 rationalp)))
                       (:expansion-level 1)))
             (:python-file "delta-rewrite-1-lemma1") ;;mktemp
             (:let ((expt_gamma_2n
                     (expt (gamma) (* 2 n))
                      rationalp)
                    (expt_gamma_2n_minus_1
                     (expt (gamma) (- (* 2 n) 1))
                      rationalp)
                    (expt_gamma_n_minus_1
                     (expt (gamma) (1- n))
                      rationalp)
                    (expt_gamma_1_minus_n
                     (expt (gamma) (1+ (- n)))
                      rationalp)
                    ))
             (:hypothesize ()))
          state)
    )))
)

(local
(defthm delta-rewrite-1
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (delta n v0 dv g1 dc)
        (delta-1 n v0 dv g1 dc))))
)

(local
(defthm delta-rewrite-2-lemma1
```

```
  (implies (basic-params n 3)
      (equal (* (expt (gamma) (1- n))
           (expt (gamma) (1+ (- n))))
        1))
  :hints (("Goal"
      :use ((:instance expt-minus
                (r (gamma))
                (i (- (1+ (- n))))))
      )))
)

(local
(defthm delta-rewrite-2-lemma2
  (implies (basic-params n 3)
      (equal (* (expt (gamma) (1- n))
           (expt (gamma) (1- n)))
        (expt (gamma) (+ -1 n -1 n))))
  :hints (("Goal"
      :do-not-induct t
      :use ((:instance exponents-add-for-nonneg-exponents
                (i (1- n))
                (j (1- n))
                (r (gamma))))
      :in-theory (disable exponents-add-for-nonneg-exponents)
      ))
  )
)

(local
(defthm delta-rewrite-2-lemma3
  (implies (basic-params n 3)
      (equal (+ A
           B
           (* (* (expt (gamma) (1- n))
            (expt (gamma) (1+ (- n))))
         C)
           (* (* (expt (gamma) (1- n))
```

```
             (expt (gamma) (1- n)))
          D))
         (+ A B C
            (* (expt (gamma) (+ -1 n -1 n)) D))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-2-lemma1)
       (:instance delta-rewrite-2-lemma2)))))
)

(local
(defthm delta-rewrite-2
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (delta-1 n v0 dv g1 dc)
        (delta-2 n v0 dv g1 dc)))
  :hints (("Goal"
      :use ((:instance delta-rewrite-2-lemma3
             (A (* (expt (gamma) (* 2 n))
              (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
                 (fdco (m n v0 g1) v0 dv g1 dc))))
             (B (* (expt (gamma) (- (* 2 n) 1))
              (- (fdco (m n v0 g1) v0 dv g1 dc)
                 (fdco (1+ (m n v0 g1)) v0 dv g1 dc))))
             (C (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))) 1))
             (D (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0))))) 1)))))))
)

(local
(defthm delta-rewrite-3-lemma1-lemma1
  (implies (basic-params n 3)
      (equal (expt (gamma) (+ (+ -1 n -1 n) 2))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 2))))
  :hints (("Goal"
```

```
      :use ((:instance exponents-add-for-nonneg-exponents
             (i (+ -1 n -1 n))
             (j 2)
             (r (gamma))))
      :in-theory (disable exponents-add-for-nonneg-exponents
                  delta-rewrite-2-lemma2))))
)

(local
(defthm delta-rewrite-3-lemma1-stupidlemma
  (implies (basic-params n 3)
      (equal (* 2 n) (+ (+ -1 n -1 n) 2))))
)

(local
(defthm delta-rewrite-3-lemma1
  (implies (basic-params n 3)
      (equal (expt (gamma) (* 2 n))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 2))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3-lemma1-lemma1)
       (:instance delta-rewrite-3-lemma1-stupidlemma)))))
)

(local
 (defthm delta-rewrite-3-lemma2-lemma1-lemma1
   (implies (basic-params n 3)
       (>= (+ n n) 2))))

(local
 (defthm delta-rewrite-3-lemma2-lemma1-stupidlemma
   (implies (basic-params n 3)
       (>= (+ -1 n -1 n) 0))
   :hints (("GOal"
       :use ((:instance
    delta-rewrite-3-lemma2-lemma1-lemma1))))))
```

```
(local
 (defthm delta-rewrite-3-lemma2-lemma1-lemma2
   (implies (basic-params n 3)
       (integerp (+ -1 n -1 n)))
   ))

(local
 (defthm delta-rewrite-3-lemma2-lemma1-lemma3
   (implies (basic-params n 3)
       (>= (+ -1 n -1 n) 0))
   :hints (("Goal"
       :use ((:instance
    delta-rewrite-3-lemma2-lemma1-stupidlemma))))))

(local
(defthm delta-rewrite-3-lemma2-lemma1
  (implies (basic-params n 3)
      (equal (expt (gamma) (+ (+ -1 n -1 n) 1))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 1))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3-lemma2-lemma1-lemma2)
       (:instance delta-rewrite-3-lemma2-lemma1-lemma3)
       (:instance exponents-add-for-nonneg-exponents
             (i (+ -1 n -1 n))
             (j 1)
             (r (gamma))))
      )))
)

(local
(defthm delta-rewrite-3-lemma2-stupidlemma
  (implies (basic-params n 3)
      (equal (- (* 2 n) 1)
        (+ (+ -1 n -1 n) 1))))
)
```

```
(local
(defthm delta-rewrite-3-lemma2
  (implies (basic-params n 3)
      (equal (expt (gamma) (- (* 2 n) 1))
        (* (expt (gamma) (+ -1 n -1 n))
           (expt (gamma) 1))))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3-lemma2-lemma1)
       (:instance delta-rewrite-3-lemma2-stupidlemma))
      :in-theory (disable delta-rewrite-2-lemma2)))
  )
)

