FORMALLY VERIFYING THE ACCURACY OF NUMERICAL APPROXIMATIONS IN SCIENTIFIC SOFTWARE

by

Timothy K. Zirkel

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Spring 2014

FORMALLY VERIFYING THE ACCURACY OF NUMERICAL APPROXIMATIONS IN SCIENTIFIC SOFTWARE

by

Timothy K. Zirkel

Approved: _____
Errol L. Lloyd, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____
James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Stephen F. Siegel, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

James A. Clause, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Louis F. Rossi, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

B. David Saunders, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to all those who have guided and encouraged me throughout my Ph.D. studies. Friends, family, and mentors too numerous to mention have provided invaluable advice and support over the last six years.

Special thanks go to my advisor, Dr. Stephen F. Siegel. From my first meeting with him as a prospective student in spring of 2008, he has been a great source of guidance and an excellent academic role model. Dr. Siegel's attention to detail and tireless work ethic have driven me time and again to produce better results. Working with him has been exciting, illuminating, and tremendously rewarding.

I would also like to thank my committee members, Dr. James A. Clause, Dr. Louis F. Rossi, and Dr. B. David Saunders, for their time, patience, flexible schedules, and feedback throughout this process.

TABLE OF CONTENTS

Appendix

LIST OF FIGURES

ABSTRACT

Numerical computation has broad application to a variety of fields. Typically a numerical method yields an approximation to an exact mathematical value, since programs cannot generally handle evaluation of continuous functions at all points. The common way of creating such a method is to discretize continuous functions by restricting them to a mesh. Performing calculations on the mesh provides an approximation to performing calculations on the original function. However, this introduces error. While not the only source of error (round-off error in floating-point operations can be a major consideration), the error in the method itself is in some sense more fundamental. In practice, programs utilizing these approximations often contain defects which introduce additional error.

The order of accuracy of a numerical method relates the scheme's error to the discretization parameters. Scientists must know the accuracy of any numerical approximation, and often prove that the method satisfies the claimed order of accuracy by hand. However, the actual code to implement a method might be more complex and veer from the abstract mathematics.

We show that the claimed order of accuracy of a numerical method implemented in a C program can be (largely) automatically verified using formal methods. The automation cannot be complete, because the problem is undecidable in general and because the programmer must provide some annotations relating the code to the underlying mathematics. These annotations can be kept to a minimum. We have extended the Concurrency Intermediate Verification Language (CIVL) model checker to verify the order of accuracy of a numerical computation. Our method requires annotating C code with information specifying the function and the order of accuracy of the approximation. CIVL parses the annotations with the C code to form a model

of the program. The model is symbolically executed, and techniques such as Taylor expansion are then used to relate the program data to the mathematical function. The verifier, with the assistance of a theorem prover, determines either that the assertions hold at all states, or else that they may not hold. If the assertions may not hold, CIVL provides diagnostic information.

Chapter 1

INTRODUCTION

There are a variety of research avenues dealing with the specification and verification of numerical programs.

One avenue is equivalence checking. That technique takes two programs and attempts to establish their functional (input-output) equivalence. Typically, one program is "trusted" and serves as the specification of an algorithm, while the other is a complex, optimized, possibly parallel, implementation. This is the main technique used by TASS [67]. Equivalence checking can be effective for both floating-point and real number notions of equivalence.

Another approach uses rich specification languages to formulate assertions and code contracts concerning the numerical computations in a program. This is the approach taken by Frama-C [3]. The formulas can specify precise relationships between inputs and outputs to functions, and can refer to both the floating-point and real semantics of numeric operations; they can be verified using deductive techniques which rely on automated theorem provers and/or proof assistants. This approach has been particularly effective at verifying precise bounds on round-off errors.

One important aspect of numerical programs that has received relatively little attention in these research efforts is the notion of *order of accuracy*. This concept is essential to the analysis of a broad range of numerical programs, especially partial differential equation solvers. The order of accuracy of an algorithm is an integer which measures how quickly the solution computed using a discrete approximation converges to the continuous mathematical solution as grid resolution increases. In particular, order of accuracy deals with "discretization error" and depends solely on the real (not

floating-point) semantics of the code. Since discretization error often dominates the error in numerical computations, it is seen as essential to get the order of accuracy "right" before focusing on floating-point error.

Many journals have strict requirements concerning the order of accuracy of methods presented in their submissions. The American Institute of Aeronautics and Astronautics requires that any article appearing in one of its journals that deals with the numerical solution to PDEs "should state the formal accuracy of the numerical method for interior points as well as the formal accuracy of the numerical boundary conditions," which should be "at least formally second-order accurate" and that "some level of verification testing" be performed on implementations [1]. The Journal of Fluids Engineering requires "[t]he numerical method used must be at least formally second-order accurate in space (based on a Taylor series expansion) for nodes in the interior of the computational grid" [42]. Authors are usually expected to carry out testing-based strategies which vary parameters in order to ascertain that the code meets the theoretical order of accuracy, but these are subject to well-known limitations of testing and cannot provide a proof.

1.1 Thesis Statement

Using symbolic execution and theorem proving techniques, it is possible to provide automatic formal verification of the order of accuracy of a numerical program.

We present a new technique, based on symbolic execution, for verifying the claimed order of accuracy of a numerical method. Our solution involves

1. a new differential accuracy specification language, described in Ch. 5, and
2. the technique of symbolic differential accuracy verification, introduced in Ch. 6, which provides a method for verifying or refuting assertions expressed in the differential accuracy specification language.

This technique takes as input an annotated C program implementing the method, but it treats all of the floating-point computations in the programs as full precision (mathematical) operations. The annotations specify the input-output signature of the

program, as well as accuracy claims, such as "the output u is 3^{rd}-order accurate in input x and 2^{nd}-order accurate in input t."

The remainder of this thesis is organized as follows:

- Ch. 2 gives some background and definitions of numerical accuracy and symbolic execution, which is the core technique used by the verification tool.
- Related work not covered in Ch. 2 is mentioned in Ch. 3.
- Ch. 4 describes the CIVL model checker, which has been extended to support symbolic differential accuracy verification.
- Ch. 5 presents the differential accuracy specification language.
- Ch. 6 describes the technique of symbolic differential accuracy verification.
- Ch. 7 discusses evaluation of symbolic differential accuracy verification.
- Ch. 8 makes some concluding remarks.

Chapter 2

BACKGROUND

2.1 Numerical Accuracy

Numerical methods involve taking a problem from a continuous domain and accurately approximating it using a discrete set of parameters. This discretization introduces error. Analyzing this error is an essential component of any investigation using a numerical scheme. In this chapter, we provide precise definitions for the numerical concepts we wish to treat formally in programs.

2.1.1 Asymptotic behavior and order of accuracy

We begin by discussing the asymptotic behavior of functions. First we will look at functions of one variable. The following are standard; see for example [44].

Definition 1 (Big-O)**.** *Let $a > 0$ and $I = (0, a)$. Suppose we have two functions $\phi : I \to \mathbb{R}$ and $\psi : I \to \mathbb{R}$. We write*

$$\phi(h) = O(\psi(h)) \text{ as } h \to 0$$

if there exist positive real numbers C and ϵ such that $|\phi(h)| \leq C|\psi(h)|$ whenever $0 < h < \epsilon$.

In the following definitions, assume $a > 0$, $I = (0, a)$, and $D \subseteq \mathbb{R}$.

Definition 2 (Order of Accuracy)**.** *Let n be a positive integer. Given a function $f : D \to \mathbb{R}$, consider a function $g : D \times I \to \mathbb{R}$. Fix $x \in D$. We say g is an n^{th}* order accurate approximation to f at x *if*

$$f(x) - g(x, h) = O(h^n) \text{ as } h \to 0.$$

The idea is that the left hand side is approaching 0 at least as fast as h^n. The order of accuracy quantifies the rate at which the numerical method will converge to the exact solution as h decreases.

Notice that the constants in Def. 2 are dependent on the particular point x. A stronger notion is of having a single ϵ and C for the entire domain. This is the concept of *uniformly n^{th} order accurate*.

Definition 3 (Uniform Order of Accuracy)**.** *Let n be a positive integer, $f : D \to \mathbb{R}$, and $g : D \times I \to \mathbb{R}$. Define $\phi : I \to \mathbb{R}$ by*

$$\phi(h) = \sup_{x \in D} |f(x) - g(x, h)|.$$

We say that g is a uniformly n^{th} order accurate approximation of f on D *if*

$$\phi(h) = O(h^n) \text{ as } h \to 0.$$

Clearly if g is uniformly n^{th} order accurate on f, then it is n^{th} order accurate at each point. However, the converse is not necessarily true. The stronger condition is usually the desired one.

2.1.2 Example: estimating $\sin'(x)$ with central differencing

Approximating the derivative of $\sin(x)$ is a simple example that illustrates the meaning of order of accuracy. The technique we use for this is central differencing, where the derivative at a point x is estimated as the slope of the line through the values of the function at $x - h$ and $x + h$. In this case $D = \mathbb{R}$ and

$$f(x) = \cos(x) \tag{2.1}$$

$$g(x, h) = \frac{\sin(x + h) - \sin(x - h)}{2h}. \tag{2.2}$$

Fig. 2.1 depicts results of approximating the derivative of $\sin(x)$ using central differencing. In Fig. 2.1(a), $f(x) = \cos(x)$ and $g(x, h)$ are given for various values of h. Fig. 2.1(b) gives the error

$$\phi(x, h) = |f(x) - g(x, h)| \tag{2.3}$$

for the same three values of h.

Per Definition 3 in order to see that g is a uniformly second order accurate approximation of f on $\mathbb{R}$, we need to show that there exist $C > 0$ and $\epsilon > 0$ such that $\forall x \in \mathbb{R}$

$$\phi(x,h) = \left|\cos(x) - \frac{\sin(x+h) - \sin(x-h)}{2h}\right| \leq Ch^2 \tag{2.4}$$

whenever $0 < h < \epsilon$. We claim we can take $C = 0.2$ and $\epsilon = 1$. In Appendix A we show that

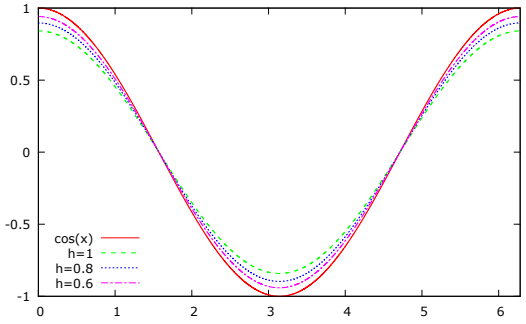
$$0.15h^2 \leq \phi(x,h) \leq 0.2h^2. \tag{2.5}$$

This proves that these values satisfy the condition required by Def. 3, and the lower bound demonstrates that g is not a 3^{rd} or higher order accurate approximation of f.

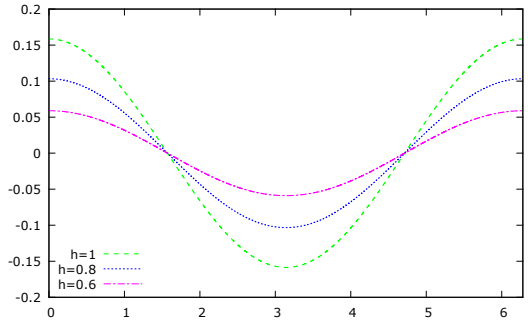
Fig. 2.1(c) shows that 0.2 is a constant demonstrating that the approximation is $O(h^2)$. Fig. 2.1(c) also shows that the approximation is not higher order, since there is a lower bound on the error that is a constant times h^2.

2.1.3 Grid approximations

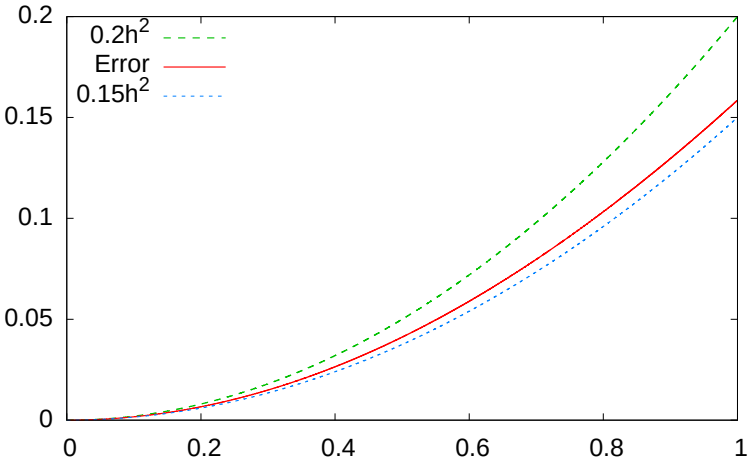
Definition 3 is often used in the analysis of finite difference methods where the solution is approximated on a grid. A grid is a discrete subset of the domain where an approximate solution is computed. For example, a domain of interest might be an interval $D = [b_0, b_1]$. One possible discretization of this domain is choosing m uniformly spaced points to form $\Delta(h) = \{b_0 + hk | 0 \leq k \leq m\}$ where $h = b_1/m$. Note that the grid resulting from a smaller h is not necessarily a refinement of that resulting from a larger h (e.g., the grid from $h = 1/4$ does not refine the grid from $h = 1/3$). For a convergent scheme, the approximation converges to the exact solution on this discrete set, as we refine the grid. While the previous definitions are standard, Definition 4 applies these notions to the computation actually performed by a program.



(a) $\cos(x)$ and approximations.



(b) Graph of $f(x) - g(x, h)$.



(c) Error ϕ at $x = \pi$ vs h.

Figure 2.1: Approximations of $\sin'(x) = \cos(x)$ and resulting error ϕ. Graph (a) shows $\cos(x)$ and the approximations obtained using central differencing. Graph (b) gives the values of ϕ for $x \in [0, 2\pi]$ and $h = 1, 0.8, 0.6$. This error is periodic with period 2π. Graph (c) shows that ϕ at $x = \pi$ is $O(h^2)$. It also gives a lower bound on the error that is a constant times h^2. This shows that the error is not, for example, $O(h^3)$ or some higher order.

Definition 4 (n^{th} order Δ convergence)**.** *Let n be a positive integer, $D \subseteq \mathbb{R}$, $f\colon D \to \mathbb{R}$, $a > 0$, and $I = (0, a)$. Suppose $\Delta\colon I \to \wp(D)$, where $\wp(D)$ is the set of all subsets of D. Let $S = \bigcup_{h \in I} (\Delta(h) \times \{h\}) \subseteq D \times I$. Suppose $g\colon S \to R$. Define $\phi\colon I \to \mathbb{R}$ by*

$$\phi(h) = \sup_{x \in \Delta(h)} |f(x) - g(x, h)|.$$

We say g is a Δ-uniformly n^{th} order accurate approximation of f *if*

$$\phi(h) = O(h^n) \text{ as } h \to 0.$$

Given the parameter h, Δ returns a subset of D that is the grid. For problems of interest in our analysis, the grid will typically consist of evenly spaced points. We are interested in a numerical method's error as h goes to 0.

2.1.4 Functions of several variables

Functions of multiple variables may have different orders of accuracy in each variable. This may be represented by a separate big-O term for each variable.

Definition 5. *Let $I_0 = (0, a), I_1 = (0, b)$, with $a, b > 0$. Suppose we have functions $\phi : I_0 \times I_1 \to \mathbb{R}$, $\psi_0 : I_0 \to \mathbb{R}$, and $\psi_1 : I_1 \to \mathbb{R}$. We write*

$$\phi(h_0, h_1) = O(\psi_0(h_0)) + O(\psi_1(h_1)) \text{ as } h_0 \to 0 \text{ and } h_1 \to 0$$

if there exist positive real numbers $C_0, C_1, \epsilon_0, \epsilon_1$ such that

$$|\phi(h_0, h_1)| \leq C_0 |\psi_0(h_0)| + C_1 |\psi_1(h_1)|$$

whenever $0 < h_0 < \epsilon_0$ and $0 < h_1 < \epsilon_1$.

Definitions 2, 3, and 4 generalize to several variables in the obvious way. For these definitions, we either give the accuracy for each variable separately or say the approximation is accurate of order $(n_0, n_1, \ldots, n_m)$.

2.2 Symbolic Execution

The main technique used by our verifier is *symbolic execution* [19,45]. Symbolic execution is an abstraction of a program in which symbolic expressions are used in place of concrete values. Our approach borrows ideas from *abstract interpretation* [21, 22], and is adapted from [67].

For symbolic execution, a program is first translated into a program graph, which is an intermediate representation that captures the concrete semantics of the program. Next, the program graph is extended to a symbolic transition system that represents the symbolic semantics of the program. This chapter defines and provides short examples of a program graph and a symbolic transition system, then discusses the symbolic execution literature.

2.2.1 Program graphs and the concrete transition system

We describe a *program graph*. Our definition is based on [4, Def. 2.13].

Suppose V is a set of *program variables*. A program graph over V is a tuple consisting of:

1. A set of *locations*
2. A set of *actions*
3. An *effect function* which takes an action and a boolean-valued expression of the variables in V and produces a new evaluation of the variables
4. A *conditional transition relation* which describes the transition from one location to another given a boolean-valued expression over V and an action
5. A set of *initial locations*
6. A boolean-valued *initial condition*.

A *state* $s = \langle l, \eta \rangle$ of a program graph comprises a location l and an evaluation of variables η. The state is *initial* if l is in the set of initial locations and the initial condition evaluates to true under the variable values η. The *next-state function* takes a state s and a transition t with start location l. If the boolean-valued expression in

```
   input int x,y;
   int z;
1  if (x > 0) {
2    if (x > y)
3      z = x-y;
     else
4      z = x+y;
   } else {
5    z = -x;
   }
6  assert z >= 0;
```

Figure 2.2: Pseudocode for a simple program. The numbers on the left indicate locations in the program.

t evaluates to true, the next-state function returns the state resulting from executing the transition's action. Otherwise, the next-state function returns the empty set.

Fig. 2.2 gives pseudocode for a simple program. The program has two input variables, x and y, and one regular variable z. The set of locations is $\{1, 2, 3, 4, 5, 6\}$ and the set of initial locations is $\{1\}$. The actions are no-op, assignment, and checking the assertion. A program graph for this example is given in Fig. 2.3. The edges are labeled with a shorthand for the conditional transition relation. For edges (1,2), (2,3), (2,4), and (1,5), the label gives the boolean-valued guard and the action is a no-op. For all other edges, the guard is *true* and the action is given in the label. The assertion at location 6 is labeled in red.

This notion of a program graph extends naturally to a parallel program by having the location be a tuple whose components are the location in each thread or process.

