2016

ON THE APPLICATIONS OF INTERACTIVE THEOREM PROVING IN COMPUTATIONAL SCIENCES AND ENGINEERING

Amer Tahat
Michigan Technological University, atahat@mtu.edu

ON THE APPLICATIONS OF INTERACTIVE THEOREM PROVING IN COMPUTATIONAL SCIENCES AND ENGINEERING

By

Amer Tahat

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computational Science and Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2016

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computational Science and Engineering.

Department of Computer Science

Dissertation Advisor : *Dr. Ali Ebnenasir*

Committee Member : *Dr. César Muñoz*

Committee Member : *Dr. Jean Mayo*

Committee Member : *Dr. Charles Wallace*

Department Chair : *Dr. Min Song*

To my mother Nafal, to my father Naser, to my wife Niveen and to my son Mohammad I could not have done this work without your love, prayers, support, and patience. I owe you everything.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Dr. Ali Ebnenasir without whom this dissertation could not have been written and to whom I am very grateful.

Also, I would like to extend my gratitude to Dr. César Muñoz from NASA Langley formal methods group, for introducing me to the world of interactive theorem proving, for his priceless explanations, and valuable opinions that have inspired my work and enhanced its quality astronomically. There are no sufficient words in my dictionary to thank you Dr.Muñoz; indeed, I owe you a lot.

Dr. Jean Mayo and Dr. Charles Wallace, thanks so much, for your continued collaboration and support. I will always be very proud of having you in my graduate committee. I am really grateful to you.

Furthermore,I would like to extend my sincere acknowledgments to Dr. John Rushby, Dr. Natrajan Shankar, Dr. Sam Owre from SRI international, Dr. Anthony Narkawicz from NASA Langley, Dr. Naser al Araje Chair of School of Technology at Michigan Technological University, and Dr. Ossama Abdelkhalik from Mechanical Engineering (space systems research group) at Michigan Technological University, for encouraging me a lot, and for their interesting discussions, valued inputs on different parts and stages of my research. Your words have been very meaningful to me.

Furthermore, I am also thankful to all people who patiently gave me part of their precious times and provided me with their valuable comments on different parts of this work.

Thank you so much.

Abstract

Interactive Theorem Proving (ITP) is one of the most rigorous methods used in formal verification of computing systems. While ITP provides a high level of confidence in the correctness of the system under verification, it suffers from a steep learning curve and the laborious nature of interaction with a theorem prover. As such, it is desirable to investigate whether ITP can be used in unexplored (but high-impact) domains where other verification methods fail to deliver. To this end, the focus of this dissertation is on two important domains, namely design of parameterized self-stabilizing systems, and mechanical verification of numerical approximations for Riemann integration. Self-stabilization is an important property of distributed systems that enables recovery from any system configuration/state. There are important applications for self-stabilization in network protocols, game theory, socioeconomic systems, multi-agent systems and robust data structures. Most existing techniques for the design of self-stabilization rely on a 'manual design and after-the-fact verification' method. In a paradigm shift, we present a novel hybrid method of 'synthesize in small scale and generalize' where we combine the power of a finite-state synthesizer with theorem proving. We have used our method for the design of network protocols that are self-stabilizing irrespective of the number of network nodes (i.e., parameterized protocols). The second domain of application of ITP that we are investigating concentrates on formal verification of the numerical propositions of Riemann integral in formal proofs. This is a high-impact problem as Riemann Integral is considered one of the most indispensable tools of modern calculus. That has significant applications in the development of mission-critical systems in many Engineering fields that require rigorous computations such as aeronautics, space mechanics, and electrodynamics. Our contribution to this problem is three fold: first, we formally specify and verify the fundamental Riemann Integral inclusion theorem in interval arithmetic; second, we propose a general method to verify numerical propositions on Riemann Integral for an uncountably infinte class of integrable functions; third, we develop a set of practical automatic proof strategies based on formally verified theorems. The contributions of Part II have become part of the ultra-reliable NASA PVS standard library.

Chapter 1

Introduction

1.1 Preface

Interactive Theorem Proving (ITP) represents the most rigorous formal verification technique for mission-critical computing systems [1]. Nonetheless, ITP is known to be time consuming given the burdensome nature of interaction with a theorem prover. As such, it is desirable to investigate whether ITP can be combined with more automatic techniques in unexplored (but high-impact) domains where other verification methods fail to deliver. To this end, the focus of this dissertation is on two important domains, namely design of parameterized self-stabilizing systems, and mechanical verification of numerical approximations for Riemann integration. A solution for these two problems is highly desirable in the design and verification of many mission-critical systems (e.g., air traffic management systems). Specifically, the interaction with the physical environment implies complex models of continuous and discrete mathematical nature [2]. For instance, the design of a safe algorithm to avoid collision between unmanned aircrafts includes the verification of non-linear arithmetic propositions on real-valued continuous functions, which is a well-known problem that faces arduous complications in automated reasoning systems [3, 4, 2, 5, 6]. In addition to this, in the absence of a human pilot in an unmanned aircraft/spaceship there will be vital enquiries to design ultra-reliable distributed protocols [7, 8, 9, 10, 11]. Self-Stabilization (SS) is a highly desirable property of these protocols [9]. However, a chief challenge associated with their formal designs is the complexity of verifying the correctness of their safety properties [9]. For example, the design of parameterized self-stabilizing protocols algorithmically is known to be an open problem. Moreover, the problem of adding convergence to finite state automata is NP-hard (in the size of state space) [12, 13, 14]. Most existing techniques for the design of self-stabilization rely on a 'manual design and after-the-fact verification' methods which are limited to specific heuristics. Automatic finite-state synthesizers do exist [13, 15, 16, 17] but they are efficient only in small scopes given the complexity of adding convergence problem

Table 1.1: Total size of PVS theories developed in the dissertation.

	Total number of lemmas	Total size PVS + proof scripts
Part I	319	4473
Part II	99	2928
Total	418	7401

(i.e., small finite number of processes) [15, 16]. This dissertation includes two parts. First, we propose a hybrid method for the synthesis and verification of parameterized self-stabilizing protocols, under the concept of synthesize in small scale then generalize. The first step of this method is done by automatic synthesizer [15, 16, 17]. By contrast, the generalization step is done by means of interactive prototype verification system (PVS). The mechanical convergence proofs we provide are based on symbolic mathematical reasoning. Thus there are no restrictions on the number of processes or their variables. In the second part of the dissertation we provide a novel automatic strategy to prove numerical propositions on Riemann Integral within a formal proof. A solution for this problem is demanding and very challenging (e.g., for the design of unmanned aircraft system) [6, 4]. Our design for this strategy required a proof for the fundamental Riemann Integral Inclusion Theorem for a specific type of integrable functions. The strategy combines several theories of real analysis and interval analysis in an automatic fashion. This yields a significant reduction in the human exertions that are required for the mechanical but not automatic verifications, saving the users thousands of proof steps. Table 1.1 presents a summary of the complexity of the PVS theories we developed in this dissertation in terms of the total sizes of the PVS files and their proof scripts.

Organization of the first part of the dissertation. Section 1.2 provides a brief overview of PVS. In Chapter 2 we introduce the first problem, backgound and related work. Chapter 3 provides basic concepts of protocols and presents their formal specifications in PVS. Then, Chapter 4 formally presents the problem of adding convergence to protocols. Chapter 5 present the specification and verification of our approach of the design and verification of parameterized weakly self-stabilizing protocols in PVS. Chapter 6 demonstrates reusability and generalizability for strongly self-stabilizing protocol of in the context of sorting protocol on rings and chains. Chapter 7 discusses the significance of this research,concluding remarks, and presents future extensions of part I. **Organization of the second part of the dissertation**. Chapter 8 provides some basic concepts and results from existing work. Chapter 9 presents an abstract algorithm for the approximation of Riemann Integral with its proof of soundness. Chapter 10 discusses the usability of RiemannSum R2I and its soundness statement in formal proofs of numerical propositions automatically,

and presents some case studies. Finally, Chapter 11 makes concluding remarks and presents possible research expansions of the results of Part II.

1.2 The Prototype Verification System (PVS)

Since in this disertation we use the Prototype Verification System (PVS) [18, 19] as the theorem prover of choice, we provide a brief introduction to PVS in this section. PVS is an environment for formal specification and verification that includes a specification language, a type checker and a theorem prover (that enables automated deduction). The basic unit of abstraction in PVS is a theory whose parameters may include constants, types or theory instances. Each THEORY contains axioms, definitions, assumptions and theorems. The following specification represents an example template of a PVS theory.

```
Example: THEORY

  BEGIN
     // The definitions, axioms, assumptions and theorems come here
  END Example
```

PVS also contains built-in libraries of theories whose theorems can be imported in specifications. A PVS expression can be considered as a predicate logic formula that may include the usual universal/existential quantifiers, arithmetic and logical operators, lambda abstraction and functions. In general, the PVS specification language is based on typed higher-order logic that supports some built-in and uninterpreted user-defined types. For example, we specify the basic concepts of state, variables and their domain as uninterpreted types state: Type+, Variable: Type+, Dom: Type+, where '+' denotes the non-emptiness of the declared type. (In PVS, "*userDefinedType* : TYPE+ = []" declares the type *userDefinedType*.) The PVS language also enables us to define constrained types (e.g., prime numbers) using predicate subtypes and dependent types. During type checking, these constrained types may cause proof obligations (e.g., proving non-emptiness) that are called Type-Correctness Conditions (TCCs). The PVS theorem prover includes two sets of automated reasoning tools, namely the primitive inference procedures and utilities. The primitive (inference) procedures include rules for propositional and quantified expressions, term rewriting, induction, simplification and data and predicate abstraction. There are also utility tools integrated with the primitive procedures to improve the efficiency of automated reasoning. These tools include a symbolic model checker, a Satisfiability Modulo Theory (SMT) solver, a random testing tool and a tool for evaluating ground expressions, called PVSio. The users should use the aforementioned machinery in a semi-automatic fashion to prove theorems. The users can create scripts that combine inference procedures, called *proof strategies*. The proofs are saved as scripts that can be edited by users for reuse and greater efficiency.

Part I

Parameterized Self-Stabilizing Protocols

Chapter 2

Introduction

2.1 Preface

In this part we present a novel hybrid method for verification and synthesis of parameterized self-stabilizing protocols where algorithmic design and mechanical verification techniques/tools are used hand-in-hand. The core idea behind the proposed method includes the automated synthesis of self-stabilizing protocols in a limited scope (i.e., fixed number of processes) and the use of theorem proving methods for the generalization of the solutions produced by the synthesizer. Specifically, we use the Prototype Verification System (PVS) to mechanically verify a synthesis algorithm for automated generation of weakly stabilizing protocols. Then, we reuse the proof of correctness of the synthesis algorithm to establish the correctness of the generalized versions of synthesized protocols for an arbitrary number of processes. We demonstrate the proposed approach in the context of an agreement and a graph coloring protocol on the ring topology [1]. We also mechanically verify a strongly stabilizing sorting[2] protocol on unidirectional rings and chains using the idea of convergence stairs [3].

2.2 Background and Related Work

Self-stabilization is an important property of dependable distributed systems as it guarantees *convergence* in the presence of transient faults. That is, from *any* state/-configuration, a Self-Stabilizing (SS) system recovers to a set of legitimate states

[1] Published in [20]
[2] In preparation for submission to the Journal of Formal Methods in System Design
[3] Section 6.1.1

(a.k.a. *invariant*) in a finite number of steps. Moreover, from its invariant, the executions of an SS system satisfy its specifications and remain in the invariant; i.e., *closure*. Nonetheless, design and verification of convergence are difficult tasks [21, 22] in part due to the requirements of (i) recovery from arbitrary states; (ii) recovery under distribution constraints, where processes can read/write only the state of their neighboring processes (a.k.a. their *locality*), and (iii) the non-interference of convergence with closure. Methods for algorithmic design of convergence [23, 24, 25, 26] can generate only the protocols that are correct up to a limited number of processes and small domains for variables. Thus, it is desirable to devise methods that enable automated design of parameterized SS systems, where a *parameterized* system includes several families of symmetric processes that have a similar code up to variable renaming. This paper presents a novel method based on the philosophy of *synthesize in small scale and generalize*, where we exploit (1) algorithmic methods for the design of small SS systems, and (2) theorem proving techniques for generalizing small solutions. The proposed method has important applications in both hardware [27] and software [28] networked systems, be it a network-on-chip system or the Internet. Numerous approaches exist for mechanical verification of self-stabilizing systems most of which focus on synthesis and verification of specific protocols. For example, Qadeer and Shankar [29] present a mechanical proof of Dijkstra's token ring protocol [21] in the Prototype Verification System (PVS) [19]. Kulkarni *et al.* [30] use PVS to mechanically prove the correctness of Dijkstra's token ring protocol in a component-based fashion. Prasetya [31] mechanically proves the correctness of a self-stabilizing routing protocol in the HOL theorem prover [32]. Tsuchiya *et al.* [33] use symbolic model checking to verify several protocols such as mutual exclusion and leader election. Kulkarni *et al.* [34, 35] mechanically prove (in PVS) the correctness of algorithms for automated addition of fault tolerance; nonetheless, such algorithms are not tuned for the design of convergence. Most existing automated techniques [36, 37, 38, 25] for the design of fault tolerance enable the synthesis of non-parametric fault-tolerant systems. For example, Kulkarni and Arora [37] present a family of algorithms for automated design of fault tolerance in non-parametric systems, but they do not explicitly address self-stabilization. Abujarad and Kulkarni [24] present a method for algorithmic design of self-stabilization in locally-correctable protocols, where the local recovery of all processes ensures the global recovery of the entire distributed system. Farahat and Ebnenasir [26, 25] present algorithms for the design of self-stabilization in non-locally correctable systems. Jacobs and Bloem [39] show that, in general, synthesis of parameterized systems from temporal logic specifications is undecidable. They also present a semi-decision procedure for the synthesis of a specific class of parameterized systems in the absence of faults. To summarize, there are two important challenges: (1) most existing methods for mechanical verification of self-stabilization are problem specific, which provide little room for reuse of verification efforts, and

(2) existing synthesis methods do not directly address the design of SS parameterized systems.

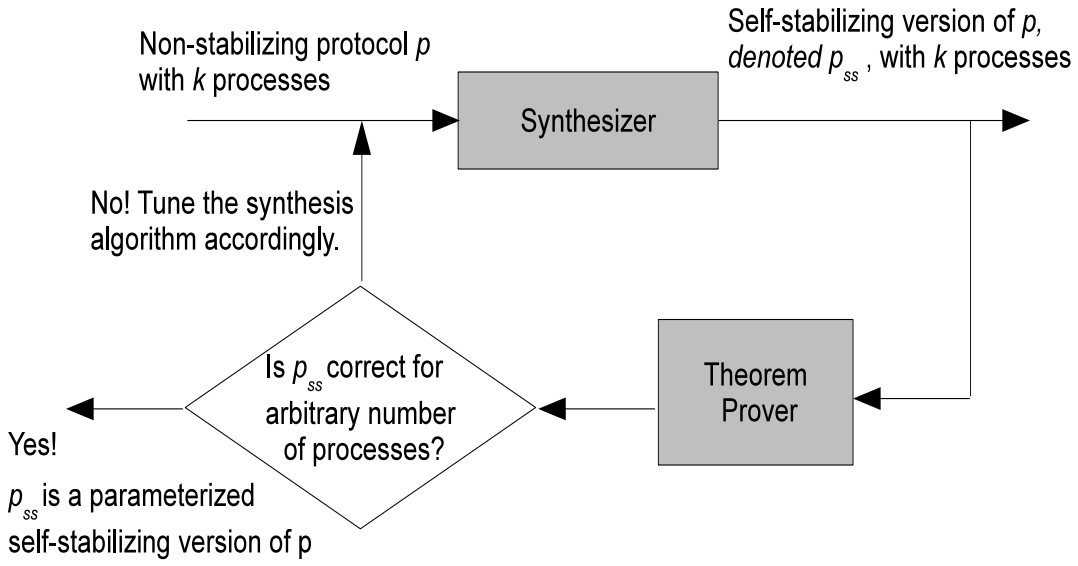

Figure 2.1: A Hybrid method for the synthesis of parameterized self-stabilizing protocols.

The **contributions** of this part are two-fold: a hybrid method (Figure 2.1) for the synthesis of parameterized self-stabilizing systems and a reusable PVS theory for mechanical verification of self-stabilization. The proposed method includes a synthesis step and a theorem proving step. In [26, 25, 14, 40] the authors enable the synthesis step where they take a non-stabilizing protocol and generate a self-stabilizing version thereof that is correct by construction up to a certain number of processes. This paper investigates the second step where we use the theorem prover PVS to prove (or disprove) the correctness of the synthesized protocol for an arbitrary number of processes; i.e., *generalize* the synthesized protocol. The synthesis algorithms in [26, 25, 14, 40] incorporate weak and strong convergence in existing network protocols; i.e., adding convergence. *Weak* (respectively, *Strong*) convergence requires that from every state there exists an execution that (respectively, every execution) reaches an invariant state in finite number of steps. To enable the second step, we first mechanically prove the correctness of the Add_Weak algorithm from [25] that adds weak convergence. As a result, any protocol generated by Add_Weak will be correct by construction. Moreover, the mechanical verification of Add_Weak provides a reusable theory in PVS that enables us to verify the generalizability of small instances of different protocols generated by Add_Weak. If the mechanical verification succeeds, then it follows that the synthesized protocol is in fact correct for an arbitrary number of processes. Otherwise, we use the feedback of PVS to determine why the synthesized

Table 2.1: Total size of PVS theories developed in Part I.

File	Size PVS		Proof-script size
Add_Weak	247		341
Coloring_m_n	232		342
Binary_Agreement	297		701
ChaiSort_m_n	607		1706
Total(319 lemmas)	1381	+	3020 = 4473

protocol cannot be generalized and re-generate a protocol that addresses the concerns reported by PVS. We continue this cycle of *synthesize and generalize* until we have a parameterized protocol. Large part of the framework that we developed to prove the soundness of Add_Weak algorithm for weak convergence, can also be reused for

the mechanical specification and verification of self-stabilizing protocols designed by means other than our synthesis algorithms. We demonstrate the reusability of the framework in the context of a coloring protocol (Section 5.3) and a binary agreement protocol (Section D.1). Furthermore, we present a method for mechanical verification of strong convergence and we demonstrate it for the design of a self-stabilizing sorting algorithm on unidirectional rings and chains. These case studies illustrate how the PVS theory we developed can be reused for the verification of strong stabilization. Nevertheless, the mechanical verifications of these protocols is really lengthy. This is due to the very precise type theoretic nature of theorem proving, the complexity of adding convergence problem, in addition to the involvement of lengthy inductive proofs. For instance, even a slight change on the actions of the protocol or on the topology, would change the proofs fundamentally. Thus, as our synthesis in the small scope is automatic, the actions and the topologies may change from one case to another unpredictably. Consequently, the automation of the proofs was highly challenging. In particular, it requires proving general theories that are applicable to larger families of automatically synthesized protocols. To illustrate the difficulty of this task, we present a summary of the mechanical proofs sizes of our theories in PVS (Listing 2.1).**Organization**. Section 1.2 provides a brief overview of PVS. Section 8.2 introduces basic concepts and presents their formal specifications in PVS. Then, Section 4.2 formally presents the problem of adding convergence to protocols. Sections 5.1 and 5.2 respectively present the specification and verification of Add_Weak in PVS. Sections 5.3, D.1, 6.1 and 6.2 respectively demonstrate reusability and generalizability in the context of a graph coloring protocol, a binary agreement protocol and a sorting algorithm on rings and chains. Section 7 discusses the significance of this research and related work. Finally, Section 7.2 makes concluding remarks and presents future extensions of this work.

Chapter 3

Preliminaries and Basic Concepts

3.1 General Framework for Shared Memory Protocols in PVS

In this section, we present a reusable frame-work for shared memory protocols that is used throughout this dissertation. Particularly, we present the fundamental types, functions, and parameters to specify basic concepts such as protocols, state predicates, computations prefix and convergence. We also present their formal specifications in PVS. The definitions of protocols and convergence are adapted respectively from [34, 21].

3.1.1 Protocols

A protocol includes a set of processes, a set of variables and a set of transitions. Since we would like the specification of a protocol to be as general as possible, we impose little constraints on the notions of state, transitions, etc. Thus, the concepts of *state*, *variable*, and *domain* are all abstract and nonempty. Formally, we specify them by uninterpreted types state: Type+, Variable: Type+, Dom: Type+. A *state predicate* is a set of states specified as StatePred: TYPE = set[state]. The concept of *transition* is modeled as a tuple type of a pair of states Transition: Type = [state,state] [34]. Likewise, the type of the set of transitions that are constructed by the *actions* of the protocol is defined naturally as a set of transitions, Action:Type =set[Transition]. An action of a protocol can be considered as an atomic guarded command "$grd \rightarrow stmt$", where grd denotes a Boolean expression in terms of protocol variables and $stmt$ is a set of statements that atomically update protocol variables when grd holds. An action is *enabled iff* (if and only if) its guard grd evaluates to true. To capture this restriction throughout this dissertation we formalize an action of a process j in a protocol by means of tabular construct of PVS where we use blank entries whenever the protocol

is silent. Hence a transition of a protocol - from a state s - will be defined as (s, action(j,s)). We provide concrete examples in sections 5.3 and 6.1. The types Dom, Variable, and Action can be finite or infinite types in our PVS specifications [1].

3.1.2 Distribution and Atomicity Models

We model the impact of distribution in a shared memory model by considering read and write restrictions for processes with respect to variables. Due to the inability of a process P_j in reading some variables, each transition of P_j belongs to a *group* of transitions. For example, consider two processes P_0 and P_1 each having a Boolean variable that is not readable for the other process. That is, P_0 (respectively, P_1) can read and write x_0 (respectively, x_1), but cannot read x_1 (respectively, x_0). Let $\langle x_0, x_1 \rangle$ denote a state of this protocol. Now, if P_0 writes x_0 in a transition $(\langle 0,0\rangle, \langle 1,0\rangle)$, then P_0 has to consider the possibility of x_1 being 1 when it updates x_0 from 0 to 1. As such, executing an action in which the value of x_0 is changed from 0 to 1 is captured by the fact that a group of two transitions $(\langle 0,0\rangle, \langle 1,0\rangle)$ and $(\langle 0,1\rangle, \langle 1,1\rangle)$ is included in P_0. In general, a transition is included in the set of transitions of a process iff its associated group of transitions is included. Formally, any two transitions (s_0, s_1) and (s'_0, s'_1) in a group of transitions formed due to the read restrictions of a process P_j meet the following constraints, where r_j denotes the set of variables P_j can read: $\forall v : v \in r_j : (v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1))$ and $\forall v : v \notin r_j : (v(s_0) = v(s_1)) \wedge (v(s'_0) = v(s'_1))$, where $v(s)$ denotes the value of a variable v in a state s that we represent by the Val:[Variables,state -> Dom] i.e Val(v, s) function in PVS.

To enable consistent reusability of our PVS specifications, we will specify our distribution model set of axioms as predicates in the abstract formal types and definitions of a process, protocol, and group transition, so one should mechanically prove them whenever a constant of these types is declared, as explained in Sections 5.3 and 6.1.1. Moreover, this technique will help the the user to use them in the mechanical proofs using the command *typepred* of PVS whenever the need be.