(local
(defthm delta-rewrite-3-lemma3
  (implies (basic-params n 3)
      (equal (* (expt (gamma) (- 2 (* 2 n)))
           (expt (gamma) (+ -1 n -1 n)))
        1))
  :hints (("Goal"
      :use ((:instance expt-minus
             (r (gamma))
             (i (- (- 2 (* 2 n)))))))))
)

(local
(defthm delta-rewrite-3
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (+ (* (expt (gamma) (* 2 n))
         (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
            (fdco (m n v0 g1) v0 dv g1 dc)))
           (* (expt (gamma) (- (* 2 n) 1))
         (- (fdco (m n v0 g1) v0 dv g1 dc)
            (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
           (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
```

```
v0))))) 1)
        (* (expt (gamma) (+ -1 n -1 n))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
v0))))) 1)))
     (* (expt (gamma) (+ -1 n -1 n))
        (+ (* (expt (gamma) 2)
         (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
            (fdco (m n v0 g1) v0 dv g1 dc)))
      (* (expt (gamma) 1)
         (- (fdco (m n v0 g1) v0 dv g1 dc)
            (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
      (* (expt (gamma) (- 2 (* 2 n)))
         (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
          (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
v0))))) 1))
      (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
v0))))) 1)))))
:hints
(("Goal"
  :in-theory (disable delta-rewrite-2-lemma1)
  :do-not-induct t
  :clause-processor
  (Smtlink clause
        '( (:expand ((:functions ((m integerp)
                    (gamma rationalp)
                    (mu rationalp)
                    (equ-c rationalp)
                    (fdco rationalp)
                    (dv0 rationalp)))
                 (:expansion-level 1)))
           (:python-file "delta-rewrite-3")
           (:let ((expt_gamma_2n
              (expt (gamma) (* 2 n))
               rationalp)
             (expt_gamma_2n_minus_1
```

```
                 (expt (gamma) (- (* 2 n) 1))
                  rationalp)
                (expt_gamma_2n_minus_2
                 (expt (gamma) (+ -1 n -1 n))
                  rationalp)
                (expt_gamma_2
                 (expt (gamma) 2)
                  rationalp)
                (expt_gamma_1
                 (expt (gamma) 1)
                  rationalp)
                (expt_gamma_2_minus_2n
                 (expt (gamma) (- 2 (* 2 n)))
                  rationalp)
                ))
              (:hypothesize ((equal expt_gamma_2n
                       (* expt_gamma_2n_minus_2 expt_gamma_2))
                     (equal expt_gamma_2n_minus_1
                       (* expt_gamma_2n_minus_2 expt_gamma_1))
                     (equal (* expt_gamma_2_minus_2n
      expt_gamma_2n_minus_2)
                       1)))
              (:use ((:type ())
                (:hypo ((delta-rewrite-3-lemma1)
                   (delta-rewrite-3-lemma2)
                   (delta-rewrite-3-lemma3)))
                (:main ()))))
          state))))
)

(local
(defthm delta-rewrite-4
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (delta-2 n v0 dv g1 dc)
        (delta-3 n v0 dv g1 dc)))
  :hints (("Goal"
      :use ((:instance delta-rewrite-3)))))
```

```
)

(defthm delta-rewrite-5
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (delta n v0 dv g1 dc)
        (delta-3 n v0 dv g1 dc)))
  :hints (("Goal"
      :use ((:instance delta-rewrite-1)
       (:instance delta-rewrite-2)
       (:instance delta-rewrite-3)
       (:instance delta-rewrite-4)))))
)

(encapsulate ()

(local
(defthm delta-<-0-lemma1-lemma
  (implies (basic-params n 3 dc v0 dv g1)
      (implies (< (+ (* (expt (gamma) 2)
              (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
            (fdco (m n v0 g1) v0 dv g1 dc)))
           (* (expt (gamma) 1)
              (- (fdco (m n v0 g1) v0 dv g1 dc)
            (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
           (* (expt (gamma) (- 2 (* 2 n)))
              (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
               (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))) 1))
           (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0))))) 1))
            0)
          (< (* (expt (gamma) (+ -1 n -1 n))
           (+ (* (expt (gamma) 2)
            (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
               (fdco (m n v0 g1) v0 dv g1 dc)))
              (* (expt (gamma) 1)
```

```
            (- (fdco (m n v0 g1) v0 dv g1 dc)
               (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
             (* (expt (gamma) (- 2 (* 2 n)))
            (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                  (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))) 1))
             (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
               (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0))))) 1)))
            0)))
  :hints (("Goal"
      :clause-processor
      (Smtlink clause
            '( (:expand ((:functions ((m integerp)
                          (gamma rationalp)
                          (mu rationalp)
                          (equ-c rationalp)
                          (fdco rationalp)
                          (dv0 rationalp)))
                       (:expansion-level 1)))
             (:python-file "delta-smaller-than-0-lemma1-lemma")
             (:let ((expt_gamma_2n
                  (expt (gamma) (* 2 n))
                   rationalp)
                 (expt_gamma_2n_minus_1
                  (expt (gamma) (- (* 2 n) 1))
                   rationalp)
                 (expt_gamma_2n_minus_2
                  (expt (gamma) (+ -1 n -1 n))
                   rationalp)
                 (expt_gamma_2
                  (expt (gamma) 2)
                   rationalp)
                 (expt_gamma_1
                  (expt (gamma) 1)
                   rationalp)
```