2.2.2 The symbolic transition system

The symbolic transition system is an interpretation of the program graph suitable for symbolic execution. It adds to the usual concrete values a set of *symbolic*

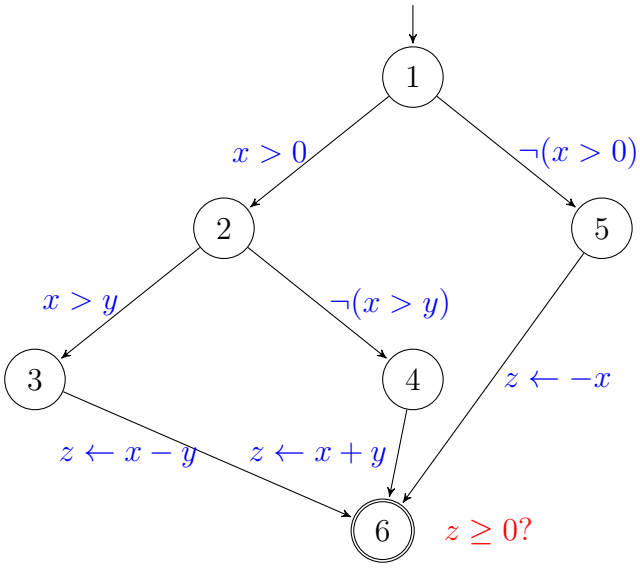


Figure 2.3: A program graph for the program in Fig. 2.2.

constants. An initial value function assigns a unique symbolic constant to each variable used as an "input" to the program in the initial state.

In the symbolic transition system, every expression is evaluated symbolically. The evaluator works in a way that is consistent with the concrete evaluation. That is, the same concrete value will be obtained by first symbolically evaluating the program expression and then replacing each symbolic constant with a concrete value as would be obtained by first replacing each symbolic constant with a concrete value and then evaluating the concrete semantics of the program function.

A *symbolic state* is composed of a boolean-valued expression called the *path condition*, a location, and a function mapping variables to the symbolic expressions for their values. The path condition keeps track of branches and other assumptions made during a particular path of symbolic execution of the program. The *symbolic next-state function* is the natural analogue of the concrete next-state function in the symbolic state space.

Fig. 2.4 shows all of the symbolic states encountered in the symbolic execution of the program graph in Fig. 2.3. In the initial state, the path condition is *true*,

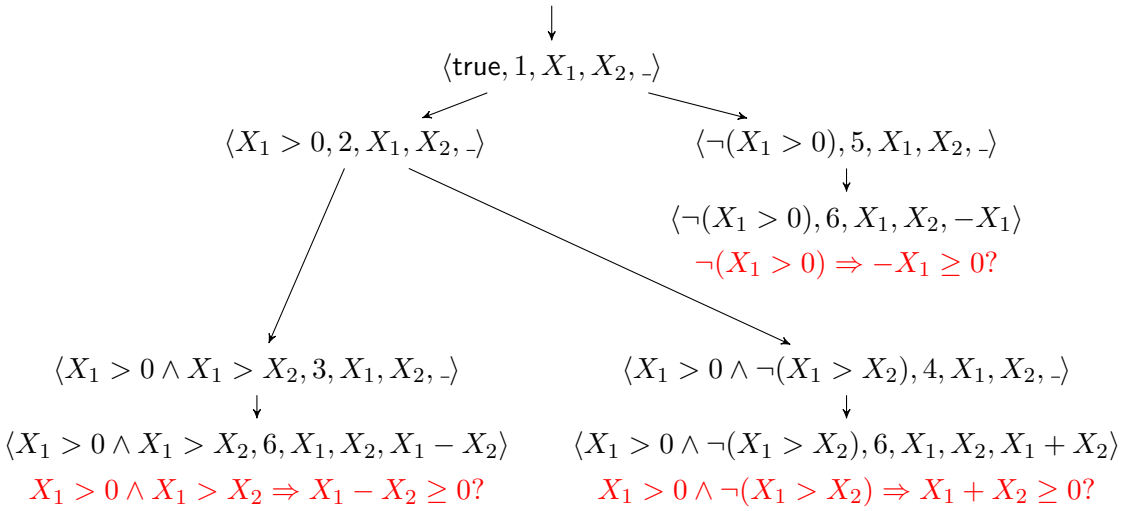


Figure 2.4: Symbolic execution over the program graph in Fig. 2.3. The components of the symbolic state are the path condition, the location, the value of x, the value of y and the value of z. _ represents an undefined value. The assertions checked at location 6 are given in red.

the program is at location 1, and the initial value function has assigned the symbolic constant X_1 to x and the symbolic constant X_2 to y. Initially, z has no assigned value. It will obtain a value during later transitions in the program. For example, consider the right branch from the initial state. This is the false branch of the outer `if` statement. When taking this branch, the path condition gets set to $\neg(X_1 > 0)$ and the program moves to location 5. The value of z is then set by the symbolic effect function during the transition between location 5 and location 6. The lines given in red in the figure indicate queries that must be verified by the theorem prover. These queries are all from the assertion at location 6, and have the form $\phi \Rightarrow z \geq 0$, where ϕ is the value of the path condition and z is the symbolic expression for the value of z.

2.2.3 Concretization

There is a *concretization map* from the symbolic state space to the concrete one. The concretization map takes a symbolic state and returns the set of all concrete states which are represented by that symbolic state. We say that a symbolic state s is

vacuous if the concretization map applied to s returns $\emptyset$. A symbolic state is vacuous if and only if the path condition for that state is unsatisfiable. Two symbolic states s and s' are equivalent if their concretizations are identical. Since a symbolic state can, in general, represent infinitely many concrete states, most tools return a single representative concrete state when concretizing a symbolic state.

Consider the symbolic state $\langle X_1 > 0 \wedge X_1 > X_2, 6, X_1, X_2, X_1 - X_2 \rangle$ from Fig. 2.4. The path condition $(X_1 > 0 \wedge X_1 > X_2)$ is satisfiable, so the state is not vacuous. It represents a number of concrete states where the program is at location 6. The following list is just a few of the possible concrete states.

- $x = 1$, $y = 0$, $z = 1$
- $x = 5$, $y = 3$, $z = 2$
- $x = 2$, $y = -10$, $z = 12$

However, any concrete state where $x < 0$ or where $x < y$ is not represented by this symbolic state.

2.2.4 A brief history of symbolic execution

Symbolic execution was originally proposed in the context of program testing [13, 19, 45]. Since then it has seen many extensions and generalizations, such as its combination with model checking [27, 43].

The ability of symbolic execution to reason about all possible inputs to a program gives it the capability to detect defects that could be missed even with extensive testing on concrete inputs. Unfortunately, this capability comes with a cost. In order to completely cover the execution space, every path through the program must be symbolically executed. The number of paths is exponential in the number of branches, and in general may be infinite, which hampers efforts to exhaustively explore all paths with symbolic execution. In practice, many symbolic execution tools sacrifice completeness in order to obtain a reasonable runtime for their analysis by using techniques such as random testing guided by control-flow graph analysis [14] or dynamic analysis

of program behavior [33]. These incomplete techniques are often effective at finding bugs missed by standard testing, but of course cannot guarantee the code to be free of defects.

Another approach to reduce the path explosion problem is to use *compositional symbolic execution* [63, 72]. In this method, each path through a procedure is symbolically executed only once. The results of symbolically executing individual procedures are then combined along feasible paths through the program.

Several symbolic execution efforts in recent years have involved concolic testing [49, 62] or execution-generated testing [15, 16, 24]. In concolic testing, programs are executed on a random concrete input, but the symbolic path condition is accumulated over the execution path. Once the execution has completed, some component of the path condition is negated, and a new set of concrete inputs is generated satisfying the modified path condition. This process continues until some path coverage criterion is met.

In execution-generated testing, the concrete and symbolic state components are maintained separately. When executing the code, any statement involving only concrete state components is executed concretely. The symbolic reasoning only comes into play when a statement involves a symbolic state component.

In [25], it is observed that for certain safety properties, most paths through a program are irrelevant. Sound path pruning algorithms can eliminate the majority of paths through the program. The pruning can provide an exponential speedup for the verification of those safety properties.

Probabilistic symbolic execution combines symbolic execution with model counting to determine the probability of a path being taken [32]. A path's probability can provide guidance for where to focus verification and test generation efforts.

Comparative symbolic execution, the method combining model checking with symbolic execution to verify the functional equivalence of two programs, was introduced in [64, 65]. This technique is used in [67] and is also supported in CIVL.

Chapter 3

OTHER RELATED WORK

This chapter discusses some related work not covered in the previous background chapters. The work falls into three categories: analysis of error in numerical software, general model checking tools, and tools aimed at verifying properties of numerical programs.

3.1 Error in numerical software

Numerical analysis is a broad field with many applications. In practice, the field must address both the real mathematical error and stability of an algorithm and also the error resulting from the use of floating point arithmetic [8, 34, 39].

Numerical approximation schemes are a crucial tool to provide solutions to hard mathematical problems, and require careful analysis of their accuracy. Several methods exist for analyzing accuracy and stability properties for various classes of problems. These include the modified equation approach [75], backward error analysis [36, 55], and grid convergence error analysis [59]. Often, an approximation scheme must handle the boundary of a domain differently from the interior points. This requires additional analysis of accuracy and stability [69]. For a parallel algorithm implemented using domain decomposition, extra care must be taken to account for all boundaries of the subdomains [56].

In addition to error resulting from approximation schemes, scientific program results contain error introduced by the use of floating point operations. Early work in this area by Wilkinson describes the error for some fundamental floating point operations [76]. Scientists must not only be aware of floating point error, but often

must adjust algorithms to mitigate it. Even simple operations such as summation can be approached in different ways, with different resulting accuracy [38].

Combined, the mathematical and floating-point errors present significant challenges for scientists designing complex simulations [53]. Thus, a variety of approaches for the validation and verification of numerical software have been employed. These include the use of probabilistic information [35,73], error estimators [78], testing protocols [68], method of nearby problems [61], and a range of other techniques [50,54,60].

One of the issues arising from rounding error is a possible effect on the convergence of values. Work has been done on proving the correctness of algorithms in spite of this problem. One approach is to specify requirements and use a computer algebra system [29,30].

3.2 General model checking tools

Spin [40] is one of the most widely-used model checking tools, and introduced a large array of techniques to reduce the time and memory consumed by explicit state model checking. For example, Spin's "collapse" compression algorithm allows global states to share common process states to reduce the memory footprint.

Bandera [20] and the related tool Bogor [57] innovated many methods in software model checking and were among the first tools to apply these techniques to Java programs. Bogor's "collapse" compression extended Spin's technique by allowing states to share sub-structures at various levels of the state hierarchy [58]. In Bogor, states are encoded (typically as bit vectors) before being saved and/or checked for being seen, to conserve memory. Also, instead of storing states on the DFS stack, Bogor stores transitions, and a method to invert a transition to obtain the previous state is used when popping the stack. (A similar technique is used by Spin.) Essentially, this allows Bogor to maintain only one uncompressed, mutable state during the search. In contrast, CIVL avoids the computational expense associated with compressing states and inverting transitions by maximizing opportunities for sharing among stored states—at every node in the state hierarchy, flyweighting is used to obtain a unique representative

of the equivalence class for that node—and limiting the number of states saved and pushed onto the stack.

3.3 Numerical program verification tools

Many tools exist for verifying properties of numerical programs. We mention some relevant ones here.

ASTRÉE is an abstract interpretation-based static analyzer for a subset of C [23]. It can reason precisely about floating-point and limited-precision integer arithmetic and verify absence of many runtime errors, but does not deal with dynamic memory allocation or recursion; its main applications have been to real-time embedded software. Improvements to ASTRÉE have used linearization and symbolic constant propagation to improve the precision of the numerical static analyses, resulting in tighter bounds [51]. FLUCTUAT [26] is another AI-based static analyzer providing information about rounding errors in C programs.

KLEE [24] is a symbolic execution tool for generating tests to improve test coverage; it can also check functional equivalence in some cases. It differs from CIVL in several ways. For example, it does not deal with parallel programs, and it uses "bit-precise" reasoning instead of mathematical real arithmetic. The GKLEE [48] extension to KLEE supports the analysis of C++ GPU programs.

Another tool using symbolic execution for test-case generation [43] is built on top of the Java PathFinder model checker [74] and can handle dynamically allocated structures and thread level parallelism. Java PathFinder also has a symbolic execution extension called JPF–SE that generates tests and proves light-weight properties of Java programs based on annotations of method specifications and loop invariants [2].

TVOC [6] is a tool for checking the correctness of compiler optimizations for sequential programs; it takes a functional equivalence verification approach based on a set of pre-defined transformation patterns. The Why/Krakatoa/Caduceus [11, 31] and Frama-C [3] frameworks provide a set of tools for checking Java and C programs. These

use special comments or JML-style [47] annotations (specifications of pre- and post-conditions for Java source code) to specify numerical accuracy requirements. None of these tools applies to message-passing based parallel programs or the problem of functional equivalence. Frama-C has also been used for verifying numerical C code to function identically across multiple architectures and compilers [12]. ISP [71], on the other hand, is geared specifically for MPI programs. It uses a modified runtime system to explore all relevant interleavings, but like ordinary testing only operates on concrete inputs, so cannot establish functional equivalence. MARMOT [46] is a tool to check for race conditions, deadlocks, and other issues in MPI programs, but is not targeted at general numerical properties.

F-Soft [41] is a model checking tool for C programs. It primarily checks for various runtime safety properties, but can also check for satisfaction of user written annotations. It does not currently support parallel codes. CBMC [18] is a bounded model checker for C and C++ programs. It checks for runtime safety properties and user specified assertions. BLAST [9,37] is a model checker for C programs that verifies temporal safety properties and automatically generates test suites.

Model checking techniques have been used successfully in the verification of safety properties and functional equivalence for mature scientific codes [66]. This project connects the code to the original numerical scheme.

Chapter 4

CIVL MODEL CHECKER

The original intention was to implement order of accuracy verification as an extension to the Toolkit for Accurate Scientific Software (TASS) [67]. However, the TASS specification language and C+MPI specific model proved limiting for this and other projects. Other popular symbolic execution tools (e.g. [2, 24]) lack, among other features, the ability to reason about floating point values under the semantics of the real numbers. As a result, we have developed a more general framework for software modeling called the Concurrency Intermediate Verification Language (CIVL). CIVL is targeted at a wide range of verification challenges, and as such has a number of features (like support for multiple processes) that are outside of the scope of this thesis. The CIVL framework consists of:

- A programming language called CIVL-C, which is an extension of C that adds a variety of useful features for verification, along with the ability to declare procedures in any scope.
- A model checker which uses symbolic execution to verify a number of safety properties of CIVL-C programs. It can also be used to verify that two CIVL-C programs are functionally equivalent.
- A number of translators from commonly used languages and APIs to CIVL-C. This part is still in progress.

We next discuss the CIVL-C input language, then give a description of the CIVL model and the semantics of that model, and conclude this chapter with a discussion of the features of the CIVL model checking tool.

4.1 CIVL-C

The input language to CIVL is called CIVL-C. CIVL-C is an extension to C which includes a number of features useful for describing programs in a variety of languages and for annotating programs with verification information.

Keywords specific to CIVL-C begin with a `$` to avoid namespace collisions with programs converted from other languages.

4.1.1 CIVL-C types

CIVL-C includes the standard C types, plus a few additional primitives. Sometimes multiple C types map to a single CIVL type when the code is translated to a model.

- *Integer types*: All of C's integer types (`int`, `long`, `unsigned short`, etc.) are supported in CIVL-C. They map to a single integer type in the CIVL model representation. The CIVL integer type represents the mathematical integers. As such, arithmetic in CIVL models is not checked for overflow or underflow by default. Assertions could be added to ensure that given quantities don't exceed a particular value. However, due to the symbolic nature of inputs, default checking of overflows and underflows would cause numerous spurious error reports whenever unconstrained symbolic values are used in arithmetic.

- *Real types*: Similarly, all of the floating point types in C, as well as a new type `$real`, map to a single real number type in the CIVL model which represents the mathematical real numbers. When providing program specifications, it is often desirable to write assertions equating various quantities. Under the semantics of floating-point arithmetic, two quantities computed in different ways (even with just different associativity) will almost never be equal. By treating all floating-point values as mathematical reals, developers can more easily specify the intended result of computations without worrying about bit-level details of floating-point representations.

- *Boolean type*: The type `_Bool` is supported. A variable of type `_Bool` can have values 1 and 0, which are also denoted by `$true` and `$false`, respectively.

- *Character type*: The `char` type is the same as C.

- *Scope type*: CIVL-C has a type `$scope` which is the type of a reference to a dynamic scope. It may be thought of as a scope ID, but it is neither necessary nor permissible to cast this type to an integer.

- *Proc type*: CIVL-C also has a type `$proc` which is the type of a reference to a process. Like the `$scope` type, it may not be cast to an integer.
- *Bundle type*: The CIVL-C `$bundle` type is used to hold an arbitrary contiguous chunk of data.
- *Pointer type*: Pointer types in CIVL-C are the same as in C.
- *Array type*: Array types may be derived from any element type, and as in C any array index operations are converted to pointer operations.
- *Struct or union types*: Struct and union types are as in C.

4.1.2 CIVL-C expressions

CIVL-C supports the standard C expressions (including pointer (de)referencing and arithmetic) with the exception of bit-wise operations. In addition, there are a number of new expressions that do not occur in C.

- *Self*: The expression `$self` is an expression of type `$proc` evaluating to a reference to the currently executed process.
- *Here*: The expression `$here` is an expression of type `$scope` evaluating to a reference to the local most dynamic scope.
- *Scope of*: The expression `$scopeof(expr)` evaluates to the dynamic scope containing the object specified by `expr`.
- *Spawn*: The expression `$spawn f(expr1, ..., exprn)` is an expression with side effects. It creates a new process executing `f` with the given arguments, and returns an object of type `$proc` that is a reference to the new process.
- *Wait*: The system function `void $wait($proc p)` will block until the process referenced by the expression `p` returns.
- *Exit*: The function `void $exit(void)` causes the calling process to immediately terminate.
- *Quantified expressions*: CIVL-C supports universal and existential quantifiers. The syntax for a universally quantified expression is

```
$forall { type identifier | restriction} expr
```

where `type` is a type name, `identifier` is the name of the variable bound by the quantified expression, `restriction` is a boolean expression expressing some restriction on the values that the bound variable may take, and `expr` is a formula. The quantified expression evaluates to true if and only if the formula is true whenever the bound variable is within the range specified by the restriction.

In the case where the bound variable has integer type and ranges over a finite interval of integers, it may be written as

```
$forall { identifier=lower .. upper } expr
```

where `lower` and `upper` are integer-valued expressions.

Existentially quantifiers use the same syntax but with `$exists` instead of `$forall`.

For the purposes of accuracy verification, we have added a third quantifier called `$uniform`. Its syntax and semantics are described in Ch. 5.

4.1.3 CIVL-C statements

CIVL-C supports all standard C statements. Among the additional statements are assumptions and assertions.

The syntax of an *assume statement* is

```
$assume expr;
```

where `expr` is a boolean-valued expression. When an assume statement is encountered during verification, the expression is assumed to hold. If the assumption later causes a contradiction on some execution, that execution is simply ignored. A violation is not reported, the contradiction just restricts the set of possible executions.

Assume statements can be used wherever a statement is expected, but can also be used in the file scope of a program to place restrictions on global variables. When used in this way, the assumption applies for the remainder of the program, including for any declarations or initializations that appear after the assumption in the file scope.

The *assert statement* is structured as a system function, and has the form

```
void $assert (_Bool expr);
```

When an assert statement is encountered, the verifier checks whether it can prove that the expression must hold in the current context. If that proof obligation cannot be discharged, a violation is reported.