For instance, since a *process* p is a tuple of a subset of variables that are readable by that process, a subset of variables that are writable by that process, and its set of transitions. Then the axiom:

AXIOM Every writable variable in a process p is a readable variable.

is implemented formally in the definition of the type of a process as a predicate as illustrated in Listing 3.1. Observe that if a user instantiated the first set of variables with a set that is not a superset of the second set of variables then PVS will issue unprovable TCC (type check condition) obligation. This technique in PVS will discover

[1] `https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/addweakconv`

any possible inconsistency in the specification immediately thus the user should be able to revise the specification accordingly, saving huge time and effort.

```
1 p_process:  TYPE = {p:[ set[Variables],set[Variables],Action ]
2                                    |subset?(proj_2(p),proj_1(p))}
```

Listing 3.1: Processes

Similarly, a *protocol prt* is a tuple of a set of processes, variables, and set of transitions. However, the property that *x of type Transition belongs to the transitions of prt iff x belongs to the transitions of one of its processes* must be valid for any protocol. Thus we implement this property in the protocol [2] type definition as given in Listing 3.2:

```
1    nd_Protocol:  TYPE =  {prt:[ set[p_process], set[ Variables],
        Action ]|
2    (forall (proc:{p:p_process|member(p,proj_1(prt))}):
3    proj_3(proc)⊆ proj_3(prt)
4    ∩ (proj_3(prt)={x:Transition|
5                        Exists(pp:p_process|pp ∈ proj_1(prt)):
6                           x ∈ proj_3(pp))})}
```

Listing 3.2: Protocols

Transition groups of a transition x due to a process p. In the following formal specifications, v is of type Variable, p is of type p_process, t and t' are of type Transition, and non_read Listing 3.4 and transition_group Listing 3.3 are functions that respectively return the set of unreadable variables of the process p and the set of transitions that meet $\forall v : v \in r_j : (v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1))$ and $\forall v : v \notin r_j : (v(s_0) = v(s_1)) \wedge (v(s'_0) = v(s'_1))$ for a transition $t = (s_0, s_1)$ and its groupmate $t' = (s'_0, s'_1)$.

```
1 transition_group(p:p_process,x:Transition,prt:nd_Protocol):set[
     Transition] = { x1:Transition| member(p,proj_1(prt))
2 ∩ (FORALL (v1:Variables| member(v1,proj_1(p))):
3                             Val(v1,proj_1(x)) = Val(v1,proj_1(x1))
4                    ∩ Val(v1,proj_2(x)) = Val(v1,proj_2(x1)))

6 ∩ (FORALL (v:Variables|member(v,Non_read(p,prt))):
7                             Val(v,proj_1(x)) = Val(v,proj_2(x))
8                    ∩ Val(v,proj_1(x1)) = Val(v,proj_2(x1)))}
```

Listing 3.3: transition_group(p,x): returns transition group of a transition x due to a process p

[2] We use the name *nd_protocol* to remind the reader that all considered protocols in this part are non-deterministic.

Where member(x,X) and subset?(X,Y) respectively represent the set membership and subset predicates in a set-theoretic context, and proj_k is a built-in function in PVS that returns the k-th element of a tuple x. Moreover, in PVS x'i can be used alternatively with proj_i. We use both of the two notations of PVS throughout the dissertation.

```
1 Non_read( pp:p_process,pprt:nd_Protocol):set[Variables] = {v|member
      (pp,proj_1(pprt)) AND member(v,proj_2(pprt))AND NOT member(v,
      proj_1(pp)) }
```

Listing 3.4: non_read: function

The *projection of a protocol prt on a state predicate I*, denoted $Proj(prt, I)$, includes the set of transitions of *prt* that start in I and end in I. One can think of the projection of *prt* as a protocol that has the same set of processes and variables as those of *prt*, but its transition set is a subset of *prt*'s transitions confined in I. We model this concept by defining the following function, where Z is instantiated by transitions of *prt*. (proj_k is a built-in function in PVS that returns the k-th element of a tuple.)

$$\begin{array}{l}\mathsf{Proj}(Z\text{: Action, } I\text{: StatePred})\text{: Action} = \\ \{t\text{:Transition} \mid t \in Z \bigwedge \mathsf{t`1} \in I \bigwedge \mathsf{t`2} \in I\}\end{array}$$

Example: Coloring on a ring of $n > 3$ processes with $m > 2$ colors. The coloring protocol, denoted $Coloring(m, n)$, includes $n > 3$ processes located along a bidirectional ring. Each process P_j has a local variable c_j with a domain of $m > 2$ values representing m colors. Thus, the set of variables of $Coloring(m, n)$ is $V_{Coloring(m,n)} = \{c_0, c_1, ..., c_{n-1}\}$. As an example of a state predicate, consider the states where no two neighboring processes have the same color. Formally, $I_{coloring} = \forall j : 0 \leq j < n : c_j \neq c_{j\oplus 1}$, where $\oplus$ and $\ominus$ denote addition and subtraction modulo n. Each process P_j $(0 \leq j < n)$ has the following action:

$$A_j : (c_j = c_{j\ominus 1}) \vee (c_j = c_{j\oplus 1}) \rightarrow c_j := other(c_{j\ominus 1}, c_{j\oplus 1}) \tag{3.1}$$

If P_j has the same color as that of one of its neighbors, then P_j uses the function "$other(c_{j\ominus 1}, c_{j\oplus 1})$" to non-deterministically set c_j to a color different from $c_{j\ominus 1}$ and $c_{j\oplus 1}$. The projection of the actions A_j (for $0 \leq j < n$) on the predicate $I_{coloring}$ is empty because no action is enabled in $I_{coloring}$. The coloring protocol has applications in several domains such as scheduling, bandwidth allocation, register allocation, etc. It is known that if $m > d$, where d is the max degree in the topology graph of the

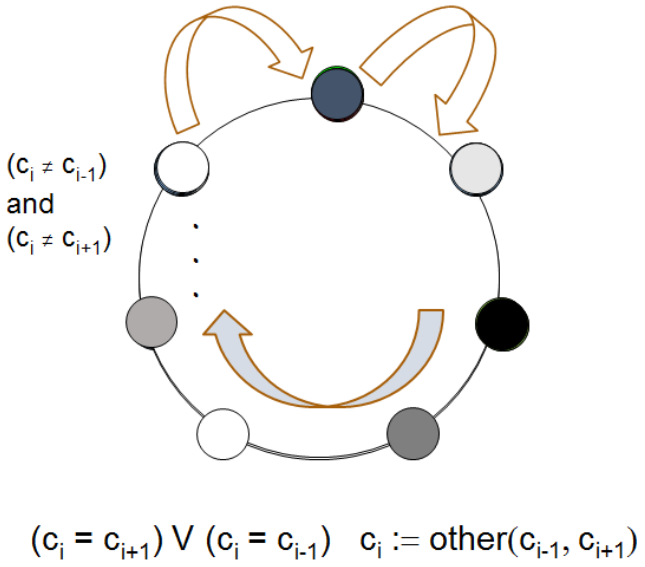

Figure 3.1: The action of the Coloring(m,n) protocol

protocol, then the coloring problem is solvable. For this reason, we have $m > 2$ for the ring.[3].

Example: Read/Write restrictions in $Coloring(3, 5)$. In the coloring protocol, each process P_j can read $\{c_{j\ominus 1}, c_j, c_{j\oplus 1}\}$, and is allowed to write only c_j. For a process P_j, each transition group includes 3^{n-3} transitions because P_j can read only the state of itself and its left and right neighbors; there is one transition in the group corresponding to each valuation of unreadable variables.

A **computation prefix** of a protocol *prt* is a *finite* sequence of states of positive length, where each state is reached from its predecessor by a transition of *prt*. Kulkarni *et al.* [34] specify a prefix as an infinite sequence in which only a finite number of states are used. By contrast, we specify a computation prefix using the PVS finite sequence type. We believe that it is more natural and more accurate to model the concept of prefix by finite sequences. Our experience also shows that modeling computation prefixes as finite sequences simplifies formal specification and verification of reachability and convergence while saving us several definitions that were required in [34] to capture the length of the prefix. We first define a subtype for finite sequences of states of positive length; i.e., Pos_F_S. Then, we use the predicate Condi_prefix?(A,Z) that holds when all transitions $(A(i), A(i+1))$ of a sequence A belong to a set of transitions Z. The notation A`length denotes the length of the

[3]We present the formal specification of this protocol in Section 5.3.1

sequence A, A‘seq(i) returns the i-th element of sequence A and below[k] is a PVS type that represents natural values less than k. The function PREFIX returns the set of computation prefixes generated by transitions of Z.

Pos_F_S: TYPE = {c:finite_sequence[state] | c‘length > 0}

Condi_Prefix?(A:Pos_F_S,Z:Action):bool= FORALL(i: below[A‘length-1]):
member((A‘seq(i), A‘seq(i+1)), Z)

PREFIX(Z: Action): set[Pos_F_S]= {A:Pos_F_S | Condi_Prefix?(A,Z) }

Example: A computation of $Coloring(3, 5)$. Consider an instance of $Coloring(m, n)$ where $n = 5$ and $m = 3$. Thus, $c_j \in \{0, 1, 2\}$ for $0 \leq j < 5$. Let $\langle c_0, c_1, c_2, c_3, c_4\rangle$ denote a state of the protocol. Starting from a state $\langle 0, 1, 2, 2, 0\rangle$, the following sequence of transitions could be taken: P_2 executes $(\langle 0, 1, 2, 2, 0\rangle, \langle 0, 1, 0, 2, 0\rangle)$ and P_0 executes $(\langle 0, 1, 0, 2, 0\rangle, \langle 2, 1, 0, 2, 0\rangle)$.

3.1.3 Closure and Convergence

A state predicate I is *closed* in a protocol *prt* iff every transition of *prt* that starts in I also terminates in I [22, 41]. The closed predicate checks whether a state predicate I is actually closed in a set of transitions Z.

closed?(I: StatePred, Z: Action): bool = FORALL (t:Transition | (member(t,Z)) AND
member(proj_1(t), I)) : member(proj_2(t), I)

A protocol *prt weakly converges* to a non-empty state predicate I iff from every state s, there exists at least one computation prefix that reaches some state in I [22, 41]. A *strongly converging* protocol guarantees that for any initial state s all possible prefixes from s will reach some state in I. Notice that any strongly converging protocol is also weakly converging, but the reverse is not true in general. A protocol *prt* is weakly (respectively, strongly) self-stabilizing to a state predicate I iff (1) I is closed in *prt*, and (2) *prt* weakly (respectively, strongly) converges to I.

```
1 Reach_from?(Z:Action,A:PREFIX_T(Z),s:state, I:StatePred):bool
2              = Exists (j:below[A`length]):
3                   A`seq(0)= s  AND member(A`seq(j),I)
```

Listing 3.5: Reachability of a prefix from a state s to I.

In Listing 3.5 (Line 1), A:PREFIX_T(Z) indicates that A is a computation prefix of Z. PREFIX_T(Z) is a *dependant type* i.e., its definition depends on a parameter, in this case Z. This is a powerful feature in PVS for declaring variables of dependant types (e.g., Listing 3.7 Line 3).

```
1 PREFIX_T(Z: Action): TYPE = {A:Pos_F_S| Condi_Prefix?(A,Z) }
```

Listing 3.6: Prefix_T (Z) a dependant Type.

```
1 weak_converge_s?(Z:Action,I:StatePred,s:state):bool
2                         = Exists( A:PREFIX_T(Z)|
3                                     A(0)=s and member(A,PREFIX(Z))):
4                                                 Reach_from?(Z,A,s,I)
```

Listing 3.7: Weak Convergence Predicate

For a given protocol $prt : nd\_Protocol$ the parameter Z in Listing 3.7 (line 1) should be instantiated with the third projection of the protocol i.e. proj_3(prt) (the set of its transitions)

Example: Closure and convergence of $Coloring(3, 5)$. Notice that the predicate $I_{coloring}$ is closed in actions A_j, where $0 \leq j < 5$, since no action is enabled in $I_{coloring}$. Moreover, starting from any state (in the 3^5 states of the state space of $Coloring(3, 5)$), there will be a computation prefix that reaches a state in $I_{coloring}$. The $Coloring(3, 5)$ presented in this section can been seen as an example of a weakly converging protocol.[4]

[4]We provide the full formal proof in Section5.3.

Chapter 4

Problem

4.1 Adding Weak Convergence Problem

The problem of adding convergence (from [25]) is a transformation problem that takes as its input a protocol *prt* and a state predicate I that is closed in *prt*. The output of Problem 4.2.1 is a revised version of *prt*, denoted prt_{ss}, that converges to I from any state. Starting from a state in I, prt_{ss} generates the same computations as those of *prt*; i.e., prt_{ss} behaves similar to *prt* in I.

Problem 4.1.1. Add Convergence

- **Input**:
  - *(1) A protocol prt;*
  - *(2) A state predicate I such that I is closed in prt;*
  - *(3) A property of L_s converging, where $L_s \in \{$weakly, strongly$\}$.*
- **Output**: *A protocol prt_{ss} such that :*
  - *(1) I is unchanged;*
  - *(2) the projection of prt_{ss} on I is equal to the projection of prt on I,*
  - *(3) prt_{ss} is L_s converging to I. Since I is closed in prt_{ss}, it follows that prt_{ss} is L_s self-stabilizing to I.*

4.2 Problem Statement

4.2.1 The Design of Parameterized Self-Stabilizing Protocols

Algorithmic Design of Parameterized Self-Stabilizing Protocols (PSSP) is an open problem. Existing methods can generate solutions that are correct up to a finite scope [23, 24, 25, 26, 17]. Even the design self-stabilizing protocols (in small scope) is a challenging task. For instance, designing strong convergence algorithmically is NP-complete [14]. Our major objective is to devise a method that enables the design of Parameterized Self-Stabilizing Protocols (PSSPs) that are correct by construction. Particularly, this part of the dissertation is studying the following problem:

Problem 4.2.1. The Design of Parameterized Self-Stabilizing Protocols (PSSPs)

- **Input***:*
  - *(1) A property of L_s converging, where $L_s \in \{$weakly, strongly$\}$;*
  - *(2) A protocol prt;*
  - *(3) A state predicate I such that I is closed in prt.*
- **Output***: A generalized parametric protocol $prt_{general}$ of prt(2)and a generalized state predicate $I_{general}$ of I such that :*
  - *(1) $I_{general}$ is closed in $prt_{general}$;*
  - *(2) $prt_{general}$ is L_s converging to $I_{general}$, i.e., $prt_{general}$ is L_s self-stabilizing to $I_{general}$.*

Where prt is given with specific number of processes and finite sets of variables and domains, a generalized protocol of prt is a parameterized version of prt with respect to the number of processes, number of variables, and the size of the domain.

Chapter 5

Parameterized Weakly Self-Stabilizing Protocols

In [22, 25] the authors showed that weak convergence can be added in polynomial time (in the size of the state space), whereas adding strong convergence is known to be an NP-complete problem [14]. Farahat and Ebnenasir [25, 26] present a sound and complete algorithm for the addition of weak convergence and a set of heuristics for efficient addition of strong convergence (in small scopes). The main focus of Sections 5.1 and 5.2 is the mechanical verification of the soundness of the Add_Weak algorithm 1. Algorithm 1 provides an informal and self-explanatory representation of the Add_Weak algorithm presented in [25]. Moreover, Table 5.1 summaries the major formal types, predicates,variables, and functions used in in the proof of the soundness of the algorithm.

Algorithm 1 : Add_Weak

Input: prt:nd_Protocol, I: statePred;

Output: set[Transition]; // Set of transitions of a weakly self-stabilizing version of prt.

1: Let Δ_{prt}={proj_3(prt)}.
2: Let $\Delta_{converge}$ be the set of transition groups that adhere to read/write restrictions of processes of prt, but exclude any transition starting in I;
3: $prt_{ws} = \Delta_{prt} \cup \Delta_{converge}$;
4: $no\_Prefix := \{s : state \mid (s \notin I) \wedge$(there is no computation prefix using transitions of prt_{ws} that can reach a state in I)$\}$
5: $\Delta_{ws} =$ (Union(prt_{ws}))
6: If ($no\_Prefix \neq \emptyset$) then weak convergence cannot be added to prt; return;
7: return Δ_{ws};

Table 5.1: Major types, variables, predicates, and functions for Add_Weak algorithm

Name in PVS	Type	Description	
prt	nd_Protocol	VAR	
I	StatePred	VAR	
s	State	VAR	
Δ_{prt}(prt)	set[set[Transition]]	Func	{prt'3}
PREFIX_T(Z:set[transition])		Type	Lisiting 3.6
transition_groups_proc(p,prt)	set[set[Transition]]	Func	Listing 5.1
$\Delta_{converge}$(prt,I)	set[set[Transition]]	Func	Lisiting 5.2
prt_{ws}(prt,I)	set[set[Transition]]	Func	Listing 5.3
Δ_{ws}(prt,I)	set[Transition]	Func	Listing 5.3
Condi_no_Prefix?(prt,I,s)	bool	predicate	Listing5.4

Mechanical verification of the soundness of Add_Weak ensures that any protocol synthesized by Add_Weak is correct by construction. Moreover, the lemmas and theorems developed in mechanical verification of Add_Weak provide a reusable framework for mechanical verification of different protocols that we generate using our synthesis tools [25, 17]. The verification of synthesized protocols increases our confidence in the correctness of the *implementation* of Add_Weak and helps us to generalize small instances of weakly converging protocols to their parameterized versions.

5.1 Specification of Add_Weak

This section presents the highlights of the formal specification of Add_Weak in PVS. (The complete PVS specifications are available at `https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/addweakconv`.) We start by specifying the basic components used in the Add_Weak algorithm, namely the functions $\Delta_{prt}, \Delta_{converge}$ and Δ_{ws}, and the state predicate *no_Prefix* Table 5.1.

Notation. In the subsequent formal specifications, we use the identifiers Delta_prt, Delta_Converge and Delta_ws corresponding to the functions $\Delta_{prt}, \Delta_{converge}$ and Δ_{ws} in Add_Weak. The function transition_groups_proc(p,prt) returns the set of transition groups of a process p of a protocol prt.

```
1 transition_groups_proc(p: p_process, prt: nd_Protocol):
2    set[set[Transition]] = {gg: set[Transition]| p ∈ PROJ_1(prt) ∩
3          (∃ (x: Transition): transition_group(p, xx, prt) = g)}
```

Listing 5.1: transition groups of a process.

```
1 Delta_converge(prt:nd_Protocol, I:StatePred): set[set[Transition]]
    =
2       {g: set[Transition] |
3          EXISTS (p: p_process | member(p, PROJ_1(prt))):
4            member(g, transition_groups_proc(p, prt)) ∩
5             (FORALL (x: Transition | member(x, g)): PROJ_1(x)∉ I)}
```

Listing 5.2: $\Delta_{converge}$

Finally, the functions of Steps 3 and 5 of algorithm Add_Weak 1 are formalized as follows:

```
1 Delta_pws(prt: nd_Protocol, I: StatePred): set[set[Transition]] =
2       union({prt'3}, Delta_converge(prt, I))

4  Delta_ws(prt: nd_Protocol, I: StatePred): set[Transition] =
5       Union(Delta_pws)
```

Listing 5.3: Δ_{pws} and Δ_{ws}

Where *union* in PVS is the union of two sets, whereas $Union(\mathfrak{F})$ is the union of a family of sets $\mathfrak{F}$ which can contain more than two sets. To specify the set of states no_Prefix, we first specify the condition of membership of this set as a predicate Condi_noPrefix that returns true only if for a protocol *prt*, a state predicate I and a state s, no state in I can be reached from s by computation prefixes of *prt* Listing 5.4.

```
1 condi_no_Prefix?(prt: nd_Protocol, I: StatePred, s: state): bool =
2       FORALL (g: Action,
3               A: PREFIX_T(g)
4                   | member(g, pws(prt, I)) AND
5                      member(A, PREFIX(g)) AND A`seq(0) = s0):
6         NOT (Reach_from?(g, A, s0, I))
```

Listing 5.4: Condition of membership of no_prefix.

We then specify the set no_Prefix using this predicate (Listing 5.5 Line 4) as follows:

```
1  no_Prefix(prt:nd_Protocol,I:StatePred):set[state]=
2      {s:state |
3            NOT( member(s,I)) AND
4            Condi_noPrefix?(prt,I,s)}
```

Listing 5.5: The set no_Prefix.

We finally specify Add_Weak using Table construction of PVS. Thus we left a blank Listing 5.6 (Line 11) when the algorithm does not do anything (Step 6 algorithm 1).

```
1  Add_weak(pprt: nd_Protocol,
2              I:
3                {II: StatePred |
4                         closed?(II, PROJ_3(pprt)) AND
5                          empty?(no_Prefix(pprt, II))}):
6           set[Transition] =
7         TABLE
8           %|-----------------------------------------+-----------------------%
9           | empty?(no_Prefix(pprt, I))       |Delta_ws(pprt, I) ||
10          %|-----------------------------------------+-----------------------%
11          | NOT empty?(no_Prefix(pprt, I))   |                   ||
12          %|-----------------------------------------+-----------------------%
13        ENDTABLE
```

Listing 5.6: Add Weak formalization using Table construction.

5.2 Verification of Add_Weak

In order to prove the soundness of Add_Weak, we check if (1) I is unchanged; (2) the projection of Δ_{ws} on I is equal to the projection of Δ_{prt} on I, and (3) Δ_{ws} is weakly converging to I. The first constraint holds trivially since no step of Add_Weak adds/removes a state to/from I. Next, we present a set of lemmas and theorems that prove the other two constraints of Problem 4.2.1.

5.2.1 Verifying the Equality of Projections on Invariant

In this section, we prove that Constraint 2 of Problem 4.2.1 holds for the output of Add_Weak, denoted by a protocol whose set of transitions is Δ_{ws}. Our proof obligation is to show that the projection of Δ_{ws} on I is equal to the projection of Δ_{prt} on I. We decompose this into two set inclusion obligations of Proj(Delta_prt,I) $\subseteq$ Proj(Delta_ws,I) and Proj(Delta_ws,I) $\subseteq$ Proj(Delta_prt,I). Notice that, by assumption, closed?(I,Delta_prt) is true.

Lemma 5.2.1. Proj(Delta_prt,I) *is a subset of* Proj(Delta_ws,I).

Proof. The proof is straightforward since by construction we have $\Delta_{ws} = \Delta_{prt} \cup \Delta_{converge}$. □

Lemma 5.2.2. Proj(Delta_ws,I) *is a subset of* Proj(Delta_prt,I).

Proof. If a transition $t = (s_0, s_1)$ is in Proj(Delta_ws,I) then $s_0 \in I$. Since $\Delta_{ws} = \Delta_{prt} \cup \Delta_{converge}$, either $t \in \Delta_{prt}$ or $t \in \Delta_{converge}$. By construction, $\Delta_{converge}$ excludes any transition starting in I including t. Thus, t must be in Δ_{prt}. Since $s_0 \in I$, it follows that $t \in$ Proj(Delta_prt,I). □

Theorem 5.2.1 (Sound_Projection)**.** Proj(Delta_ws,I) = Proj(Delta_prt,I).

Or in in PVS this theorem is explicitly formalized as follows:

```
1  Sound_projction: Theorem
2     closed?(I, prt'3) AND empty?(no_Prefix(prt, I)) IMPLIES
3      (member(x, proj(Add_weak(prt, I), I)) =
4        member(x, proj(prt'3, I)))
```

Listing 5.7: Sound projections property in PVS.

5.2.2 Verifying Weak Convergence

In this section, we prove the weak convergence property (i.e., Constraint 3 of Problem 4.2.1) of the output of Add_Weak. Specifically, we show that from any state $s_0 \in \neg I$, there is a prefix A in PREFIX(Delta_ws) such that A reaches some state in I. Observe that, an underlying assumption in this section is that closed?(I,Delta_prt) holds. For a protocol *prt* and a predicate I that is closed in *prt* and a state $s \notin I$, we have:

Lemma 5.2.3. *If* empty?(no_Prefix(prt,I)) *holds then* Condi_noPrefix?(prt,I,s) *returns false for any* $s \notin I$.

Lemma Weak_converg_pws_2 Listing 5.8 implies that when Add_Weak returns, the revised version of *prt* guarantees that there exists a computation prefix to I from any state outside I; hence weak convergence. This is due to the fact that A is a prefix of Δ_{ws}. In PVS we represent this theorem explicitly in Listing 5.8 as follows:

```
1 Weak_converg_pws_2: THEOREM
2     closed?(I, PROJ_3(prt)) AND
3      empty?(no_Prefix(prt, I)) AND NOT member(s, I)
4      IMPLIES
5      (EXISTS (A: PREFIX_T(Delta_ws(prt, I))
6                    | A`seq(0) = s ∩ A ∈ PREFIX(Delta_ws(prt, I))):
7         Reach_from?(Delta_ws(prt, I), A, s, I))
```

Listing 5.8: Convergence property in PVS.