```
                  (expt_gamma_2_minus_2n
                   (expt (gamma) (- 2 (* 2 n)))
                    rationalp)
                  ))
                (:hypothesize ((> expt_gamma_2n_minus_2 0))))
              state))))
)

(local
(defthm delta-<-0-lemma1
  (implies (basic-params n 3 dc v0 dv g1)
      (implies (< (delta-3-inside n v0 dv g1 dc) 0)
          (< (delta-3 n v0 dv g1 dc) 0))))
)

(local
(defthm delta-<-0-lemma2-lemma
  (implies (basic-params n 3 dc v0 dv g1)
      (implies (< (/ (+ (* (expt (gamma) 2)
            (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
               (fdco (m n v0 g1) v0 dv g1 dc)))
              (* (expt (gamma) 1)
            (- (fdco (m n v0 g1) v0 dv g1 dc)
               (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
              (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
               (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
    v0))))) 1))
           (- 1
              (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
    v0)))))))
             (expt (gamma) (- 2 (* 2 n))))
          (< (+ (* (expt (gamma) 2)
              (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
            (fdco (m n v0 g1) v0 dv g1 dc)))
           (* (expt (gamma) 1)
              (- (fdco (m n v0 g1) v0 dv g1 dc)
```

```
           (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
          (* (expt (gamma) (- 2 (* 2 n)))
             (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
              (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c v0))))) 1))
          (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
           (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c v0))))) 1))
            0)))
 :hints (("Goal"
     :clause-processor
     (Smtlink clause
           '( (:expand ((:functions ((m integerp)
                      (gamma rationalp)
                      (mu rationalp)
                      (equ-c rationalp)
                      (fdco rationalp)
                      (dv0 rationalp)))
                   (:expansion-level 1)))
             (:python-file "delta-smaller-than-0-lemma2-lemma")
             (:let ((expt_gamma_2n
                (expt (gamma) (* 2 n))
                 rationalp)
               (expt_gamma_2n_minus_1
                (expt (gamma) (- (* 2 n) 1))
                 rationalp)
               (expt_gamma_2n_minus_2
                (expt (gamma) (+ -1 n -1 n))
                 rationalp)
               (expt_gamma_2
                (expt (gamma) 2)
                 rationalp)
               (expt_gamma_1
                (expt (gamma) 1)
                 rationalp)
               (expt_gamma_2_minus_2n
```

```
                 (expt (gamma) (- 2 (* 2 n)))
                  rationalp)
                ))
              (:hypothesize ((> expt_gamma_2_minus_2n 0))))
            state))))
)

(local
(defthm delta-<-0-lemma2
  (implies (basic-params n 3 dc v0 dv g1)
      (implies (< (delta-3-inside-transform n v0 dv g1 dc)
             (expt (gamma) (- 2 (* 2 n))))
          (< (delta-3-inside n v0 dv g1 dc) 0)))
  :hints (("Goal"
      :use ((:instance delta-<-0-lemma2-lemma)))))
)

(local
;; This is for proving 2n < gamma^(2-2n)
(defthm delta-<-0-lemma3-lemma1
  (implies (and (integerp k)
      (>= k 6))
      (< k (expt (/ (gamma)) (- k 2)))))
)

(local
 (defthm delta-<-0-lemma3-lemma2-stupidlemma
   (implies (basic-params n 3)
       (>= n 3))))

(local
 (defthm delta-<-0-lemma3-lemma2-stupidlemma-omg
   (implies (and (rationalp a) (rationalp b) (>= a b))
       (>= (* 2 a) (* 2 b)))))

(local
 (defthm delta-<-0-lemma3-lemma2-lemma1
```

```
   (implies (basic-params n 3)
       (>= (* 2 n) 6))
   :hints (("Goal"
       :use ((:instance delta-<-0-lemma3-lemma2-stupidlemma)
        (:instance delta-<-0-lemma3-lemma2-stupidlemma-omg
              (a n)
              (b 3))
        ))))
 )

(local
(defthm delta-<-0-lemma3-lemma2
  (implies (basic-params n 3)
      (< (* 2 n)
         (expt (/ (gamma)) (- (* 2 n) 2))))
  :hints (("Goal"
      :use ((:instance delta-<-0-lemma3-lemma1
            (k (* 2 n)))
       (:instance delta-<-0-lemma3-lemma2-lemma1))))
  :rule-classes :linear)
)

(local
(defthm delta-<-0-lemma3-lemma3-stupidlemma
  (equal (expt a n) (expt (/ a) (- n))))
)

(local
(defthm delta-<-0-lemma3-lemma3
  (implies (basic-params n 3)
      (equal (expt (/ (gamma)) (- (* 2 n) 2))
        (expt (gamma) (- 2 (* 2 n)))))
  :hints (("Goal"
      :use ((:instance delta-<-0-lemma3-lemma3-stupidlemma
           (a (/ (gamma)))
           (n (- (* 2 n) 2))))
      :in-theory (disable
```

```
    delta-<-0-lemma3-lemma3-stupidlemma))))
)

(local
(defthm delta-<-0-lemma3-lemma4-stupidlemma
  (implies (and (< a b) (equal b c)) (< a c)))
)