The CIVL-C assert statement may take additional arguments to print a message if the assertion is violated. The additional arguments take a form similar to C's `printf` statement: a format string, followed by some number of arguments which are evaluated and substituted for successive codes in the format string. For example,

```
$assert(x<=B, "x-coordinate %f exceeds bound %f", x, B);
```

Assumptions and assertions are critical tools for specifying the accuracy of numerical programs.

4.1.4 Input/output specifications

CIVL-C has the additional type qualifiers `$input` and `$output`. These specifiers can be used for any variables in the outermost scope of the program and indicate to the verifier that the variable should be treated as an input or output to the program. Input variables may be read from but never written to. Output variables may be written to but never read.

The description here only provides a subset of the elements of CIVL-C. For further information, see [17].

4.2 CIVL model

The CIVL model is a "guarded command" style representation [28] that provides simple primitives for dynamic process creation, function calls, nondeterminism, and message-passing. It also adds to the usual model a notion of scopes, which have both a static and a dynamic aspect, and is based in part on our previous model for Chapel verification [79].

A *CIVL model* consists of the following components. First, there is a set Σ of (static) *scopes*, which has the structure of a rooted tree with root σ_0. These correspond to the lexical scopes in the source code, plus scopes that may be added to translate complex statements. The root scope represents the outermost scope encompassing the entire program. If τ is a child of σ, the lexical scope represented by τ is immediately contained in that represented by σ.

The model associates to each $\sigma \in \Sigma$ a set of typed variables and a set of function symbols. We say these variables and function symbols are *declared in* σ. All of these sets are pairwise disjoint; in particular, the variables declared in σ do not include those declared in any child of σ. We say a variable or function symbol is *visible in* σ if it is declared in σ or an ancestor of σ.

Types include *boolean*, *real*, *int*, *char*, arrays of any element type, a type *scope* for scope IDs, and a type *process* for process IDs. In this thesis, the *real* and *int* types represent the mathematical real numbers and integers, though no fundamental changes are required in the model to incorporate finite-precision or other types.

For each function symbol f declared in the model, there is a *function scope*, which is a child of the scope in which f is declared. A scope can be the function scope of at most one function. The *system function* has σ_0 as its function scope, and is the only function that does not have a declaration scope.

Every scope σ "belongs to" a unique function: if σ is the function scope for some f then σ belongs to f, else σ belongs to the function to which the parent of σ belongs.

The model associates to each function symbol f a *function signature*, which consists of a return type (possibly "void") and a sequence of parameter types. Finally, there is a program graph associated to f. The program graph includes a set of locations, including a start location. Each location l has an associated scope $\mathsf{lscope}(l)$ which must belong to f; there is no other restriction on $\mathsf{lscope}(l)$. The location also has some number (possibly 0) of outgoing transitions. Each transition comprises (1) a *guard*, a boolean-valued expression specifying when the transition is enabled, (2) a destination location, and (3) an *atomic CIVL statement*.

4.3 CIVL composite model

CIVL can compare two CIVL-C programs for functional (i.e. input-output) equivalence. In order to accomplish this, a *composite model* must be created. CIVL

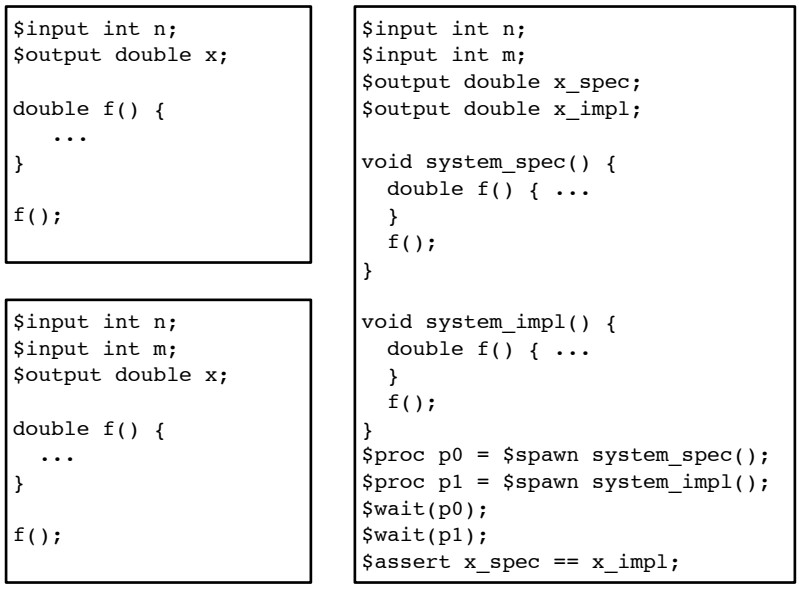


Figure 4.1: Conversion of two CIVL models to a composite model for comparison. top left: a specification program; bottom left: an implementation using a superset of the inputs of the specification; right: a composite model constructed form the specification and implementation.

can then run the verification algorithm on the composite model. Currently, the composite model must be constructed manually. However, the capability to automatically construct the composite model is under development. This section describes the algorithm that will be used in the automatic composite model construction, which will be available in an upcoming CIVL release. The manual construction follows much the same pattern. We refer to the first program provided for comparison as the *specification program*, and the second as the *implementation program*.

The first step in creating a composite model is building models for the specification and implementation programs. Next, the system functions for the specification and implementation models are renamed. CIVL will then create a new system function to wrap the renamed functions. The body of this new system function will spawn both the specification and implementation system functions, then wait for each to finish. What remains is to handle program inputs and outputs.

As discussed in Sec. 4.1, variables in the outermost scope of a CIVL-C program

may be specified as `$input` or `$output` variables. CIVL requires that

1. corresponding input and output variables have the same name
2. the input variables to the specification are a subset of the input variables to the implementation
3. the sets of output variables for the specification and implementation are the same

In constructing the composite model, CIVL will move the input variables from the specification and implementation models to the new system function. Whenever the same input variables exist in the specification and implementation, they are unified into a single variable in the new model.

CIVL will rename each output variable in the specification and implementation and move them to the new outermost scope. After the wait statements for the processes executing the specification and implementation system functions, the composite model system function has a series of assertions checking the equivalence of corresponding output variables.

Fig. 4.1(right) gives an example of a composite model formed from the two models on the left. The specification has an input variable n and an output x. The implementation has two input variables, n and m, and an output x. These variables meet the three requirements on input and output variables. The composite model shows the renamed system functions and output variables, and the unified set of input variables.

4.4 CIVL model semantics

A *state* of a CIVL model is a tuple composed of

1. a set of *dynamic scopes*
2. the *root dynamic scope*
3. a mapping from each dynamic scope to its parent in the dynamic scope tree
4. a mapping, called *static* from each dynamic scope to the corresponding static scope
5. a function assigning a valuation to each variable in each dynamic scope

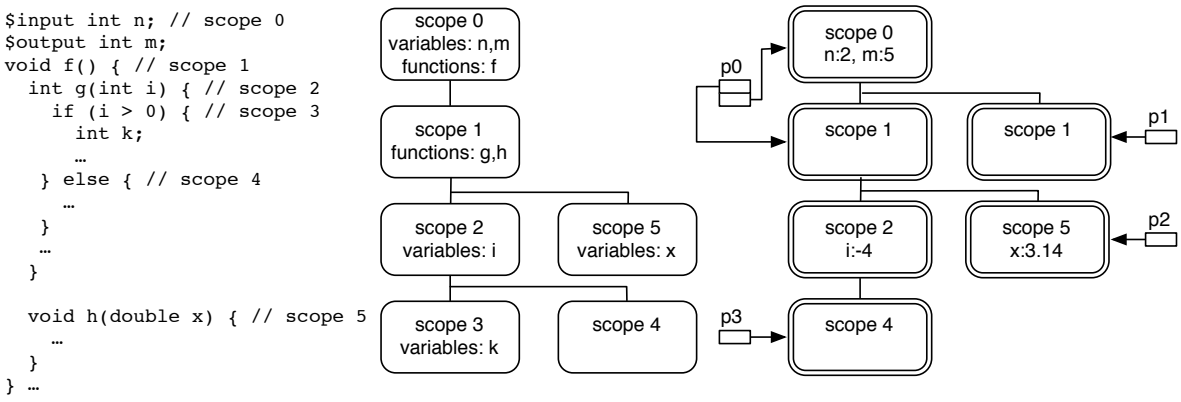


Figure 4.2: CIVL scopes. left: partial code for a CIVL program with lexical scopes numbered; center: the static scope tree; right: a state consisting of 4 processes and 6 dynamic scopes.

6. a set of process IDs
7. for each process, a stack of 0 or more frames, where each frame gives a location and a corresponding dynamic scope.

The mapping from dynamic scopes to static scopes preserves the static scope tree structure. That is, if δ is a dynamic scope with parent δ' in the dynamic scope tree and $\sigma = \mathit{static}(\delta)$, then $\sigma' = \mathit{static}(\delta')$ is the parent of σ in the static scope tree.

Figure 4.2(right) shows a state of the model to its left. This state has two dynamic scopes which are instances of scope 1, no instance of scope 3, and 1 instance of each of the remaining scopes. There are 4 processes, whose call stacks are illustrated (the locations are not shown).

When control moves from one location to another within a function's transition system, the scope may change. When this happens, new dynamic scopes are created and added to the state. This is carried out in such a way that the correspondence between dynamic and static scopes is preserved. The protocol requires computing the "join" in the static scope tree of the old and new scopes, considering the path from the old scope to the join to the new scope, and then creating a corresponding structure in the dynamic tree; see Fig. 4.3(a-c).

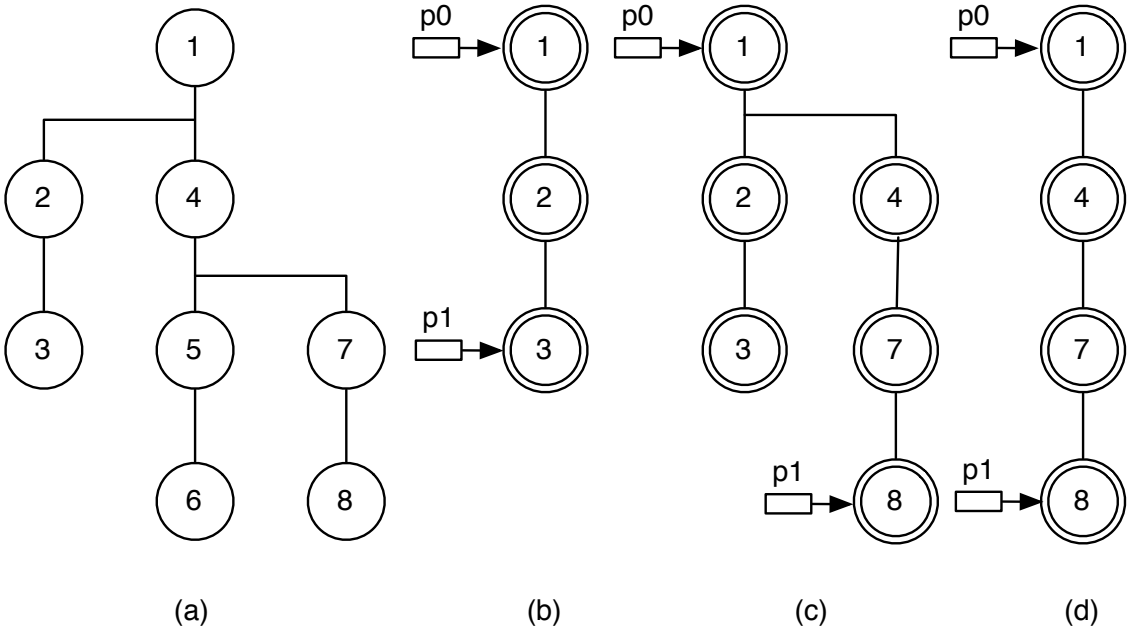


Figure 4.3: Jump protocol. (a) a static scope tree; (b) a dynamic scope tree; p_1 is about to move from a location in scope 3 to a location in scope 8; (c) new dynamic scopes are added corresponding to the path from scope 1 (the join of 3 and 8) to 8; (d) dynamic scopes 2 and 3 became unreachable and so were removed.

A dynamic scope s is *reachable* from a process p if

1. s is referenced in some frame of p's call stack, or
2. s is the parent of a dynamic scope s' that is reachable from p.

When a dyscope is not reachable from any process, we say that it is *unreachable*. Such a dyscope may be removed form the state; see Fig. 4.3(d).

If a call or spawn of a function f is executed in dynamic scope δ, then since f is visible, it must be the case that f is declared in $static(\delta')$, where δ' is either δ or an ancestor of δ. A new dynamic scope is created whose parent is δ' and whose (corresponding) lexical scope is the function scope of f. A new frame is created referring to the new dynamic scope. For a function call, the frame is pushed onto the existing stack; for a spawn, a new process is created whose stack consists of the single frame.

If a process has terminated and there are no references to that process in the state, it can be removed from the state.

When dynamic scopes or processes are removed from the state, scope IDs and process IDs are renumbered to put the state into a canonical form.

4.5 CIVL tool

The CIVL model checker is a tool written in Java for the verification of CIVL-C programs. It performs model checking with symbolic execution on CIVL models. It utilizes the following components:

- *ABC.* ABC is a Java-based C front-end. It provides a preprocessor and parser for C programs. ABC has been extended to also parse CIVL-C.
- *GMC: The Generic Model Checker.* GMC is a package that provides interfaces and basic search capability for model checking.
- *SARL: The Symbolic Algebra and Reasoning Library.* SARL is a library for creating, manipulating, and reasoning about symbolic expressions.

To verify a program, CIVL uses ABC to parse the CIVL-C source and create an abstract syntax tree. This AST then undergoes some analysis and transformation,

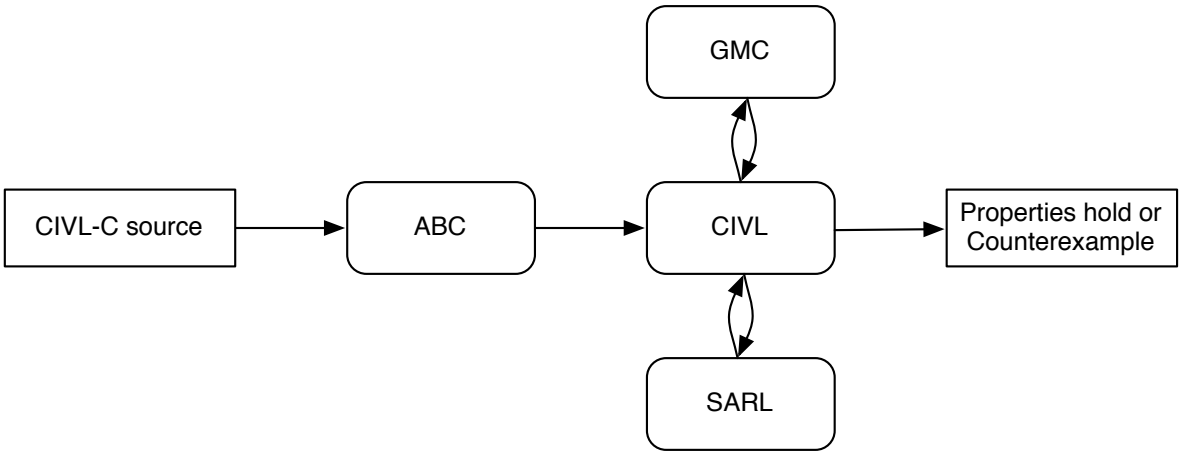


Figure 4.4: Data flow through the CIVL model checker.

and is converted into a CIVL model. CIVL has a representation of the state of a CIVL model and transitions between states. Values of state components are symbolic expressions, which are manipulated using SARL. GMC provides the functionality of searching the state space. After exhaustively exploring the state space, CIVL will either report that all specified properties hold or else provide a counterexample. Fig. 4.4 depicts the organization of CIVL.

When the CIVL model checker runs or verifies a CIVL-C program, it checks that the program is free of a variety of errors. These include:

- Deadlocks
- Assertion violations
- Division by 0
- Illegal pointer dereferences
- Out-of-bounds array indexes
- Invalid casts
- Use of uninitialized objects.

The CIVL tool runs from the command line in OS X or Linux, and has several commands:

- `help`: print usage information
- `run`: run the program using random simulation
- `verify`: verify the program
- `compare`: compare two programs for functional equivalence
- `replay`: replay a program trace
- `parse`: show the result of preprocessing and parsing the file
- `preprocess`: show the result of preprocessing the file only.

For full information on available CIVL commands, see App. B.

Chapter 5

DIFFERENTIAL ACCURACY SPECIFICATION

We present a new *differential accuracy specification language*, which adds several elements to the CIVL specification language. The new quantifier `$uniform` was mentioned above. The other enhancements are abstract functions, derivatives, and big O expressions.

Great care was taken in crafting these annotations to provide programmers with easy and readable ways to express the information necessary for verifying the order of accuracy. This involved examining the structure of proofs that mathematicians write, and abstracting the components of that information which would be useful to the verifier. For most proofs, there are three key pieces of information.

1. The function being reasoned about, including its input-output signature and some hints about how many terms to use in the truncated Taylor expansion.
2. The relationship between program variables and the values of the function evaluated at various grid points.
3. An assertion relating the values of output program variables to the actual mathematical computation.

The differential accuracy specification language was designed to facilitate concise statement of those three types of information.

5.1 Abstract functions and derivatives

Abstract functions are functions that have no body, but are used to represent the mathematical notion of a function (rather than the program notion). The syntax for an abstract function is

`$abstract contin(` *int* `)` *type name* `(` *param_list* `);`

where *int* is an integer indicating the number of partial derivatives that exist and are continuous, *type* is the return type of the abstract function, *name* is the name of the abstract function, and *param_list* is a (possibly empty) comma-separated list of parameter declarations. The `contin(`*int*`)` can be omitted when the value of the continuity is 0.

Abstract functions can be used in combination with assume statements to express the relationship between input variables to the program and the mathematical function whose results those input variables are representing. They are also useful for modeling functions performing mathematical computations (such as those described in C's `math.h`) without being dependent on a specific implementation.

Of course, when reasoning about numerical solutions to partial differential equations, a notion of abstract functions is insufficient. One also needs to refer to derivatives of those functions. We use notation somewhat similar to Mathematica [77], and specify a derivative by its function and the number of partial derivatives in terms of each parameter to the function. The syntax for a *derivative expression* is

`$D[`*func*`,`*partial_list*`](`*arg_list*`)`

where *func* is the name of the abstract function whose derivative is being taken, *partial_list* is a comma separated list of partials specified by the form {*var*`,`*int*}. For each partial, *var* is the name of the parameter and *int* indicates how many partial derivatives to take in terms of that parameter. Finally, *arg_list* is a comma separated list of expressions that are arguments to the function evaluation. The type of a derivative expression is the return type of the abstract function.

5.2 Big-O expressions

We have added a *big-O expression*. The big-O expression has the form `$O(`*exp*`)`. It represents the condition in Def. 1. Big-O expressions can be used in assumptions and assertions. They are also added automatically as a component of Taylor expansions in

our modifications to the CIVL verifier. The verifier is aware of various properties of big-O expressions, which allows them to be used in polynomial arithmetic. In particular, the verifier knows that

- for an expression x and integer $n > 0$, $xO(x^n) = O(x^{n+1})$, and
- for an expression x, integer $n > 0$, and non-zero constant c, $cO(x^n) = O(x^n)$.