5.2.3 Explicit Representation of the Convergence Property

The formalization in Listing 5.8 is sufficient to show the safety property we want to prove i.e. convergence property. However, given the very rich specification language of PVS, one can build alternative equivalent but more explicit formalization for the same result. For example, using the predicate weak_converge_s?[1] (Listing 5.9 (Line 4)) we prove the following theorem in PVS:

```
1  Weak_Converge_pws_3: THEOREM
2     closed?(I, prt'3) AND
3      empty?(no_Prefix(prt, I)) AND NOT member(s, I)
4      IMPLIES weak_converge_s?(Delta_ws(prt, I), I, s)
```

Listing 5.9: Convergence property in PVS alternative representation 1.

Furthermore, observe that for $s1 \in I$, the property Weak_Converge_s? on Delta_ws(prt, I), I, and s1 can be proved to return true easily because of the closure property that is assumed on I i.e. closed?(I, prt'3). This also can be seen explicitly in PVS. In particular, we proved the following theorem Listing 5.10 by using a boolean predicate called *weak_converge* (Line 5):

```
1 Addweak_Sound: THEOREM
2           closed?(I, prt'3) AND empty?(no_Prefix(prt, I))
3      AND member(s, I) AND member(x, Delta_ws(prt, I)) AND s = x'1
4      AND NOT member(s1, I)
5      IMPLIES Weak_Converge?(Delta_ws(prt, I), I, s, s1)
```

Listing 5.10: Convergence property in PVS alternative representation 2.

Where the predicate *Weak_Converge?* returns true iff Weak_Converge_s?(Delta_ws(prt, I), I, s) returns true, and Weak_Converge_s?(Delta_ws(prt, I), I, s1) returns true. Which completes the proof. By proving these theorems we finish this section. We are ready now to see more concrete examples on the usability of Add Weak and the abstract framework we introduced thus far with concrete actual examples.

[1] Defined previously in Listing 3.7

Table 5.2: Major declarations of Coloring(m,n)

Name in PVS	Description	
COLORS	TYPE+	= below[m]
STC	TYPE+	{s:finseq(COLORS)| s‘length = n } a state
ndx_varb	TYPE+	[COLORS,below[n]]
K,L	VAR	ndx_varb
is_nbr?(K,L)	predicate	mod(abs(K‘2-L‘2),n)<= 1
is_bad_nbr?(K,L)	predicate	mod(abs(K‘2-L‘2),n) = 1 ∧ K‘1 = L‘1
nbr_v(K)	TYPE+	{C:ndx_varb |is_nbr?(K,C)}
bad_nbr_v(K)	TYPE	{C:ndx_varb|is_bad_nbr?(K,C)}
bad_nbr_s(K):set[ndx_varb]	Func	{C:ndx_varb|is_bad_nbr?(K,C)} set of all bad nbr of K
nbr_is_bad?(s,j)	predicate	nonempty?(bad_nbr_s(K(s,j)))
nbr_is_good?(s,j)	predicate	NOT nbr_is_bad?(s,j)
is_LEGT?(s)	predicate	∀ (j:below[n]): nbr_is_good?(s,j)
S_ill	TYPE	{ s:STC| not is_LEGT?(s)}
K(s:STC,j:below[n]):ndx_varb	Func	(s‘seq(j),j)
VAL(L,s):COLORS	Func	K(s,L‘2)‘1 valuation function
nbr_colors(K):set[COLORS]	Func	{cl:COLORS| ∃(c:nbr_v(K)):c‘1=cl }
fullset_colors:set[COLORS]	const	{cl:COLORS|TRUE}
ϵ (epsilon)	Func	the Choice func in PVS
other(K):COLORS	Func	ϵ(fullset_colors - nbr_colors(K))

5.3 Generalized Coloring Protocol *Coloring(m,n)*

In this section, we show how a synthesized 3-coloring protocol (that is correct by construction for rings up to 40 processes) is proved to be generalizable for rings of arbitrary size with more than 2 colors (i.e., $m > 2$). Subsection 5.3.1 presents the reusability of the abstract framework for formal specification of 3-coloring in PVS. Subsection 5.4 verifies the generalization of 3-coloring to rings of arbitrary size with 3 or more colors. In Table 5.2 we list the major types, variables, and functions that we use to formalize the Coloring(m,n) protocol 8.2 [2].

[2] `https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/coloring_m_n`

5.3.1 PVS Specification of Coloring

Local status at process j in the ring. For instance,the type COLORS is the type below[m], the number of processes is n, and we assume $n > m$ and $m >= 3$. A variable of an arbitrary process j can be evaluated based on two pieces of information, namely a color and the process $j \leq n$, thus we represent its type *ndx_varb* as a tuple of two components (COLOR, below[n]).
Defining a state of $Coloring(m, n)$. States of the ring are represented as a finite sequence of colors of length n. But since this protocol has a bidirectional ring topology, thus for an arbitrary state s, we define the *neighborhood* for arbitrary variable at process j, by means of $-,+$ *mod(n)* operations. Thus the left and the right neighbors are defined relatively locally to j.
Detecting good/bad neighbors. The predicate is_nbr?(K,L) returns true iff K and L are variables of two neighboring processes; i.e., mod(abs(K‘2-L‘2),n) ≤ 1, where K‘2 denotes the index of K. Likewise, we define the predicate is_bad_nbr?(K,L) that holds iff K,L are neighbors indices and they hold the same colors i.e., K‘1=L‘1. Finally we define the set bad_nbr_s(K) to be the set of all bad neighbors of variable K. The set nbr_colors(K) returns the set of the colors of all neighbors of K of a process j. By saying the state s *is corrupted at process* j we mean K(s,j) has a nonempty set of bad neighbors and s is called illegitimate state.Particularly, s will be of type S_ill. A legitimate state has an empty set of bad neighbors set.
Reusability of the abstract framework to specify the general protocol Coloring(m,n). We now define formally the protocol's main parameters. To reuse the abstract framework we defined previously, we will import the PVS theory of Add_Weak.pvs with the following concrete values STC, below[m], ndx_varb, VAL instantiated into its abstract parameters. This enables us to reuse its abstract types in specifying $Coloring(m, n)$. Particularly, we will define the sets of readable, writable variables, and the transitions of the protocol. We have to do that in such a way that adhere to the read/write axioms of the model. To keep the consistency of the formalization of the protocol these axioms were implemented as predicates in the types process, nd_protocol. Thus, PVS will generate several TCCs which their proofs guarantee the satisfiability of the axioms for the concrete declared protocol.
Specification of the action of Coloring(m,n). In Listing 5.11 we present the action of the protocol to return the state reached when process j acts to correct its corrupted state. Formally, we define action using tabular type of PVS. Because of the very precise types of PVS, this type allows explicit representation when the protocol is silent by using blank entries corresponding to the appropriate cases. However, the type checker will not generate the coverage tccs that are required by using COND internment. Instead, it will require to ensure that the cases for blank entries are not accessible. We do that in Listing 5.11 (line 2), as s must be illegitimate state and

has a bad neighbor(i.e. corrupted at process j). We specify the function *other* Table 5.2 to choose a new color for the variable K(s,j) other than the corrupted one. To this end, we use the *epsilon* function (the choice function in PVS)over the full set of colors minus the set of colors of the neighbors of the corrupted process. Observe that the very precise type in Listing 5.11 Line 2, ensures the choice set will not be empty.

```
1   action(j: below[n],
2          s: {state: STC | NOT is_LEGT?(state) AND nbr_is_bad?(state
               , j)},
3          K: ndx_varb, C: bad_nbr_v(K)):
4         STC =
5       TABLE
6                    +-------------------------------+---------------------+
7                    |[ nbr_is_bad?(s, j)            |nbr_is_good?(s, j)]|
8  +-------------------+-------------------------------+---------------------+
9  | NOT is_LEGT?(s) |(#length:=n,
10                                seq
11                                := (LAMBDA
12                                (i: below[n]):
13                                     IF i = j
14                                     THEN
15                                     other(K(s, j))
16                                     ELSE s(i)
17                                     ENDIF) #)       |                   ||
18 +-------------------+-------------------------------+---------------------+
19 | is_LEGT?(s)      |                               |                   ||
20 +-------------------+-------------------------------+---------------------+
21 ENDTABLE
```

Listing 5.11: Table construction for transition systems shows blank entries when the protocol is silent.

Specification of a process of coloring(m,n). In Table 5.3 we list the sets that are required to define a process j in this protocol. In particular, for an arbitrary global state s and a process j, we define the function READ_p which returns all readable variables of process j. Similarly, we define the function WRITE_p which returns the variables that process j can write. We finally define the function DELTA_p that returns the set of transitions belonging to process j if process j is corrupted in the global state s; i.e., j has a bad neighbor. Thus, the specification of a process of the protocol $Coloring(m, n)$ is as follows:

- PRS_p(s,j) = (READ_p(s,j), WRITE_p(s,j),DELTA_p(s,j))

The parameterized specification of the Coloring(m,n) protocol. Finally, to define the $Coloring(m, n)$ protocol as a constant of the type nd_Protocol. It is

Table 5.3: Defining a process using general framework

Name in PVS	Description	
READ_p(s, j):set[ndx_varb]	Func	{L:nbr_v(K(s,j))|True} readable variables of process j
WRITE_p(s,j):set[ndx_varb]	Func	{L:ndx_varb|L=K(s,j)} variables process j can write
Delta_p(s,j):set[Transition]	Func	{tr:Transition| ∃ (c:bad_nbr_v(K(s,j))): tr=(s,action(j,s,K(s,j),c))}

Table 5.4: Major functions in the proof of convergence of Coloring(m,n)

Name in PVS	Description	
corrected(s, j): COLORS	Func	Listing 5.12 correct color at j
LocallyCorrected(s, j): STC	Func	Listing 5.13 Locally correct state at j
corrected_up_to_j(s, j):STC	Func	Listing 5.14 Recursive construction of finite prefix states converging to I

sufficient to determine the sets of processes,variables,transitions of the protocol. For instance,

- PROC_prt(s:STC): set[p_process]={p:p_process | ∃ (j:below[n]):p = Process_j(s,j)}
- VARB_prt(s:STC): set[ndx_varb]= {v:ndx_varb | ∃ (j:below[n]):v ∈ WRITE $_p$(s,j))}
- Delta_prt(s:STC): set[Transition]= {tr:Transition | ∃ (j:below[n]):tr ∈ DELTA $_p$(s,j)} .

Thus:

Coloring(m,n)(s:STC):nd_Protocol =(PROC_prt(s),VARB_prt(s),DELTA_prt(s)).

5.4 Mechanical Verification of Parameterized Coloring

In this section we provide a summary of the major functions, lemmas and theorems, that we use for the proof of convergence of Coloring(m,n) for $m \geq 3$ and $n > m$.

```
1  corrected(s: STC, j: below[n]): COLORS =
2       TABLE
3                            %+------------------------+-------------------------+
4                            |[ nbr_is_good?(s, j)| nbr_is_bad?(s, j)]|
5          %-----------------+------------------------+-------------------------+
6          |NOT is_LEGT?(s)| s`seq(j)                | other(K(s, j))    ||
7          %-----------------+------------------------+-------------------------+
8          |is_LEGT?(s)     | s`seq(j)               |                   ||
9          %-----------------+------------------------+-------------------------+
10      ENDTABLE
```

Listing 5.12: colors status at process j after applying the correction of the *action* function.

The proof is based on a recursive prefix construction that demonstrates the existence of a computation prefix σ of $Coloring(m, n)$ from any arbitrary illegitimate state s such that σ includes a state in I. We illustrate the construction procedure by means of three functions listed in Table 5.4 and explained in subsection 5.4.1. Later in subsection 5.4.2 we prove the convergence by induction.

5.4.1 Recursive Construction

```
1   locallyCorrected(s: STC, j: below[n]): STC =
2       TABLE
3         %-----------------+-------------------------------------------------+
4         | NOT is_LEGT?(s) | (# length := n,
5                               seq
6                                 := (LAMBDA
7                                      (i: below[n]):
8                                      IF i = j
9                                      THEN corrected(s, j)
10                                     ELSE s`seq(i)
11                                     ENDIF) #)                          ||
12        %-----------------+-------------------------------------------------+
13        | is_LEGT?(s)     | s                                            ||
14        %-----------------+-------------------------------------------------+
15      ENDTABLE
```

Listing 5.13: Locally correct state at process j.

corrected(s,j): correct color at j. By studying the behavior of the action function of this protocol Listing 5.11 (line 6–18), we can capture the status of the colors on the ring after passing a state s to the action function by one out of four cases, illustrated in Listing 5.12 (lines 6–8). First, if s is corrupted at j then there will be a new correct color other than the original bad_one. Second, if the state s is corrupted but not

at j then the color will not change thus the color at j will be s‘seq(j). Third, if the state is not corrupted at all then all colors will remain unchanged at any process$j$, thus s‘seq(j). Observe that it is impossible to have a legitimate state that has any corrupted color. Thus the fourth case in the listing is left blank. In the all possible three cases we will have a color that does not have bad neighbors.

LocallyCorrected(s,j): A state that has a correct color at j is called *locally correct at j*. Thus the earlier three cases define a state under three conditions expressed in Listing 5.13 (Line 4–11, 13). We call this state *LocallyCorrected(s,j)* as it is always uncorrupted at process j.

corrected_up_to_j(s,j) The function coorected_up_to_j 5.14 captures a recursive correction procedure illustrated in (Line 3–9) that ensures the convergence to I in a finite number of steps from any illegitimate state. In section 5.4.2 we demonstrate the convergence proof by means of induction.

```
1  corrected_up_to_j(s: STC, j: below[n]): RECURSIVE STC =
2      TABLE
3                  +-------------------------------------------------------+
4                  |[ j = 0                 | j > 0                      ]|
5  +---------------+------------------------+----------------------------+
6  |not is_LEGT?(s)| locallyCorrected(s, 0)| locallyCorrected
7                                              (corrected_up_to_j
8                                                 (s, j - 1), j) ||
9  +---------------+------------------------+----------------------------+
10 |is_LEGT?(s)    | s                      | s                          ||
11 +---------------+------------------------+----------------------------+
12 ENDTABLE
13  MEASURE j
```

Listing 5.14: corrected_up_to_j returns a state that has j locally correct neighbors.

5.4.2 Convergence Proof

Based on the axiom of choice (in the PVS prelude library), we proved that the *other* function -Table 5.2 last row and Listing 5.11(Line 15)- will be able to choose a new correct color (Listing 5.15). Thus, applying the action of process j of a state s will guarantee the local correctness of its color i.e. the color will not have bad neighbors. Particularly, the color of the state LocallyCorrected(s,j) will always have a good neighborhood at process j. More precisely, we verified the following three lemmas in PVS:

```
nonempty_choice_set: lemma
  nonempty?(bad_nbr_s(K(sl, j)))⇒
    nonempty?(difference(fullset_colors, nbr_colors(K(sl, j))))

other_can_choose : lemma
   nonempty?(bad_nbr_s(K(sl, j))) IMPLIES
     empty?({C: ndx_varb| is_bad_nbr?((other(K(sl, j)), j), C)})

corrected_color_at_proc_i_has_no_bad_nbr1: Theorem
                          nbr_is_good?(locallyCorrected(s, j), j)

%|- corrected_color_at_proc_i_has_no_bad_nbr1: PROOF
%|-(lemma "other_can_choose")(skosimp*)(grind) Q.E.D
```

Observe that the mechanical proofs of the first two lemmas are a bit lengthy (about 70 proof steps combined). Nevertheless, they are based on two simple facts. Namely, there are m colors, and for a corrupted process j the neighborhood (i.e., $j\ominus 1$,j ,$j\oplus 1$) can not hold more than 2 distinct colors. Thus, by the non-emptiness of the choice set (the first lemma), the second lemma is a consequence of the axiom of choice (Listing 5.15 Lines 29–30).

```
Coloring_n_m.other_can_choose: proved - complete [shostak](0.43 s)
1  ("" (lemma "nonempty_choice_set")
2     (skosimp*)
3  (case "member(other(K(sl!1,j!1)),difference(fullset_colors,
   nbr_colors(K(sl!1, j!1))))")
4   ("1"
:     :
24  ("2"
25   (inst -1 " j!1" "sl!1")
26 (musimp)
27 (expand "member" 1)
28 (expand "other" 1)
29 (lemma "epsilon_ax[COLORS]")
30 (lemma "choose_is_epsilon[COLORS]")
31 (inst -1
32 "difference(fullset_colors, nbr_colors(K(sl!1, j!1)))")
33 (inst -2
34 "difference(fullset_colors, nbr_colors(K(sl!1, j!1)))")
35 (expand "nonempty?" -4)
36 (expand "empty?" 2)
37 (skosimp*)
38 (musimp)
39 (inst 1 "x!1")
40 (assert))))
```

Listing 5.15: Other can choose proof

Furthermore, it is possible that the new state i.e. *LocallyCorrected(s,j)* to be a legitimate state or an illegitimate one. If it was legitimate the proof is complete as we can easily define a transition from outside I to I namely (s,LocallyCorrected(s,j)). But if it was illegitimate at a neighbor of j, say the right neighbor $j+1$, then $j+1$ must be corrupted from its right (i.e. the color at j+2) as its left neighbor's color is correct by the previous move. The action will choose a new color that is different than both of its neighbors which guarantees the local correctness at $j+1$ and it preserves the correctness of the color at j, call the new state s2. If the new state s2 is legitimate then the prefix (s,LocallyCorrected(s,j),$s2$) reaches I, and the proof is complete. If it was not, then by induction and the fact that s has originally a finite length n, applying this correction procedure n-times recursively will ensure the convergence to I from any illegitimate state s. We prove this fact formally in two steps: First, we show that the state corrected_up_to_j(s1, j) has always good neighbors at process j for all $j < n$.

```
induct: THEOREM FORALL (j: below[n]):
                nbr_is_good?(corrected_up_to_j(s1, j), j)
```

The formal proof of this theorem is explained in Listing 5.16[3].
Second, based on the fact that a state that has good colors at all process j is legitimate, and by using the above induct theorem, we show that the state that is constructed after applying the recursion n times is legitimate Listing 5.17. We formalize this as follows:

```
s_has_all_good_nbr_is_LEGT: lemma
  (FORALL (j: below[n]): nbr_is_good?(s, j)) IMPLIES is_LEGT?(s)
     Proof: (grind) Q.E.D

constructed_LEGT: STC =
    (# length := n,
       seq := (LAMBDA (j: below[n]): corrected_up_to_j(s1, j)`seq
           (j)) #)

correcting_recursively_up_to_n_minus_1_constructs_LEGT: THEOREM
                  is_LEGT?(constructed_LEGT)
```

This ensures the convergence to I as required.[4]

[3] The full formal proof is available here **https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/coloring_ring_m_n**

[4] Appendix D has a proof of a generalized binary agreement protocol, it has similar proof construction of the coloring protocol.

```
(induct "j")
 (("1"
   (flatten)
   (expand "corrected_up_to_j")
   (case "is_LEGT?(s1) ")
   (("1" (assert) (grind))
    ("2"
     (case "NOT is_LEGT?(s1) ")
     (("1"
       (musimp)
       (lemma "corrected_color_at_proc_i_has_no_bad_nbr1")
       (inst?))
      ("2" (propax))))))
  ("2"
   (skosimp*)
   (expand 'corrected_up_to_j '1)
   (assert)
   (case "NOT is_LEGT?(s1)")
   (("1"
     (assert)
     (expand 'corrected_up_to_j '2)
     (musimp)
     (("1"
       (assert)
       (expand "nbr_is_good?")
       (expand "corrected_up_to_j")
       (expand "nbr_is_bad?")
       (lemma "corrected_color_at_proc_i_has_no_bad_nbr1")
       (inst?)
       (expand "nonempty?")
       (expand "nbr_is_good?")
       (expand "nbr_is_bad?")
       (expand "nonempty?")
       (lemma "corrected_color_at_proc_i_has_no_bad_nbr1")
       (inst -1 "1+jb!1 " "locallyCorrected(s1, 0)")
       (expand "nbr_is_good?")
       (expand "nbr_is_bad?")
       (assert)
       (grind))
      ("2"
       (lemma "corrected_color_at_proc_i_has_no_bad_nbr1")
       (inst -1 "1+jb!1 "
        "locallyCorrected(corrected_up_to_j(s1, jb!1 - 1), jb!1)")
       (typepred "jb!1" 1)
       (grind))))
    ("2" (grind))))))
```

Listing 5.16: Proof of induct theorem for the coloring protocol.

```
1  constructed_is_LEGT: THEOREM is_LEGT?(constructed_LEGT)
2 (lemma "induct_corrected_up_to_n_minus_1_is_LEGT")
3  (lemma "s_has_all_good_nbr_is_LEGT ")
4  (expand "is_LEGT?")
5  (skosimp*)
6  (expand 'nbr_is_good? '1)
7  (expand 'nbr_is_bad? '-3)
8  (expand 'nonempty? '-3)
9  (expand 'K '1)
10 (case-replace
11  "constructed_LEGT`seq(j!1)= corrected_up_to_j(s1, j!1)`seq(j!1)")
12 (("1"
13   (inst -3 "j!1")
14   (expand "nbr_is_good?")
15   (expand "nbr_is_bad?")
16   (grind))
17  ("2" (grind)) ("3" (grind))))
```

Listing 5.17: Proof of the coloring convergence protocol.

Chapter 6

Parameterized Strongly Self-Stabilizing Protocols

This chapter presents two case studies on the *synthesis in small scale and generalization* of strongly self-stabilizing sorting algorithms presented in [17] using the idea of convergence stairs. For instance, Section 6.1 and 6.2 include the formal implementation and verification of on Unidirectional Chains and Ring topologies respectively. Moreover,we explain the reusability of the verification of Add_Weak presented in the last chapter on the verification of these two cases. [1].

6.1 PSS Sorting on Unidirectional Chains

Section 6.1.1 introduces the sorting algorithm of [17], and Subsection 6.1.2 presents its formal specification in PVS. Subsection 6.1.3 represents the notion of convergence stairs, which we shall use to prove the strong convergence of sorting on unidirectional chains. Finally, Subsection 6.1.4 provides a mechanical proof for the correctness of the sorting algorithm for arbitrary number of processes n and variable domain m, where $m \leq n^2$.

6.1.1 Synthesized Sorting Algorithm

Klinkhamer and Ebnenasir use the Protocon tool [17] to automatically generate a self-stabilizing sorting algorithm on a unidirectional chain topology. Protocon implements a sound and complete backtracking algorithm (presented in [40]) for the synthesis of

[1] In preparation for submission to the Journal of Formal Methods in System Design

[2] For full PVS specifications and proofs visit `https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/chain-sort-m-n`

self-stabilizing protocols. Consider a version of the chain sort with 5 processes; i.e., $n = 5$. Each process p_j has a variable c_j, where $0 \leq j \leq 4$, with the domain of $\{0, 1, 2, 3\}$; i.e., $m = 4$. We use the notation *ChainSort*(m, n) to represent the instances of the chain sort for specific values of m and n. As such, the set of variables of *ChainSort*$(4, 5)$ equals to $\{c_0, c_1, ..., c_4\}$. Each process p_j, where $0 \leq j \leq 3$, can read and write $\{c_j, c_{j+1}\}$. The action of the process p_j $(0 \leq j \leq 3)$ is as follows: (Notice that, the last process does not have any actions.)

$$A_j : \quad (c_j > c_{j+1}) \quad \longrightarrow \quad c_j := c_{j+1}; \;\; c_{j+1} := c_j; \qquad (6.1)$$

If the variable c_j has a greater value than its right neighbor, then the process p_j swaps the values of c_j and c_{j+1}. The set of legitimate states of the unidirectional chain sort is the state predicate $I_{ChainSort}$, where $I_{ChainSort} = \{s \mid \forall i : 0 \leq i < n-1 : c_i(s) \leq c_{i+1}(s)\}$. Since this sorting protocol has been automatically generated by Protocon, it is strongly self-stabilizing up to a fixed number of processes. To prove that this protocol is correct for larger values of n, we use the theorem prover PVS to formally verify that the chain sort is correct for arbitrary (but finite) values of m and n, where $m \leq n$.