(local
(defthm delta-<-0-lemma3-lemma4
  (implies (basic-params n 3)
      (< (* 2 n)
         (expt (gamma) (- 2 (* 2 n)))))
  :hints (("Goal"
      :do-not '(preprocess simplify)
      :use ((:instance delta-<-0-lemma3-lemma2)
       (:instance delta-<-0-lemma3-lemma3)
       (:instance delta-<-0-lemma3-lemma4-stupidlemma
             (a (* 2 n))
             (b (expt (/ (gamma)) (- (* 2 n) 2)))
             (c (expt (gamma) (- 2 (* 2 n))))))
      :in-theory (disable delta-<-0-lemma3-lemma2
                delta-<-0-lemma3-lemma3
                delta-<-0-lemma3-lemma4-stupidlemma)))
  :rule-classes :linear)
)

(local
(defthm delta-<-0-lemma3
  (implies (basic-params n 3 dc v0 dv g1)
      (implies (< (/ (+ (* (expt (gamma) 2)
             (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
                (fdco (m n v0 g1) v0 dv g1 dc)))
               (* (expt (gamma) 1)
             (- (fdco (m n v0 g1) v0 dv g1 dc)
                (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
               (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
```

```
                (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
   v0))))) 1))
            (- 1
               (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
             (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
   v0)))))))
              (* 2 n))
           (< (/ (+ (* (expt (gamma) 2)
             (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
                (fdco (m n v0 g1) v0 dv g1 dc)))
               (* (expt (gamma) 1)
             (- (fdco (m n v0 g1) v0 dv g1 dc)
                (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
               (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
   v0))))) 1))
            (- 1
               (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
             (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
   v0)))))))
              (expt (gamma) (- 2 (* 2 n))))))
 :hints (("Goal"
     :clause-processor
     (Smtlink clause
           '( (:expand ((:functions ((m integerp)
                           (gamma rationalp)
                           (mu rationalp)
                           (equ-c rationalp)
                           (fdco rationalp)
                           (dv0 rationalp)))
                       (:expansion-level 1)))
               (:python-file "delta-smaller-than-0-lemma3")
               (:let ((expt_gamma_2n
                  (expt (gamma) (* 2 n))
                   rationalp)
                 (expt_gamma_2n_minus_1
                  (expt (gamma) (- (* 2 n) 1))
```

```
                   rationalp)
                 (expt_gamma_2n_minus_2
                  (expt (gamma) (+ -1 n -1 n))
                   rationalp)
                 (expt_gamma_2
                  (expt (gamma) 2)
                   rationalp)
                 (expt_gamma_1
                  (expt (gamma) 1)
                   rationalp)
                 (expt_gamma_2_minus_2n
                  (expt (gamma) (- 2 (* 2 n)))
                   rationalp))
                 )
               (:hypothesize ((< (* 2 n)
    expt_gamma_2_minus_2n)))
               (:use ((:type ())
                      (:hypo ((delta-<-0-lemma3-lemma4)))
                      (:main ())))
               )
             state)
      :in-theory (disable delta-<-0-lemma3-lemma1
                          delta-<-0-lemma3-lemma3-stupidlemma
                          delta-<-0-lemma3-lemma2
                          delta-<-0-lemma3-lemma3
                    delta-<-0-lemma3-lemma4-stupidlemma)
      )))
)

(local
(defthm delta-<-0-lemma4
  (implies (basic-params n 3 dc v0 dv g1)
      (< (/ (+ (* (expt (gamma) 2)
             (- (fdco (1- (m n v0 g1)) v0 dv g1 dc)
           (fdco (m n v0 g1) v0 dv g1 dc)))
          (* (expt (gamma) 1)
             (- (fdco (m n v0 g1) v0 dv g1 dc)
```

```
            (fdco (1+ (m n v0 g1)) v0 dv g1 dc)))
           (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
            (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
 v0))))) 1))
        (- 1
           (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
              (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
 v0)))))))
          (* 2 n)))
:hints (("Goal"
    :clause-processor
    (Smtlink clause
          '( (:expand ((:functions ((m integerp)
                      (gamma rationalp)
                      (mu rationalp)
                      (equ-c rationalp)
                      (fdco rationalp)
                      (dv0 rationalp)))
                  (:expansion-level 1)))
            (:python-file "delta-smaller-than-0-lemma4")
            (:let ((expt_gamma_2
               (expt (gamma) 2)
                rationalp)
              (expt_gamma_1
               (expt (gamma) 1)
                rationalp))
              )
            (:hypothesize ((equal expt_gamma_1 1/5)
                 (equal expt_gamma_2 1/25)
                 )
             ))
           state)
    :in-theory (disable delta-<-0-lemma3-lemma1
                 delta-<-0-lemma3-lemma3-stupidlemma
                 delta-<-0-lemma3-lemma2
                 delta-<-0-lemma3-lemma3
              delta-<-0-lemma3-lemma4-stupidlemma
```

```
                 delta-<-0-lemma3-lemma4))))
)


(defthm delta-<-0
  (implies (basic-params n 3 dc v0 dv g1)
      (< (delta n v0 dv g1 dc) 0))
  :hints (("Goal"
      :use ((:instance delta-rewrite-5)
       (:instance delta-<-0-lemma4)
       (:instance delta-<-0-lemma3)
       (:instance delta-<-0-lemma2)
       (:instance delta-<-0-lemma1))
      :in-theory (disable delta-<-0-lemma3-lemma1
                   delta-<-0-lemma3-lemma3-stupidlemma
                   delta-<-0-lemma3-lemma2
                   delta-<-0-lemma3-lemma3
                delta-<-0-lemma3-lemma4-stupidlemma
                delta-<-0-lemma3-lemma4)
     )))
) ;; delta < 0 thus is proved

;; prove phi(2n+1) = gamma^2*A+gamma*B+delta
(encapsulate ()