5.3 The uniform quantifier

The `$uniform` quantifier has the same syntax as the existential or universal quantifiers. Its general form is

`$uniform {`*decl*`=`*lower* `..` *upper*`}` *expression* `=` *O-expression*,

where *decl* declares the name of a variable v, *lower* and *upper* are expressions bounding the values of v. `$uniform` indicates that the constant C (from Def. 1) in the big-O term is independent of any local variables or any values inside of the big-O.

Chapter 6

SYMBOLIC DIFFERENTIAL ACCURACY VERIFICATION

The technique of *symbolic differential accuracy verification* is based on symbolic execution. It uses symbolic representations of elements of the new specification language and heuristics to automatically add assumptions relating specified abstract functions evaluated at certain points to their Taylor expansions.

Every abstract function is represented by a unique symbolic constant. The symbolic constant has a type which is a symbolic representation of a function type. A symbolic function type has a list of input types corresponding to the function parameters, and a return type. Although an abstract function has no body, it can still be "called." An *abstract function call expression* looks like a regular program function call, but returns a symbolic expression representing the application of the abstract function to the arguments. The type of the abstract function call is the same as the return type of the function, and an abstract function call can occur anywhere an expression of that type is permitted.

As described in Ch. 5, derivative expressions take an abstract function and a list of partials, as well as an argument list. Each particular derivative (that is, each combination of an abstract function and a given set of partials) is represented by a unique symbolic constant. A derivative expression is then evaluated in the same way as an abstract function call expression, with the type of the resulting symbolic expression again being the return type of the abstract function. Note that the reasoner, upon encountering an abstract function call expression and an expression for a derivative of that abstract function, has no inherent way of knowing that the two are in any way related. A derivative expression is just an application of a different function to

some list of arguments. Any relationships between functions and their derivatives are explicitly inserted into the model as assumptions.

During the symbolic execution, a big-O expression also gets translated to a type of abstract function call. In order to capture the semantics of arithmetic with big-O terms, abstract function calls to the symbolic constant representing big-O are handled in special ways in some cases. The reasoner recognizes the patterns described in Sec. 5.2.

Having described the symbolic evaluation of abstract functions, derivatives, and big-O terms, what remains is to modify the model with extra information to relate these notions. One of the features of CIVL-C is the `$assume` statement, which allows the programmer to provide information to the verifier. Our technique uses heuristics to analyze existing `$assume` statements containing abstract function call expressions and to add new information to the model. The general procedure is

1. find assume statements containing abstract function calls
2. generate appropriate Taylor expansions based on the form of the assumption and the abstract function declaration
3. add new assumptions about the Taylor expansions to the model.

In step (2), the verifier applies some heuristics to generate the expansions. There are two heuristics used in the analysis, a spatial argument heuristic and a time argument heuristic. Whenever an `$assume` statement is added to the model, the assumed expression is examined to determine whether it contains an abstract function call. Only limited types of expressions are checked. The verifier recursively examines all arguments to binary and quantifier expressions and collects a list of the abstract function calls encountered. The recursive check terminates on any other type of expression. During the recursive check, quantifier expressions are also accumulated.

Each of the collected abstract function calls is then processed. For each function call, all arguments are checked to see whether they match either the spatial argument heuristic or the time argument heuristic. The spatial argument heuristic matches

arguments of the form $v_q * expr$ or $expr * v_q$, where v_q is one of the variables bound by a quantifier and $expr$ is any other expression. The idea behind the spatial argument heuristic is that a quantifier will often be used to relate an abstract function call at successive grid points to the values in an array or other data structure.

The time argument heuristic matches arguments of the form $v_i * v_{dt}$ or $v_{dt} * v_i$, where v_i is a variable of integer type and v_{dt} is an input variable. The intuition behind the time argument heuristic is that in an iterative numerical scheme, the time argument to an abstract function call is usually an integer counter (v_i) times the size of a time step (v_{dt}).

For each abstract function call argument that matches one of the heuristics, new assumptions are created and added to the model describing a truncated Taylor expansion around that argument. Whenever an expansion around an argument is added, all other arguments are kept the same as the original call. Since other arguments may contain variables bound by quantifiers, the new Taylor expansion expression is nested inside duplicates of all of the previously accumulated quantifiers.

Suppose an abstract function call for a function f with continuity c has an argument matching the spatial heuristic. Then the call is of the form $f(..., v_q * expr, ...)$. Two new assumptions will be added. They are

$$f(..., (v_q + 1) * expr, ...) = \left(\sum_{i=0}^{i=c-1} \frac{f^{(i)}(..., v_q * expr, ...)}{i!} expr^i \right) + O(expr^c) \tag{6.1}$$

$$f(..., (v_q - 1) * expr, ...) = \left(\sum_{i=0}^{i=c-1} (-1)^i \frac{f^{(i)}(..., v_q * expr, ...)}{i!} expr^i \right) + O(expr^c) \tag{6.2}$$

where $f^{(i)}$ indicates the i^{th} derivative of f.

Suppose an abstract function call for a function f has an argument matching the time heuristic. The the call is of the form $f(..., v_i * v_{dt}, ...)$. One new assumption will be added. It is

$$f(..., (v_i + 1) * v_{dt}, ...) = f(..., v_i * v_{dt}, ...) + \frac{f'(..., v_i * v_{dt}, ...)}{2} v_{dt} + O(v_{dt}^2) \tag{6.3}$$

The heuristics currently in use are relatively simple, but are able to provide sufficient information for the analysis of many finite difference schemes. Future work may involve additional static analysis to support more complex heuristics.

In the next part, we present a number of case studies and describe their mathematical analysis and the representation of accuracy claims in the specification language.

Chapter 7

EVALUATION

This chapter evaluates the technique of symbolic differential accuracy verification. We first compare symbolic differential accuracy verification to recent work on order of accuracy verification, then present several case studies of applying our technique to sample programs, and conclude with a discussion of scaling.

7.1 Comparison to existing techniques

There have been very few efforts to formally verify the order of accuracy of numerical programs. One notable exception is [10], which gives the results of a comprehensive formal verification of a 1d wave equation code.Using the Frama-C platform and various automated theorem provers and proof assistants, the authors of that study produced a mechanized proof of all correctness properties of the program, including convergence, order of accuracy, and bounds on floating-point error. This work differs from ours in two significant ways. First, many of the proofs required extensive interaction with the proof assistant. For example, a 5000-line long Coq [7] proof of method error was required (though around half of this may be re-usable for similar problems), and 32 verification conditions required Coq interaction to discharge. Second, the annotational burden is much larger than in our approach: for example, there are 174 lines of annotations (including axioms, lemmas, and definitions) added to a program which is 32 lines of uncommented C code (see `http://fost.saclay.inria.fr/coq_total/dirichlet.c.html`). In contrast, our method is fully automatic after adding the relatively small number of annotations we have shown. On the other hand, the method of [10] provides an extremely

high level of assurance (a proof based on first principles, with no bounds on input parameters), and deals with floating-point issues in addition to order of accuracy.

While symbolic differential accuracy verification only checks order of accuracy, the severely reduced annotational requirements (typically about three lines on a similarly sized differential equation solver) and fully automated nature of the proof should make it more accessible to prospective users of accuracy verification tools.

7.2 Case studies

This section examines the application of symbolic differential accuracy verification to a number of representative numerical codes. Each case study presents

1. the mathematical analysis that a mathematician or developer might do to prove that the algorithm is n^{th} order accurate
2. an excerpt of code implementing the algorithm, including differential accuracy specification annotations
3. a description of the symbolic differential accuracy verification steps performed by CIVL.

Note that (1) is independent of the code, but an understanding of the mathematical analysis is a prerequisite to writing a numerical program and making a claim about its accuracy (regardless of whether any formal verification of that accuracy will be attempted). Symbolic differential accuracy verification does not release the programmer from the obligation of performing the manual analysis. What is does instead is utilize the information from that mathematical analysis to provide a level of assurance that the code does indeed faithfully implement the technique to the claimed order of accuracy. The differential accuracy specification needed to enable the verification is not difficult to extract from the mathematical analysis. Symbolic differential accuracy verification takes the specification annotations and automatically applies techniques similar to the manual mathematical analysis to prove that the assertions hold. CIVL is able to verify that the assertions hold (with the exception of some erroneous versions that are mentioned, where the assertions fail as expected).

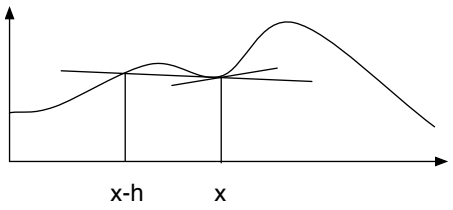


Figure 7.1: Backward differencing approximates the derivative of $\rho(x)$ as the slope through the points $(x-h, \rho(x-h))$ and $(x, \rho(x))$.

First we discuss case studies on differentiation in one and two dimensions. Then we describe solving a a useful and frequently studied parabolic partial differential equation, the diffusion equation. Finally we present a number of techniques for solving hyperbolic partial differential equations. In particular, we will focus on the advection operator

$$L[u] = \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} \tag{7.1}$$

in the case where $L[u] = 0$. The advection equation is useful for describing the transport of a substance via currents in fluid.

7.2.1 First derivative, backward

Our first case study is a backward difference approximation to the first derivative. The backward finite difference method approximates a derivative at a point using that point and the left neighbor.

7.2.1.1 Mathematical analysis

Let $\rho : \mathbb{R} \to \mathbb{R}$ be two times differentiable and assume there exists $M > 0$ such that $|\rho''(x)| < M$ for all x. A discrete approximation of the derivative is given by

$$g(x,h) \equiv \frac{\rho(x) - \rho(x-h)}{h}. \tag{7.2}$$

In Definition 3, $f(x) = \rho'(x)$. We claim that g is a uniformly 1^{st} order approximation for f on $\mathbb{R}$. To verify, we use Taylor polynomials with Lagrangian remainders. Given $x \in \mathbb{R}$ and $h > 0$, there exists $\xi \in [x-h, x+h]$ such that

$$\rho(x-h) = \rho(x) - \rho'(x)h + \frac{1}{2}\rho''(\xi)h^2.$$

From this we conclude

$$\left|\frac{\rho(x) - \rho(x-h)}{h} - \rho'(x)\right| = \frac{|\rho''(\xi)|}{2}h \leq \frac{1}{2}Mh.$$

We see that the difference between the discrete approximation and the exact derivative is bounded by a constant times h, and thus g is a uniformly 1^{st} order accurate approximation of f on $\mathbb{R}$.

7.2.1.2 Specification

Fig. 7.2 is an excerpt of CIVL-C code for the backward finite difference scheme for differentiation. The code takes as input h and an array which holds values of the function at the points ih, where $i \in \mathbb{Z}$ and $0 \leq i \leq n$. These points form the grid $\Delta(h)$ from Definition 4. The values stored in `result` at the end are the output of g_h. Since backward differencing cannot be performed on the left end of the array, we use forward differencing at the first position.

Three annotations are needed to provide the differential accuracy specification. Line 4 is a declaration of the abstract function ρ, which is declared to have 2 continuous, bounded derivatives. The assumption at line 7 relates the values in the input array `y` to the function ρ. Line 13 is the assertion about the relationship between `result` and the actual derivative. Note the `$uniform` quantifier at line 13, which indicates that the code is Δ-uniformly 1^{st} order accurate.

By applying the symbolic differential accuracy verification technique to the given annotations, the verifier can automatically add information about Taylor expansions of ρ to the model, and then prove that the results computed by the code are Δ-uniformly 1^{st} order accurate.

```
1 $input double h;
2 $input int num_elements;
3 $assume h > 0;
4 $abstract $contin(2) $real rho($real x);
5
6 void differentiate(double h, int n, double y[], double result[]){
7   $assume $forall {m=0 .. n-1} y[m] == rho(m*h);
8   for(int i = 1; i < n; i++) {
9     result[i] = (y[i]-y[i-1])/h;
10  }
11  // forward at endpoint
12  result[0] = (y[1] - y[0])/h;
13  $assert($uniform {k=0 .. n-1} result[k]-$D[rho,{x,1}](k*h) == $O(h));
14 }
```

Figure 7.2: Annotated CIVL-C code for differentiation. The code does backward differencing on the array except for index 0, where it does forward differencing.

7.2.1.3 Verification

Consider lines 4 and 7 from Fig. 7.2:

```
$abstract $contin(2) $real rho($real x);
$assume $forall {m=0..n-1} y[m] == rho(m*h);
```

From the declaration of the abstract function the verifier knows that two derivatives of ρ exist and are continuous. The assumption provides a clue about which points to expand around. In this case, the verifier recognizes the argument to the abstract function as matching the spatial argument heuristic. The new assumptions will quantify over the same range, and expand around points $(m+1)h$ and $(m-1)h$. The automatically generated assumptions are:

```
$assume $forall {m=0..n-1} rho((m+1)*h)==rho(m*h)+$D[rho,{x,1}](m*h)*h
       +$O(h*h);
$assume $forall {m=0..n-1} rho((m-1)*h)==rho(m*h)-$D[rho,{x,1}](m*h)*h
       +$O(h*h);
```

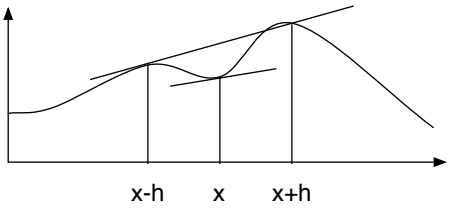


Figure 7.3: Central differencing approximates the derivative of $\rho(x)$ as the slope through the points $(x-h, \rho(x-h))$ and $(x+h, \rho(x+h))$.

These assumptions are added to the model immediately following the assumption at line 8. Recall that a `$assume` statement adds the expression to the path condition. Thus the Taylor expansions are carried in the path condition throughout the remainder of the program. The information isn't needed until the evaluation of the assertion at line 13. At this point the theorem prover is able to use these assumptions that are encoded in the path condition to prove the assertion. The actual query passed to the prover for a CIVL verifier run on this program is given in App. C. In the query, `BIG_O` is the big-O function and `rhox1` is the first derivative of `rho` with respect to `x`. It should be stressed that the query is automatically generated during symbolic differential accuracy verification, and is the tool's response to the few lines of annotation provided.

7.2.2 First derivative, centered

Our next case study is a centered difference approximation to the first derivative.

7.2.2.1 Mathematical analysis

Let $\rho : \mathbb{R} \to \mathbb{R}$ be three times differentiable and assume there exists $M > 0$ such that $|\rho'''(x)| < M$ for all x. A discrete approximation of the derivative is given by

$$g(x,h) \equiv \frac{\rho(x+h) - \rho(x-h)}{2h}. \tag{7.3}$$

In Definition 3, $f(x) = \rho'(x)$. We claim that g is a uniformly 2^{nd} order approximation for f on $\mathbb{R}$. To verify, we use Taylor polynomials with Lagrangian remainders. Given $x \in \mathbb{R}$ and $h > 0$, there exist $\xi_1, \xi_2 \in [x-h, x+h]$ such that

$$\rho(x+h) = \rho(x) + \rho'(x)h + \frac{1}{2}\rho''(x)h^2 + \frac{1}{6}\rho'''(\xi_1)h^3$$
$$\rho(x-h) = \rho(x) - \rho'(x)h + \frac{1}{2}\rho''(x)h^2 - \frac{1}{6}\rho'''(\xi_2)h^3.$$

From this we conclude

$$\left|\frac{\rho(x+h) - \rho(x-h)}{2h} - \rho'(x)\right| = \frac{|\rho'''(\xi_1) + \rho'''(\xi_2)|}{12}h^2 \leq \frac{1}{6}Mh^2.$$

We see that the difference between the discrete approximation and the exact derivative is bounded by a constant times h^2, and thus g is a uniformly 2^{nd} order accurate approximation of f on $\mathbb{R}$.

7.2.2.2 Specification

Fig. 7.4 is an excerpt of CIVL-C code. As in Sec. 7.2.1, the code takes as input h and an array which holds values of the function at the points ih, where $i \in \mathbb{Z}$ and $0 \leq i \leq n$. These points form the grid $\Delta(h)$ from Definition 4. The values stored in `result` at the end are the output of g_h. Since central differencing cannot be performed on the endpoints, we use forward and backward differencing at the first and last positions, respectively. These are first order accurate methods. Hence the result is not Δ-uniformly second order accurate. Instead we will only consider points in the interior of the domain, which we will call Δ'. Define $\Delta'(h) = \{ih | 1 \leq i < n-1\}$. All points in Δ' are then computed using central differencing. Note that the bounds on i are the same as the bounds on the quantified variable `k` in the assertion at line 13.

Line 4 is a declaration of the abstract function ρ, which is declared to have 3 continuous, bounded derivatives. The assumption at line 7 relates the values in the input array `y` to the function ρ. Line 12 is the assertion about the relationship between `result` and the actual derivative, namely that the code is Δ'-uniformly 2^{nd} order accurate.

```
1 $input double h;
2 $input int num_elements;
3 $assume h > 0;
4 $abstract $contin(3) $real rho($real x);
5
6 void differentiate(int n, double y[], double result[]){
7   $assume $forall {m=0 .. n-1} y[m] == rho(m*h);
8   for(int i = 1; i < n-1; i++)
9     result[i] = (y[i+1]-y[i-1])/(2*h);
10  result[0] = (y[1]-y[0])/h;
11  result[n-1] = (y[n-1] - y[n-2])/h;
12  $assert($uniform{k=1..n-2} result[k]-$D[rho,{x,1}](k*h)==$O(h*h));
13 }
```

Figure 7.4: Annotated CIVL-C code for differentiation. The code does central differencing on the interior of the array and forward/backward differencing for the endpoints.

7.2.2.3 Verification

Lines 4 and 7 in Fig. 7.4 look verify similar to the corresponding lines in Fig. 7.2:

```
$abstract $contin(3) $real rho($real x);
$assume $forall {m=0..n-1} y[m] == rho(m*h);
```

In the backward differencing case, the function `rho` was specified to have two continuous derivatives. In the centered example, three continuous derivatives are specified. It is, of course, possible to apply these programs to the same `rho`. By declaring the minimum number of continuous derivatives that must exist for the analysis of the numerical method, the programmer assists the tool in choosing the correct Taylor expansions. The argument to `rho` in the assumption at line 7 matches the spatial argument heuristic, so the automatically generated assumptions for this program are:

```
$assume $forall {m=0..n-1} rho((m+1)*h)==rho(m*h)+$D[rho,{x,1}](m*h)*h
       +$D[rho,{x,2}](m*h)*h*h/2+$O(h*h*h);
$assume $forall {m=0..n-1} rho((m-1)*h)==rho(m*h)-$D[rho,{x,1}](m*h)*h
       +$D[rho,{x,2}](m*h)*h*h/2+$O(h*h*h);
```

When the verifier encounters the assertion at line 12, it is able to use the assumptions to verify the assertion holds. Recall that this example is only second order accurate on Δ', not on the whole domain. If the assertion is modified to include the endpoints, the tool correctly reports that the properties may not hold.