6.1.2 Specification of Chain Sort in PVS

This section demonstrates how we reuse the abstract concepts (e.g., state, state predicate) defined in the PVS specification of Add_Weak to specify the chain sort protocol. Let s be a global state of the chain sort. We model s as a finite sequence of length n with elements from the domain below[m], where below[m] in PVS denotes values in modulo m. Thus, a legitimate state s is a sorted sequence; i.e., $s(i) \leq s(i+1)$ for all $0 \leq i < n-1$. We specify the following function of Table construction in PVS to capture the above action:

```
1  action(s:{state:Glob_State
2                    | not is_LEGT?(state)},
3            j:subrange(0,n-2)): Glob_State =
4            TABLE
5                 +-------------------------------------------------+
6                 | active_LocallyCorrupted?(s,j)    | swap(s,j) ||
7                 +-------------------------------------------------+
8                 |is_LEGT?(s)                       |           ||
9                 +-------------------------------------------------+
10                |not active_LocallyCorrupted?(s,j) |           ||
11                +-------------------------------------------------+
12           ENDTABLE
```

Listing 6.1: action of generalized Chain Sort protocol using PVS Table construction.

where active_LocallyCorrupted is a predicate that returns true if the j-th value in the sequence corresponding to s is greater than the $(j+1)$-th value.

active_LocallyCorrupted?(s:Glob_State, j:subrange(0,n-2)):bool = s`seq(j) > s`seq(j+1)

To ease the presentation of mechanical verification, we define the following notations. Let *Range*(s) be a function that maps a global state s to a subset of the set $\{0, ..., m-1\}$ such that *Range*(s) contains all numbers that appear in the sequence corresponding to s.

Definition 6.1.1. *Range:* Glob_State → set[below[m]], *where* *Range*(s) = {x : below[m] | x = s`seq(i) *for some* i: below[n]}.

Let $min_s(0)$ denote the minimum value that appears in the global state s. That is, $min_s(0)$= min[*Range*(s)]. We also define $Range_s(0)$= *Range*(s). Moreover, let *remove*(α,B) denote a function that removes an element α from a set B and returns the resulting set; i.e., *remove*(α,B) = B-{α}. We define $Range_s(1)$= *remove*($min_s(0)$, *Range*(s)). Inductively, we define the following concepts:

Definition 6.1.2. $\mathsf{Range}_s(i)$ = remove(min[$\mathsf{Range}_s(i-1)$], $\mathsf{Range}_s(i-1)$)

When the cardinality of $\mathsf{Range}_s(i-1)$ is 1, the remove function returns the empty set and PVS generates a TCC. To resolve this TCC, we define the function $\mathsf{Range}_s(i)$ to return Max($\mathsf{Range}_s(i)$.

Definition 6.1.3. $\mathsf{min}_s(i)$= min[$\mathsf{Range}_s(i)$], *the minimum value in the set* $\mathsf{Range}_s(i)$.

Since repeated values in s are allowed, we define the multiplicity of a natural value t in a state s by the number of times that t occurs in s denoted by multi(t)= l_k, where k is a positive integer. Notice that, in this notation, we have multi($\mathsf{min}_s(0)$)= l_0. We demonstrate these concepts with the following simple example.

Example 6.1.1. *Let* $s = (0,0,1,1,1,3)$ *then* Range(s)=*{0,1,3}*, Range_s*(1)= {1,3}, and* Range_s*(2)={3}. Moreover, we have* $\mathsf{min}_s(0)$= *0,* multi($\mathsf{min}_s(0)$)= l_0 = *2 since we have two zeros in s, whereas* multi*(1)*=l_1=*3, and* multi*(3)*=l_2 = *1. Further, we have* $\mathsf{min}_s(1)$= min($\{0,1,3\} - \{0\}$)= *1, while* $\mathsf{min}_s(2)$= min($\{0,1,2\} - \{0,1\}$)= *3.*

In order to define multi(t) more precisely, we first define the function index_multi that returns the set of indices of all occurrences of t in s.

Definition 6.1.4. indexed_multi(t:below[m]) = *If* $t \in$ Range(s) ⇒ { i: below[n]| s`$seq(i) = t$}, *else* ϕ.

Now, we simply define the multiplicity multi(t) as the cardinality of the set indexed_multi(t), denoted by multi(t)= Card(indexed_multi(t)).

Example 6.1.2. *Assume t is not in the range of s, which means there is no index i below* s'length *such that s'$seq(i) = t$. As such,* indexed_multi(t) *is the empty set ϕ; hence* multi(t) *= 0. Moreover, in Example 6.1.1, we have* indexed_multi(0)*= {0,1} since the first two positions of s have zeros. Thus,* Card(indexed_multi(0)) *= 2. Similarly,* indexed_multi(1) *= {2,3,4} and thus* Card(indexed_multi(1)) *= 3. Further, we have* indexed_multi(3) *= {5} and thus* multi(3) *= 1.*

Finally, we define the function pos_multi(i), where $i \in$ below[Card(Range(s))], that returns the summation of all multiplicities from $\mathsf{min}_s(0)$ up to $\mathsf{min}_s(i)$.

Definition 6.1.5. *For $i <$* Card(Range(s))*:*

$$pos\_multi(i) = \begin{cases} multi(min_s(0)) & i = 0 \\ multi(min_s(i)) + pos\_multi(i-1) & else \end{cases}$$

6.1.3 Convergence Stairs for Proving Strong Convergence

Gouda and Multari [42] show that a protocol p is strongly self-stabilizing to a state predicate I iff there is a sequence of state predicates $S_1, \cdots S_N$ such that the following properties hold for some $N > 0$:

1. *Boundary*: $S_1 = true$ and $S_N = I$;
2. *Closure*: Each S_i is closed, where $1 \leq i \leq N$, and
3. *Convergence*: Each S_i converges to S_{i+1}, for $1 \leq i < N$.

We adopt a proof strategy based on the idea of convergence stairs. Specifically, to prove closure and strong convergence of the chain sort, we define the suitable stairs and then prove the closure of each stair and the convergence of stair S_i to S_{i+1}, for all $1 \leq i < N$. For a global state s to be in the j-th stair (i.e., S_j), the predicate sorted$_j$?(s, pos_multi(j)-1) must evaluate to *true*, which means s is sorted up to the position pos_multi(j)-1, where $1 \leq j < N$. Nonetheless, this is insufficient alone to define convergence stairs since the closure property may not be satisfied! This phenomenon could occur if the sub-sequence 1 to pos_multi(j)-1 is sorted, but does not contain the first pos_multi(j)-1 smallest values. Thus, we add the condition $\forall i : 0 \leq i \leq j$: s(pos_multi($i$)-1) = $\mathsf{min}_s(i)$ to the definition of the j-th stair. Formally, we define the following predicate:

Definition 6.1.6. n_Stair?*(s, j:*below[Card(Range(s))]*): bool =* sorted$_j$?(s,pos_multi(j) *- 1)* $\cap \forall\ 0 \leq i \leq j$*:* s(pos_multi(i)-1) = min_s(i)

Example 6.1.3. *Let s be an illegitimate state say $s = (1,1,0,2,4,5,3,3,0,1)$. To compute* n_Stair*?$(s,3)$, we first find the summation of the multiplicities $l_0+l_1+l_2 =$* pos_multi*(3). In this case, it is equal to 2+3+1=6. Then s must be sorted up to the position 6-1; i.e., the first 6 elements in s must be sorted. That is,* sorted$_j$*?$(s$, 6-1) holds. Moreover, $\forall i : 0 \leq i < 3$: s(*pos_multi*(i)-1) = $min_s(i)$. This means, for instance, if s is in* n_Stair*?$(s,3)$ then at positions l_0, l_1, l_2 it must have the values $min_s(0)$, $min_s(1)$, $min_s(2)$ respectively. For example,* n_Stair*?$(s,3)$ is given as follows:*

n_Stair*?$(s,3)$={(0,0=x_{2-1},1,1,1=x_{5-1},2=x_{6-1},$\cdots$,x_9) | x_i= 3,4 or 5 for i > 6-1}*

6.1.4 Mechanical Proof of Closure and Convergence

The mechanical verification of these properties in PVS consumed 1706 proof command including all the TCC and all required lemmas, this follows the very lengthy inductive proofs that were required. Thus we will only explain the main ideas of the proofs in this sections, [3].

In order to mechanically prove the strong convergence of the chain sort, we use Definition 6.1.6 and show that the sequence of predicates introduced in this definition creates the necessary convergence stairs for proving the strong convergence of *Chain_Sort*(m,n) in a parametric sense. We first provide helpful lemmas and theorems that state important properties of some of the basic concepts and functions introduced in Section 8.2.

Remark 1. To simplify the mechanical proofs, we separate the trivial cases of Card(Range(s)) $= 1, 2$. Thus, the global state s that is passed as the first parameter to the predicates n_Stair? and sorted? meets the requirement that Card(Range(s))> 2.

Lemma 6.1.1. ∀ j:below[Card(Range(s))]: (n_Stair?(s, j) ⇒ sorted$_j$?(s, pos_multi(j)-1))

Lemma 6.1.1 shows that if a state s belongs to the j-th stair, then the sequence of numbers representing s must be sorted up to the position pos_multi(j)-1.

Lemma 6.1.2. ∀ (j:below[Card(Range(s))]): (n_Stair?(s, j) ⇒ ∀ (i:subrange[0,j]): s‘seq(pos_ multi(i)-1) = min$_s(i)$)

Lemma 6.1.2 requires that if the global s is in the j-th stair, then each value in position j of the sequence representing s must be the $(j+1)$-th smallest value of the sequence. The proofs of Lemmas 6.1.1 and 6.1.2 follow by definition; hence omitted. The following theorem states that min$_s$(i-1)< min$_s$(i).

[3] The full formal proof is available here `https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/chain-sort-m-n`

Theorem 6.1.1. *For any* $0 < i <$ Card(Range(s))*, we have* Card(Range$_s$*(i-1))>1* $\Rightarrow$ min(Range$_s$(i-1)) $<$ min(Range$_s$(i))

Proof. For $i = 1$ the proof is trivial since Range$_s(0)$= Range(s) and Range$_s(1)$ = remove(min(Range(s)), Range(s)), which is a proper subset of Range$_s$(0) that excludes min(Ran- ge(s)). Thus, min(Range$_s(0)$) $<$ min$_s(1)$. For $i > 1$, the proof splits into two cases: for any $i > 1$, either Card(Range$_s(i-1)$) > 1 holds or not. If Card(Range$_s(i-1)$) > 1 does not hold for some $i > 1$, then Card(Range$_s(i-1)$) $= 1$ and by definition, Range$_s(i)$ is max(Range(s)); hence certainly everything is less than max(Range(s)). Otherwise, Range$_s(i)$ = remove(min(Range$_s(i-1)$), Range$_s(i-1)$), which is a proper subset of Range$_s(i-1)$. Therefore, we have min(Range$_s(i-1)$) $<$ min(Range$_s(i)$), which completes the proof. □

We also show that the sum of multiplicities of the values in the state s preserves the order of the values with respect to the ordering relation $<$. For instance, since min$_s(j-1) <$ min$_s(j)$, we have pos_multi(j-1) $<$ pos_multi(j).

Lemma 6.1.3. *(*$j_2 = j_1 + 1 \wedge j_2 <$ Card(Range(s))*)* $\Rightarrow$ pos_multi(j_1) $<$ pos_multi(j_2).

Proof. The proof is by induction on j_1. For $j_1 = 0$, the proof is straightforward. Since by definition we have pos_multi(1)= multi(1) + pos_multi(0) and pos_multi(0)= multi(0), the proof of the case $j_1 = 0$ follows. The proof becomes complete by induction hypothesis, a careful rewriting, expansions, if-lifting, and instantiations as in the base case. □

Theorem 6.1.2. *(*$j_2 = j_1 + 1 \wedge j_2 <$ Card(Range(s)) $\wedge$ sorted$_j$?(s, pos_multi(j_2)))* $\Rightarrow$ sorted$_j$?(s, pos_multi(j_1))

Proof. Again, the proof of this theorem is by induction on j_1 starting from $j_1 = 0$. The base case is trivial; hence omitted. For $j_1 > 0$, Lemma 6.1.3 implies that if $j_2 = j_1 + 1$ and $j_2 <$ Card(Range(s)), then we have pos_multi(j_1) $<$ pos_multi(j_2). Let pos_multi(j_1) $= k_1$ and pos_multi(j_2) $= k_2$. We then prove that, for $k_1, k_2 > 1$, if $k_2 > k_1$, then we have sorted$_j$?(s, k_2) $\Rightarrow$ sorted$_j$?(s, k_1), which is achieved by simple skolemization, expansion and instantiation. □

It is worth mentioning that Theorem 6.1.2 has a simple proof, which would not have been possible without reusing Lemma 6.1.3. We also show that the stairs are nested. In particular, if $j_1 < j_2$ and s is in the j_2-th stair, then s must be in the j_1-th stair as well.

Theorem 6.1.3. *(*$j_2 = j_1 + 1 \wedge j_2 <$ Card(Range(s)) $\wedge$ n_Stair?(s, j_2)*)* $\Rightarrow$ n_Stair?(s, j_1)

Proof. To prove this theorem, we expand the definition of n_Stair?, and use Lemma 6.1.2, Theorem 6.1.2, and Lemma 6.1.3 along with instantiations, which completes the proof. □

This theorem is very important since it shows that if the two conditions of Definition 6.1.6 are satisfied for j_2, then they must hold for all $j < j_2$. In fact, to simplify the proof of this theorem, we found it useful to prove the first sorting property of Definition 6.1.6 independently, and then we reused it in the proof of Theorem 6.1.3. Now, we show that if s is in the j-th stair, then for all $i <$ pos_multi$(j)-1$ the actions of the process i will not change the values on s for all $i <$ pos_multi$(j) - 1$.

Lemma 6.1.4. $\neg$is_LEGT?$(s_0) \ \wedge \ j_1 <$ pos_multi$(j_2) - 1 \wedge$ n_Stair?$(s_0, j_2) \Rightarrow$ action$(s_0, j_1) = s_1 \ \wedge \ s_1$`seq(j$_1$) $= s_0$`seq(j$_1$)

Finally, the following theorem shows that the difference between pos_multi(j_2) and pos_multi (j_1) is multi(min$_s(j_2)$).

Theorem 6.1.4. *(*$j_2 = j_1$+1 $\wedge$ $j_2 <$ Card(Range(s))) $\Rightarrow$ pos_multi*(*j_2*)* - pos_multi*(*j_1*)* $=$ multi(min$_s(j_2)$)

Proof. By expansion and rewriting the definition of pos_multi, we get multi(min$_s(j_2)$) + pos_multi($j_2 - 1$)-pos_multi (j_1). Now, substitution will give us multi(min$_s(j_2)$) + pos_multi (j_1) - pos_multi(j_1), which is equal to multi(min$_s(j_2)$). In PVS, automatic expansion, rewriting, instantiations, with if-lifting will complete the proof. □

Theorem 6.1.5. *Each stair defined for Chain_Sort(m, n) (in Definition 6.1.6) is closed in the synthesized action presented in Section 6.1.1.*

Proof. For any $1 \leq j_1, j_2 < N$, where $j_1 < j_2$, and for a global state s in the j_2-th stair, the number of values between the positions pos_multi(j_2) and pos_multi(j_1) is equal to multi$(min_s(j_2))$ (based on Theorem 6.1.4). Moreover, Theorem 6.1.3 implies that we have the value of $min_s(j_2)$ at position pos_multi$(j_2) - 1$ and the value (min$_s$(j$_1$)) at position pos_multi$(j_1) - 1$. As such, any occurrence of $min_s(j_2)$ must be located between the two portions as explained in Fig(6.1) because it is sorted up to pos_multi$(j_2) - 1$. Thus, by Theorems 6.1.4, 6.1.2,6.1.3 and 6.1.1, and Lemmas 6.1.1 and 6.1.2, we conclude that all occurrences of min$_s(j_2)$ are in positions pos_multi(j_1) up to pos_multi$(j_2) - 1$. For this reason, all values in positions $k >$ pos_multi(j_2) cannot be less than min$_s(j_2)$. Thus, the state s could be locally corrupted only for $k >$ pos_multi(j_2). Therefore, applying the action of process k on s does not change the order of the values up to pos_multi$(j_2) - 1$; hence, the closure of the i-th stair, where $\forall \ 0 \leq i \leq$ Card(Range(s))-1. □

It remains to show the convergence of the stairs defined in Definition 6.1.6. First, we prove that for a global state s, if s is in the (Card(Range(s))-1)-th stair, then s is a legitimate state. Formally, we have:

Theorem 6.1.6. n_Stair?(s, Card(Range(s))-1) $\Rightarrow$ is_LEGT?(s)

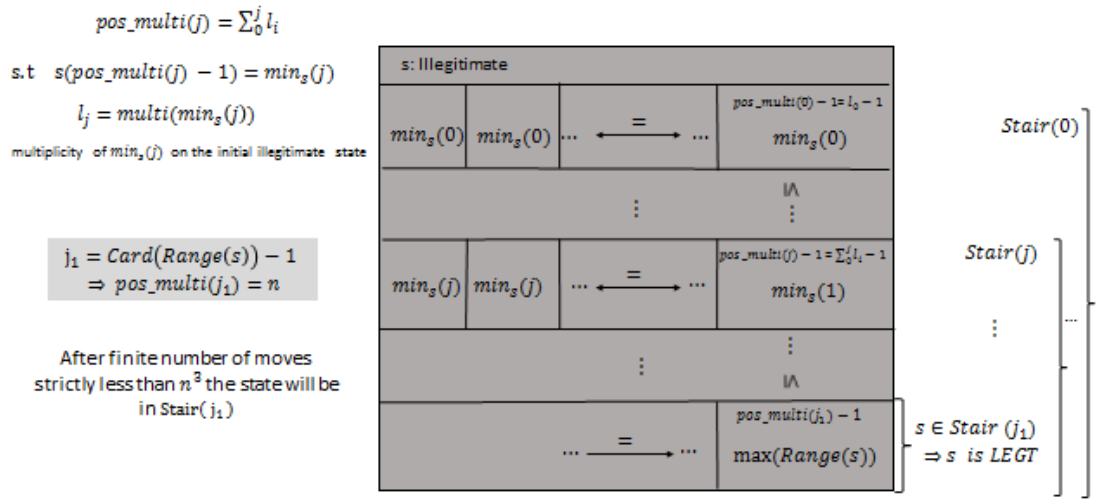

Figure 6.1: Chain_Sort(m, n) proof of convergence using convergence stairs.

Proof. Let n be the length of the sequence of values representing s. By Lemma 6.1.1 and the fact that the summation of all multiplicities of the values in Range(s) is n, we state that if n_Stair?(s, Card(Range(s))-1) holds, then sorted$_j$?(s, n-1) holds as well. Thus, the state s must be sorted up to pos_multi(Card(Range(s))) $= n - 1$. If sorted$_j$?(s, n-1) holds, then s is a legitimate state. □

Lemma 6.1.5. ¬is_LEGT?(s_0) ∧ active_LocallyCorrupted?(s_0, j) ⇒ action$(s_0, j) = s_1$ ∧ s_1‘seq$(j) = s_0$‘seq$(j+1)$ ∧ s_1‘seq$(j+1) = s_0$‘seq(j)

Proof. The proof is completed by expanding the definition of action, swap, is_LEGT?, active_LocallyCorrupted?. □

Notice that, the function action(s, j) returns a global state generated by the execution of the synthesized action in position j of the sequence of values representing s. As such, action(s, j)‘seq(j) is the j-th value in the sequence of values representing the global state action(s, j). Lemmas 6.1.5 simply states that if process j in the chain is enabled and executes its action, then in the resulting global state the values j and $j + 1$ are swapped.

Lemma 6.1.6. ¬is_LEGT?(s_0) ∧ active_LocallyCorrupted?(s_0, j) ⇒ action$(s_0, j) = s_1$ ∧ s_1‘seq$(j) < s_1$‘seq$(j + 1)$

Proof. The proof follows directly from expanding the definitions of action, swap, is_LEGT?, active_LocallyCorrupted? and Lemma 6.1.5. □

Lemmas 6.1.6 stipulates that if process j in the chain is enabled and executes its action, then in the resulting global state the value at position j will be strictly smaller than the value at position $j+1$.

Theorem 6.1.7. *Based on Definition 6.1.6, each stair j converges to stair $j+1$, for $1 \leq j < \mathsf{Card}(\mathsf{Range}(s)) - 1$.*

Proof. We show that $\forall\ 0 \leq j \leq \mathsf{Card}(\mathsf{Range}(s)) - 1$: the j-th stair converges to the $(j+1)$-th stair. First, we make the following observations:

1. If s is an illegitimate state and locally corrupted at position i (i.e., the local state of process i is corrupted), then s‘seq(i) > s‘seq(i+1). Thus, the action of process i will swap the values s‘seq(i) and s‘seq(i+1). As a result, the position of the smaller value will decrease strictly by 1 in the new state (based on Lemmas 6.1.5 and 6.1.6).

2. Let s be an illegitimate state that is sorted in a non-increasing order (i.e., worst case input for a sorting algorithm that sorts increasingly). Let K be the number of moves/actions that are enabled at s. We observe that the number of enabled actions in any other illegitimate state cannot be greater than K.

Let $\mathsf{Chain\_Sort}(m, n)$ be an instance of the chain sort protocol with n processes in the chain and m values in the domain of variables, where $m \leq n$. Moreover, let s be a global state that is ordered in the opposite direction (i.e., the minimum value is on the right most position in the sequence representing s). Since $m \leq n$, we may assume that there are at most n *distinct* values in s, starting from 0 ending at $n-1$. Thus, the multiplicity of each value in Range(s) is 1 in s. Now, since the sequence of s is sorted decreasingly, $min_s(0)$ is located at position $n-1$. By Lemmas 6.1.5 and 6.1.6, when process $n-1$ executes, the position of $min_s(0)$ will decrease strictly by 1. Thus, for $min_s(0)$ to move to position 0, there must be $n-2$ moves. After this occurs, process 0 will not be locally corrupted anymore (based on Theorems 6.1.1 and 6.1.5). That is, convergence from the first stair to the second is achieved in at most $n-2$ steps. Based on a similar argument, it takes $n-3$ moves for $min_s(1)$ to move to position 1. In other words, convergence from the second stair to the third one is attained in $n-3$ steps in the worst case. In general, the number of moves for placing $min_s(i)$ in its right position (i.e., convergence from $(i+1)$-th stair to $(i+2)$-th stair) is $(n-2-i)$, where $0 \leq i \leq \mathsf{Card}(\mathsf{Range}(s)) - 1$. Based on Theorem 6.1.1, we know that $\mathsf{min}_s(i) < \mathsf{min}_s(i+1)$ holds. Therefore, based on the above argument (which follows from Theorem 6.1.5 and Lemmas 6.1.5 and 6.1.6), each stair converges to its consequent stair, and convergence to legitimate states can be achieved in at most $(n^2-n)/2$ steps (see Figure 6.1). □

To illustrate the proof, we provide an example.