(local
(defthm split-phi-2n+1-lemma1-lemma1
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (equal (A (+ n 1) phi0 v0 dv g1 dc)
        (+ (* (expt (gamma) (+ (* 2 n) 1)) phi0)
           (* (expt (gamma) (* 2 n))
         (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
           (* (expt (gamma) (- (* 2 n) 1))
         (- (fdco (m n v0 g1) v0 dv g1 dc) 1))))))
)

(local
```

```
(defthm split-phi-2n+1-lemma1-lemma2
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (equal (+ (* (expt (gamma) (+ (* 2 n) 1)) phi0)
           (* (expt (gamma) (* 2 n))
         (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
           (* (expt (gamma) (- (* 2 n) 1))
         (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
        (+ (* (+ (* (expt (gamma) (- (* 2 n) 1)) phi0)
            (* (expt (gamma) (- (* 2 n) 2))
               (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
            (* (expt (gamma) (- (* 2 n) 3))
               (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))
         (expt (gamma) 2))
           (- (* (expt (gamma) (* 2 n))
            (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
         (* (expt (gamma) (* 2 n))
            (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
           (- (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
         (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1))))))
  )
)

(local
(defthm split-phi-2n+1-lemma1-A
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (equal (A (+ n 1) phi0 v0 dv g1 dc)
        (+ (* (A n phi0 v0 dv g1 dc) (gamma) (gamma))
           (- (* (expt (gamma) (* 2 n))
            (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
         (* (expt (gamma) (* 2 n))
            (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
           (- (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
         (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))))))
```

```
)

(local
(defthm split-phi-2n+1-lemma2-lemma1
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (B (+ n 1) v0 dv g1 dc)
        (* (expt (gamma) (- n 1))
           (B-sum 1 (- n 1) v0 dv g1 dc)))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma2
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (B (+ n 1) v0 dv g1 dc)
        (* (expt (gamma) (- n 1))
           (+ (B-term (- n 1) v0 dv g1 dc)
         (B-term (- (- n 1)) v0 dv g1 dc)
         (B-sum 1 (- n 2) v0 dv g1 dc))))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma3
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (B (+ n 1) v0 dv g1 dc)
        (+ (* (expt (gamma) (- n 1))
         (B-sum 1 (- n 2) v0 dv g1 dc))
           (* (expt (gamma) (- n 1))
         (B-term (- n 1) v0 dv g1 dc))
           (* (expt (gamma) (- n 1))
         (B-term (- (- n 1)) v0 dv g1 dc))))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma4
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (B (+ n 1) v0 dv g1 dc)
        (+ (* (gamma) (expt (gamma) (- n 2))
```

```
          (B-sum 1 (- n 2) v0 dv g1 dc))
            (* (expt (gamma) (- n 1))
          (+ (B-term (- n 1) v0 dv g1 dc)
             (B-term (- (- n 1)) v0 dv g1 dc)))))))
)

(local
(defthm split-phi-2n+1-lemma2-lemma5
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (B (+ n 1) v0 dv g1 dc)
        (+ (* (gamma) (B n v0 dv g1 dc))
           (* (expt (gamma) (- n 1))
         (+ (B-term (- n 1) v0 dv g1 dc)
            (B-term (- (- n 1)) v0 dv g1 dc)))))))
)

(local
(defthm split-phi-2n+1-lemma2-B
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (B (+ n 1) v0 dv g1 dc)
        (+ (* (gamma) (B n v0 dv g1 dc))
           (* (expt (gamma) (- n 1))
         (+ (* (expt (gamma) (- (- n 1)))
               (B-term-rest (- n 1) v0 dv g1 dc))
            (* (expt (gamma) (- n 1))
               (B-term-rest (- (- n 1)) v0 dv g1 dc))))))))
)

(local
(defthm split-phi-2n+1-lemma3-delta-stupidlemma
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (+ (- (* (expt (gamma) (* 2 n))
            (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
         (* (expt (gamma) (* 2 n))
            (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
           (- (* (expt (gamma) (- (* 2 n) 1))
            (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
```

```
          (* (expt (gamma) (- (* 2 n) 1))
             (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))
            (* (expt (gamma) (- n 1))
          (+ (* (expt (gamma) (- (- n 1)))
                (B-term-rest (- n 1) v0 dv g1 dc))
             (* (expt (gamma) (- n 1))
                (B-term-rest (- (- n 1)) v0 dv g1 dc)))))
         (+ (- (* (expt (gamma) (* 2 n))
             (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
          (* (expt (gamma) (* 2 n))
             (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
            (- (* (expt (gamma) (- (* 2 n) 1))
             (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
          (* (expt (gamma) (- (* 2 n) 1))
             (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))
            (* (expt (gamma) (1- n))
          (+ (* (expt (gamma) (1+ (- n)))
                (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (+ (1- n) dc)) (equ-c
    v0))))) 1))
             (* (expt (gamma) (1- n))
                (- (/ (* (mu) (1+ (* *alpha* (+ v0 dv))))
                 (1+ (* *beta* (+ (* g1 (+ (- 1 n) dc)) (equ-c
    v0))))) 1))))))))
)

(local
(defthm split-phi-2n+1-lemma3-delta
  (implies (basic-params n 3 dc v0 dv g1)
      (equal (+ (- (* (expt (gamma) (* 2 n))
             (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
          (* (expt (gamma) (* 2 n))
             (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
            (- (* (expt (gamma) (- (* 2 n) 1))
             (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
          (* (expt (gamma) (- (* 2 n) 1))
             (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))
```