7.2.3 Second derivative

We next consider approximating a second derivative using central differencing. The analysis is similar to Section 7.2.2 above, but requires an additional term in the Taylor expansions of ρ.

7.2.3.1 Mathematical analysis

Let $\rho : \mathbb{R} \to \mathbb{R}$ be four times differentiable. Suppose there exists $M > 0$ such that $|\rho''''(x)| < M$ for all x. The approximation is given by

$$g(x,h) \equiv \frac{\rho(x+h) - 2\rho(x) + \rho(x-h)}{h^2}. \tag{7.4}$$

In Def. 3, $f(x) = \rho''(x)$. We claim that g is a uniformly 2^{nd} order approximation for f on $\mathbb{R}$. To verify, we use Taylor polynomials with Lagrangian remainders. Given $x \in \mathbb{R}$ and $h > 0$, there exist $\xi_1, \xi_2 \in [x-h, x+h]$ such that

$$\begin{aligned}
\rho(x+h) &= \rho(x) + \rho'(x)h + \frac{1}{2}\rho''(x)h^2 + \frac{1}{6}\rho'''(x)h^3 + \frac{1}{24}\rho''''(\xi_1)h^4 \\
\rho(x-h) &= \rho(x) - \rho'(x)h + \frac{1}{2}\rho''(x)h^2 - \frac{1}{6}\rho'''(x)h^3 + \frac{1}{24}\rho''''(\xi_2)h^4.
\end{aligned}$$

From this we compute the accuracy:

$$\left| \frac{\rho(x+h) - 2\rho(x) + \rho(x-h)}{h^2} - \rho''(x) \right| = \frac{|\rho'''(\xi_1) + \rho'''(\xi_2)|}{24} h^2 \leq \frac{1}{12} M h^2.$$

We see that the difference between the discrete approximation and second derivative is bounded by a constant times h^2, and thus g is a uniformly 2^{nd} order accurate approximation of f on $\mathbb{R}$.

```
1 $input double h;
2 $input int num_elements;
3 $input double initial[num_elements];
4 $abstract $contin(4) $real rho($real x);
5 $assume h > 0;
6
7 void secondDerivative(double h, int n, double y[], double result[]){
8   $assume $forall {m=0 .. n-1} y[m] == rho(m*h);
9   for(int i = 1; i < n-1; i++)
10    result[i] = (y[i+1]-2*y[i]+y[i-1])/(h*h);
11  result[0] = (y[2]-2*y[1]+y[0])/h;
12  result[n-1] = (y[n-3] - 2*y[n-2]-y[n-1])/h;
13  $assert($uniform{k=1 .. n-2} result[k]-$D[rho,{x,2}](k*h)==$O(h*h));
14 }
```

Figure 7.5: Annotated CIVL-C code for second derivative. The code does central differencing on the interior of the array and forward/backward differencing for the endpoints.

7.2.3.2 Specification

Fig. 7.5 is an excerpt of the CIVL-C code for computing the second derivative. As in Sections 7.2.1 and 7.2.2, the code takes the grid separation h and an array of inputs holding the values of $\rho(ih)$.

Line 4 is the declaration of the abstract function ρ. Unlike the previous example, ρ is specified here to have four continuous derivatives. In practice this ρ might be the same ρ as the differentiation examples, but specifying four continuous derivatives gives the verifier a useful hint on how many terms to create in the Taylor expansions. The assumption at line 8 tells the verifier the relationship between the array values and the function ρ. Line 13 asserts that the approximation is uniformly 2^{nd} order accurate in h.

7.2.3.3 Verification

The second derivative code is a natural extension of the centered first derivative. An additional term is needed in the truncated Taylor expansion in order to prove the

result, and thus Line 4 in Fig. 7.5 indicates four continuous derivatives. The argument to the abstract function call in the assumption at line 13 matches the spatial argument heuristic. The automatically generated assumptions are:

```
$assume $forall {m=0..n-1} rho((m+1)*h)==rho(m*h)+$D[rho,{x,1}](m*h)*h
        +$D[rho,{x,2}](m*h)*h*h/2+$D[rho,{x,3}](m*h)*h*h*h/6+$O(h*h*h*h);
$assume $forall {m=0..n-1} rho((m-1)*h)==rho(m*h)-$D[rho,{x,1}](m*h)*h
        +$D[rho,{x,2}](m*h)*h*h/2-$D[rho,{x,3}](m*h)*h*h*h/6+$O(h*h*h*h);
```

7.2.4 Laplace Operator

The Laplace operator ∇^2 is a differential operator that is useful for describing a wide range of problems in mathematics, science and engineering.

7.2.4.1 Mathematical Analysis

The two-dimensional cartesian form of the Laplace operator is

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

Using a grid size of h in both dimensions, the Laplace operator can be approximated using a finite difference scheme on the five point stencil shown in Fig. 7.6. The approximation is given by

$$g(x, y, h) \equiv \frac{u(x-h, y) + u(x, y-h) - 4u(x, y) + u(x+h, y) + u(x, y+h)}{h^2}.$$

The truncated Taylor polynomials used here are analogous to those in Section 7.2.3, but expansions must happen about $u(x, y)$ in both the x and y directions. Substituting the appropriate expansions into the finite difference scheme and simplifying shows that the approximation is $O(h^2)$ accurate.

7.2.4.2 Specification

Code for computing the Laplace operator is given in Fig. 7.7. The abstract function $\phi(x, y)$ is declared at line 6. It indicates that four bounded, continuous derivatives

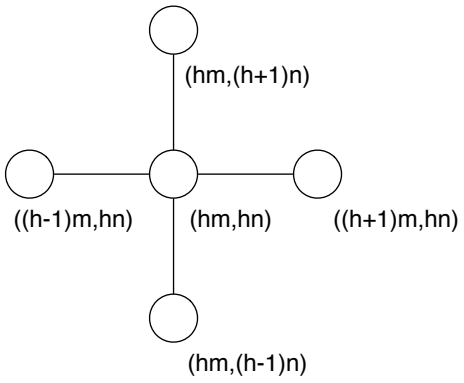


Figure 7.6: Five point stencil for the 2D Laplace operator.

may be taken. The assumption at line 7 equates elements in the two-dimensional array `u` to values of the abstract function. Note that variables from the quantifiers are involved in both arguments to ϕ. That information assists the verifier in making choices about which Taylor polynomials to create. Lines 14–15 assert that the result is uniformly 2^{nd} order accurate in h.

7.2.4.3 Verification

The code at lines 6 and 7 in Fig. 7.7 leads the verifier to create more and higher order Taylor polynomials.

```
$abstract $contin(4) $real phi($real x, $real y);
$assume $forall{m=0..rows-1} $forall{n=0..cols-1} u[m][n]==phi(m*h,n*h);
```

The specification of four continuous derivatives in the definition of ϕ tells the verifier to expand the Taylor polynomial until the $O(h^4)$ term. In this case, both arguments to the abstract function call match the spatial argument heuristic, so the verifier adds four Taylor expansions. The automatically generated assumptions are:

```
1 $input double h;
2 $input int rows, cols;
3 $input double u[rows][cols];
4 double result[rows][cols];
5 $assume h > 0;
6 $abstract $contin(4) $real phi($real x, $real y);
7 $assume $forall{m=0..rows-1} $forall{n=0..cols-1} u[m][n]==phi(m*h,n*h);
8
9 void laplace() {
10   for (int i=1; i < rows-1; i++)
11     for (int j=1; j < cols-1; j++)
12       result[i][j]=(u[i-1][j]+u[i][j-1]-4*u[i][j]+u[i+1][j]+u[i][j+1])\
13                       /(h*h);
14   $assert($uniform{i=1..rows-2} $uniform{j=1..cols-2} result[i][j]-\
15           ($D[phi,{x,2}](i*h,j*h)+$D[phi,{y,2}](i*h,j*h))==$O(h*h));
16 }
```

Figure 7.7: Annotated CIVL-C code for the Laplace operator in two dimensions. The code does central differencing on the interior of the array. The boundary is held constant.

```
$assume $forall{m=0..rows-1} $forall{n=0..cols-1} phi((m+1)*h,n*h)==
       phi(m*h,n*h)+$D[phi,{x,1}](m*h,n*h)*h
       +$D[phi,{x,2}](m*h,n*h)*h*h/2+$D[phi,{x,3}](m*h,n*h)*h*h*h/6
       +$O(h*h*h*h);
$assume $forall{m=0..rows-1} $forall{n=0..cols-1} phi((m-1)*h,n*h)==
       phi(m*h,n*h)-$D[phi,{x,1}](m*h,n*h)*h
       +$D[phi,{x,2}](m*h,n*h)*h*h/2-$D[phi,{x,3}](m*h,n*h)*h*h*h/6
       +$O(h*h*h*h);
$assume $forall{m=0..rows-1} $forall{n=0..cols-1} phi(m*h,(n+1)*h)==
       phi(m*h,n*h)+$D[phi,{y,1}](m*h,n*h)*h
       +$D[phi,{y,2}](m*h,n*h)*h*h/2+$D[phi,{y,3}](m*h,n*h)*h*h*h/6
       +$O(h*h*h*h);
$assume $forall{m=0..rows-1} $forall{n=0..cols-1} phi(m*h,(n-1)*h)==
       phi(m*h,n*h)+$D[phi,{y,1}](m*h,n*h)*h
       +$D[phi,{y,2}](m*h,n*h)*h*h/2+$D[phi,{y,3}](m*h,n*h)*h*h*h/6
       +$O(h*h*h*h);
```

7.2.5 Diffusion

Next we consider solving the 1-dimensional diffusion equation.

7.2.5.1 Mathematical analysis

We again begin by describing the math to give insight into what the tool must do automatically during symbolic differential accuracy verification. This example is a differential equation presented in terms of a differential operator. Solutions to differential equations may be numerically computed using approximation schemes for the differential operator. We present the notion of order of accuracy for a numerical approximation scheme.

In the discussion below, $Func(X, Y)$ denotes the set of functions from X to Y. If X and Y are continuous domains, we assume the functions are sufficiently smooth to take the necessary number of derivatives.

Definition 6 (Accuracy of a Scheme)**.** *Let n be a positive integer, $D \subseteq \mathbb{R}$, and $L : Func(D, \mathbb{R}) \to Func(D, \mathbb{R})$. Let $I = (0, a)$, where a is a positive real number and suppose $\Delta : I \to \wp(D)$. Let $r_{\Delta_h} : Func(D, \mathbb{R}) \to Func(\Delta(h), \mathbb{R})$ be the operator which restricts a function to $\Delta(h)$. Suppose for each h there is an operator $\hat{L}_h : Func(\Delta(h), \mathbb{R}) \to Func(\Delta(h), \mathbb{R})$. For any smooth function $u : D \to \mathbb{R}$, define $\psi_u : I \to \mathbb{R}$ by*

$$\psi_u(h) = \sup_{x \in \Delta(h)} \left| r_{\Delta_h}[L[u]](x) - \hat{L}_h[r_{\Delta_h}[u]](x) \right|.$$

We say $\hat{L}$ is a Δ-uniformly n^{th} order accurate scheme for L *if for all u*

$$\psi_u(h) = O(h^n) \text{ as } h \to 0.$$

See [70] Def. 3.1.1. This is the special case when the forcing function $f = 0$.

This definition can be generalized to multiple variables by defining an appropriate Δ and modifying ψ accordingly.

We now apply the notion of accuracy of a scheme to the diffusion example. Let $D \subseteq \mathbb{R} \times \mathbb{R}$. Define an operator L which takes a function $v : D \to \mathbb{R}$ that is twice differentiable in x and once in t and returns a function $L[v] : D \to \mathbb{R}$ as follows:

$$L[v] = \frac{\partial v}{\partial t} - \kappa \frac{\partial^2 v}{\partial x^2} \tag{7.5}$$

where κ is a positive constant. The 1-dimensional diffusion equation is the equation $L[u] = 0$. A typical problem specifies boundary conditions in addition to the basic equation $L[u] = 0$. The goal is to find a function u which satisfies all of these constraints. We will also suppose that the solution u must be four times differentiable in x and twice differentiable in t.

Given positive real numbers h_0, h_1, we define a uniform mesh

$$\Delta = \Delta(h_0, h_1) = \{(ih_0, nh_1) \in D | i, n \in \mathbb{Z}\}. \tag{7.6}$$

Given a function f from Δ to $\mathbb{R}$, we will write f_i^n for the value of f at the point (ih_0, nh_1). We define a restriction operator r_Δ which takes a function $v : D \to \mathbb{R}$ and returns the restriction $r_\Delta[v]$ of v to Δ. To summarize:

$$r_\Delta[v]_i^n = v(ih_0, nh_1) \tag{7.7}$$

where i and n are integers.

Using finite differences, we obtain the discretized operator $\hat{L} = \hat{L}_{h_0,h_1}$ (we will typically omit the subscript for brevity) as follows. $\hat{L}$ takes a function $\hat{v} : \Delta \to \mathbb{R}$ and returns a function $\hat{L}[\hat{v}] : \Delta \to \mathbb{R}$ and is defined by

$$\hat{L}[\hat{v}]_i^n = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} - \kappa \frac{\hat{v}_{i+1}^n - 2\hat{v}_i^n + \hat{v}_{i-1}^n}{h_0^2}. \tag{7.8}$$

We claim that $\hat{L}$ is a scheme for L that is Δ-uniformly accurate of order (2,1). That is, it is second order accurate in h_0 and first order accurate in h_1. In order to show

this, we must do several Taylor expansions. Given $i, n \in \mathbb{Z}$ and $h_0, h_1 > 0$, there exist $\xi_0, \xi_1 \in [(i-1)h_0, (i+1)h_0]$ and $\xi_2 \in [(n-1)h_1, (n+1)h_1]$ such that the following hold:

$$\begin{aligned} r_\Delta[u]_{i+1}^n &= u(ih_0, nh_1) + h_0 \frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2 \frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\ &\quad + \frac{1}{6}h_0^3 \frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4 \frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) \qquad (7.9) \\ r_\Delta[u]_{i-1}^n &= u(ih_0, nh_1) - h_0 \frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2 \frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\ &\quad - \frac{1}{6}h_0^3 \frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4 \frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1) \qquad (7.10) \\ r_\Delta[u]_i^{n+1} &= u(ih_0, nh_1) + h_1 \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2 \frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2). \qquad (7.11) \end{aligned}$$

Assume that on the domain of interest, the absolute value of the fourth derivative of u with respect to x is bounded by $M_0 > 0$ and the absolute value of the second derivative of u with respect to t is bounded by $M_1 > 0$. Substituting the expansions into (7.8),

$$\begin{aligned} \hat{L}[r_\Delta[u]]_i^n &= \frac{r_\Delta[u]_i^{n+1} - u(ih_0, nh_1)}{h_1} \\ &\quad - \kappa \frac{r_\Delta[u]_{i+1}^n - 2u(ih_0, nh_1) + r_\Delta[u]_{i-1}^n}{h_0^2} \qquad (7.12) \\ &= \frac{h_1 \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2 \frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2)}{h_1} \\ &\quad - \kappa \frac{h_0^2 \frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{24}h_0^4 (\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) + \frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1))}{h_0^2} \qquad (7.13) \\ &= \frac{\partial u}{\partial t}(ih_0, nh_1) - \kappa \frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{2}h_1 \frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2) \\ &\quad + \frac{1}{24}h_0^2 (\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) + \frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1)). \qquad (7.14) \end{aligned}$$

As in Def. 6, we subtract $L[u]$ from (7.14) to get the desired result:

$$\left|\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n\right| = \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_2) + \frac{1}{24}h_0^2\left(\frac{\partial^4 u}{\partial x^4}(\xi_0,nh_1) + \frac{\partial^4 u}{\partial x^4}(\xi_1,nh_1)\right)\right| \tag{7.15}$$

$$\leq \frac{1}{2}h_1\left|\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_2)\right| + \frac{1}{24}h_0^2\left|\frac{\partial^4 u}{\partial x^4}(\xi_0,nh_1) + \frac{\partial^4 u}{\partial x^4}(\xi_1,nh_1)\right| \tag{7.16}$$

$$\leq \frac{M_1}{2}h_1 + \frac{M_0}{12}h_0^2. \tag{7.17}$$

By Def. 5, we conclude that

$$\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n = O(h_1) + O(h_0^2). \tag{7.18}$$

7.2.5.2 Specification

Fig. 7.8 gives an example snippet of a diffusion CIVL-C code annotated for accuracy. The view of the computation in the program is slightly different than the presentation of the math above. In the code, time steps are computed iteratively. This involves rearranging (7.8). Recall that $\hat{L}[\hat{u}] = 0$. That is,

$$0 = \frac{\hat{u}_i^{n+1} - \hat{u}_i^n}{h_1} - \kappa\frac{\hat{u}_{i+1}^n - 2\hat{u}_i^n + \hat{u}_{i-1}^n}{h_0^2} \tag{7.19}$$

$$0 = \hat{u}_i^{n+1} - \hat{u}_i^n - h_1\kappa\frac{\hat{u}_{i+1}^n - 2\hat{u}_i^n + \hat{u}_{i-1}^n}{h_0^2} \tag{7.20}$$

$$\hat{u}_i^{n+1} = \hat{u}_i^n + h_1\kappa\frac{\hat{u}_{i+1}^n - 2\hat{u}_i^n + \hat{u}_{i-1}^n}{h_0^2}. \tag{7.21}$$

Define $\Phi : Func(\Delta,\mathbb{R}) \to Func(\Delta,\mathbb{R})$ by

$$\Phi[\hat{v}] = \hat{v}_i^n + h_1\kappa\frac{\hat{v}_{i+1}^n - 2\hat{v}_i^n + \hat{v}_{i-1}^n}{h_0^2}$$

This is what is actually computed in Fig. 7.8 at lines 16–17. CIVL can extract Φ by symbolically executing the update function.

We define a shift operator $S : Func(\Delta,\mathbb{R}) \to Func(\Delta,\mathbb{R})$ by $S[\hat{v}]_i^n = \hat{v}_i^{n+1}$. Eq. (7.21) then becomes $S[\hat{u}] = \Phi[\hat{u}]$. We rewrite $\hat{L}$ in terms of S and Φ to get

$$\hat{L}[\hat{v}] = \frac{S[\hat{v}] - \Phi[\hat{v}]}{h_1}. \tag{7.22}$$

```
$input int n; /* Number of points */
$input double h; /* Distance between points */
$input double dt; /* Size of a time step */
$input double k; /* Constant for rate of diffusion */
$abstract $contin(4) $real u($real x, $real t);
$assume h > 0 && dt > 0 && k > 0;
double v[n], v_new[n];
int iter;

void update() {
  $assume $forall {j=0 .. n-1} v[j] == u(j*h, iter*dt);
  for (int i = 1; i < n-1; i++)
    v_new[i] = v[i]+dt*k*(v[i+1]-2*v[i]+v[i-1])/(h*h);
  for (int i = 1; i < n-1; i++)
    v[i] = v_new[i];
  $assert($uniform{m=1 .. n-2} (u(m*h, (iter+1)*dt)-v[m])/dt \
   -$D[u,{t,1}](m*h,iter*dt)+k*$D[u,{x,2}](m*h,iter*dt)==$O(dt)+$O(h*h));
}
```

Figure 7.8: Annotated CIVL-C code for iterative diffusion in one dimension.