Example 6.1.4. *Consider the* Chain_Sort$(6,6)$*, and let* $s = (5,4,3,2,1,0)$*. Then* s‘seq*(5) = 0,* Range(s)*=\{ 0, 1,* $\cdots$*, 5\} and* multi$(i) = 1$ *for each* i *in* Range(s)*. All processes except* $n-1 = 5$ *are locally corrupted. Thus, the swapping action will make* s‘seq*(4) equal to 0. Hence, the position of 0 must decrease strictly by one; i.e., from 5 to 4. Since 0 is the minimum in* Range*(s), any process that has 0 as its right neighbor will be locally corrupted. Thus, its swapping action will make the new position of 0 less than the previous position by 1. Swapping of 0 will continue until* s‘seq(0) *= 0; i.e., 0 does not have a left neighbor. The same argument applies to 1 in* Range*(s)-\{0\} (since 1 is the minimum value of* Range*(s)-\{0\}). Moreover, once 0 reaches position 0, s will be in the second stair, where it will not change its place due to the closure of the stairs. This means that the corrupted processes must be to the right of 0; i.e., process(j) is corrupted implies* $j > 0$*. Since the multiplicities are equal to 1, the values that are located at positions* $j > 0$ *are greater than 0. Now, any process* $j > 1$ *that has 1 as its right neighbor is locally corrupted and starts the wave of swapping. The swapping continues until 1 has a left neighbor less than it, which is 0. At this point, s is sorted increasingly up to the second position; i.e., s is in the stair 2. In general, the same phenomenon occurs with all values on s because (1)* $\mathsf{min}_s(i-1) < \mathsf{min}_s(i)$*, (2) the stairs are closed, (3) the actions reduce the positions strictly by 1, and (4) multiplicities are equal to 1. Thus, s will be sorted in finite number of moves; i.e., convergence is achieved.*

6.2 PSS Sorting on Unidirectional Rings

In this section, we consider a special case of the Ring sort where there exist a unique zero. The topology of the network is a unidirectional ring instead of a unidirectional chain. That is, we have a protocol *Ring_Sort*(m,n), where n denotes the number of processes and m represents the domain of each variable and $m = n$. Consider a version of the Ring sort with 5 processes; i.e., $n = 5$. Each process p_j has a variable c_j, where $0 \leq j < 5$, with the domain of $\{0,1,2,3,4\}$; i.e., $m = 5$. Moreover, each process p_j, where $0 \leq j \leq 4$, can read and write $\{c_j, c_{j+1}\}$, where addition is in modulo 5. We assume that the multiplicity of the value 0 is 1; i.e., there is exactly one 0 in the ring. We call the process that holds the 0, the 0-process. Klinkhamer and Ebnenasir [17] automatically generate the following action ($0 \leq j \leq 4$) for the *Ring_Sort*(m,n), where $m = 5$ and $n = 5$.

$$A'_j: \quad (c_{j+1} \neq 0) \wedge (c_j > c_{j+1}) \quad \longrightarrow \quad c_j := c_{j+1};\ c_{j+1} := c_j; \quad (6.2)$$

If the variable c_j has a greater value than its right neighbor and the value of the

right neighbor is non-zero, then the process p_j swaps the values of c_j and c_{j+1}. Notice that, contrary to the chain sort, the all processes can see their right neighbors. Nevertheless, the 0-process will be always silent since 0 is the smallest number. Moreover, the set of legitimate states of the unidirectional ring sort is the state predicate $I_{RingSort}$, where $I_{RingSort} = \{s \mid \forall i : 0 \leq i \leq n-1 : (c_i(s) \leq c_{i+1}(s)) \vee (c_{i+1}(s) = 0)\}$. Thus, PVS specifications of the action, the states, and the legitimate states are similar to the chain sort, as in List 6.2; hence we omitted them.

The generalization of the ring sort protocol is as follows: There are n processes located in a unidirectional ring, each having a local variable c_j with a domain of m values, where $m \leq n$. Each process j can read and write the values $\{c_j, c_{j+1}\}$ and includes the action A'_j, for $0 \leq j \leq n-1$. The *Ring_Sort*(m, n) protocol has three constraints. First, there is exactly one 0 in any global state s. We call the process that holds 0, proc(0). Second, the action of the predecessor of the process that initially holds 0 is disabled. In particular, 0 represents a mark that labels the beginning and the end of the sorting procedure. Third, the $(n-1)$-th process (to the right of proc(0)) does have an action (contrary to the chain sort where process $n-1$ excludes any actions). Since $c_0 = 0$, this action becomes disabled and the entire ring sort turns into an instance of chain sort. In the case where $m = n$, we can represent each legitimate state s as a sequence of values from {0, 1, $\cdots$, n-1} where s‘seq(0)=0 and the rest of the values to right of 0 are sorted increasingly. This in turn makes the proof very similar to the proof of convergence from a state in the first stair of the chain sort! Thus, we can reuse the mechanical verification of the chain sort.

6.2.1 PVS Specification and Verifications of Ring Sort

In this section, we present the proof of generality of *Ring_Sort*(m, n), where $m = n$. The PVS specification of *Ring_Sort*(n, n) greatly benefits from the PVS specification of *Chain_Sort*(m, n). Importing *Chain_Sort*(m, n) significantly simplifies the specification and verification of *Ring_Sort*(n, n) as shown in Listing 6.2. For instance, to prove the following two lemmas, we import the corresponding lemmas from chain sort.

Lemma 6.2.1. ($j_2 = j_1 + 1 \wedge j_2 <$ Card(Range(s))) $\Rightarrow$ pos_multi(j_2) - pos_multi(j_1) = 1

Lemma 6.2.2. $\forall$(j:below[Card(Range(s))]):n_Stair?(j) $\Rightarrow \forall$ (i$\in$[0,j]):s‘seq(pos_multi(i) - *1)* = i

The proof of Lemma 6.2.1 reuses Theorem 6.1.4 while the proof of Lemma 6.2.2 directly depends on Lemma 6.1.2. The mechanical proof follows after some simple instantiations and applying the constraints of *Ring_Sort*(n, n) on s.

Theorem 6.2.1. *Ring_Sort*(n, n) *is convergeing for any* $n \geq 5$.

Or as shown in Listing 6.2 in PVS language.[4]

```
1  Ring_n_converge0[m1: upfrom[4], n1: upfrom[5]]: THEORY
2  BEGIN

4  ASSUMING
5  Ring_n_converge: ASSUMPTION m1 = n1
6  IMPORTING Chain_Sort_m_n[m1, n1]
7  j1, j2, i3: VAR below[Card(Range(fs))]
8  Ring_n_converge1: ASSUMPTION
9  Card(Range(fs)) = n1 ∩ (member(i3, Range(fs)) ⇒ multi(i3) = 1) ∩
10 Range(fs) = fullset[below[n1]] ∩ min_s(i3) = i3 ∩ fs`seq(0) = 0
11 ENDASSUMING

13  :
14 Stair_converges11: THEOREM
15  n_Stair?(Card(Range(fs)) − 1) IMPLIES is_LEGT?(fs)

17 END Ring_n_converge0
```

Listing 6.2: Specification of Ring Sort Assumptions when m = n.

[4] The full formal proof is available here `https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/chain-sort-m-n`

Chapter 7

Extensions, Conclusions and Future Work

7.1 Discussion and Related Work

This section discusses the impact of the proposed approach and the related work.
Significance. Self-stabilization is an important property for networked systems, be it a network-on-chip system or the Internet. There are both hardware [27] and software systems [28] that benefit from the resilience provided by self-stabilization. Thus, it is important to have an abstract specification of self-stabilization that is independent from hardware or software. While several researchers [29, 30] have utilized theorem proving to formally specify and verify the self-stabilization of specific protocols, this paper presents a problem-independent specification of self-stabilization that enables potential reuse of efforts in the verification of convergence of different protocols. (As demonstrated by our case studies.)

One of the fundamental impediments before automated synthesis of self-stabilizing protocols from their non-stabilizing versions is the scalability problem. While there are techniques for parameterized synthesis [39, 43] of concurrent systems, such methods are not directly useful for the synthesis of self-stabilization due to several factors. First, such methods are mostly geared towards synthesizing concurrent systems from formal specifications in some variant of temporal logic. Second, in the existing parameterized synthesis methods the formal specifications are often parameterized in terms of local liveness properties of individual components (e.g., progress for each process), whereas convergence is a global liveness property. Third, existing methods often consider the synthesis from a set of initial states that is a proper subset of the state space rather than the entire state space itself (which is the case for self-stabilization). With this motivation, our contributions in this paper enable a hybrid method based on synthesis and theorem proving that enables the generalization of small instances of self-stabilizing protocols generated by our synthesis tools [17].

Related work. Kulkarni and Bonakdarpour's work [34, 35] is the closest to the proposed approach in this paper. As such, we would like to highlight some differences between their contributions and ours. First, in [34], the authors focus on mechanical verification of algorithms for the addition of fault tolerance to concurrent systems in a high atomicity model where each process can read and write all system variables in one atomic step. One of the fault tolerance requirements they consider is nonmasking fault-tolerance, where a nonmasking system guarantees recovery to a set of legitimate states from states reachable by faults and not necessarily from the entire state space. Moreover, in [35], Kulkarni and Bonakdarpour investigate the mechanical verification of algorithms for the addition of multiple levels of fault tolerance in the high atomicity model. In this paper, our focus is on self-stabilization in distributed systems where recovery should be provided from any state and high atomicity actions are not feasible.

Methods for the verification of parameterized systems can be classified into the following major approaches, which do not directly address SS systems. Abstraction techniques [44, 45, 46] generate a finite-state model of a parameterized system and then reduce the verification of the parameterized system to the verification of its finite model. Network invariant approaches [47, 48, 49] find a process that satisfies the property of interest and is invariant to parallel composition. Logic program transformations and inductive verification methods [50, 51] encode the verification of a parameterized system as a constraint logic program and reduce the verification of the parameterized system to the equivalence of goals in the logic program. In regular model checking [52, 53], system states are represented by grammars over strings of arbitrary length, and a protocol is represented by a transducer. Abdulla *et al.* [54] also investigate reachability of unsafe states in symmetric timed networks and prove that it is undecidable to detect livelocks in such networks. Bertrand and Fournier [55] also focus on the verification of safety properties for parameterized systems with probabilistic timed processes.

7.2 Conclusions and Future Work

We presented a novel method for the design of self-stabilizing parameterized network protocols. The proposed method is based on the philosophy of *synthesize in small scale and generalize*, which includes two components (see Figure 2.1), namely synthesis of small scale protocols and the proof of their generality. The synthesis component has been the focus of our previous work [26, 25, 14, 40]. This paper focused on exploiting theorem proving for the generalization of synthesized self-stabilizing protocols that are correct in a finite scope (i.e., up to a small number of processes). We specifically presented a mechanical proof for the correctness of the Add_Weak algorithm from [25] that synthesizes weak convergence. This mechanical proof provides a reusable theory in PVS for the proof of weakly stabilizing systems in general (irrespective of how they

have been designed). The success of mechanical proof for a small synthesized protocol shows the generality of the synthesized solution for arbitrary number of processes. We have demonstrated the proposed approach in the context of a graph coloring protocol (Section 5.3) and a binary agreement protocol (Section D.1). Moreover, we presented a method for mechanical verification of strong convergence and we demonstrated it for the design of a self-stabilizing sorting algorithm on rings and chains. These case studies illustrate how the PVS theory developed for weak stabilization can be reused for the verification of strong stabilization.

We will extend this work by reusing the existing PVS theories for mechanical proof of a complete backtracking algorithm (introduced in [40]) that synthesizes strongly stabilizing protocols in a small scope. We expect that the mechanical verification of this algorithm will greatly facilitate the design of parameterized self-stabilizing systems. We will also develop algorithms that will automatically generate Verification Conditions (VCs) from small synthesized solutions towards facilitating the process of generalization using theorem proving. Such algorithms will enable us to classify verification conditions into two categories: general and problem-specific. The general verification conditions specify constraints whose proofs can directly be reused from one problem to another. Examples of such VCs include the property of closure, where we must prove that a set of states is closed in a set of transitions. The problem-specific VCs (e.g., convergence stairs) depend on the problem at hand and should be generated from the small scale self-stabilizing solutions that the synthesizer generates. The problem-specific VCs will be used in mechanical proof of convergence (using theorem provers). In addition to the aforementioned extensions, we will continuously expand the repository of our case studies to more complicated protocols (e.g., leader election, maximal matching, consensus) with topologies other than ring or chain.

Part II

Rigorous Numerical Riemann Integral In PVS

Chapter 8

Introduction

INtegral calculations of real-valued functions are hard to handle in mechanical proofs. In this part, we provide formal verification of an abstract algorithm that is used to approximate definite Riemann Integral. We have verified the soundness of the algorithm using Prototype Verification System (PVS). The verification yields practical automated proof strategies to estimate definite Riemann Integral for a large class of real-valued integrable functions. We illustrate the application of the algorithm and the proof strategies in the context of continuous functions on $\mathfrak{R}$.[1]

8.1 Preface

There is a vital need of rigorous approximation for Riemann integral in wide range of applications of life-critical *cyber-physical* systems. For example, in air traffic management systems (Fig 8.1) given the complexity of the trajectories there are increasing demands for formal verification of real valued integral calculus, which is known to be a challenging task [6, 56, 57]. For instance, all trigonometric functions in PVS were constructed using Riemann Integral based on the formalization of arc-tangent function [6]:

$$atan(x) = \int_0^x \frac{1}{t^2+1} dt \tag{8.1}$$

These functions have a central role in air traffic management algorithms [6, 58]. Furthermore, aircraft trajectories are commonly modeled with differential equations for which Riemann Integral is part of their analytical solutions [6, 59, 58].

[1]The contribution of this part of the dissertation has become part of NASA PVS standard library, under the GNU GPL license, copyright(C)Michigan Technological University and Amer Tahat, innovation disclosure Tech ID# 1617.00. for Amer Tahat.`https://github.com/nasa/pvslib`

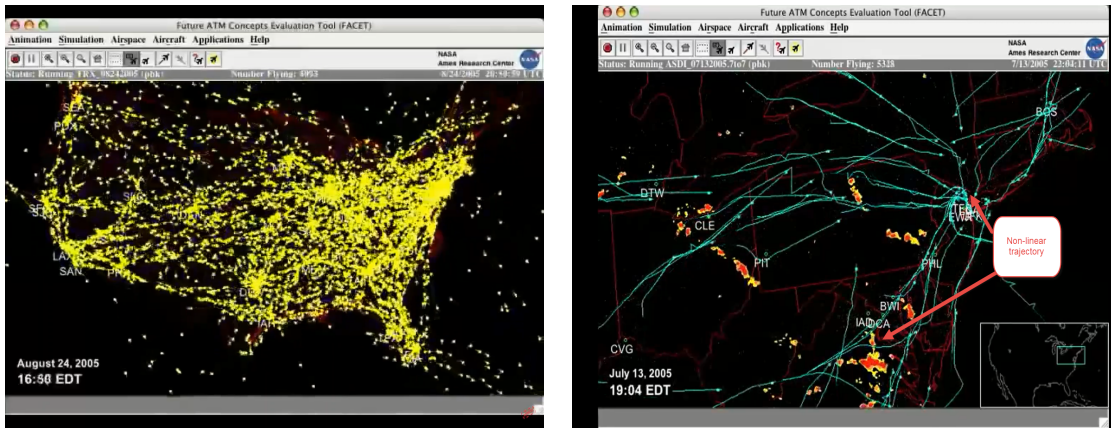

Figure 8.1: The complexity of air traffic management systems.[2]

In the last two decades, there has been a significant development in the mechanical formalization and verification of real-valued calculus in various verification systems [57]. For instance, fundamental analysis libraries for integral calculus were implemented in several theorem provers such as PVS [6, 60], HOL [61], Coq [62], Isabelle/HOL [63], and Isabelle/Isar [64]. In his landmark work [6], Butler provided a comprehensive discussion on the intricacy of the mechanical proofs that involve integral calculus. Supporting them with many examples of formulas of lengthy mechanical proofs of several hundred to a few thousand proof-commands. He spotted the light on two major reasons that lead to these hurdles in the mechanical proofs, unlike the manual proofs. First, working on partitions will force lengthy inductive proofs. Second, using type theory seeking consistency in place of set theory will complicate the formalizations of various functions and generate many proof obligations in terms of TCCs (type correctness conditions). Due to the same reasons, we found out that these challenges are inherited in the mechanical proofs of numerical propositions on Riemann Integral. Thus, it is very commanding to automate these proofs within a formal verification system. In general, proving a numerical proposition on a real-valued function mechanically is known to be a challenging task [4, 58, 6]. Thus, practical approaches that combine computational evaluations within theorem provers have become very appealing. Recently, Muñoz and Narkawicz (from NASA Langley) have implemented and verified a rigorous approximation technique for specific type of functions. For instance, they provided a generic *branch-and-bound* algorithm to estimate the maximum and the minimum values of real valued expressions for a large

[2]The animation is available in the public domain `http://www.aviationsystemsdivision.arc.nasa.gov/research/modeling/facet.shtml`. Please see Appendix E for copyright documentation.

scope of functions like trigonometric, exponential, polynomials and their combinations. The verification yields highly sophisticated proof strategies to prove linear and non-linear inequalities of the form:

$$\mathsf{x} \in \mathsf{X} \Rightarrow \mathsf{f} \in \mathsf{F}(\mathsf{X}) \tag{8.2}$$

where $X \in \mathbb{R}^n$ and F is an *enclosure method* such as *interval* or *affine* evaluations for the function f over the *Box* X,where a Box is a list of intervals [58]. For a function f that satisfies the formula 8.2, the formula 8.2 instantiated by f is called the *fundamental inclusion* lemma for f. The major proof step in these proof strategies,*e.g numerical, affine-numerical* [58, 65], is based on *ground evaluation* of PVS [65], which means the numeric evaluations of symbolic expressions in PVS after being instantiated with concrete values. These proof strategies preserve the soundness of PVS.

Other proof strategies for real valued inequalities which include *transcendental* functions do exist in PVS NASA library[3], which in fact represents the standard PVS library. For example, the proof strategy *metit* [66], unlike *interval*[4] and numerical, uses theorem prover MetiTarsky [67] as a trusted oracle and requires an external arithmetic decision procedure such as Z3[4] [66]. Another example, is the proof strategy *Bernstein* which is based on Bernstein polynomials to approximate the values of multivariate polynomials in PVS [68]. These proof strategies enable a practical approach called *model animation* to validate numerical software implementations against their formal models [5]. Recently, NASA's formal methods group has used model animation in validating software implementations of Detect and Avoid Alerting Logic for Unmanned Systems (DAIDALUS) algorithms with respect to their formal specifications [2, 5].

Organization. Section 8.2 provides some basic concepts and results from existing work. Section 9.1 presents an abstract algorithm for the approximation of Riemann Integral. Section 9.3 proves the soundness statement of the proposed algorithm. Section 10.1 discusses the usability of RiemannSum R2I and its soundness statement in formal proofs of numerical propositions automatically. Section 10.4 presents some case studies. Finally, Section 11.1 makes concluding remarks and presents some extensions of this work.

8.2 Preliminaries

In this section, we provide a brief review of the basic concepts and theorems which we use throughout this part of the dissertation.

[3] `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`

[4] The full distribution of the NASA PVS 6.0 Library includes pre-installed binaries of MetiTarski 2.2 and Z3 4.3.1 for Mac OSX 10.7.3 and 64-bits Linux

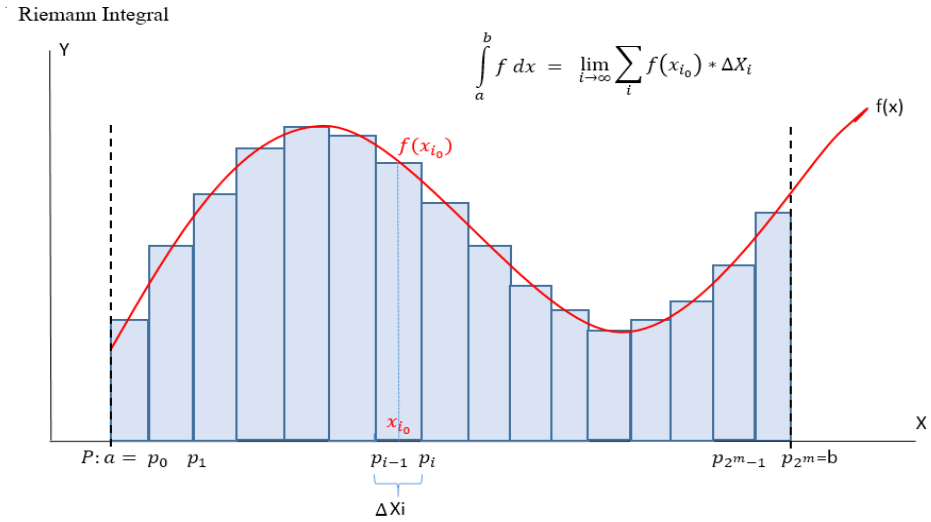

Figure 8.2: Riemann integral of the function $f(x)$ from a to b.

8.2.1 Riemann Integral

Riemann Integral was formalized in PVS by Butler [6] based on the definition from Maxwell Rosenlicht [69]. In the rest of Part II of the dissertation we consider $a, b \in \Re$ and $a < b$ where a and b are the two endpoints of the integral. We also assume that f is a univariate real-valued function, which is well defined on the interval $X = [|a, b|]$, and P is a partition of X of length $N + 1$ such that $P(0) = a$, $P(N) = b$.

Definition 8.2.1. *If $x' \in [|P(i-1), P(i)|]$ then for all $i \in \{1, ..., N\}$ the quantity $f(x')\times(P(i) - P(i-1))$ is called Riemann-section of the i-th subdivision of the partition P. We denote it by Riemann-sec(a,b,f,i).*

Definition 8.2.2. *The sum of all Riemann-sections of the partition P is called Riemann-sum of f on $[|a, b|]$. For instance, Riemann_sum(a,b,f,N)=$\sum_1^N$ Riemann-sec(a,b,f,i).*

Observe that in definition 8.2.1 of Riemann-section the quantity $(P(i) - P(i-1))$ is called the *width* of the i-th Riemann section of the partition P of the interval $[|a, b|]$, and it is denoted by ΔP_i. If ΔP_i is equal to the same constant δ for all i then the partition is called equal partition of width δ.

Definition 8.2.3. *If the limit of Riemann-sum when $\Delta P(i)$ approaches 0 -or equivalently when N approaches ∞- exists, then this limit is called Definite Riemann Integral of the function f from a to b, denoted by $\int_a^b f(x)dx$.*

Notice that if the limit in definition 8.2.3 exists then the function f is called *integrable* over $[|a, b|]$. Butler [6] proved many fundamental properties of Riemann Integral in PVS. We represent the following three theorems from [6] since they are central in our verifications.

Theorem 8.2.1 (Integral Bounds)**.** *Let $a, b \in \mathfrak{R}$ such that $a < b$ and f is integrable on $[|a, b|]$. If $\forall$ $(x \in [|a, b|])$: $m_0 \leq f(x)$, and $f(x) \leq M_0$) then $m_0 \times (b - a) \leq \int_a^b f(x)dx \leq M_0 \times (b - a)$.*

Theorem 8.2.2 (Integral Split)**.** *Let a, b, $c \in \mathfrak{R}$ such that $a < b < c$; and let f be a real-valued function on $[|a, c|]$. Then f is integrable on $[|a, c|]$ if and only if it is integrable on both $[|a, b|]$ and $[|b, c|]$, in which case $\int_a^c f(x)dx = \int_a^b f(x)dx + \int_b^c f(x)dx$.*

Theorem 8.2.3. *If f is a continuous real-valued function on the interval $[|a, b|]$ then $\int_a^b f(x)dx$ exists.*

In the PVS analysis library Theorem 8.2.1 is called *integral_bound*. It establishes a lower and an upper bound for the integral given a lower and an upper bound for the integrand f on a given strict interval $[|a, b|]$. Furthermore, Theorem (8.2.2) is called *integral_split*. It represents one of the fundamental properties of Riemann Integral. In the rest of this Part II, to keep our terminology consistent with PVS analysis, we will use the same names and notations which were used in [6]- whenever need be, unless specified otherwise.

8.2.2 Basics of Interval arithmetic

Interval operations are defined in such a way that guarantees the inclusion of the result of the correspondence regular arithmetic operation in their output. In particular, the addition of two intervals X,Y is defined as Add(X,Y) = $[|lb(X) + lb(Y), ub(X) + ub(Y)|]$. Thus, if $x \in X$ and $y \in Y$, then the inclusion $x + y \in$ Add(X,Y) is valid. This property is called the *fundamental inclusion theorem of interval addition*. The rest of the operations for two intervals X,Y are defined as follows:

$$
\begin{aligned}
\text{X-Y} &= [|\text{lb(X) - ub(Y), ub(X) - lb(Y)}|], \\
\text{X*Y} &= [|\ \min\{\text{lb(X)lb(Y), lb(X)ub(Y), ub(X)lb(Y), ub(X)ub(Y)}\}, \\
&\quad \max\{\ \text{lb(X)lb(Y), lb(X)ub(Y), ub(X)lb(Y), ub(X)ub(Y)}\}|], \\
\text{X/Y} &= \text{X*}[|\ \tfrac{1}{ub(Y)}, \tfrac{1}{lb(Y)}|], \text{ if lb(y)ub(y)} > 0.
\end{aligned}
$$

In [4] the formalization in PVS of all interval operations and the verifications of all inclusion theorems for each operation is provided in details. We present the following inclusion theorem for the four basic interval operations from [4] since it lies at the heart of our proofs. Observe that, it is assumed that $0 \notin Y$ for interval division.