```
           (* (expt (gamma) (- n 1))
         (+ (* (expt (gamma) (- (- n 1)))
               (B-term-rest (- n 1) v0 dv g1 dc))
            (* (expt (gamma) (- n 1))
               (B-term-rest (- (- n 1)) v0 dv g1 dc)))))
        (delta n v0 dv g1 dc)))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1-lemma3-delta-stupidlemma)
       (:instance delta)))))
)

(local
(defthm split-phi-2n+1-lemma4
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1 dc)
        (+ (A (+ n 1) phi0 v0 dv g1 dc)
           (B (+ n 1) v0 dv g1 dc)))))
)

(local
(defthm split-phi-2n+1-lemma5
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1 dc)
        (+ (+ (* (A n phi0 v0 dv g1 dc) (gamma) (gamma))
         (- (* (expt (gamma) (* 2 n))
               (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
            (* (expt (gamma) (* 2 n))
               (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
         (- (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
            (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1))))
           (+ (* (gamma) (B n v0 dv g1 dc))
         (* (expt (gamma) (- n 1))
            (+ (* (expt (gamma) (- (- n 1)))
             (B-term-rest (- n 1) v0 dv g1 dc))
               (* (expt (gamma) (- n 1))
```

```
              (B-term-rest (- (- n 1)) v0 dv g1 dc))))))))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1-lemma1-A)
       (:instance split-phi-2n+1-lemma2-B)))))
)

(local
(defthm split-phi-2n+1-lemma6
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1 dc)
        (+ (* (A n phi0 v0 dv g1 dc) (gamma) (gamma))
           (* (gamma) (B n v0 dv g1 dc))
           (+ (- (* (expt (gamma) (* 2 n))
               (- (fdco (1- (m n v0 g1)) v0 dv g1 dc) 1))
            (* (expt (gamma) (* 2 n))
               (- (fdco (m n v0 g1) v0 dv g1 dc) 1)))
         (- (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (m n v0 g1) v0 dv g1 dc) 1))
            (* (expt (gamma) (- (* 2 n) 1))
               (- (fdco (1+ (m n v0 g1)) v0 dv g1 dc) 1)))
         (* (expt (gamma) (- n 1))
            (+ (* (expt (gamma) (- (- n 1)))
             (B-term-rest (- n 1) v0 dv g1 dc))
               (* (expt (gamma) (- n 1))
             (B-term-rest (- (- n 1)) v0 dv g1 dc)))))))))
)

(defthm split-phi-2n+1
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (equal (phi-2n-1 (1+ n) phi0 v0 dv g1 dc)
        (+ (* (gamma) (gamma) (A n phi0 v0 dv g1 dc))
           (* (gamma) (B n v0 dv g1 dc)) (delta n v0 dv g1
    dc))))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1-lemma6)
       (:instance split-phi-2n+1-lemma3-delta)))))

```

```
)

;; prove gamma^2*A + gamma*B < 0
(encapsulate ()

(local
(defthm except-for-delta-<-0-lemma1
  (implies (and (and (rationalp c)
                     (rationalp a)
                     (rationalp b))
                (and (> c 0)
                     (< c 1)
                     (< (+ A B) 0)
                     (< B 0)))
                (< (+ (* c c A) (* c B)) 0))
  :hints (("Goal"
           :clause-processor
           (Smtlink clause
                    '( (:expand ((:function ())
                                 (:expansion-level 1)))
                      (:python-file
    "except-for-delta-smaller-than-0-lemma1")
                      (:let ())
                      (:hypothesize ()))
                    state)))
  :rule-classes :linear)
)

(defthm except-for-delta-<-0
  (implies (basic-params n 3 dc v0 dv g1 phi0 (< (phi-2n-1 n
    phi0 v0 dv g1 dc) 0))
           (< (+ (* (gamma) (gamma) (A n phi0 v0 dv g1 dc))
            (* (gamma) (B n v0 dv g1 dc)))
              0))
  :hints (("Goal"
           :do-not-induct t
           :use ((:instance except-for-delta-<-0-lemma1
```

```
             (c (gamma))
             (A (A n phi0 v0 dv g1 dc))
             (B (B n v0 dv g1 dc)))
       (:instance B-neg)))))
)

;; for induction step
(encapsulate ()

(defthm phi-2n+1-<-0-inductive
  (implies (basic-params n 3 dc v0 dv g1 phi0 (< (phi-2n-1 n phi0 v0 dv g1 dc) 0))
      (< (phi-2n-1 (1+ n) phi0 v0 dv g1 dc) 0))
  :hints (("Goal"
      :use ((:instance split-phi-2n+1)
       (:instance delta-<-0)
       (:instance except-for-delta-<-0)))))

(defthm phi-2n+1-<-0-inductive-corollary
  (implies (basic-params (- i 1) 3 dc v0 dv g1 phi0
          (< (phi-2n-1 (- i 1) phi0 v0 dv g1 dc) 0))
      (< (phi-2n-1 i phi0 v0 dv g1 dc) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-inductive
             (n (- i 1)))))))

(defthm phi-2n+1-<-0-inductive-corollary-2
  (implies (basic-params (- i 1) 3 dc v0 dv g1 phi0
          (< (phi-2n-1 (- i 1) phi0 v0 dv g1 dc) 0))
      (< (+ (A i phi0 v0 dv g1 dc)
       (* (B-expt i)
          (B-sum 1 (- i 2) v0 dv g1 dc))) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-inductive-corollary)))))

(defthm phi-2n+1-<-0-base
    (implies (basic-params-equal n 2 dc v0 dv g1 phi0)
```