Note that $\hat{L}[\hat{u}] = 0$ (since $\hat{u}$ satisfies $S[\hat{u}] = \Phi[\hat{u}]$), but $\hat{L}[\hat{v}]$ is not necessarily 0 for arbitrary $\hat{v}$.

Since Φ is what the code actually computes, we want to rewrite (7.18) in terms of Φ. We use (7.22) to get

$$\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n = \frac{S[r_\Delta[u]]_i^n - \Phi[r_\Delta[u]]_i^n}{h_1} - r_\Delta[L[u]]_i^n. \tag{7.23}$$

The assertion in the code is

$$\frac{S[r_\Delta[u]]_i^n - \Phi[r_\Delta[u]]_i^n}{h_1} - r_\Delta[L[u]]_i^n = O(h_1) + O(h_0^2). \tag{7.24}$$

By phrasing the assertion in this way, the information about $\hat{L}$ is determined implicitly.

7.2.5.3 Verification

This diffusion solver is an iterative scheme. The abstract function `u` has two parameters: one for space and one for time. In the assumption at line 11, the arguments to the abstract function call have slightly different forms. The argument `j*h` matches the spatial argument heuristic. During symbolic differential accuracy verification, the verifier will recognize the form of the argument and combined with the continuity specification to create the following assumptions:

```
$assume $forall {j=0..n-1} u((j+1)*h,iter*dt)==u(j*h,iter*dt)
       +$D[u,{x,1}](j*h,iter*dt)*h+$D[u,{x,2}](j*h,iter*dt)*h*h/2
       +$D[u,{x,3}](j*h,iter*dt)*h*h*h/6+$O(h*h*h*h);
$assume $forall {j=0..n-1} u((j-1)*h,iter*dt)==u(j*h,iter*dt)
       -$D[u,{x,1}](j*h,iter*dt)*h+$D[u,{x,2}](j*h,iter*dt)*h*h/2
       -$D[u,{x,3}](j*h,iter*dt)*h*h*h/6+$O(h*h*h*h);
```

The other argument, `iter*dt` does not match the spatial argument heuristic (because no component of the expression is a variable bound by a quantifier). Instead, it matches the time argument heuristic, resulting in the addition of the following assumption:

```
$assume $forall {j=0..n-1} u(j*h,(iter+1)*dt)==u(j*h,iter*dt)
       +$D[u,{t,1}](j*h,iter*dt)*dt+$O(dt*dt);
```

When the assertion at lines 16-17 is passed to the prover, these three expansions in the path condition provide enough information for the prover to determine that the query is valid.

In this chapter we present a number of techniques for solving hyperbolic partial differential equations. In particular, we will focus on the advection operator

$$L[u] = \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} \tag{7.25}$$

in the case where $L[u] = 0$. The advection equation is useful for describing the transport of a substance via currents in fluid.

7.2.6 Upwind scheme, first order

The upwind scheme for one dimensional advection uses forward differencing in time and makes a choice of forward or backward differencing in space based on the direction of the fluid flow. The stencil in Fig. 7.9 represents the points involved in the upwind scheme computation.

7.2.6.1 Mathematical analysis

The discretized operator $\hat{L}$ for the first order upwind method takes a function $\hat{v} : \Delta \to \mathbb{R}$ and returns a function $\hat{L}[\hat{v}] : \Delta \to \mathbb{R}$. It is defined by

$$\hat{L}[\hat{v}] = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} + a\frac{\hat{v}_i^n - \hat{v}_{i-1}^n}{h_0} \text{ for } a > 0 \tag{7.26}$$

$$\hat{L}[\hat{v}] = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} + a\frac{\hat{v}_{i+1}^n - \hat{v}_i^n}{h_0} \text{ for } a < 0. \tag{7.27}$$

We claim that $\hat{L}$ is a scheme for L that is Δ-uniformly accurate of order $(1, 1)$. Given $i, n \in \mathbb{Z}$ and $h_0, h_1 > 0$, there exist $\xi_0, \xi_1 \in [(i-1)h_0]$ and $\xi_2 \in [(n-1)h_1, (n+1)h_1]$ such that the following hold:

$$r_\Delta[u]_{i+1}^n = u(ih_0, nh_1) + h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(\xi_0, nh_1) \tag{7.28}$$

$$r_\Delta[u]_{i-1}^n = u(ih_0, nh_1) - h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(\xi_1, nh_1) \tag{7.29}$$

$$r_\Delta[u]_i^{n+1} = u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2). \tag{7.30}$$

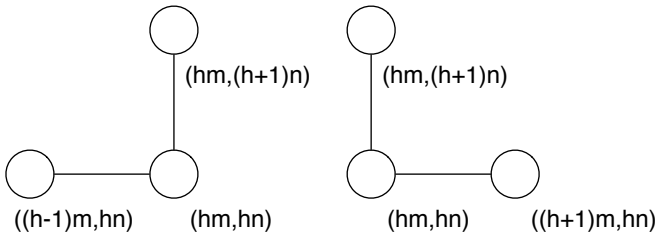


Figure 7.9: Stencils for the first order upwind scheme for linear advection when a is positive (left) or negative (right).

Assume that on the domain of interest, the absolute values of the second derivatives of u with respect to both x and t are bounded by $M > 0$. Substituting the expansions into Eq. 7.26,

$$\begin{aligned}\hat{L}[r_\Delta[u]]_i^n &= \frac{u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2) - u(ih_0, nh_1)}{h_1} \\ &\quad + a\frac{u(ih_0, nh_1) - (u(ih_0, nh_1) - h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(\xi_1, nh_1))}{h_0} \qquad (7.31) \\ &= \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2) \\ &\quad + a\frac{\partial u}{\partial x}(ih_0, nh_1) - \frac{a}{2}h_0\frac{\partial^2 u}{\partial x^2}(\xi_1, nh_1) \qquad (7.32)\end{aligned}$$

Similarly, substituting the expansions into Eq. 7.27,

$$\begin{aligned}\hat{L}[r_\Delta[u]]_i^n &= \frac{u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2) - u(ih_0, nh_1)}{h_1} \\ &\quad + a\frac{u(ih_0, nh_1) + h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(\xi_0, nh_1) - u(ih_0, nh_1)}{h_0} \qquad (7.33) \\ &= \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_2) \\ &\quad + a\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{a}{2}h_0\frac{\partial^2 u}{\partial x^2}(\xi_1, nh_1) \qquad (7.34)\end{aligned}$$

Subtracting $L[u]$ from Eq. 7.32 and 7.34 yields

$$\left|\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n\right| = \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_2) - \frac{a}{2}h_0\frac{\partial^2 u}{\partial x^2}(\xi_1,nh_1)\right| \tag{7.35}$$

$$\leq \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_2)\right| + \left|\frac{a}{2}h_0\frac{\partial^2 u}{\partial x^2}(\xi_1,nh_1)\right| \tag{7.36}$$

$$\leq \frac{M}{2}h_1 + \frac{aM}{2}h_0 \text{ for } a > 0 \tag{7.37}$$

$$\left|\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n\right| = \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_2) + \frac{a}{2}h_0\frac{\partial^2 u}{\partial x^2}(\xi_1,nh_1)\right| \tag{7.38}$$

$$\leq \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_2)\right| + \left|\frac{a}{2}h_0\frac{\partial^2 u}{\partial x^2}(\xi_1,nh_1)\right| \tag{7.39}$$

$$\leq \frac{M}{2}h_1 + \frac{aM}{2}h_0 \text{ for } a < 0 \tag{7.40}$$

By Def. 5, we conclude that

$$\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n = O(h_1) + O(h_0). \tag{7.41}$$

7.2.6.2 Specification

Figure 7.10 gives an excerpt of CIVL-C code for solving the one dimensional advection equation using the first order upwind scheme. Note the branching on the value of a at line 27. The branch indicates the direction of the flow, and the code chooses to do forward or backward differencing based on its sign. The assumption added to the path condition at line 8 precludes the trivial condition of $a = 0$ (i.e. the case when there is no flow).

7.2.6.3 Verification

The abstract function `u` is declared to have two continuous derivatives, and the first argument to the call of `u` at line 26 matches the spatial argument heuristic. Thus, the verifier adds the following assumptions:

```
$assume $forall {j=0..n-1} u((j+1)*h,iter*dt)==u(j*h,iter*dt)
       +$D[u,{x,1}](j*h,iter*dt)*h+$O(h*h);
$assume $forall {j=0..n-1} u((j-1)*h,iter*dt)==u(j*h,iter*dt)
       -$D[u,{x,1}](j*h,iter*dt)*h+$O(h*h);
```

```
$input int n; /* Number of points */
$input double h; /* Distance between points */
$input double dt; /* Size of a time step */
$input double a; /* Constant for wave velocity */
$abstract $contin(2) $real u($real x, $real t);
$assume h > 0;
$assume dt > 0;
$assume a != 0;
double v[n];
double v_new[n];
int iter;

void upwindForward() {
  for (int i = 1; i < n-1; i++) {
    v_new[i] = v[i]-dt*a*(v[i+1]-v[i])/h;
  }
}

void upwindBackward() {
  for (int i = 1; i < n-1; i++) {
    v_new[i] = v[i]-dt*a*(v[i]-v[i-1])/h;
  }
}

void upwind() {
  $assume $forall {j=0 .. n-1} v[j] == u(j*h, iter*dt);
  if (a > 0)
    upwindBackward();
  else
    upwindForward();
  for (int i = 1; i < n-1; i++) {
    v[i] = v_new[i];
  }
  $assert($uniform{m=1..n-2} (u(m*h,(iter+1)*dt)-v[m])/dt- \
      ($D[u, {t, 1}](m*h, iter*dt)+a*$D[u,{x,1}](m*h, iter*dt)) \
      ==$O(dt)+$O(h));
}
```

Figure 7.10: Excerpt of annotated CIVL-C code for the first order upwind scheme.

The second argument to the call of `u` at line 26 match the time argument heuristic. This prompts the verifier to add the assumption:

```
$assume $forall {j=0..n-1} u(j*h,(iter+1)*dt)==u(j*h,iter*dt)
    +$D[u,{t,1}](j*h,iter*dt)*dt+$O(dt*dt);
```

For each iteration of the upwind scheme, the assertion will be checked twice; once for each result of the branch at line 27. The values in `v` will differ depending on whether they were computed using `upwindForward()` or `upwindBackward()`. However, just like in the manual mathematical analysis, the same set of assumptions provides enough information to the prover to check the assertion regardless of whether the forward or backward computation is used.

7.2.7 Upwind scheme, second order

The second order upwind scheme functions similarly to the first order scheme given in Sec. 7.2.6. In order to gain the additional accuracy, the second order upwind scheme uses an extra point in the opposite direction from the flow. The stencil in Fig. 7.11 describes the points used in the scheme.

7.2.7.1 Mathematical analysis

The discretized operator $\hat{L}$ for the second order upwind method takes a function $\hat{v} : \Delta \to \mathbb{R}$ and returns a function $\hat{L}[\hat{v}] : \Delta \to \mathbb{R}$. It is defined by

$$\hat{L}[\hat{v}] = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} + a\frac{3\hat{v}_i^n - 4\hat{v}_{i-1}^n + \hat{v}_{i-2}^n}{2h_0} \text{ for } a > 0 \tag{7.42}$$

$$\hat{L}[\hat{v}] = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} + a\frac{-\hat{v}_{i+2}^n + 4\hat{v}_{i+1}^n - 3\hat{v}_i^n}{2h_0} \text{ for } a < 0. \tag{7.43}$$

We claim that $\hat{L}$ is a scheme for L that is Δ-uniformly accurate of order $(2, 1)$. Given $i, n \in \mathbb{Z}$ and $h_0, h_1 > 0$, there exist $\xi_0, \xi_1, \xi_2, \xi_3 \in [(i-1)h_0]$ and $\xi_4 \in [(n -$

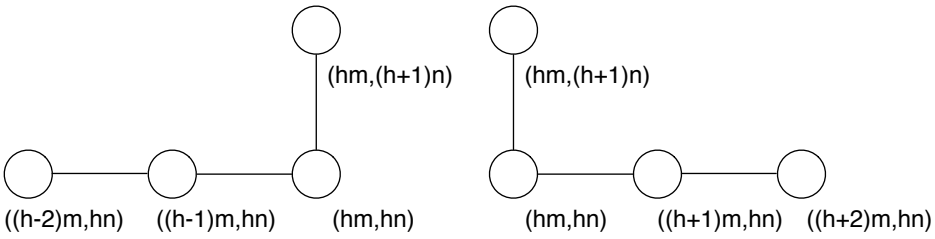


Figure 7.11: Stencils for the second order upwind scheme for linear advection when a is positive (left) or negative (right).

$1)h_1, (n+1)h_1]$ such that the following hold:

$$r_\Delta[u]_{i+1}^n = u(ih_0, nh_1) + h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_0, nh_1) \tag{7.44}$$

$$r_\Delta[u]_{i-1}^n = u(ih_0, nh_1) - h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_1, nh_1) \tag{7.45}$$

$$r_\Delta[u]_{i+2}^n = u(ih_0, nh_1) + 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_2, nh_1) \tag{7.46}$$

$$r_\Delta[u]_{i-2}^n = u(ih_0, nh_1) - 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_3, nh_1) \tag{7.47}$$

$$r_\Delta[u]_i^{n+1} = u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4). \tag{7.48}$$

Assume that on the domain of interest, the absolute value of the third derivative of u with respect to x is bounded by $M_0 > 0$, and that the absolute value of the second derivative of u with respect to t is bounded by $M_1 > 0$. Substituting the expansions

into Eq. 7.42,

$$\begin{aligned}
\hat{L}[r_\Delta[u]]_i^n &= \frac{u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) - u(ih_0, nh_1)}{h_1} \\
&\quad + \frac{a}{2h_0}(3u(ih_0, nh_1) - 4(u(ih_0, nh_1) - h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&\quad + \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_1, nh_1)) + (u(ih_0, nh_1) - 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&\quad + \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_3, nh_1))) \qquad (7.49) \\
&= \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) \\
&\quad + \frac{a}{2h_0}(2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) - \frac{2}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_1, nh_1) + \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_3, nh_1)) \qquad (7.50) \\
&= \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) \\
&\quad + a\frac{\partial u}{\partial x}(ih_0, nh_1) - \frac{a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_1, nh_1) + \frac{2a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_3, nh_1) \qquad (7.51)
\end{aligned}$$

Similarly, substituting the expansions into Eq. 7.43,

$$\begin{aligned}
\hat{L}[r_\Delta[u]]_i^n &= \frac{u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) - u(ih_0, nh_1)}{h_1} \\
&\quad + \frac{a}{2h_0}(-(u(ih_0, nh_1) + 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&\quad + \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_2, nh_1)) + 4(u(ih_0, nh_1) + h_0\frac{\partial u}{\partial x}(ih_0, nh_1) \\
&\quad + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_0, nh_1)) - 3u(ih_0, nh_1)) \qquad (7.52) \\
&= \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) \\
&\quad + \frac{a}{2h_0}(2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) - \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_2, nh_1) + \frac{2}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(\xi_0, nh_1)) \qquad (7.53) \\
&= \frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) \\
&\quad + a\frac{\partial u}{\partial x}(ih_0, nh_1) - \frac{2a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_2, nh_1) + \frac{a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_0, nh_1) \qquad (7.54)
\end{aligned}$$

Subtracting $L[u]$ from Eq. 7.51 and 7.54 yields

$$\begin{aligned}
\left|\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n\right| &= \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4) - \frac{a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_1,nh_1) + \frac{2a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_3,nh_1)\right| \\
&\leq \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4)\right| + \left|\frac{a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_1,nh_1)\right| \\
&\quad + \left|\frac{2a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_3,nh_1)\right| \qquad (7.55) \\
&\leq \frac{M_1}{2}h_1 + aM_0h_0^2 \text{ for } a > 0 \qquad (7.56) \\
\left|\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n\right| &= \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4) - \frac{2a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_2,nh_1) + \frac{a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_0,nh_1)\right| \\
&\leq \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4)\right| + \left|\frac{2a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_2,nh_1)\right| \\
&\quad + \left|\frac{a}{3}h_0^2\frac{\partial^3 u}{\partial x^3}(\xi_0,nh_1)\right| \qquad (7.57) \\
&\leq \frac{M_1}{2}h_1 + aM_0h_0^2 \text{ for } a < 0. \qquad (7.58)
\end{aligned}$$

By Def. 5, we conclude that

$$\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n = O(h_1) + O(h_0^2). \qquad (7.59)$$

7.2.7.2 Specification

The abstract function is declared in at line 5 to have three continuous derivatives. The assumption relating the array `v` to u is at line 28. The assertion at lines 36-38 checks that the code is Δ-uniformly first order accurate in time and second order accurate in space.

7.2.7.3 Verification

The abstract function `u` is declared to have three continuous derivatives, and the first argument to the call of `u` at line 28 matches the spatial argument heuristic. Thus, the verifier adds the following assumptions:

```
$input int n; /* Number of points */
$input double h; /* Distance between points */
$input double dt; /* Size of a time step */
$input double a; /* Constant for wave velocity */
$abstract $contin(3) $real u($real x, $real t);
$assume h > 0;
$assume dt > 0;
$assume a != 0;
double v[n];
double v_new[n];
int iter;

void upwindForward() {
  for (int i = 1; i < n-2; i++) {
    v_new[i] = v[i]-dt*a*(-v[i+2]+4*v[i+1]-3*v[i])/(2*h);
  }
  v_new[n-2] = v[n-2]-dt*a*(v[n-1]-v[n-2])/h;
}

void upwindBackward() {
  for (int i = 2; i < n-1; i++) {
    v_new[i] = v[i]-dt*a*(3*v[i]-4*v[i-1] + v[i-2])/(2*h);
  }
  v_new[1] = v[1]-dt*a*(v[1]-v[0])/h;
}

void upwind() {
  $assume $forall {j=0 .. n-1} v[j] == u(j*h, iter*dt);
  if (a > 0)
    upwindBackward();
  else
    upwindForward();
  for (int i = 1; i < n-1; i++) {
    v[i] = v_new[i];
  }
  $assert($uniform{m=2..n-3} (u(m*h,(iter+1)*dt)-v[m])/dt- \
      ($D[u,{t,1}](m*h,iter*dt)+a*$D[u,{x,1}](m*h,iter*dt)) \
      ==$O(dt)+$O(h*h));
}
```

Figure 7.12: Excerpt of annotated CIVL-C code for the second order upwind scheme.

```
$assume $forall {j=0..n-1} u((j+1)*h,iter*dt)==u(j*h,iter*dt)
       +$D[u,{x,1}](j*h,iter*dt)*h+$D[u,{x,2}](j*h,iter*dt)*h*h/2
       +$O(h*h*h);
$assume $forall {j=0..n-1} u((j-1)*h,iter*dt)==u(j*h,iter*dt)
       -$D[u,{x,1}](j*h,iter*dt)*h+$D[u,{x,2}](j*h,iter*dt)*h*h/2
       +$O(h*h*h);
```

The second argument to the call of `u` at line 28 match the time argument heuristic. This prompts the verifier to add the assumption:

```
$assume $forall {j=0..n-1} u(j*h,(iter+1)*dt)==u(j*h,iter*dt)
       +$D[u,{t,1}](j*h,iter*dt)*dt+$O(dt*dt);
```

As in the first order scheme, the assertion will be checked at each iteration for both branches of the code. CIVL is able to verify the second order upwind scheme for small configurations, but the queries rapidly become untenable for the current prover.