Theorem 8.2.4. *If $x \in X$ and $y \in Y$ then $x \otimes y \in X \otimes Y$, where $\otimes \in \{+, -, *, \div\}$.*

8.3 Problem Statement

In this section we formally state the problem which we inspect its solution in this part of the dissertation by means of interactive theorem prover PVS(Fig 8.2). Let f be a real valued *integrable function* on the strict real valued interval [|a,b|]. Our goal is to verify univariate real-valued numerical integral proposition of the form

$$\mathsf{x} \in [|\mathsf{a}, \mathsf{b}|] \Rightarrow \int_{\mathsf{a}}^{\mathsf{b}} \mathsf{f}\mathsf{d}\mathsf{x} \in [|\alpha, \beta|] \tag{8.3}$$

where α, β are distinct real numbers. Thus the problem takes as its input a real-valued function f and a closed strict interval $X = [|a, b|]$. The output of Problem 8.3.1 is either true or false according to the correctness of the input expression.

Problem 8.3.1. Approximate Riemann Integral within a Formal Proof in PVS

- **Input***: (1) A real valued function f; (2) A strict closed interval $[|a, b|]$; and (3) An expression $IntegExpr$ of form 8.3.*
- **Output***: True, or False based on the correctness of the input $IntegExpr$.*

8.4 Exemplification

We developed the example 8.4.1 in order to explain the need for a proof strategy that can handle the proof of an expression of the form 8.3. More intuitively, in this example we illustrate the complexity of formal mechanical proofs for a proposition of the form 8.3 versus informal manual proofs. Moreover, we explain the need for *parametric* lemmas, theorems, proof strategies, and the ground evaluations of PVS within the mechanical proof of these expressions. In the example[8] 8.4.2 we show how an actual proof strategy can simplify the proof of a similar lemma, however, of the form 8.2.

8.4.1 Informal Manual Proof

Our[7] method to solve problem 8.3.1 is similar in spirit to the work of Muñoz and Narkawicz [70] which is based on evaluating numerical enclosure functions of the form 8.2. For instance, we first make a partition, say P, of the interval $X = [|a, b|]$.

[8]This example was taken from NASA PVS interval_arith lib 6.09. The library includes more examples of this form.

[7]We would like to thank Dr. César Muñoz and Dr. Anthony Narkawicz from NASA Langley formal method group for their valued inputs through out this part.

Then we use the enclosure function F to calculate an interval approximation of the function f over each subdivision of P. Then we rely on the enclosure property of $f(x)$ in $F(X)$ and the linear property of Riemann integral to inductively verify the inequality. We refer to this strategy by *numerical-Riemann*.

8.4.2 Mechanical but not Automatic Formal Proof

Example 8.4.1. To show that

$$\int_0^1 cos(x)dx \in [|0.841, 0.852|] \tag{8.4}$$

In PVS we prove the following lemma:

Lemma 8.4.1. $integral(0, 1, cos) \in [|0.841, 0.852|]$

The mechanical proof of this lemma in PVS[9] can be given as illustrated in Listing 8.1. Observe that, in example 8.4.1, to prove lemma 8.4.1 we need to prove the lemma "integral_bounds_cos" (Line 2) and the lemma "Incl_Member" (Line 12). Moreover, we need to call an auxiliary function to evaluate the symbolic expression in PVS with concrete numeric values - namely *RiemannSum_r2i*. Then we instantiate them with the valid parameters. Then we use ground evaluations of PVS, *ground* and *eval-formula* [71], to simplify the symbolic formulas and evaluate them over their concrete numeric values. Thus, the proof obligation can be discharged by PVS decision procedure. We emphasize that the function RiemannSum_r2i in this example was specified for *cos* function only. The function F_1 is the interval function Cos; which we use to produce the interval evaluation of the function *cos* over each subinterval of a partition of length 2^m; where m, here, is 12. The number 2 is used as the depth of the approximation for the function *cos* in PVS. Unlike the manual informal proof 8.4.1, the formal proof of the lemma, using the *primitive proof rules* of PVS only, is too lengthy. For instance, as it can be seen in appendices A and B, the proof of lemma 8.4.1 required several hundred line of code and proof command! This was necessary to complete the proof. Observe that this proof is valid only for the *cos* function and x∈[|0,1|]. On the top of this, all concrete instantiations in the proof are done manually[9]. Thus, if we modified the lemma 8.4.1 with other parameters, say $[|1, 2|], sin, [| - 1, 1|]$, then we will need to reprove everything from scratch for the new parameters. This will inevitably happen despite the fact that the two proofs are almost identical except that the names of the parameters do differ. The proof of these lemmas using a copy-paste technique then doing the changes manually will

[9]Appendix A contains the complete code in PVS

[9] Appendix B contains the complete PVS proof scripts

try the patience of any proof engineer. For these reasons, PVS does provide a powerful strategy language [72] that enables the definitions of parametric proof strategies. Thus, they can be used as *proof rules* which can show a lemma like 8.4.1 in one line command as an atomic step rather than rewriting all the proof steps every time we need it during the verification of the system.

```
1  Proof:
2   %|- (then (lemma "integral_bounds_cos")
3   %|-  (spread (inst -1 "[|0,1|]" "12" "2")
4   %|-   ((spread
5   %|-     (case "RiemannSum_r2i(lb([|0, 1|]),
6   %|-                           ub([|0, 1|]),
7   %|-                           12,
8   %|-                           2 ^ 12,
9   %|-                           F_1,
10  %|-                           2) << [| 0.841,0.852|]")
11  %|-     ((spread (case "StrictInterval?([|0,1|])")
12  %|-       ((then (assert) (lemma "Incl_Member")
13  %|-         (inst -1 " RiemannSum_r2i(0, 1, 12, 2 ^ 12, F_1, 2)"
14  %|-          "[|841/1000, 213/250|]")
15  %|-         (assert) (inst -1 "integral(0, 1, cos)") (ground))
16  %|-        (then (expand 'StrictInterval? '1) (ground))))
17  %|-      (then (hide-all-but 1) (eval-formula))))
18  %|-    (then (expand 'StrictInterval? '1) (ground)))))
19  %|- QED
```

Listing 8.1: Mechanical proof of lemma 8.4.1

8.4.3 Automatic Proof Strategy

Example 8.4.2. To show that:

$$x \in [|\ 0,\ 8.17 * 10^{-8}\ |]) \Rightarrow \text{-0.0006427443996} * \exp(\text{-2.559889987} * (10^8 * x)) - 0.07600397043 + 0.1443470449 * \exp(\text{-4.211001275} * (10^6 * x)) \in [|\ 0,\ 0.08\ |]$$

One can use the sophisticated proof strategy interval as follows:

Proof. PROOF (interval) QED. □

Interval can handle a large family of problems quite easily. As can be seen in example 8.4.2, the proof strategy completed the proof without any extra effort from the user- as one atomic step. In fact, in PVS there are several ways to automate a proof [72]. For instance, PVS has the capabilities to redefine a list of proof steps into a proof strategy such that it can be instantiated with the required parameters

at the top level by the user once and for all. Or it can do the instantiations more automatically [72, 73]. However, either way, to apply these techniques for "numerical-Riemann" we will need the development of more generalized mathematical theorems. In which the variables are of general types. The membership predicates of these types should guarantee the correctness properties of expressions of the form 8.3.

Chapter 9

Generic Algorithm to Estimate Riemann Integral in PVS

9.1 Preview

Using enclosure methods -such as interval arithmetic- to approximate Riemann Integral in PVS is motivated by the need to provide guaranteed bounds of the integral within a formal proof assistant. This work extends the results of [3, 74] toward integral calculus in PVS. However, we designed the algorithm *RiemannSum_R2I* Listing 9.1 to be generic with respect to the bounding function to estimate Riemann Integral. In Section 9.2, we illustrate the inputs of RiemannSum_R2I and their types; furthermore, we provide a full description of this algorithm. In Section 9.3, we show the soundness statement of the algorithm. Finally, in Section 10.1 we elaborate on the usability of RiemannSum_R2I and its soundness statement in formal proofs of numerical propositions of the form (8.3).

9.2 Inputs of the Algorithm and its Formalization

We list the inputs to the algorithm and their types in Table 9.1. Moreover, we provide the formalization of this algorithm in Listing 9.1. For instance, the variables a and b represent the two endpoints of the definite integral over the strict interval $X = [|a, b|]$. The algorithm depends on dividing the given interval X into an equal partition P of 2^m subdivisions where m is a positive natural number. The input F is a bounding function of type *EVAL*; it gives an interval-estimate of the integrand over a given interval. If a numerical series expansion is required to define the integrand in PVS then the parameter n represents the depth of the series expansion (e.g. Taylor series). However, in the case of exact functions the parameter n can be instantiated by any natural number. Finally, R of type *Riem_sec_r2i*, is a function that computes a

Table 9.1: Inputs and Their Types to RiemannSum_R2I

Input	Type
a	real
b	$\{x : real \mid x > a\}$
m	posnat
n	nat
F	EVAL:$[n \to [Interval \to Interval]]$
i	subrange$[1, 2^m]$
R	Riem_sec_r2i:
	[real, {x:real$\mid x > a$}, posnat, subrange[1,2^m], nat, EVAL $\to$ Interval]

coarse estimate - but correct- of Riemann-sections over the ith subdivision of the partition P. The function R is evaluated based on the parameters a, b, m, n, i, and the bounding function F as follows:

$$R(a, b, m, i, n, F) = F(n)([|P(i-1), P(i)|]) * [|(b-a)/2^m|] \tag{9.1}$$

The algorithm adds the outputs of R from 1 to 2^m recursively. The sum is an interval which will bound the required integral. The formalization of the algorithm in higher order logic of PVS is given by the function RiemannSum_R2I of type Interval Listing 9.1. From now on, in this Part II, we refer to the algorithm by name of this function.

```
1  RiemannSum_R2I(a: real, b: {x: real | a < x}, m: posnat,
2                 i: subrange[1, 2 ^ m], n: nat,
3                 F: [nat -> [Interval -> Interval]],
4                 R: Riem_sec_r2i): RECURSIVE
5        Interval =
6    IF i = 1 THEN R(a, b, m, 1, n, F)
7    ELSE Add(R(a, b, m, i, n, F),
8             RiemannSum_R2I(a, b, m, i - 1, n, F, R))
9    ENDIF
10    MEASURE i
```

Listing 9.1: Generic algorithm to Estimate Riemann Integral

9.3 Soundness of the Algorithm

In order to use RiemannSum_R2I in the proof of any formula such as (8.3), the algorithm must satisfy a safety property. That is required to ensure the inclusion of the given definite Riemann Integral in the output of RiemannSum_R2I. Moreover,

Table 9.2: Major declarations and formulas in the proof of the soundness theorem of RiemannSum_R2I

Names in PVS	description	
Eval	VAR	EVAL
F_Bound?(X,f,n,Eval)	predicate	$\forall\ (x \in X)$: $f(x) \in Eval(n)(X)$
integ_inclus_fun?(a,b,n,f,Eval)	predicate	integrable? $\wedge$ $\exists$ Eval: F_Bound?
integ_inclusion_fun	Type+	{f|$\forall$(a,b,n): integ_inclus_fun?}
g	VAR	integ_inclusion_fun
sum_n_split	function	sum of integral(P(i - 1),P(i),g)
general_integ_split$_1$		Theorem(9.3.2)
Fundamental_Riemann_inclusion$_1$		Theorem(9.3.3)
Simple_Riemann_Soundness		Soundness Theorem(9.3.1)

the integrand function f needs to pass two specific conditions. First, *integrability*, i.e., it must be integrable on $X = [|a, b|]$. Second, *inclusion isotonicity*, which means, it must have a bounding function $Eval$ of type EVAl such that if $x \in [|a, b|]$ then $f(x) \in Eval(n)([|a, b|])$ for some natural number n. In Table 9.2 we list a summary of the variables, functions, predicates, types, key lemmas and major theorems that were used for the proof of the soundness property of the algorithm. For instance, in PVS the integrability condition is formalized by the predict $integrable?(a, b, f)$ [6]. We formalized the inclusion property by an existential quantifier on the predicate $F\_Bound?(X, f, n, Eval)$. We called the type of the functions that satisfy these two conditions *integ_inclusion_fun*. To keep the consistency of the proof, we showed that this type is non-empty, particularly, it contains the function *cos*. Thus, we formalized the soundness property of the algorithm for a strict interval $X = [|a, b|]$ and a function g of type $integ\_inclusion\_fun$ as follows:

Theorem 9.3.1 (Soundness Theorem)**.**
integral(a,b,g) ∈ RiemannSum_R2I(a,b,m,2^m,n,Eval,R)

Although the formalization of the RiemannSum_R2I in PVS higher order logic was not hard (Listing 9.1), it was significantly difficult to mechanically prove the correctness statement of this algorithm. Precisely, the proof of Theorem (9.3.1) required 16 lemmas. The size of the proof script was about 941 steps, including the proof commands of the generated TCCs and the proof commands of all these lemmas. Due to space constraints, we present a proof sketch that includes the key lemmas and theorems that were pivotal to complete the formal proof of Theorem(9.3.1).[1]

[1]Appendix C displays the full proof scripts of the soundness theorem and its TCCs.

9.3.1 Proof Sketch

The proof is based on the fact that for any subdivision $[|P(i-1), P(i)|]$ of the equal partition P, if $x \in [|P(i-1), P(i)|]$ then $f(x) \in$ Eval$(n)([|P(i-1), P(i)|])$. This property is guaranteed by the type of the integrand f. Thus by *integ_bound* Theorem(8.2.1) we can get an interval that is guaranteed to bound the integral $\int_{P(i-1)}^{P(i)} f(x)dx$. This can be done by substituting the lower and the upper bounds of the interval Eval(n)($[|P(i-1), P(i)|]$) for m_0 and M_0 respectively. Observe that P includes 2^m subdivisions of equal widths hence $\Delta P(i) = \frac{(b-a)}{2^m}$. The interval bound from the Theorem (8.2.1) will be equal to $R(a, b, m, i, n, Eval)$. For instance, we have proved the following key lemma in PVS.

Lemma 9.3.1. $\int_{P(i-1)}^{P(i)} f(x)dx \in R(a, b, m, i, n, Eval)$.

Then the proof proceeds by using induction twice. First, it is necessary to use induction to show that the sum of the sub-integrals $\int_{P(i-1)}^{P(i)} f(x)dx$ for i from 1 to 2^m is equal to $\int_a^b f(x)dx$, which is a generalization to theorem integ_split Theorem (8.2.2), where we formalize the required sum as follows:

```
1    sum_n_split(a:real, b: {x: real | a < x},m: nat,i: subrange[1, 2
         ^ m],
2                g: Integ_Inclusion_fun): RECURSIVE
3          real =
4     LET P = eq_partition(a, b, 2 ^ m + 1) IN
5       IF i = 1
6         THEN integral(P(0),P(1),g)
7       ELSE integral(P(i - 1),P(i),g)+ sum_n_split(a, b, m, i - 1, g
            )
8       ENDIF
9      MEASURE i
```

Listing 9.2: PVS formalization of the sum of the sub-integrals over a partition P of length 2^m.

Thus, we formally have verified this theorem in PVS. Listing 9.3 includes the formal specification of the key inductive lemma to show Theorem 9.3.2 in PVS.

```
1  general_integ_split:THEOREM
2     a < b IMPLIES
3      LET P = eq_partition(a, b, 2 ^ m + 1) IN
4        FORALL (i: subrange(1, 2 ^ m)):
5          integral(P(0),P(i),g)= sum_n_split(a, b, m, i, g)
```

Listing 9.3: PVS formalization of Theorem 9.3.2.

Theorem 9.3.2. $integral(a, b, g) = sum\_n\_split(a, b, m, 2^m, g)$

Second, we apply induction to show that the sum of the sub-integrals over the partition P is included in the output of the algorithm RiemannSum_R2I.

Theorem 9.3.3 (Inclusion Theorem)**.**
$sum\_n\_split(a, b, m, i, g) \in RiemannSum\_R2I(a, b, m, i, n, Eval, R)$

Then the proof of this theorem follows by applying interval arithmetic fundamental theorems of inclusion for addition and multiplication i.e., Theorem 8.2.4. Listing 9.4 shows its representation in PVS.

```
1  Fundamental_Riemann_inclusion: THEOREM
2     LET P = eq_partition(lb(X), ub(X), 2 ^ m + 1) IN
3        (FORALL (i: subrange(1, 2 ^ m)):
4           (FORALL (Eval:
5                          {F: [nat -> [Interval -> Interval]] |
6                                FORALL (i: subrange(1, 2 ^ m)):
7                                  F_Bound?([|P(i - 1), P(i)|],
8                                           g,
9                                           n,
10                                          F)}):
11                sum_n_split(lb(X), ub(X), m, i, g) ##
12                 RiemannSum_R2I(lb(X), ub(X), m, i, n, Eval, R)))
```

Listing 9.4: PVS formalization of Theorem 9.3.3.

Finally, by substituting 2^m into i in the inclusion Theorem 9.3.3 and by using Theorem 9.3.2 we complete the proof of the main soundness Theorem 9.3.1. We specify this theorem in PVS as given in Listing 9.5 and 9.6 respectively:

```
1  Simple_Riemann_Soundness: THEOREM
2      LET P = eq_partition(lb(X), ub(X), 2 ^ m + 1) IN
3        (FORALL (Eval:
4                    {F: [nat -> [Interval -> Interval]] |
5                             FORALL (i: subrange(1, 2 ^ m)):
6                               F_Bound?([|P(i - 1),P(i)|],
7                                        g,
8                                        n,
9                                        F)}):
10          integral(lb(X), ub(X), g) ##
11           RiemannSum_R2I(lb(X), ub(X), m, 2 ^ m, n, Eval, R))
```

Listing 9.5: PVS formalization of the soundness theorem .

```
928 Riemann_integ_interval_approx.Simple_Riemann_Soundness: proved-
    complete
930 ("
931 (skosimp*)
932 (assert)
933 (skosimp*)
934 (lemma "Fundamental_Riemann_inclusion1")
935 (inst?)
936 (inst -1 "X!1" "g!1" "n!1")
937 (assert)
938 (inst -1 "2^m!1")
939 (inst -1 "Eval!1")
940 (case "integral(lb(X!1), ub(X!1), g!1) = sum_n_split(lb(X!1),
    ub(X1), m!1, 2 ^ m!1,g!1)")
941 (("1" (rewrite -1)) ("2" (lemma "general_integ_split1") (inst?)
    )))
```

Listing 9.6: Sample of the mechanical proof of the soundness theorem Appendix C.

9.3.2 Why the Mechanical Proofs are too Lengthy.

In [6] Butler provided an in depth discussion to the complexity of the mechanical proofs for integral calculus in PVS. In particular, he gave theorem (8.2.2) integ_split as an example in which the mechanical proof required 4000 proof commands. He explained that it was the most difficult theorem to prove in the formalizations of integral calculus library. He emphasized two major reasons for that. For instance, using partitions will force lengthy inductive proofs. Moreover, using type theory will lead to several complications in the formalizations of many functions leading to many proof obligations in terms of TCCs. Then he provided a very helpful lemma to simplify proofs that include partitions; namely *parts_order* Listing 9.7(Line 667). Similarly, any proof to any property that is correct for a partition will need induction. We illustrate this in our formalization by an example of an inductive proof of theorem general_split Listing 9.7 as follows:

Example 9.3.1. *The informal representation of an equal-partition* $P1$ *of an interval* $[|a, b|]$ *in a manual proof will be simple as follows:*

$$\textit{Let } P1{:}a{=}x_0{<}x_1{<}\ldots{<}x_{n-1}{=}b \textit{ be an equal-partition of } [|a, b|]$$

However, in a formal proof of PVS the type of the function eq_partition $P1$ *is a finite sequence, which does not inform PVS about the order of its elements. Thus, by expanding the definition of eq_partition (Line 673) there will be an obligation, labeled*

by "4" (Line 674), to show that $x_{i-1}<x_i$ for all $i < P$'length. We prove it by means of lemma parts_order and induction (Lines 675–677).

Indeed, the generalized version of integ_split i.e Theorem 9.3.2 is the longest part of the proof of the soundness theorem in our formalization with roughly 760 proof command. The same reasons apply to Theorem 9.3.3 as well.

```
664 "1")
665 (("1" (expand "integ_inclus_fun?") (assert))
666 ("2"
667 (lemma "parts_order")
668 (expand 'strictinterval? '1)
669 (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)" "k!1"
670 "1+k!1")
671 (("1" (assert) (ground) (expand "P_1") (propax))
672 ("2" (expand 'eq_partition '1) (field 1))
673 ("3" (expand 'eq_partition '1) (field 1))))))
674 ("4"
675 (lemma "parts_order")
676 (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)" "k!1"
677 "1+k!1")
```

Listing 9.7: Sample from the proof of generl_split_integ Theorem 9.3.2.

Chapter 10

Automation

10.1 Preview

It is worth mentioning that before we implemented the soundness Theorem 9.3.1 we had proved mechanically but not automatically a few numerical propositions of the form (8.3) such as $\int_0^1 cos(x)dx \in [|0.8414, 0.8417|]$. The proofs were very similar in their structure but they differ only in the names of the integrands and the ends points of the integrals. Using copy/paste technique is impractical and it will try the patience of any proof engineer. In this section, we illustrate how to use RiemannSum_R2I algorithm and the soundness theorem in order to develop automated proof strategies that significantly help proving propositions of form (8.3). In particular, in Section 10.2, we present a general automation method. In Section 10.3, we explain an implementation for this method in the context of real valued continuous functions. We call the implemented proof strategy *conts-numerical-Riemann*. In Sections 10.4.1 and 10.4.2 we provide two case studies to illustrate the usability of the strategy conts-numerical-Riemann.

10.2 General Method

We provide a general automation method based on Algorithm RiemannSum_R2I and the soundness Theorem 9.3.1 to prove numerical properties of form (8.3). The method has seven steps as follows.

1. Claim soundness Theorem 9.3.1 is correct for the given instantiations.
2. PVS will require proving the claim and one TCC with 2 subgoals:
    - integrable?(a,b,f).

- ∃ Eval:F_Bound?(a,b,f,n,Eval).

3. Claim that $RiemannSum\_R2I << [|\alpha, \beta|]$.
4. Use *eval-formula* to prove the claim in Step 3.
5. If Step 4 succeeded then prove the two TCC's subgoals.
6. If Step 5 succeeded, the proof is complete.
7. If Step 5 or 4 failed then exit.

In other words, this means if the claim in Step 3 is provable, then proving the formula depends only on proving the correctness of the TCC's subgoals. If a user was able to discharge the TCC but the method failed, then it is inevitable that the inclusion in Step 3 is not valid, hence the user may consider increasing the parameters m (or/and) n to enhance the accuracy of the approximation. Steps 1,3 and 4 in the general method would be a routine. In particular, in Listing 10.1 we present a proof strategy called *Eval-step* that considerably automates these steps.

```
1  (defstep  eval-step ()
2   (spread (invoke (case "%1  << %2") (! -1 2) (! 1 2))
3       (
4        (then
5         (lemma "Incl_Member")
6         (invoke (inst -1 "%1" "%2") (! -2 1) (! -2 2))
7         (assert)
8         (invoke (inst -1 "%1") (! -3 1))
9         (ground))
10       (then (hide-all-but 1) (eval-formula)) ) ) "" "" )
```

Listing 10.1: Eval-step automates Steps 1–4 in the general proof strategy.

For example, Line 2 will invoke Step 3 in the general method. For instance, it will generate the command:

```
1  (case "RiemannSum_R2I << [|α,β|]" )
```

Listing 10.2: The result of applying Line 2 of Eval-step Listing 10.1.

Whereas, Line 10 the strategy (eval-formula) will capture Step 4 of the general method. The %1,%2 (Lines 2,6,8) are descriptor variables in PVS common manipulations strategy package [75], they will capture strings of the appropriate instantiation.

Table 10.1: Parameters of conts-numerical-Riemann

Parameter	An Instantiation	Usability
Eval	Cos	a bounding function to the integrand
inclusion_lemma	Cos_inclusion	a lemma to prove F_bound? TCC
cont_lemma	cos_cont_fun	a lemma to prove the continuity TCC
$[\|a, b\|]$	$[\|0, 1\|]$	the end points of the integral
f	cos	the integrand
m	12	2^m is the number of subdivisions
n	2	expansion depth

We design the instantiation in this example to be recognized by PVS automatically based on the *PVS location determinations strategies*. In particular, in Listing 10.1 Line 2, the command (inst -1 "%1" "%2") (! -2 1) (! -2 2) will instantiate %1 with the first term of formula -2 with respect to << (shown in Listing 10.2). Thus %1 will capture *RiemannSum_R2I*. Similarly, (! -2 2) will command %2 to capture the string $[|\alpha, \beta|]$ in formula -2.
However, it is more challenging to mechanize Step 5, because there are several heuristics to prove the required subgoals in this step. Nevertheless, in Section 10.3, we show that Step 5 can be automated for a large class of functions.