```
        (< (phi-2n-1 (1+ n) phi0 v0 dv g1 dc) 0))
    :hints (("Goal''"
        :clause-processor
        (Smtlink clause
              '( (:expand ((:function ())
                      (:expansion-level 1)))
                (:python-file "phi-2n+1-smaller-than-0-base")
                (:let ())
                (:hypothesize ()))
              state)))
    )

(defthm phi-2n+1-<-0-base-new
      (implies (basic-params-equal (- i 2) 1 dc v0 dv g1 phi0)
        (< (phi-2n-1 (- i 1) phi0 v0 dv g1 dc) 0))
    :hints (("Goal''"
        :clause-processor
        (Smtlink clause
              '( (:expand ((:function ())
                      (:expansion-level 1)))
                (:python-file "phi-2n+1-smaller-than-0-base-new")
                (:let ())
                (:hypothesize ()))
              state)))
    )

(defthm phi-2n+1-<-0-base-corollary
    (implies (basic-params-equal (1- i) 2 dc v0 dv g1 phi0)
        (< (phi-2n-1 i phi0 v0 dv g1 dc) 0))
    :hints (("Goal"
        :use ((:instance phi-2n+1-<-0-base
               (n (- i 1))))))
    )

(defthm phi-2n+1-<-0-base-corollary-2
    (implies (basic-params-equal (1- i) 2 dc v0 dv g1 phi0)
        (< (+ (A i phi0 v0 dv g1 dc)
```

```
       (* (B-expt i)
          (B-sum 1 (- i 2) v0 dv g1 dc))) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-base-corollary))))
  )

(defthm stupid-proof
  (implies (and (equal a f)
      (equal a i)
      (implies (and m l) l)
      (implies l (and c h))
      (implies (and c h) (and c j))
           (implies (and a b c d) e)
      (implies (and f b c d) g)
      (implies (and f b h d e) g)
      i
      m
      (implies (and a b j d) e)
      f
      b
      l
      d)
      g)
  :rule-classes nil)

(defthm phi-2n+1-<-0-lemma-lemma1
  (implies
 (and
     (implies
          (and (and (integerp (+ -2 i))
                    (rationalp g1)
                    (rationalp v0)
                    (rationalp phi0)
                    (rationalp dv)
          (rationalp dc))
               (equal (+ -2 i) 1)
               (equal g1 1/3200)
```

```
      (>= dc 0)
      (< dc 1)
           (<= 9/10 v0)
           (<= v0 11/10)
           (<= -1/8000 dv)
           (<= dv 1/8000)
           (<= 0 phi0)
           (< phi0
              (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
                    (/ (+ 1
                          (fix (* (+ 1
                                     (* (+ (fix (* (+ 1
(fix v0)) 1)) -1)
                                        (/ g1))
                                     -640 dc)
                                  g1))))))))
      (< (phi-2n-1 (+ -1 i) phi0 v0 dv g1 dc) 0))
 (implies
      (and (and (integerp (+ -1 i))
                (rationalp g1)
                (rationalp v0)
                (rationalp phi0)
                (rationalp dv)
      (rationalp dc))
           (equal (+ -1 i) 2)
           (equal g1 1/3200)
      (>= dc 0)
      (< dc 1)
           (<= 9/10 v0)
           (<= v0 11/10)
           (<= -1/8000 dv)
           (<= dv 1/8000)
           (<= 0 phi0)
           (< phi0
              (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
```

```
                    (/ (+ 1
                          (fix (* (+ 1
                                     (* (+ (fix (* (+ 1
(fix v0)) 1)) -1)
                                        (/ g1))
                                     -640 dc)
                                  g1))))))))
      (< (+ (a i phi0 v0 dv g1 dc)
            (* (/ (expt 5 (+ -2 i)))
               (b-sum 1 (+ -2 i) v0 dv g1 dc)))
         0))
 (implies
      (and (and (integerp (+ -1 i))
                (rationalp g1)
                (rationalp v0)
                (rationalp dv)
                (rationalp phi0)
      (rationalp dc))
           (<= 3 (+ -1 i))
           (<= (+ -1 i) 640)
      (>= dc 0)
      (< dc 1)
           (equal g1 1/3200)
           (<= 9/10 v0)
           (<= v0 11/10)
           (<= -1/8000 dv)
           (<= dv 1/8000)
           (<= 0 phi0)
           (< phi0
              (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
                    (/ (+ 1
                          (fix (* (+ 1
                                     (* (+ (fix (* (+ 1
(fix v0)) 1)) -1)
                                        (/ g1))
                                     -640 dc)
```

```
                                  g1)))))))
           (< (phi-2n-1 (+ -1 i) phi0 v0 dv g1 dc) 0))
      (< (+ (a i phi0 v0 dv g1 dc)
            (* (/ (expt 5 (+ -2 i)))
               (b-sum 1 (+ -2 i) v0 dv g1 dc)))
         0))
 (not (or (not (integerp i)) (< i 1)))
 (implies
      (and (and (integerp (+ -1 -1 i))
                (rationalp g1)
                (rationalp v0)
                (rationalp dv)
                (rationalp phi0)
      (rationalp dc))
           (<= 2 (+ -1 -1 i))
           (<= (+ -1 -1 i) 640)
      (>= dc 0)
      (< dc 1)
           (equal g1 1/3200)
           (<= 9/10 v0)
           (<= v0 11/10)
           (<= -1/8000 dv)
           (<= dv 1/8000)
           (<= 0 phi0)
           (< phi0
              (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
                    (/ (+ 1
                          (fix (* (+ 1
                                     (* (+ (fix (* (+ 1
(fix v0)) 1)) -1)
                                        (/ g1))
                                     -640 dc)
                                  g1))))))))
      (< (+ (a (+ -1 i) phi0 v0 dv g1 dc)
            (* (/ (expt 5 (+ -2 -1 i)))
               (b-sum 1 (+ -2 -1 i) v0 dv g1 dc)))
```