7.2.8 Upwind scheme, third order

The third order upwind scheme uses an additional point compared to the second order scheme, but this time the extra point is in the direction of the flow, as in the stencil in Fig. 7.13.

7.2.8.1 Mathematical analysis

The discretized operator $\hat{L}$ for the third order upwind method takes a function $\hat{v} : \Delta \to \mathbb{R}$ and returns a function $\hat{L}[\hat{v}] : \Delta \to \mathbb{R}$. It is defined by

$$\hat{L}[\hat{v}] = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} + a\frac{2\hat{v}_{i+1}^n + 3\hat{v}_i^n - 6\hat{v}_{i-1}^n + \hat{v}_{i-2}^n}{6h_0} \text{ for } a > 0 \tag{7.60}$$

$$\hat{L}[\hat{v}] = \frac{\hat{v}_i^{n+1} - \hat{v}_i^n}{h_1} + a\frac{-\hat{v}_{i+2}^n + 6\hat{v}_{i+1}^n - 3\hat{v}_i^n - 2\hat{v}_{i-1}^n}{6h_0} \text{ for } a < 0. \tag{7.61}$$

Proving the accuracy of the scheme requires Taylor expansions around the same

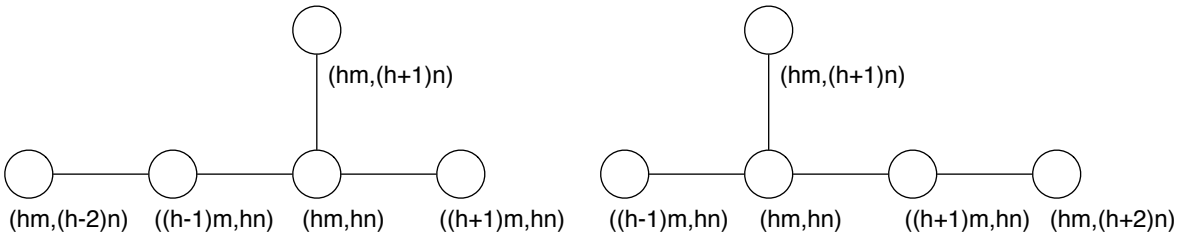


Figure 7.13: Stencils for the third order upwind scheme for linear advection when a is positive (left) or negative (right).

points as Sec. 7.2.7, but with an extra term in the expansion.

$$r_\Delta[u]_{i+1}^n = u(ih_0, nh_1) + h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) \tag{7.62}$$

$$r_\Delta[u]_{i-1}^n = u(ih_0, nh_1) - h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1) \tag{7.63}$$

$$r_\Delta[u]_{i+2}^n = u(ih_0, nh_1) + 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{2}{3}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_2, nh_1) \tag{7.64}$$

$$r_\Delta[u]_{i-2}^n = u(ih_0, nh_1) - 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) + \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{2}{3}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_3, nh_1) \tag{7.65}$$

$$r_\Delta[u]_{i}^{n+1} = u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4). \tag{7.66}$$

Assume that on the domain of interest, the absolute value of the fourth derivative of u with respect to x is bounded by $M_0 > 0$, and that the absolute value of the second derivative of u with respect to t is bounded by $M_1 > 0$. Substituting the expansions

into Eq. 7.60,

$$\begin{aligned}
\hat{L}[r_\Delta[u]]_i^n = &\frac{u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) - u(ih_0, nh_1)}{h_1} \\
&+ \frac{a}{6h_0}(2(u(ih_0, nh_1) + h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&+ \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1)) \\
&+ 3u(ih_0, nh_1) - 6(u(ih_0, nh_1) - h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&+ \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1)) \\
&+ u(ih_0, nh_1) - 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&+ \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{2}{3}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_3, nh_1)) \qquad (7.67)
\end{aligned}$$

$$\begin{aligned}
= &\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) + \frac{a}{6h_0}(6h_0\frac{\partial u}{\partial x}(ih_0, nh_1) \\
&+ \frac{1}{12}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) - \frac{1}{4}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1)) + \frac{2}{3}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_3, nh_1)) \qquad (7.68) \\
= &\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) + a\frac{\partial u}{\partial x}(ih_0, nh_1) \\
&+ \frac{a}{72}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) - \frac{a}{24}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1) + \frac{a}{9}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_3, nh_1) \qquad (7.69)
\end{aligned}$$

Similarly, substituting the expansions into Eq. 7.61,

$$\begin{aligned}
\hat{L}[r_\Delta[u]]_i^n = &\frac{u(ih_0, nh_1) + h_1\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1^2\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) - u(ih_0, nh_1)}{h_1} \\
&+ \frac{a}{6h_0}(-(u(ih_0, nh_1) + 2h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + 2h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&+ \frac{4}{3}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{2}{3}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_2, nh_1)) \\
&+ 6(u(ih_0, nh_1) + h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&+ \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1)) - 3u(ih_0, nh_1) \\
&- 2(u(ih_0, nh_1) - h_0\frac{\partial u}{\partial x}(ih_0, nh_1) + \frac{1}{2}h_0^2\frac{\partial^2 u}{\partial x^2}(ih_0, nh_1) \\
&+ \frac{1}{6}h_0^3\frac{\partial^3 u}{\partial x^3}(ih_0, nh_1) + \frac{1}{24}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1))) \qquad (7.70) \\
= &\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) + \frac{a}{6h_0}(6h_0\frac{\partial u}{\partial x}(ih_0, nh_1) \\
&- \frac{2}{3}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_2, nh_1) + \frac{1}{4}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) - \frac{1}{12}h_0^4\frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1) \qquad (7.71) \\
= &\frac{\partial u}{\partial t}(ih_0, nh_1) + \frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0, \xi_4) + a\frac{\partial u}{\partial x}(ih_0, nh_1) \\
&- \frac{a}{9}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_2, nh_1) + \frac{a}{24}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_0, nh_1) - \frac{a}{72}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_1, nh_1) \qquad (7.72)
\end{aligned}$$

Subtracting $L[u]$ from Eq. 7.69 and 7.72 yields

$$\left|\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n\right| = \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4) + \frac{a}{72}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_0,nh_1) - \frac{a}{24}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_1,nh_1)\right.$$
$$\left. + \frac{a}{9}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_3,nh_1)\right| \tag{7.73}$$
$$\leq \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4)\right| + \left|\frac{a}{72}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_0,nh_1)\right| + \left|\frac{a}{24}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_1,nh_1)\right|$$
$$+ \left|\frac{a}{9}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_3,nh_1)\right| \tag{7.74}$$
$$\leq \frac{M_1}{2}h_1 + \frac{aM_0}{6}h_0^2 \text{ for } a > 0 \tag{7.75}$$
$$\left|\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n\right| = \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4) - \frac{a}{9}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_2,nh_1) + \frac{a}{24}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_0,nh_1)\right.$$
$$\left. - \frac{a}{72}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_1,nh_1)\right| \tag{7.76}$$
$$\leq \left|\frac{1}{2}h_1\frac{\partial^2 u}{\partial t^2}(ih_0,\xi_4)\right| + \left|\frac{a}{9}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_2,nh_1)\right| + \left|\frac{a}{24}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_0,nh_1)\right|$$
$$+ \left|\frac{a}{72}h_0^3\frac{\partial^4 u}{\partial x^4}(\xi_1,nh_1)\right| \tag{7.77}$$
$$\leq \frac{M_1}{2}h_1 + \frac{aM_0}{6}h_0^2 \text{ for } a < 0. \tag{7.78}$$

By Def. 5, we conclude that

$$\hat{L}[r_\Delta[u]]_i^n - r_\Delta[L[u]]_i^n = O(h_1) + O(h_0^3). \tag{7.79}$$

7.2.8.2 Specification

The code is structured the same way as the first and second order upwind schemes. In this case, the abstract function is declared at line 5 to have four continuous derivatives. The assumption relating the array `v` to u is at line 28. The assertion at lines 36-38 checks that the code is Δ-uniformly first order accurate in time and third order accurate in space.

7.2.8.3 Verification

The abstract function `u` is declared to have four continuous derivatives, and the first argument to the call of `u` at line 28 matches the spatial argument heuristic. Thus,

```
$input int n; /* Number of points */
$input double h; /* Distance between points */
$input double dt; /* Size of a time step */
$input double a; /* Constant for wave velocity */
$abstract $contin(4) $real u($real x, $real t);
$assume h > 0;
$assume dt > 0;
$assume a != 0;
double v[n];
double v_new[n];
int iter;

void upwindForward() {
  for (int i = 1; i < n-2; i++) {
    v_new[i] = v[i]-dt*a*(-v[i+2]+6*v[i+1]-3*v[i]-2*v[i-1])/(6*h);
  }
  v_new[n-2] = v[n-2]-dt*a*(v[n-1]-v[n-2])/h;
}

void upwindBackward() {
  for (int i = 2; i < n-1; i++) {
    v_new[i] = v[i]-dt*a*(2*v[i+1]+3*v[i]-6*v[i-1] + v[i-2])/(6*h);
  }
  v_new[1] = v[1]-dt*a*(v[1]-v[0])/h;
}

void upwind() {
  $assume $forall {j=0 .. n-1} v[j] == u(j*h, iter*dt);
  if (a > 0)
    upwindBackward();
  else
    upwindForward();
  for (int i = 1; i < n-1; i++) {
    v[i] = v_new[i];
  }
  $assert($uniform{m=2..n-3} (u(m*h,(iter+1)*dt)-v[m])/dt- \
         ($D[u,{t,1}](m*h,iter*dt)+a*$D[u,{x,1}](m*h,iter*dt) \
         ==$O(dt)+$O(h*h*h));
}
```

Figure 7.14: Excerpt of annotated CIVL-C code for the third order upwind scheme.

the verifier adds the following assumptions:

```
$assume $forall {j=0..n-1} u((j+1)*h,iter*dt)==u(j*h,iter*dt)
        +$D[u,{x,1}](j*h,iter*dt)*h+$D[u,{x,2}](j*h,iter*dt)*h*h/2
        +$D[u,{x,3}](j*h,iter*dt)*h*h*h/6+$O(h*h*h*h);
$assume $forall {j=0..n-1} u((j-1)*h,iter*dt)==u(j*h,iter*dt)
        -$D[u,{x,1}](j*h,iter*dt)*h+$D[u,{x,2}](j*h,iter*dt)*h*h/2
        -$D[u,{x,3}](j*h,iter*dt)*h*h*h/6+$O(h*h*h*h);
```

The second argument to the call of `u` at line 28 match the time argument heuristic. This prompts the verifier to add the assumption:

```
$assume $forall {j=0..n-1} u(j*h,(iter+1)*dt)==u(j*h,iter*dt)
        +$D[u,{t,1}](j*h,iter*dt)*dt+$O(dt*dt);
```

Like the second order upwind scheme, CIVL can verify the third order scheme for very small configurations, but the complexity quickly surpasses the prover's ability to handle. Future work will research new techniques or other provers to mitigate this issue and improve scaling.

7.3 Scaling

Fig. 7.15 gives the results of some scaling experiments for verifying correct programs. For the Laplace example, the number of rows is held constant and the column dimension is scaled. In addition, CIVL was correctly unable to verify modified assertions that claimed too high of an order of accuracy.

While the scaling seems reasonable, the underlying theorem prover eventually becomes unable to verify the assertion. More research is needed to determine how to mitigate the theorem prover issues. Possibilities include reformulating the queries or applying some sort of invariant techniques to reduce the size of queries.

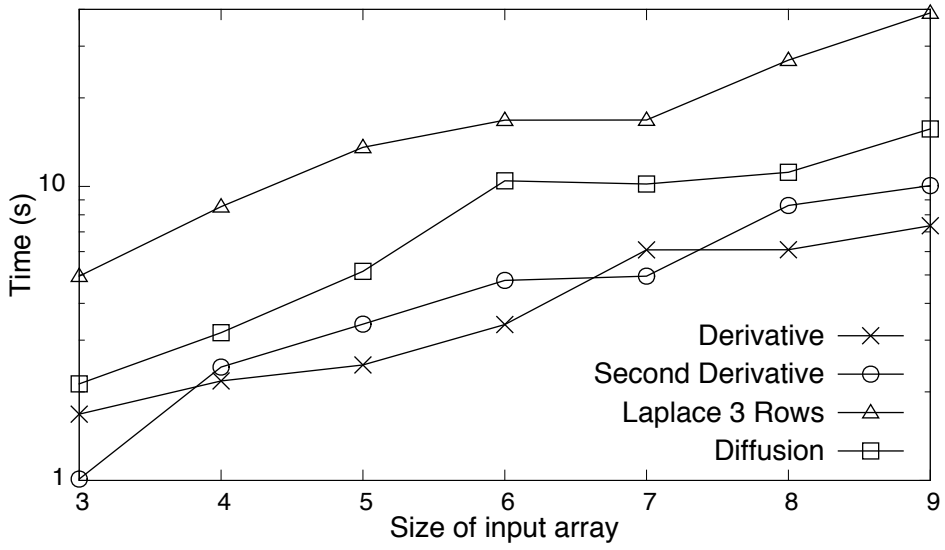


Figure 7.15: Graph of scaling experiments run on a 2.6 GHz Intel Core i7 Mac Mini; log. time axis

Chapter 8

CONCLUSION

We have shown a method, based on symbolic execution, by which code implementing a finite difference numerical approximation scheme can be specified and verified to meet order of accuracy requirements. Contrary to other approaches for accuracy verification, our approach requires only a few lines of code annotation. The lines of annotation use information that should be readily available to anybody implementing a numerical scheme.

The examples successfully verified with this approach are some commonly used finite difference approximations. They are applied to a variety of scientific and engineering problems. Our method can assist practitioners of these numerical methods with increasing their degree of confidence in their software.

Fig. 8.1 summarizes the results of applying symbolic differential accuracy verification to small configurations of each of the case studies. The "Valid calls" column gives the number of times that the symbolic algebra library was asked to check the validity of a query. The "Prover calls" column give the number of times that the symbolic algebra library's internal validity checker was unable to discharge the query, and it had to be passed to the underlying theorem prover. The "States" column shows the number of states instantiated during the verification process.

Symbolic differential accuracy verification in its current form is not a complete solution to the problem of verifying the order of accuracy of numerical codes. While we have demonstrated effectiveness for a variety of finite difference case studies on a regular mesh, we have not addressed irregular meshes or other types of numerical schemes. Nor have we considered issues such as stability. In addition, some finite difference

Name	Time (s)	Valid calls	Prover calls	Mem (MB)	States
Backward difference					
$num\_elements = 3$	1.6	22	2	257	89
$num\_elements = 5$	1.9	32	2	325	109
$num\_elements = 7$	2.1	42	2	325	129
$num\_elements = 9$	2.5	52	2	460	149
Central difference					
$num\_elements = 3$	1.8	23	1	325	86
$num\_elements = 5$	2.3	33	2	325	106
$num\_elements = 7$	3.7	43	2	460	126
$num\_elements = 9$	4.2	53	2	465	146
Second derivative					
$num\_elements = 3$	1.5	27	1	325	82
$num\_elements = 5$	2.2	39	2	325	102
$num\_elements = 7$	2.9	51	2	460	122
$num\_elements = 9$	3.7	63	2	460	142
Laplace 2d					
$rows = 3, cols = 3$	3.2	33	1	460	113
$rows = 3, cols = 4$	4.3	45	1	735	123
$rows = 3, cols = 5$	7.6	57	1	1277	133
$rows = 3, cols = 6$	11.1	69	1	1277	143
$rows = 3, cols = 7$	9.9	81	1	1594	153
$rows = 3, cols = 8$	8.7	93	1	1594	163
$rows = 3, cols = 9$	24.7	105	1	1594	173
Diffusion 1d					
$n = 3$	1.7	29	1	325	152
$n = 4$	2.8	39	2	460	182
$n = 5$	2.7	49	2	460	212
$n = 6$	3.3	59	2	460	242
$n = 7$	8.6	69	2	735	272
Upwind 1^{st} order					
$n = 3$	1.7	42	7	325	215
$n = 4$	3.1	58	9	325	265
$n = 5$	3.6	74	9	460	315
$n = 6$	3.7	90	9	460	365
$n = 7$	4.2	106	9	460	415
Upwind 2^{nd} order					
$n = 5$	4.3	70	7	460	298
Upwind 3^{rd} order					
$n = 5$	4.7	76	7	460	301

Figure 8.1: Summary of results of running CIVL on small configurations of the case studies.

problems are not amenable to our approach in its current form. For example, the Lax-Friedrichs scheme is an explicit numerical method for solving hyperbolic partial differential equations. It uses a simple stencil, and is first order accurate in time and second order accurate in space. On the surface, it appears similar in complexity to several of the examples presented here. However, the order of accuracy analysis requires additional information about the relationship between the size of the time step and space step.

Theorem proving is a challenge for this type of reasoning. The current implementation of SARL uses CVC3 as the underlying SMT solver. Some examples, such as the second and third order upwind schemes, become unmanageable by CVC3 at modest configurations. Future work will involve experimenting with different theorem provers, including CVC4 [5] and Z3 [52]. It might also be beneficial to use multiple theorem provers simultaneously since different provers might handle certain queries more efficiently.

BIBLIOGRAPHY

[1] American Institute of Aeronautics and Astronautics. Editorial policy statement on numerical and experimental accuracy. *Journal of Guidance, Control, and Dynamics*, 31(1):9–9, 2014.

[2] Saswat Anand, Corina Păsăreanu, and Willem Visser. JPF–SE: A symbolic execution extension to Java PathFinder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer Berlin / Heidelberg, 2007.

[3] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 127–141. Springer Berlin / Heidelberg, 2010.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg, 2011.

[6] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer Berlin Heidelberg, 2005.

[7] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, 2004.

[8] Martin Berz and Kyoko Makino. Verified integration of ODEs and flows with differential algebraic methods on Taylor models. *Reliable Computing*, 4:361–369, 1998.

[9] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.

[10] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. Wave equation numerical resolution: A comprehensive mechanized proof of a C program. *Journal of Automated Reasoning*, 50(4):423–456, 2013.

[11] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining coq and gappa for certifying floating-point programs. In *Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics*, Calculemus '09/MKM '09, pages 59–74, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, number NASA/CP-2010-216215 in NASA Conference Publication, pages 14–23, Washington D.C., USA, April 2010.

[13] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[14] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, SPIN'05, pages 2–23, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[17] CIVL: The Concurrency Intermediate Verification Language. http://vsl.cis.udel.edu/civl, accessed Feb. 7, 2014.

[18] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer Berlin / Heidelberg, 2004.

[19] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2:215–222, May 1976.

[20] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, USA, 2000. ACM.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[22] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[23] Patrick Cousot, Rashia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyser. In Mooly Sagiv, editor, *Proc. European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, April 2–10 2005. Springer Berlin Heidelberg.