10.3 Novel Strategy: conts-numerical-Riemann

In this section, we present an implementation of the general method in the context of continuous functions over $\mathfrak{R}$. For instance, *conts-numerical-Riemann* is a proof strategy that implements the general method and automatically uses Theorem 8.2.3 in order to prove the integrability of the integrand f. In Table 10.1 we list the parameters of conts-numerical-Riemann and we give an example of a possible instantiation for each one.
To use conts-numerical-Riemann for the resolution of a numerical query, one has to prove cont-lemma and inclusion-lemma, then the proof will follow automatically. For example, to resolve the proposition $\int_0^1 cos(x)dx \in [|0.8414, 0.8417|]$ we can now verify it in one proof command as follows.

```
cos_test1:LEMMA integral(0,1,cos) ## [|0.8414 , 0.8417|]
PROOF cos_test:
(conts_numerical_Riemann ("Cos") Cos_inclusion cos_cont_fun
   "[|0,1|]"  "cos" "12" "2")
QED
```

10.3.1 Generic Design

```
1  expcos:[real->real]= exp*cos

3  ExpCos(n)(Y):Interval = Mult(Exp(n)(Y),Cos(n)(Y))

5  expcos_cont_fun :lemma continuous?(expcos)

7   PROOF:
8  (expand "expcos") (cont-prod exp_continuous cos_cont_fun)
9  QED

11 ExpCos_inclusion:lemma x ## Y IMPLIES expcos(x) ## ExpCos(n)(Y)

13 expcos_test1:LEMMA integral(0,1,expcos) ## [|1.377, 1.379|]

15  PROOF: (conts-numerical-Riemann ("ExpCos") ExpCos_inclusion
       expcos_cont_fun   "[|0,1|]" "expcos" "12" "2") QED
```

Listing 10.3: conts-numerical-Riemann when the integrand is a combinations of continuous functions

Observe that the proof of Theorem 9.3.1 has the advantage that it is sound for any valid instantiation of the parameter Eval. Thus, we can use any bounding function for the variable F in RiemannSum_R2I (Line 3 listing 9.1 for the integrand based on any inclusion method. For example, in conts-numerical-Riemann since the integrand f is a continuous function on $\mathfrak{R}$ then it is continuous and bounded on any closed interval $[|a, b|]$. Thus, the bounding functions can be defined based on a case analysis of the analytical behavior for these functions i.e where they are decreasing or increasing. In Section 10.4.1 and 10.4.2, we illustrate this feature using two case studies. The first case depends on Taylor expansion to define the bounding function. Whereas the second case depends only on direct analysis to the integrand.

10.4 Case Studies

10.4.1 Elementary functions and reusable strategies

The PVS NASA library includes the continuity lemmas on $\mathfrak{R}$ and the inclusion lemmas verified already for some elementary functions such as *sin*,*cos*, and *exp*. Since the basic operations (+,-,×, safe ÷) on continuous functions generate a continuous function, then -with minimal human efforts- we can apply the strategy when the integrand is a continuous combination of these functions on $\mathfrak{R}$. For example, if the

integrand is the multiplications of *exp* and *cos* Listing 10.3(Line 1), we can simply prove the continuity and the inclusion theorems of the multiplication (Lines 3,5 and 11). Thus, we can apply conts-numerical-Rieamnn on the function $exp \times cos$ -as illustrated in Listing 10.3 (Lines 13 and 15). In general, the strategy applies regardless of whether the integrand's anti-derivative has a known representation by means of elementary functions or not.

10.4.1.1 cont-prod.

To help the user to discharge the continuity lemma in Listing 10.3, we developed a simple strategy to prove that the product of two continuous functions is continuous; we called it *cont-prod* Listing 10.4. This strategy has two main steps (cont-mult) and (lemma-conts-step) Listing 10.4 (Lines 2,5–6), also it requires the continuity lemmas of the two functions to be provided as parameters (Line 1). Subsequently, the first step (mult-cont)Listing 10.5(Lines 1–7) shows if two given functions are continuous then their multiplication is continuous, based on theorem *"prod_cont_fun"* from PVS library Listing 10.5 (Line 3). However, it does not complete the proof when it captures the two functions directly! Particularly, PVS will generate two TCCs for these functions to check the correctness of their continuity, but (mult-cont) will postpone this obligation Listing 10.5 (Line 7).

```
1 (defstep cont-prod (lemma1 lemma2 )
2   (then (cont-mult)
3     (assert)
4     (spread (split)
5         ((lemma-conts-step lemma1)
6          (lemma-conts-step lemma2)))) " " "" )
```

Listing 10.4: Automatic strategy to show the product of two continuous function is continuous.

The second, step (lemma-conts-step) Listing 10.5 (Lines 10–12) tells PVS how to discharge this goal for each function using the parameters lemmas provided in (Line 1). The user can use a proved lemma of PVS, or they can develop new ones- in case PVS does not have them already as explained in Listing 10.3 (Line 5). cont-prod can also be used recursively to show that the product of a finite number of continuous functions is continuous. We mention it here since it can be used as a template to define similar proof strategies for other operations over continuous functions. In [73] the author developed a strategy to check the continuity of a specific class of transcendental functions in PVS, but, unfortunately, their strategy is not supported in the current PVS library. Moreover, their approach depends on formalizing appropriate judgements to get the appropriate instantiations in their strategy. By contrast, we

designed the strategy cont-prod based on the powerful manipulation package of PVS [75, 72], which allows the instantiations depending on pattern matching and location determinations strategies of PVS. For instance, in Listing 10.5 (Lines 4) the command (inst -1 "%1" "%2") (? 1 "continuous?(%1*%2) ")) will instantiate the lemma in Line 3 with two functions that are captured by the descriptors %1,%2. We designed the descriptors to capture the strings that match the pattern continuous?(%1*%2) which appears in the first formula of the current goal "i.e., formula number 1". For example, if the goal was to prove that *continuous?(exp*cos)* as in Listing 10.3 (Line 5), then %1 will capture *exp* while %2 will capture *cos*.

```
1  (defstep cont-mult ()
2    (spread
3     (then (lemma "prod_cont_fun")
4       (invoke (inst -1 "%1" "%2") (? 1 "continuous?(%1*%2) "))
5       (invoke (case "continuous?(%1) and continuous?(%2)")
6           (? 1 "continuous?(%1*%2)")))
7     ((assert) (postpone))) """")


10 (defstep lemma-conts-step (lemma1)
11   ( let ( ( fun1-cont (list `lemma lemma1)))
12       (then fun1-cont )) "" "" )
```

Listing 10.5: Auxiliary steps of cont-prod Listing 10.4.

10.4.1.2 conts-integ-[real]

```
1  (defstep conts-integ-[real] (lemma1)
2  (then
3   (then (lemma "fundamental_indef[real]")
4    (invoke (inst -1 "%1" "%2" "%3")
5     (? 1 "integrable?[real](%1,%2,%3 )")))
6   (let ((lemma-step (list `lemma lemma1)))
7     (then lemma-step (propax) (assert) (expand `connected? `1)
8      (propax)))
9   (invoke (inst -1 "%1" "%2" "%3") (? 1 "integrable?[real](%1,%2,%3
       )"))
10  (expand "Integrable?" -1) (propax)) "" "")
```

Listing 10.6: Automatic strategy to show that every continuous function on $\mathfrak{R}$ is integrable.

Now we illustrate how conts-numerical-Riemann uses Theorem 8.2.3 of Butler to discharge the integrability property. We continue using *expcos* as the running example.

In particular, we define the strategy *conts-integ-[real]* Listing 10.6 with a lemma parameter (Line 1). This lemma is the continuity lemma of the given function. In our running example, this was proved by means of the strategy cont-prod for the function *expcos*. In Line 3 the strategy uses a fundamental theory from analysis library of PVS, particularly (lemma "fundamental_indef[real]") (i.e., Theorem 8.2.3). Then the command (invoke (inst -1 "%1" "%2" "%3") will instantiate the lemma from the current goal using pattern matching of PVS. For instance, in the formula integral(0,1,expcos) ## [|1.377, 1.379|] Listing 10.3 (Line 13) PVS will generate the goal integrable?[real](0,1,*expcos*). Thus, the command (? 1 "integrable?[real](%1,%2,%3)"))) will force %1 to capture 0, %2 to capture 1, and %3 to capture *expcos*. Hence, cont-integ-[real] will apply the lemma and thus discharge the current goal, where ## is a macro for $\in$ in PVS NASA Interval_arith library.

10.4.2 atan

There are many real-valued functions of significant applications that are defined in terms of Riemann Integral. For example, the $atan(x) = \int_0^x \frac{1}{t^2+1} dt$, *Fresnel function* S(x)=$\int_0^x \sin \frac{-\pi t^2}{2} dt$, and *Error function erf(x)*=$\frac{2}{sqrt(\pi)} \int_0^x \exp(-t^2) dt$. In this section, we provide a case study on atan function to illustrate how conts-numerical-Riemann can facilitate approximating the values of these functions at a certain point within a formal proof of an expression of the form (8.3). The atan function is defined in the PVS library as illustrated in equation(8.1). Its values over a given interval are approximated using Taylor series expansions and interval arithmetic in PVS[4]. The strategy conts-numerical-Riemann allows the approximation of atan at a certain point without the need of Taylor expansions, Listing 10.7 explains the formalization in PVS.

```
1 atan_deriv_fun(x:real):real = 1/(1+x*x)

3 atan_deriv_fun_bounds:lemma 0  < atan_deriv_fun(x) and
      atan_deriv_fun(x) <= 1

5 atan_deriv_strict_decreasing: lemma forall (x1,x2:{x0:real | 0 < x0
      }): x1 < x2  implies atan_deriv_fun(x2) <  atan_deriv_fun(x1)

7 atan_deriv_strict_increasing: lemma forall (x1,x2:{x0:real | 0 > x0
      }): x1 < x2  implies atan_deriv_fun(x2) >  atan_deriv_fun(x1)

9 atan_deriv_cont:lemma continuous?[real](atan_deriv_fun)
```

Listing 10.7: atan_deriv_fun: definition, analytical behavior, and continuity

For instance, we define the integrand *atan_deriv_fun* Listing 10.7 (Line 1). Then we prove the lemma *atan_deriv_cont* Listing 10.7 (Line 9), which states the continuity of atan_deriv_fun on $\mathfrak{R}$. Moreover, based on case analysis of the behavior of the atan_deriv_fun provided in Listing 10.7 (Lines 3, 5 and 7) we defined a bounding function of atan_deriv_fun, and we called it *Atan_Deriv* (Lines 1–6 Listing 10.8).

```
1 Atan_Deriv(n)(Y):Interval = COND

3  0 < lb(Y) and 0 < ub(Y) and lb(Y)<ub(Y)  -> [|1/(1+ub(Y)*ub(Y))
      ,1/(1+lb(Y)*lb(Y))|],
4  ub(Y) < 0 and lb(Y) < 0 and lb(Y)<ub(Y)  -> [|1/(1+lb(Y)*lb(Y))
      ,1/(1+ub(Y)*ub(Y))|],
5  lb(Y)=ub(Y)                              -> [|0,1|],
6  else -> [|min(atan_deriv_fun(lb(Y)),atan_deriv_fun(ub(Y))),1|]
                      ENDCOND

8  Atan_Deriv_inclusion: lemma x ## Y IMPLIES atan_deriv_fun(x) ##
      Atan_Deriv(n)(Y)
```

Listing 10.8: Atan_Deriv a bounding function of atan_deriv_fun

Then we proved the lemma *Atan_Deriv_inclusion* which shows that atan_deriv_fun is bounded by Atan_Deriv over its domain as explained in Listing 10.8(Line 8). Observe that this bounding method does not depend on Taylor expansion thus the value of n in the strategy conts-numerical-Riemann can be instantiated by any natural number. The continuity and the inclusion lemmas enable us to verify guaranteed bounds for atan automatically at different given points as illustrated in Listing 10.9 and table 10.2.

```
1 atan_approx_at_1_on_230: lemma  integral(0,1/230, atan_deriv_fun)
     ## [|0.00434779868, 0.004347798699 |]

3  PROOF atan_approx_at_1_on_230:
4  conts-numerical-Riemann ("Atan_Deriv") Atan_Deriv_inclusion
      atan_deriv_cont "[|0,1/230|]" "atan_deriv_fun" "12" "2")  QED

6 atan_approx_at_1_on_30: lemma  integral(0,1/30, atan_deriv_fun) ##
     [|0.03320997, 0.03333 |]

8  PROOF atan_approx_at_1_on_30:
9  (conts-numerical-Riemann ("Atan_Deriv") Atan_Deriv_inclusion
      atan_deriv_cont "[|0,1/30|]" "atan_deriv_fun" "13" "2")   QED
```

Listing 10.9: Using conts-numerical-Riemann to estimate atan(x) at x=1/230, and x=1/30

Table 10.2: atan(x)=$\int_0^x \frac{1}{t^2+1}dt$, with different values of x.

x	2^m	Lower bound	upper bound	$\backsim \Delta$	run-time
$\frac{1}{239}$	2^{13}	0.0041840759971	0.004184076007	1×10^{-12}	2.78s
$\frac{1}{230}$	2^{13}	0.00434779868	0.004347798696	1.6×10^{-11}	3.15s
$\frac{1}{30}$	2^{13}	0.0332099587	0.0332099819	2.5×10^{-10}	3.19s
$\frac{1}{5}$	2^{13}	0.19739509	0.1973960299	9.3×10^{-7}	2.24s
1	2^{13}	0.7853	0.7855	2×10^{-4}	2.93s

Table 10.3: Estimating $atan(89) = \int_0^{89} \frac{1}{t^2+1}dt$ in PVSio using RiemannSum_R2I with different numbers of subdivisions 2^m.

2^m	Lower bound	upper bound	width of estimation	run-time
2^{12}	1.548698	1.5704237	3×10^{-2}	0.6778s
2^{13}	1.5541294	1.5649923	1×10^{-2}	3.112s
2^{14}	1.5571195	1.5620022	5×10^{-3}	7.240s
2^{15}	1.5583402	1.5607815	2×10^{-3}	15.399s
2^{16}	1.5589505	1.5601711	1×10^{-3}	50.262s

To reduce the effect of the well known phenomenon of *dependency effects* of interval arithmetic [76, 77] on the estimation,we designed conts-numerical-Riemann to be configurable - but at the expense of the time- to these effects by permitting the user to modify the parameter m as much as the resources allow. For instance, as illustrated in Table 10.2 the longer the interval is the less accurate the estimation is. Where 2^m is the number of subdivisions, Δ is the width of the given interval, and CPU runtime is the time to prove the inclusion of atan in the given intervals using conts-numerical-Riemann. Enhancing the accuracy by increasing m will increase the time.

Table 10.3 provides the time, the number of subdivisions and the accuracy for the case of atan function where $t \in [|0, 89|]$. Enhancing the accuracy by increasing m will increase the time. Table 10.3 provides the time, the number of subdivisions and the accuracy for the case of atan function where $t \in [|0, 89|]$. The experimental results were conducted by using a regular Linux machine of core i7 Intel processor and 7.6 GiB RAM. It is important to distinguish between the computations we do here within PVS and the computations that are done by regular computation tools such as mathematica and matlap. Specifically, the computations that are done within a PVS lemma (e.g., see Section 10.4.3) do run a proof script; which improves their reliability. Moreover, PVSio uses the concept of *semantic attachment* [5] which guarantees the

correctness of the computations for a given function up to certain precision each time the function is called. This technique performs way more reliably than regular floating point arithmetic. In addition to these two major differences, RiemannSum_R2I has the approximation method for the integrand (if it was not an exact function) as a parameter. Thus, RiemannSum_R2I is appropriate to compare the efficiency and the performance of different approximation methods regardless to whether the algorithm has a known anti-derivative or not. Finally, the number of subdivisions is a parameter as well, thus the accumulated accuracy can be improved following to the user needs.

10.4.3 More Examples

In this section we provide more practical examples on the use of conts-numerical for different functions and parameters.

```
cos_test5:LEMMA integral(0,1/230,cos) ## [|0.00434781236,0
   .00434781241|]
%|- cos_test5 : PROOF
%|- (conts_numerical_Riemann ("Cos") Cos_inclusion cos_cont_fun
%|-  "[|0,1/230|]" "cos" "13" "7") QED

sin_test1:LEMMA integral(0,1.4142135,sin) ## [|0.843 , 0.8449|]%
   use rational approx- for irrational values: example sqrt(2)~1
   .4142135.
%|- sin_test1 : PROOF
%|- (conts_numerical_Riemann ("Sin") Sin_inclusion sin_cont_fun
%|-  "[|0, 1.4142135|]" "sin" "12" "3") QED

sin_test:LEMMA integral(0,1/30,sin) ## [|0.000555 , 0.000556|]
%|- sin_test : PROOF
%|- (conts_numerical_Riemann ("Sin") Sin_inclusion sin_cont_fun
%|-  "[|0,1/30|]" "sin" "13" "4") QED

sin_test4:LEMMA integral(-1.5,1,sin) ## [|-0.474 , -0.468|]
%|- sin_test4 : PROOF
%|- (conts_numerical_Riemann ("Sin") Sin_inclusion sin_cont_fun
%|-  "[|-1.5, 1|]" "sin" "12" "2") QED

exp_test3:LEMMA integral(0,3,exp) ## [|19.0 , 19.1|]
%|- exp_test3 : PROOF
%|- (conts_numerical_Riemann ("Exp") Exp_inclusion exp_continuous
   "[|0,3|]"
%|-  "exp" "12" "3") QED
```

Chapter 11

Conclusions

11.1 Contributions of Part II

Riemann Integral is one of the most preeminent tools in modern calculus given its wide range of applications. Using interval arithmetic to approximate Riemann Integral has been in use for long time in many computational systems [78]. PVS is a verification system that has one of the largest and most complete formally verified interval_arithmetic libraries [3]. This part of the dissertation extends the proposed verification method in [3] into integral calculus. However, it is based on a formally verified implementation of a generic algorithm that can be instantiated with different bounding functions. We explained how to implement practical and reusable automated proof strategies to verify expressions of the form (8.3) depending on the proofs of simple properties of the integrand[1]. Furthermore, the general automation method presented in this part can help to approximate the values of many special functions in PVS that are defined using Riemann Integral. We summaries our major contributions in Table 11.1 and Table 11.2.

11.2 Discussions and Future Research

The need for Riemann integral calculations often arise in mission-critical cyber-physical systems such as the integration of Unmanned Systems (UAS) in NAS (National Aero-space System) in aeronautics [2], calculating trajectories of a ballistic missile or a spacecraft [1]. Particularly, it is very useful in solving the related differential equations. We are exploring several expansions to our results:

[1] Please visit `https://github.com/nasa/pvslib` for full specifications in PVS strategy language

Table 11.1: Cost of the Proof of major results of Part II.

Lemmas/Theorems	Mechanical	Automatic
general_split	760	
Fundamental Riemann Inclusion	809	
Soundness Theorem	940	
cos_test lemma	1300	
10 test_lemmas (atan, cos, sin, exp, exp∗cos)		10 proof commands

Table 11.2: Major automatic proof strategies that are required to define conts-numerical-Riemann and their functionalities.

Main Strategies	Functionality
Eval_step	Numerical evaluation step to RiemannSound_R2I.
cont-prod	Product of two continuous functions is continuous -a template for other operations.
conts-Integ	continuous? is integrable?
Func-inclusion	Inclusion isotonicity
soundness	Applies soundness theorem continuity, inclusion, without the Eval-step.
conts-numerical-Riemann	The unified strategy that applies all of the above for numerical propositions.

1. The development of automated proof strategies for other types of integrable functions, such as discontinuous but integrable functions.

2. The evolution of PVS packages to formalize rigorous approximations for many special functions.

3. The expansion of this work toward multi-variable integrals.

4. To overcome time limitation for long intervals and expressions we are studying distributed framework for high-precision calculations of numerical queries of form 8.3 within a formal proof.

Finally we shade a light on two important motivations that prompt the future extensions, namely:

- The calculations which are computed by PVS's ground evaluator provide more accurate results than regular floating point arithmetic. Thus more reliability for the design and the verification of critical systems that involve Riemann Integrations.
- The automatic proof strategy we developed conts-numerical-Riemann cuts the verifications time and human efforts significantly for unaccountably large family of functions. Particularly, from roughly a thousand proof step to almost one line proof command. An experience that is very appealing to be applied on other types of functions. For instance, multi-variable integrable functions.

Since Riemann Integral is one of the most preeminent computational tools in modern calculus, we choose to finish the dissertation with our major results of Part II. As they have enabled automatic numerical computations for Riemann Integral within a formal rigorous proof in the widely used state-of-the-art ultra-reliable interactive verification system PVS.