```
                0))
        (integerp (+ -1 i))
        (rationalp g1)
        (rationalp v0)
        (rationalp dv)
        (rationalp phi0)
        (rationalp dc)
        (<= 2 (+ -1 i))
        (<= (+ -1 i) 640)
        (>= dc 0)
        (< dc 1)
        (equal g1 1/3200)
        (<= 9/10 v0)
        (<= v0 11/10)
        (<= -1/8000 dv)
        (<= dv 1/8000)
        (<= 0 phi0)
        (< phi0
           (+ -1
              (* (fix (+ 1 (fix (+ v0 dv))))
                 (/ (+ 1
                       (fix (* (+ 1
                                  (* (+ (fix (* (+ 1 (fix v0))
       1)) -1)
                                     (/ g1))
                                  -640 dc)
                               g1))))))))
    (< (+ (a i phi0 v0 dv g1 dc)
          (* (/ (expt 5 (+ -2 i)))
             (b-sum 1 (+ -2 i) v0 dv g1 dc)))
       0))
     :hints (("Goal"
         :use ((:instance stupid-proof
                 (a (integerp (+ -1 -1 i)))
                 (b (and (rationalp g1)
                    (rationalp v0)
                    (rationalp dv)
```

```
                (rationalp phi0)
                (rationalp dc)))
             (c (equal (+ -2 i) 1))
             (d (and (>= dc 0)
                (< dc 1)
                (equal g1 1/3200)
                (<= 9/10 v0)
                (<= v0 11/10)
                (<= -1/8000 dv)
                (<= dv 1/8000)
                (<= 0 phi0)
                (< phi0
                   (+ -1
                 (* (fix (+ 1 (fix (+ v0 dv))))
                    (/ (+ 1
                     (fix (* (+ 1
                           (* (+ (fix (* (+ 1 (fix v0)) 1)) -1)
                         (/ g1))
                           -640 dc)
                        g1)))))))))
             (e (< (+ (a (+ -1 i) phi0 v0 dv g1 dc)
                 (* (/ (expt 5 (+ -2 -1 i)))
               (b-sum 1 (+ -2 -1 i) v0 dv g1 dc)))
              0))
             (f (integerp (+ -1 i)))
             (g (< (+ (a i phi0 v0 dv g1 dc)
                 (* (/ (expt 5 (+ -2 i)))
               (b-sum 1 (+ -2 i) v0 dv g1 dc)))
              0))
             (h (and (<= 3 (+ -1 i))
                (<= (+ -1 i) 640)))
             (i (integerp i))
             (j (and (<= 2 (+ -1 -1 i))
                (<= (+ -1 -1 i) 640)))
             (l (and (<= 2 (+ -1 i))
                (<= (+ -1 i) 640)
                ))
```

```
              (m (>= i 1)))))))

(defthm phi-2n+1-<-0-lemma-lemma2
  (implies (and (or (not (integerp i)) (< i 1))
                (integerp (+ -1 i))
                (rationalp g1)
                (rationalp v0)
                (rationalp dv)
                (rationalp phi0)
           (rationalp dc)
                (<= 2 (+ -1 i))
                (<= (+ -1 i) 640)
           (>= dc 0)
           (< dc 1)
                (equal g1 1/3200)
                (<= 9/10 v0)
                (<= v0 11/10)
                (<= -1/8000 dv)
                (<= dv 1/8000)
                (<= 0 phi0)
                (< phi0
                   (+ -1
                      (* (fix (+ 1 (fix (+ v0 dv))))
                         (/ (+ 1
                               (fix (* (+ 1
                                          (* (+ (fix (* (+ 1
    (fix v0)) 1)) -1)
                                             (/ g1))
                                          -640 dc)
                                       g1))))))))
           (< (+ (a i phi0 v0 dv g1 dc)
                 (* (/ (expt 5 (+ -2 i)))
                    (b-sum 1 (+ -2 i) v0 dv g1 dc)))
              0))
  :rule-classes nil)

(defthm phi-2n+1-<-0-lemma
```

```
  (implies (basic-params (1- i) 2 dc v0 dv g1 phi0)
      (< (+ (A i phi0 v0 dv g1 dc)
       (* (B-expt i)
          (B-sum 1 (- i 2) v0 dv g1 dc))) 0))
  :hints (("Goal"
      :do-not '(simplify)
      :induct (B-sum 1 i v0 dv g1 dc))
     ("Subgoal *1/2"
     :use ((:instance phi-2n+1-<-0-base-new)
      (:instance phi-2n+1-<-0-base-corollary-2)
      (:instance phi-2n+1-<-0-inductive-corollary-2)
      ))
     ("Subgoal *1/2''"
      :use ((:instance phi-2n+1-<-0-lemma-lemma1)))
     ("Subgoal *1/1'"
      :use ((:instance phi-2n+1-<-0-lemma-lemma2)))
     )
  )

(defthm phi-2n+1-<-0
  (implies (basic-params (1- i) 2 dc v0 dv g1 phi0)
      (< (phi-2n-1 i phi0 v0 dv g1 dc) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0-lemma))
      ))
  )

(defthm phi-2n-1-<-0
  (implies (basic-params n 3 dc v0 dv g1 phi0)
      (< (phi-2n-1 n phi0 v0 dv g1 dc) 0))
  :hints (("Goal"
      :use ((:instance phi-2n+1-<-0
             (i n))))))
)
```