[24] Dawson Engler Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex system programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[25] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 329–342, New York, NY, USA, 2013. ACM.

[26] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer Berlin / Heidelberg, 2009.

[27] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 157–166. IEEE Computer Society, 2006.

[28] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[29] Richard Fateman. Continuity and limits of programs. from web, Sep. 2003.

[30] Richard Fateman. High-level proofs of mathematical programs using automatic differentiation, simplification, and some common sense. In *ISSAC '03: Proceedings of the 2003 international symposium on Symbolic and algebraic computation*, pages 88–94, New York, NY, USA, 2003. ACM.

[31] Jean Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer Berlin Heidelberg, 2007.

[32] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 166–176, New York, NY, USA, 2012. ACM.

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[34] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[35] Laurent Granvilliers, Vladik Kreinovich, and Norbert Müller. Novel approaches to numerical software with result verification. In René Alt, Andreas Frommer, R. Kearfott, and Wolfram Luther, editors, *Numerical Software with Result Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 643–657. Springer Berlin / Heidelberg, 2004.

[36] E. Hairer and C. Lubich. The life-span of backward error analysis for numerical integrators. *Numerische Mathematik*, 76(4):441–462, June 1997.

[37] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In Thomas Ball and Sriram Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 624–624. Springer Berlin / Heidelberg, 2003.

[38] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comp.*, 14(4):783, 1993.

[39] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002.

[40] Gerard J. Holzmann. *The* Spin *Model Checker*. Addison-Wesley, Boston, 2004.

[41] F. Ivančić, Z. Yang, M.K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft software verification platform. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 301–306. Springer Berlin / Heidelberg, 2005.

[42] Journal of Fluid Engineering. Editorial policy statement on the control of numerical accuracy. http://journaltool.asme.org/templates/JFENumAccuracy.pdf. accessed Feb. 9, 2014.

[43] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer Berlin Heidelberg, 2003.

[44] David Kincaid and Ward Cheney. *Numerical Analysis*. Brooks/Cole, 1996.

[45] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[46] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing: Software Technology, Algorithms, Architectures, and Applications*, Advanced in Parallel Computing, pages 493–500. Elsevier, September 2003.

[47] Gary T. Leavens, Clyde Ruby, K. Rustan, M. Leino, Erik Poll, and Bart Jacobs. Jml (poster session): notations and tools supporting detailed design in java. In *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, OOPSLA '00, pages 105–106, New York, NY, USA, 2000. ACM.

[48] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. Gklee: Concolic verification and test generation for gpus. *SIGPLAN Not.*, 47(8):215–224, February 2012.

[49] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[50] Matthieu Martel. An overview of semantics for the validation of numerical programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 59–77. Springer Berlin / Heidelberg, 2005.

[51] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2006.

[52] Leonardo Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[53] W.L. Oberkampf and F.G. Blottner. Issues in computational fluid dynamics code verification and validation. *American Institute of Aeronautics and Astronautics Journal*, 36(5):687–695, 1998.

[54] Douglass Post and Timothy Trucano. Verification and validation in computational science and engineering. *Computing in Science & Engineering*, 6(5):8–9, sep–oct 2004.

[55] Sebastian Reich. Backward error analysis for numerical integrators. *SIAM J. Numer. Anal*, 36:1549–1570, 1996.

[56] Wilson Rivera-Gallego. Stability analysis of numerical boundary conditions in domain decomposition algorithms. *Applied Mathematics and Computation*, 137(2-3):375 – 385, 2003.

[57] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, 2003.

[58] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic software. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[59] C. J. Roy. Grid Convergence Error Analysis for Mixed-Order Numerical Schemes. *AIAA Journal*, 41:595–604, April 2003.

[60] Christopher J. Roy. Review of code and solution verification procedures for computational simulation. *J. Comput. Phys.*, 205:131–156, May 2005.

[61] Christopher J. Roy and Andrew J. Sinclair. On the generation of exact solutions for evaluating numerical schemes and estimating discretization error. *J. Comput. Phys.*, 228:1790–1802, March 2009.

[62] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of*

Software Engineering, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[63] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 270–284, Berlin, Heidelberg, 2012. Springer-Verlag.

[64] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In Lori L. Pollock and Mauro Pezzé, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17–20, 2006*, pages 157–168. ACM, 2006.

[65] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology*, 17(2):Article 10, 1–34, 2008.

[66] Stephen F. Siegel and Louis F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI User's Group Meeting, Proceedings*, volume 5205 of *LNCS*. Springer, 2008.

[67] Stephen F. Siegel and Timothy K. Zirkel. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science*, 5(4):395–426, 2011.

[68] Gustaf Söderlind and Lina Wang. Evaluating numerical ode/dae methods, algorithms and software. *J. Comput. Appl. Math.*, 185:244–260, January 2006.

[69] Ercilia Sousa and Ian Sobey. On the influence of numerical boundary conditions. *Applied Numerical Mathematics*, 41(2):325–344, May 2002.

[70] John C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. SIAM, January 2004.

[71] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 285–286, 2008.

[72] Dries Vanoverberghe and Frank Piessens. Theoretical aspects of compositional symbolic execution. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software*, FASE'11/ETAPS'11, pages 247–261, Berlin, Heidelberg, 2011. Springer-Verlag.

[73] Jean Vignes. Discrete Stochastic Arithmetic for Validating Results of Numerical Software. *Numerical Algorithms*, 37(1):377–390, December 2004.

[74] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003. 10.1023/A:1022920129859.

[75] R. F. Warming and B. J. Hyett. The modified equation approach to the stability and accuracy analysis of finite-difference methods. *Journal of Computational Physics*, 14(2):159 – 179, 1974.

[76] J. H. Wilkinson. Error analysis of floating-point computation. *Numerische Mathematik*, 2(1):319–340, Dec. 1960.

[77] Wolfram Research, Inc. Wolfram Mathematica: Technical Computing Software. http://www.wolfram.com/products/mathematica/index.html. accessed Feb. 9, 2014.

[78] O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptive procedure for practical engineering analysis. *International Journal for Numerical Methods in Engineering*, 24(2):337–357, 1987.

[79] Timothy K. Zirkel, Stephen F. Siegel, and Timothy McClory. Automated verification of chapel programs using model checking and symbolic execution. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 198–212. Springer Berlin Heidelberg, 2013.

Appendix A

PROOF OF BOUNDS FOR $\sin'(X)$

Proof of upper bound. We take the Taylor expansion of $\sin(x \pm h)$.

$$\sin(x+h) = \sin(x) + h\cos(x) - \frac{h^2}{2!}\sin(x) + \sum_{i=3}^{\infty} \frac{h^i}{i!}\frac{\partial^i}{\partial x^i}\sin(x) \tag{A.1}$$

$$\sin(x-h) = \sin(x) - h\cos(x) - \frac{h^2}{2!}\sin(x) + \sum_{i=3}^{\infty} \frac{(-h)^i}{i!}\frac{\partial^i}{\partial x^i}\sin(x) \tag{A.2}$$

Substituting equations A.1 and A.2 into the left hand side of equation 2.4 and simplifying yields

$$\phi(x,h) = \left|\cos(x) - \frac{\sum_{i=0}^{\infty} \frac{2h^{2i+1}}{(2i+1)!}\frac{\partial^{2i+1}}{\partial x^{2i+1}}\sin(x)}{2h}\right| \tag{A.3}$$

$$= \left|\sum_{i=1}^{\infty} \frac{h^{2i}}{(2i+1)!}\frac{\partial^{2i+1}}{\partial x^{2i+1}}\sin(x)\right| \tag{A.4}$$

$$= \left|\sum_{i=1}^{\infty} (-1)^{i+1}\frac{h^{2i}}{(2i+1)!}\cos(x)\right|. \tag{A.5}$$

All $\sin(x)$ terms in the expansions are cancelled due to the subtraction, so a $\cos(x)$ can be factored out. Take $\epsilon = 1$. Since $h < 1$, the remaining numerators are less than h^2. Factoring out an h^2 results in all numerators in the remaining sum being less than or equal to one. The resulting sum is bounded by

$$\sum_{k=0}^{\infty}\frac{1}{k!} - \left(\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{4!}\right) = e - \left(\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{4!}\right) < 0.2$$

and therefore we get

$$\phi(x,h) = \left| \sum_{i=2}^{\infty} (-1)^i \frac{h^{2i}}{(2i-1)!} \cos(x) \right| \tag{A.6}$$

$$= \left| \cos(x) \left(\sum_{i=2}^{\infty} (-1)^i \frac{h^{2i}}{(2i-1)!} \right) \right| \tag{A.7}$$

$$< \left| 0.2 \cos(x) h^2 \right| \tag{A.8}$$

$$\leq 0.2h^2. \tag{A.9}$$

This shows that $C = 0.2$ and $\epsilon = 1$ satisfy the condition required by Definition 3. Thus we see that g is a uniformly second order accurate approximation of f on $\mathbb{R}$.

□

Proof of lower bound at $x = \pi$. Start with equation A.5 and consider the point $x = \pi$:

$$\phi(x,h) = \left| \sum_{i=1}^{\infty} (-1)^{i+1} \frac{h^{2i}}{(2i+1)!} \cos(\pi) \right| \tag{A.10}$$

$$= \left| \sum_{i=1}^{\infty} (-1)^{i} \frac{h^{2i}}{(2i+1)!} \right| \tag{A.11}$$

$$= \left| \sum_{i\geq 2, i \text{ even}}^{\infty} \frac{h^{2i}}{(2i+1)!} - \sum_{i\geq 1, i \text{ odd}}^{\infty} \frac{h^{2i}}{(2i+1)!} \right| \tag{A.12}$$

$$= \left| \sum_{i\geq 1, i \text{ odd}}^{\infty} \frac{h^{2i}}{(2i+1)!-} - \sum_{i\geq 2, i \text{ even}}^{\infty} \frac{h^{2i}}{(2i+1)!} \right| \tag{A.13}$$

$$= \left| \sum_{i\geq 1, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} - \sum_{i\geq 2, i \text{ even}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 \tag{A.14}$$

$$= \left| \frac{1}{3!} + \sum_{i\geq 3, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} - \sum_{i\geq 2, i \text{ even}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 \tag{A.15}$$

$$\geq \left| \frac{1}{3!} + \sum_{i\geq 3, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 - \left| \sum_{i\geq 2, i \text{ even}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 \tag{A.16}$$

$$\geq \left| \frac{1}{3!} + \sum_{i\geq 3, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 - \left| \sum_{i\geq 2, i \text{ even}}^{\infty} \frac{1}{(2i+1)!} \right| h^2 \tag{A.17}$$

$$\geq \left| \frac{1}{3!} + \sum_{i\geq 3, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 - \left| \sum_{i\geq 2}^{\infty} \frac{1}{(2i+1)!} \right| h^2 \tag{A.18}$$

$$= \left| \frac{1}{3!} + \sum_{i\geq 3, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 - \left| \sum_{i\geq 0}^{\infty} \frac{1}{(2i+1)!} - \frac{1}{1!} - \frac{1}{3!} \right| h^2 \tag{A.19}$$

$$\geq \left| \frac{1}{3!} + \sum_{i\geq 3, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 - \left| \sinh 1 - \frac{1}{1!} - \frac{1}{3!} \right| h^2 \tag{A.20}$$

$$\geq \left| \frac{1}{3!} + \sum_{i\geq 3, i \text{ odd}}^{\infty} \frac{h^{2i-2}}{(2i+1)!} \right| h^2 - 0.0087 h^2 \tag{A.21}$$

$$\geq \left(\frac{1}{3!} - 0.0087 \right) h^2 \tag{A.22}$$

$$> 0.15 h^2. \tag{A.23}$$

Thus the error is bounded from below by $0.15h^2$. □

Appendix B

CIVL TOOL OPTIONS

```
CIVL v0.9 of 2014-03-14 -- http://vsl.cis.udel.edu/civl
Usage: civl <command> <options> filename ...
Commands:
  verify : verify program filename
  run : run program filename
  help : print this message
  replay : replay trace for program filename
  parse : show result of preprocessing and parsing filename
  preprocess : show result of preprocessing filename
Options:
  -debug or -debug=BOOLEAN (default: false)
    debug mode: print very detailed information
  -echo or -echo=BOOLEAN (default: false)
    print the command line
  -enablePrintf or -enablePrintf=BOOLEAN (default: true)
    enable printf function
  -errorBound=INTEGER (default: 1)
    stop after finding this many errors
  -guided or -guided=BOOLEAN
    user guided simulation; applies only to run, ignored
    for all other commands
  -id=INTEGER (default: 0)
    ID number of trace to replay
  -inputKEY=VALUE
    initialize input variable KEY to VALUE
  -maxdepth=INTEGER (default: 2147483647)
    bound on search depth
  -min or -min=BOOLEAN (default: false)
    search for minimal counterexample
  -mpi or -mpi=BOOLEAN (default: false)
    MPI mode
  -por=STRING (default: std)
    partial order reduction (por) choices:
    std (standard por) or scp1 (scoped por 1) or scp1 (scoped por 2)
```

```
-random or -random=BOOLEAN
  select enabled transitions randomly; default for run,
  ignored for all other commands
-saveStates or -saveStates=BOOLEAN (default: true)
  save states during depth-first search
-seed=STRING
  set the random seed; applies only to run
-showAmpleSet or -showAmpleSet=BOOLEAN (default: false)
  print the ample set of each state
-showModel or -showModel=BOOLEAN (default: false)
  print the model
-showProverQueries or -showProverQueries=BOOLEAN (default: false)
  print theorem prover queries only
-showQueries or -showQueries=BOOLEAN (default: false)
  print all queries
-showSavedStates or -showSavedStates=BOOLEAN (default: false)
  print saved states only
-showStates or -showStates=BOOLEAN (default: false)
  print all states
-showTransitions or -showTransitions=BOOLEAN (default: false)
  print transitions
-simplify or -simplify=BOOLEAN (default: true)
  simplify states?
-solve or -solve=BOOLEAN (default: false)
  try to solve for concrete counterexample
-sysIncludePath=STRING
  set the system include path
-trace=STRING
  filename of trace to replay
-userIncludePath=STRING
  set the user include path
-verbose or -verbose=BOOLEAN (default: false)
  verbose mode
```

Appendix C

PROVER QUERY FOR BACKWARD FINITE DIFFERENCE ASSERTION

```
CVC3 assumptions 1: TRUE
(LET v_0 = rho(X_s0v2),
v_36 = X_s0v4[1],
v_23 = (-1 * v_36),
v_35 = (v_0 + v_23),
v_3 = (2 * X_s0v2),
v_2 = rho(v_3),
v_38 = X_s0v4[2],
v_20 = (-1 * v_38),
v_37 = (v_2 + v_20),
v_12 = ((1 * rhox1(0)) * X_s0v2),
v_1 = BIG_O((X_s0v2 ^ 2)),
v_11 = (-1 * v_0),
v_6 = rho(0),
v_39 = (v_12 + v_1 + v_11 + v_6),
v_14 = ((1 * rhox1(X_s0v2)) * X_s0v2),
v_9 = (-1 * v_2),
v_40 = (v_14 + v_1 + v_0 + v_9),
v_10 = ((1 * rhox1(v_3)) * X_s0v2),
v_4 = (3 * X_s0v2),
v_5 = rho(v_4),
v_7 = (-1 * v_5),
v_41 = (v_10 + v_1 + v_2 + v_7),
v_8 = ((1 * rhox1(v_4)) * X_s0v2),
v_17 = rho((4 * X_s0v2)),
v_29 = (-1 * v_17),
v_42 = (v_8 + v_1 + v_5 + v_29),
v_13 = (-1 * v_6),
v_22 = X_s0v4[0],
v_21 = X_s0v4[3],
v_16 = (-1 * v_8),
v_34 = (v_16 + v_1 + v_9 + v_5),
v_15 = (-1 * v_10),
v_33 = (v_15 + v_1 + v_11 + v_2),
v_18 = (-1 * v_12),
```

```
v_19 = (-1 * v_14),
v_43 = rho((-1 * X_s0v2)),
v_30 = (-1 * v_43),
v_31 = (v_18 + v_1 + v_30 + v_6),
v_32 = (v_19 + v_1 + v_0 + v_13),
v_24 = (2/3 * v_21),
v_25 = (1/3 * v_22),
v_26 = (1/3 * v_21),
v_27 = (2/3 * v_22),
v_28 = (-1/3 * v_22)
IN ((0 = v_35) AND (0 = v_37) AND (0 = v_39) AND (0 = v_40) AND (0 = v_41) AND (
0 = v_42) AND (0 = ((-1 * ((1 * BIG_O(X_s0v2)) * X_s0v2)) + v_1)) AND (0 = (v_13
 + v_22)) AND (0 = (v_7 + v_21)) AND (0 <= v_34) AND (0 <= v_33) AND (0 <= (v_18
 + v_1 + v_0 + v_13)) AND (0 <= (v_19 + v_1 + v_11 + v_2)) AND (0 <= (v_15 + v_1
 + v_9 + v_5)) AND (0 <= (v_16 + v_1 + v_7 + v_17)) AND (0 <= v_31) AND (0 <= v_
32) AND (0 < X_s0v2) AND (0 = (v_20 + v_24 + v_25)) AND (0 = (v_23 + v_26 + v_27
)) AND (0 = (v_9 + v_24 + v_25)) AND (0 = (v_11 + v_26 + v_27)) AND (0 = (v_18 +
 v_1 + v_26 + v_28)) AND (0 = (v_16 + v_1 + v_26 + v_28)) AND (0 = (v_15 + v_1 +
 v_26 + v_28)) AND (0 = (v_19 + v_1 + v_26 + v_28)) AND (0 = (v_29 + (4/3 * v_21
) + v_28)) AND (0 = (v_30 + (-1/3 * v_21) + (4/3 * v_22))) AND (0 = v_31) AND (0
 = v_32) AND (0 = v_33) AND (0 = v_34) AND (0 <= (v_8 + v_1 + v_2 + v_7)) AND (0
 <= (v_10 + v_1 + v_0 + v_9)) AND (0 <= v_35) AND (0 <= (v_11 + v_36)) AND (0 <=
 v_37) AND (0 <= (v_9 + v_38)) AND (0 <= v_39) AND (0 <= v_40) AND (0 <= v_41) A
ND (0 <= v_42) AND (0 <= (v_12 + v_1 + v_43 + v_13)) AND (0 <= (v_14 + v_1 + v_1
1 + v_6))))
CVC3 predicate   1: (LET v_3 = X_s0v4[1],
v_1 = (-1 * v_3),
v_4 = X_s0v4[2],
v_0 = ((1 * BIG_O(X_s0v2)) * X_s0v2),
v_2 = X_s0v4[0]
IN ((0 = (v_0 + ((1 * rhox1(0)) * X_s0v2) + v_1 + v_2)) AND (0 = (v_0 + ((1 * rh
ox1(X_s0v2)) * X_s0v2) + v_1 + v_2)) AND (0 = (v_0 + ((1 * rhox1((2 * X_s0v2)))
* X_s0v2) + v_3 + (-1 * v_4))) AND (0 = (v_0 + ((1 * rhox1((3 * X_s0v2))) * X_s0
v2) + v_4 + (-1 * X_s0v4[3])))))
CVC3 result      1: VALID
```