Bibliography

[1] R. W. Butler and S. C. Johnson, "Formal methods for life-critical software," in *Computing in Aerospace Conference*, pp. 319–329, 1993. 1, 83

[2] C. Muñoz, A. Narkawicz, G. Hagen, J. Upchurch, A. Dutle, M. Consiglio, and J. Chamberlain, "Daidalus: detect and avoid alerting logic for unmanned systems," in *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pp. 5A1–1, IEEE, 2015. 1, 56, 83

[3] C. Muñoz and D. Lester, *Real number calculations and theorem proving*, pp. 195–210. Springer, 2005. Interval Expr vs Real Expr. 1, 64, 83

[4] M. Daumas, D. Lester, and C. Muñoz, "Verified real number calculations: A library for interval arithmetic," *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 226–237, 2009. 1, 2, 55, 56, 58, 78

[5] A. M. Dutle, C. A. Muñoz, A. J. Narkawicz, and R. W. Butler, "Software validation via model animation," in *Tests and Proofs (TAP 2015), 1991. Proceedings CVPR '91., Proceedings of the 9th International Conference on*, pp. 92–108, Springer, 2015. 1, 56, 80

[6] R. W. Butler, "Formalization of the integral calculus in the pvs theorem prover.," *J. Formalized Reasoning*, vol. 2, no. 1, pp. 1–26, 2009. 1, 2, 54, 55, 57, 58, 66, 69

[7] P. S. Miner, M. Malekpour, and W. Torres, "A conceptual design for a reliable optical bus (robus)," in *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, vol. 2, pp. 13D3–1, IEEE, 2002. 1

[8] L. Pike, J. Maddalon, P. Miner, and A. Geser, "Abstractions for fault-tolerant distributed system verification," in *Theorem Proving in Higher Order Logics*, pp. 257–270, Springer, 2004. 1

[9] M. R. Malekpour, "A self-stabilizing hybrid fault-tolerant synchronization protocol," in *Aerospace Conference, 2015 IEEE*, pp. 1–11, IEEE, 2015. 1

[10] S. Owre, J. Rushby, N. Shankar, and F. Von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs," *Software Engineering, IEEE Transactions on*, vol. 21, no. 2, pp. 107–125, 1995. 1

[11] L. Pike, S. Niller, and N. Wegmann, "Runtime verification for ultra-critical systems," in *Runtime Verification*, pp. 310–324, Springer, 2011. 1

[12] J. Chen, A. Ebnenasir, and S. S. Kulkarni, "The complexity of adding multitolerance," *ACM Transactions on Adaptive and Autonomous Systems*, vol. 9, no. 3, pp. 15:1–15:33, 2014. 1

[13] A. Klinkhamer and A. Ebnenasir, "On the hardness of adding nonmasking fault tolerance," *IEEE Trans. Dependable Sec. Comput.*, vol. 12, no. 3, pp. 338–350, 2015. 1

[14] A. Klinkhamer and A. Ebnenasir, "On the complexity of adding convergence," in *Proceedings of the 5th International Symposium on Fundamentals of Software Engineering (FSEN)*, pp. 17–33, 2013. 1, 8, 18, 20, 50

[15] A. Farahat and A. Ebnenasir, "A lightweight method for automated design of convergence in network protocols," *ACM Transactions on Adaptive and Autonomous Systems*, vol. 7, no. 4, p. 38, 2012. 1, 2

[16] A. Klinkhamer and A. Ebnenasir, "Synthesizing self-stabilization through superposition and backtracking," in *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, pp. 252–267, 2014. 1, 2

[17] A. Klinkhamer and A. Ebnenasir, "Protocon: A parallel tool for automated synthesis of self-stabilization." `http://asd.cs.mtu.edu/projects/protocon/`. 1, 2, 18, 21, 37, 46, 49

[18] S. Owre, N. Shankar, J. M. Rushby, and D. W. Stringer-Calvert, "Pvs language reference," *Computer Science Laboratory, SRI International, Menlo Park, CA*, vol. 1, no. 2, p. 21, 1999. 3

[19] N. Shankar, S. Owre, and J. M. Rushby, *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. A new edition for PVS Version 2 is released in 1998. 3, 7

[20] A. Tahat and A. Ebnenasir, "A hybrid method for the verification and synthesis of parameterized self-stabilizing protocols," in *Logic-Based Program Synthesis and Transformation*, pp. 201–218, Springer, 2014. 6

[21] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974. 7, 10

[22] M. Gouda, "The theory of weak stabilization," in *Workshop on Self-Stabilizing Systems*, vol. 2194 of *LNCS*, pp. 114–123, 2001. 7, 15, 20

[23] F. Abujarad and S. S. Kulkarni, "Multicore constraint-based automated stabilization," in *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pp. 47–61, 2009. 7, 18

[24] F. Abujarad and S. S. Kulkarni, "Automated constraint-based addition of nonmasking and stabilizing fault-tolerance," *Theoretical Computer Science*, vol. 412, no. 33, pp. 4228–4246, 2011. 7, 18

[25] A. Farahat and A. Ebnenasir, "A lightweight method for automated design of convergence in network protocols," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, pp. 38:1–38:36, Dec. 2012. 7, 8, 17, 18, 20, 21, 50, 137

[26] A. Ebnenasir and A. Farahat, "Swarm synthesis of convergence for symmetric protocols," in *European Dependable Computing Conference*, pp. 13–24, 2012. 7, 8, 18, 20, 50

[27] S. Dolev and Y. A. Haviv, "Self-stabilizing microprocessor: analyzing and overcoming soft errors," *Computers, IEEE Transactions on*, vol. 55, no. 4, pp. 385–399, 2006. 7, 49

[28] S. Dolev and R. Yagel, "Self-stabilizing operating systems," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 1–2, ACM, 2005. 7, 49

[29] S. Qadeer and N. Shankar, "Verifying a self-stabilizing mutual exclusion algorithm," in *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)* (D. Gries and W.-P. de Roever, eds.), (Shelter Island, NY), pp. 424–443, Chapman & Hall, June 1998. 7, 49

[30] S. S. Kulkarni, J. Rushby, and N. Shankar, "A case-study in component-based mechanical verification of fault-tolerant programs," in *19th IEEE International Conference on Distributed Computing Systems - Workshop on Self-Stabilizing Systems*, pp. 33–40, 1999. 7, 49

[31] I. S. W. B. Prasetya, "Mechanically verified self-stabilizing hierarchical algorithms," *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), volume 1217 of Lecture Notes in Computer Science*, pp. 399–415, 1997. 7

[32] M. J. C. Gordon and T. F. Melham, *Introduction to HOL: A Theorem proving Environment for Higher Order Logic*. Cambridge University Press, 1993. 7

[33] T. Tsuchiya, S. Nagano, R. B. Paidi, and T. Kikuno, "Symbolic model checking for self-stabilizing algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 1, pp. 81–95, 2001. 7

[34] S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir, "Mechanical verification of automatic synthesis of fault-tolerance," *International Symposium on Logic-based Program Synthesis and Transformation*, vol. 3573, pp. 36–52, 2004. 7, 10, 14, 50

[35] B. Bonakdarpour and S. S. Kulkarni, "Towards reusing formal proofs for verification of fault-tolerance," in *Workshop in Automated Formal Methods*, 2006. 7, 50

[36] P. C. Attie, anish Arora, and E. A. Emerson, "Synthesis of fault-tolerant concurrent programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 1, pp. 125–185, 2004. 7

[37] S. S. Kulkarni and A. Arora, "Automating the addition of fault-tolerance," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, (London, UK), pp. 82–93, Springer-Verlag, 2000. 7

[38] A. Ebnenasir, S. S. Kulkarni, and A. Arora, "FTSyn: A framework for automatic synthesis of fault-tolerance," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 5, pp. 455–471, 2008. 7

[39] S. Jacobs and R. Bloem, "Parameterized synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 362–376, 2012. 7, 49

[40] A. Klinkhamer and A. Ebnenasir, "Synthesizing self-stabilization through superposition and backtracking," in *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pp. 252–267, Springer, 2014. 8, 37, 50, 51

[41] A. Arora and M. G. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1015–1027, 1993. 15

[42] M. G. Gouda and N. J. Multari, "Stabilizing communication protocols," *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 448–458, 1991. 40

[43] A. Khalimov, S. Jacobs, and R. Bloem, "Party parameterized synthesis of token rings," in *Computer Aided Verification*, pp. 928–933, Springer, 2013. 49

[44] C. N. Ip and D. L. Dill, "Verifying systems with replicated components in murphi," *Formal Methods in System Design*, vol. 14, no. 3, pp. 273–310, 1999. 50

[45] A. Pnueli, J. Xu, and L. D. Zuck, "Liveness with (0, 1, infty)-counter abstraction," in *International Conference on Computer Aided Verification (CAV)*, pp. 107–122, 2002. 50

[46] E. A. Emerson and K. S. Namjoshi, "On reasoning about rings," *International Journal of Foundations of Computer Science*, vol. 14, no. 4, pp. 527–550, 2003. 50

[47] P. Wolper and V. Lovinfosse, "Verifying properties of large sets of processes with network invariants," in *International Workshop on Automatic Verification Methods for Finite State Systems*, pp. 68–80, 1989. 50

[48] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck, "Network invariants in action," in *Concurrency Theory*, pp. 101–115, Springer, 2002. 50

[49] O. Grinchtein, M. Leucker, and N. Piterman, "Inferring network invariants automatically," in *Automated Reasoning*, pp. 483–497, Springer, 2006. 50

[50] A. Roychoudhury, K. N. Kumar, C. Ramakrishnan, I. Ramakrishnan, and S. A. Smolka, "Verification of parameterized systems using logic program transformations," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 172–187, Springer, 2000. 50

[51] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni, "Generalization strategies for the verification of infinite state systems.," *TPLP*, vol. 13, no. 2, pp. 175–199, 2013. 50

[52] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili, "Regular model checking," in *CAV*, pp. 403–418, 2000. 50

[53] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena, "A survey of regular model checking," in *CONCUR*, pp. 35–48, 2004. 50

[54] P. A. Abdulla and B. Jonsson, "Model checking of systems with many identical timed processes," *Theoretical Computer Science*, vol. 290, no. 1, pp. 241–264, 2003. 50

[55] N. Bertrand and P. Fournier, "Parameterized verification of many identical probabilistic timed processes," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 24, 2013. 50

[56] M. U. Sanwal and O. Hasan, "Formal verification of cyber-physical systems: coping with continuous elements," in *Computational Science and Its Applications–ICCSA 2013*, pp. 358–371, Springer, 2013. 54

[57] S. Boldo, C. Lelay, and G. Melquiond, "Formalization of real analysis: A survey of proof assistants and libraries," *Mathematical Structures in Computer Science*, pp. 1–38, 2014. 54, 55

[58] A. Narkawicz and C. Muñoz, "Formal verification of conflict detection algorithms for arbitrary trajectories," *Reliable Computing*, vol. 17, no. 2, pp. 209–237, 2012. 54, 55, 56

[59] V. Carreño and C. Muñoz, *Aircraft trajectory modeling and alerting algorithm verification*. Springer, 2000. 54

[60] A. Narkawicz, "A formal proof of the riesz representation theorem," *Journal of Formalized Reasoning*, vol. 4, no. 1, pp. 1–24, 2011. 55

[61] J. Harrison, *Theorem proving with the real numbers*. Springer Science & Business Media, 2012. 55

[62] C. Filipe and L. Calhorda, *Constructive real analysis: a type-theoretical formalization and applications*. [Sl: sn], 2004. 55

[63] J. D. Fleuriot, "On the mechanization of real analysis in isabelle/hol," in *Theorem Proving in Higher Order Logics*, pp. 145–161, Springer, 2000. 55

[64] S. Richter, "Formalizing integration theory with an application to probabilistic algorithms," in *Theorem Proving in Higher Order Logics*, pp. 271–286, Springer, 2004. 55

[65] M. M. Moscato, C. A. Muñoz, and A. P. Smith, *Affine Arithmetic and Applications to Real-Number Proving*, pp. 294–309. Springer, 2015. 56

[66] W. Denman and C. Muñoz, "Automated real proving in pvs via metitarski," in *FM 2014: Formal Methods*, pp. 194–199, Springer, 2014. 56

[67] B. Akbarpour and L. C. Paulson, "Metitarski: An automatic theorem prover for real-valued special functions," *Journal of Automated Reasoning*, vol. 44, no. 3, pp. 175–205, 2010. 56

[68] C. Muñoz and A. Narkawicz, "Formalization of bernstein polynomials and applications to global optimization," *Journal of Automated Reasoning*, vol. 51, no. 2, pp. 151–196, 2013. 56

[69] M. Rosenlicht, *Introduction to analysis*. Courier Corporation, 1968. 57

[70] A. Narkawicz and C. Muñoz, *A formally verified generic branching algorithm for global optimization*, pp. 326–343. Springer, 2014. 59

[71] C. Munoz, "Rapid prototyping in pvs," *NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, Report NIA Report*, no. 2003-03, 2003. 60

[72] M. Archer, B. Di Vito, and C. Muñoz, *Design and Application of Strategies/-Tactics in Higher Order Logics*. National Aeronautics and Space Administration, Langley Research Center, 2003. 61, 62, 77

[73] H. Gottliebsen, "Transcendental functions and continuity checking in pvs," in *Theorem Proving in Higher Order Logics*, pp. 197–214, Springer, 2000. 62, 76

[74] M. Daumas, G. Melquiond, and C. Muñoz, "Guaranteed proofs using interval arithmetic," in *17th IEEE Symposium on Computer Arithmetic*, pp. 188–195, IEEE, 2005. 64

[75] B. L. DiVito, "A pvs prover strategy package for common manipulations," 2002. 73, 77

[76] A. Neumaier, *Interval methods for systems of equations*, vol. 37. Cambridge university press, 1990. 80

[77] R. B. Kearfott, "Interval computations: Introduction, uses, and resources," *Euromath Bulletin*, vol. 2, no. 1, pp. 95–112, 1996. 80

[78] E. Moore, R. B. Kearfott, and M. J. Cloud, "Introduction to interval analysis. by ramon," 83

Appendix A

PVS Code $\int_0^1 cos(x) \in [|0.841, 0.852|]$

This code shows the complete pvs formalization for Riemann integral approximation for the function $cos(x)$ in the interval $[|0, 1|]$.

```
1  Riemann_integ_interval_approx: THEORY
2   BEGIN
3
4    IMPORTING interval_arith@interval, interval_arith@strategies,
5              reals@sigma_nat, interval_arith@symbols_as_interval,
6              analysis@integral_def[real], trig_fnd@sincos,
7              listn[Interval]
8
9    X, Y: VAR StrictInterval
10
11   x, a, b: VAR real
12
13   n: VAR nat
14
15   m: VAR posnat
16
17   f: VAR [real -> real]
18
19   G: TYPE =
20         [aa: real, {az: real | aa < az}, mm: posnat,
21         below(2 ^ mm + 1),nat -> Interval]
22
23   F(a: real, b: {x: real | a < x}, m: posnat, i: below(2 ^ m + 1),
24     n: nat):
25         Interval =
26       IF i = 0 THEN [|0|]
27       ELSE LET P = eq_partition(a, b, 2 ^ m + 1) IN
```

```
28              Sin(n)([|P(i - 1), P(i)|]) * [|(b - a) / 2 ^ m|]
29      ENDIF
30
31  F_1(a: real, b: {x: real | a < x}, m: posnat,i: below(2 ^ m + 1),
32      n: nat):
33        Interval =
34      IF i = 0 THEN [|0|]
35      ELSE LET P = eq_partition(a, b, 2 ^ m + 1) IN
36             Cos(n)([|P(i - 1), P(i)|]) * [|(b - a) / 2 ^ m|]
37      ENDIF
38
39  RiemannSum_r2i(a: real, b: {x: real | a < x}, m: posnat,
40                 i: below(2 ^ m + 1), K: G, n: nat): RECURSIVE
41        Interval =
42    IF i = 0 THEN [|0|]
43    ELSE K(a, b, m, i, n) + RiemannSum_r2i(a, b, m, i - 1, K, n)
44    ENDIF
45     MEASURE i
46
47  F_integrable_cos: LEMMA Integrable?(lb(X), ub(X), cos)
48
49  eq_part_width: LEMMA
50    LET P = eq_partition(lb(X), ub(X), 2 ^ m + 1) IN
51      (FORALL (i: subrange(1, 2 ^ m)):
52         StrictInterval?([|P(i - 1), P(i)|]) IMPLIES
53          P(i) - P(i - 1) = (ub(X) - lb(X)) / 2 ^ m)
54
55  l: VAR {l: real | l > 0}
56
57  X1: VAR Interval
58
59  Mult_r2i_dist: LEMMA
60    [|lb(X1) * l, ub(X1) * l|] = Mult([|lb(X1), ub(X1)|], [|l|])
61
62  mm, M: VAR real
63
64  integ_r2i_bound: LEMMA
65    a < b AND
66     integrable?(a, b, f) AND
67     (FORALL (x: closed_interval(a, b)): mm <= f(x) AND f(x) <= M)
68     IMPLIES integral(a, b, f) ## [|mm * (b - a), M * (b - a)|]
69
70  integ_r2i_bound: THEOREM
71    integrable?(lb(X), ub(X), f) AND
72     (FORALL (x: closed_interval(lb(X), ub(X))): f(x) ## [|mm, M|])
73     IMPLIES
74     integral(lb(X), ub(X), f) ##Mult([|mm, M|],[|ub(X) - lb(X)|])
75
76  F_inclusion_cos: LEMMA
```

```
77      LET P = eq_partition(lb(X), ub(X), 2 ^ m + 1) IN
78      (FORALL (i: subrange(1, 2 ^ m)):
79        StrictInterval?([|P(i - 1), P(i)|]) AND x ##[|P(i - 1),P(i)|]
80            IMPLIES cos(x) ## Cos(n)([|P(i - 1), P(i)|]))
81
82    F_inclusion1_cos: LEMMA
83      LET P = eq_partition(lb(X), ub(X), 2 ^ m + 1) IN
84      (FORALL (i: subrange(1, 2 ^ m)):
85        StrictInterval?([|P(i - 1), P(i)|]) AND x ##[|P(i - 1),P(i)|]
86            IMPLIES
87            cos(x) * (ub(X) - lb(X)) / 2 ^ m ##
88            Cos(n)([|P(i - 1), P(i)|]) * [|(ub(X) - lb(X)) / 2 ^ m|])
89
90    F_inclusion2_cos: THEOREM
91      LET P = eq_partition(lb(X), ub(X), 2 ^ m + 1) IN
92       FORALL (i: subrange(1, 2 ^ m)):
93        StrictInterval?([|P(i - 1), P(i)|]) AND x ##[|P(i - 1),P(i)|]
94         IMPLIES
95         integral(lb([|P(i - 1), P(i)|]), ub([|P(i - 1), P(i)|]),cos)
96         ## Cos(n)([|P(i - 1), P(i)|]) * [|(ub(X) - lb(X)) / 2 ^ m|]
97
98    integral_bounds_cos: AXIOM
99      StrictInterval?(X) IMPLIES
100      integral(lb(X), ub(X), cos) ##
101       RiemannSum_r2i(lb(X), ub(X), m, 2 ^ m, F_1, n)
102
103   cos_test1: LEMMA integral(0, 1, cos) ## [|0.841, 0.852|]
104
105   d_arith_Sum(l: list[Interval], i: below[length(l) + 1]):
106    RECURSIVE Interval =
107     IF i = 0 THEN [|0|]
108     ELSE Add(d_arith_Sum(l, i - 1),nth(l, i - 1)) ENDIF
109      MEASURE i
110
111   list_sub_integ(a: real, b: {x: real | a < x}, m: posnat,
112                  i: below(2 ^ m + 1), K: G, n: nat, dd: nat,
113                  j: below[dd]): RECURSIVE
114         list[Interval] =
115     IF j = 0 THEN null
116     ELSE LET P = eq_partition(a, b, dd) IN
117            cons(RiemannSum_r2i(P(j - 1), P(j), m, i, K, n),
118                 list_sub_integ(a, b, m, i, K, n, dd, j - 1))
119     ENDIF
120      MEASURE j
121
122   listd_arith_sum(a: real, b: {x: real | a < x}, m: posnat,
123                   i: below(2 ^ m + 1), K: G, n: nat, dd: nat,
124                   j: below[dd],
125                   d: nat, z: below[d]): RECURSIVE
```

```
126          list[Interval] =
127      IF z = 0 THEN null
128      ELSE LET P= eq_partition(a, b, d) IN
129            LET l = list_sub_integ(P(z − 1), P(z), m, i, K, n, dd, j)
130            IN
131             cons(d_arith_Sum(l, length(l)),
132               listd_arith_sum(a, b, m, i, K, n, dd, j, d, z − 1))
133      ENDIF
134       MEASURE z
135  END Riemann_integ_interval_approx
```

Appendix B

Proof Scripts $\int_0^1 cos \in [|0.841, 0.852|]$

This appendix presents the mechanical proof scripts of $\int_0^1 cos \in [|0.841, 0.852|]$.

```
1
2  Proof scripts for theory Riemann_integ_interval_approx:
3
4
5  Riemann_integ_interval_approx.IMP_integral_def_TCC1: proved -
6   complete [shostak](n/a s)
7
8  ("" (assuming-tcc))
9
10
11 Riemann_integ_interval_approx.IMP_integral_def_TCC2: proved -
12 complete [shostak](n/a s)
13
14 ("" (expand 'not_one_element? '1)
15     (skosimp*) (inst 1 "x!1+1") (assert))
16
17
18 Riemann_integ_interval_approx.F_TCC1: proved -
19 complete [shostak](n/a s)
20
21 ("" (subtype-tcc))
22
23
24 Riemann_integ_interval_approx.F_TCC2: proved -
25  complete [shostak](n/a s)
26
27 ("
28  (skosimp*)
```

```
 (typepred "i!1")
 (assert)
 (case "i!1=0")
 (("1" (propax))
  ("2"
   (case "i!1>0")
   (("1"
     (rewrite -3 3)
     (case "length (eq_partition(a!1,b!1,1+2^m!1))=2^m!1")
     (("1" (grind)) ("2" (ground) (grind))))
    ("2" (grind))))))


Riemann_integ_interval_approx.F_TCC3: proved -
complete [shostak](n/a s)

(""
 (skosimp*)
 (assert)
 (case "i!1=0")
 (("1" (propax))
  ("2"
   (case "i!1>0")
   (("1"
     (ground)
     (rewrite -2 3)
     (case "length(eq_partition(a!1, b!1, 1 + 2 ^ m!1))=1+2^m!1")
     (("1" (rewrite -1 3) (ground))
      ("2" (ground) (expand 'eq_partition '1) (propax))))
    ("2" (rewrite -1 4) (typepred "i!1") (ground))))))


Riemann_integ_interval_approx.RiemannSum_r2i_TCC1: proved -
 complete [shostak](n/a s)

("" (subtype-tcc))


Riemann_integ_interval_approx.RiemannSum_r2i_TCC2: proved -
complete [shostak](n/a s)

("" (termination-tcc))


Riemann_integ_interval_approx.F_integrable_cos: proved -
complete [shostak](0.15 s)

(""
 (lemma "fundamental_indef[real]")
```

```
78   (("1"
79     (skosimp*)
80     (inst -1 "lb(X!1)" "ub(X!1)" "cos")
81     (assert)
82     (lemma "cos_cont_fun")
83     (propax))
84    ("2" (assert) (expand 'connected? '1) (propax))))
85
86
87  Riemann_integ_interval_approx.eq_part_width_TCC1: proved -
88  complete [shostak](n/a s)
89
90  ("" (subtype-tcc))
91
92
93  Riemann_integ_interval_approx.eq_part_width_TCC2: proved -
94  complete [shostak](n/a s)
95
96  ("" (subtype-tcc))
97
98
99  Riemann_integ_interval_approx.eq_part_width_TCC3: proved -
100  complete [shostak](n/a s)
101
102 ("" (subtype-tcc))
103
104
105 Riemann_integ_interval_approx.eq_part_width_TCC4: proved -
106  complete [shostak](n/a s)
107
108 ("" (subtype-tcc))
109
110
111 Riemann_integ_interval_approx.eq_part_width: proved -
112 complete [shostak](20.13 s)
113
114 ("" 
115  (lemma "width_eq_part")
116  (skosimp*)
117  (assert)
118  (skosimp*)
119  (name-replace "P_1"
120                " eq_partition(lb(X!1), ub(X!1), 2 ^ m!1 + 1)")
121  (inst? -1)
122  (assert)
123  (typepred
124   "width(lb(X!1),
125          ub(X!1),
126          eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1))")
```

```
127  (expand "width")
128  (typepred
129   "max({l: real |
130                  EXISTS (ii: below(length
131                  (eq_partition(lb(X!1),
132                   ub(X!1),
133                   1 + 2 ^ m!1)) - 1)):
134                  l = seq(eq_partition(lb(X!1),
135                          ub(X!1),1 + 2 ^ m!1))(1 + ii)
136                             - seq(eq_partition(lb(X!1),
137                           ub(X!1), 1 + 2 ^ m!1))(ii)})
138            ")
139  (("1"
140    (skosimp*)
141    (replace -1)
142    (reveal -1)
143    (expand "P_1")
144    (expand "eq_partition")
145    (field))
146   ("2"
147    (assert)
148    (expand "P_1")
149    (field)
150    (lemma "analysis@integral_def.width_TCC3")
151    (name-replace "PP"
152     "eq_partition[real](lb(X!1), ub(X!1), 1 + (2 ^ m!1))")
153    (inst -1 "lb(X!1)" "ub(X!1)" "PP" "seq(PP)")
154    (("1"
155      (inst?)
156      (("1"
157        (assert)
158        (hide-all-but (-1 1))
159        (split)
160        (("1" (propax)) ("2" (expand "PP") (propax))))
161       ("2" (typepred "X!1") (assert))))
162     ("2" (typepred "X!1") (assert))))
163   ("3" (typepred "X!1") (assert) (skosimp*) (assert))))
164
165
166 Riemann_integ_interval_approx.Mult_r2i_dist: proved -
167  complete [shostak](0.08 s)
168
169 ("" (grind))
170
171
172 Riemann_integ_interval_approx.integ_r2i_bound: proved -
173 complete [shostak](0.14 s)
174
175 (""
```

```
176   (skosimp*)
177   (lemma "integral_bound[real]")
178   (("1" (inst?) (expand "##") (musimp) (("1" (assert))
179   ("2" (assert))))
180    ("2" (assert) (expand 'connected? '1) (propax))))
181
182
183 Riemann_integ_interval_approx.integ_r2i_bound: proved -
184 complete [shostak](0.16 s)
185
186 (""
187  (lemma "integ_r2i_bound")
188  (skosimp*)
189  (inst? -1)
190  (inst -1 " M!1" "m!1")
191  (lemma "Mult_r2i_dist")
192  (inst -1 "[|m!1, M!1|]" "ub(X!1) - lb(X!1)")
193  (("1"
194    (case "[|m!1 *(ub(X!1)-lb(X!1)), M!1*(ub(X!1)- lb(X!1))|]
195    = Mult([|lb([|m!1, M!1|]),
196             ub([|m!1, M!1|])|],
197             [|ub(X!1)-lb(X!1)|])")
198    (("1"
199      (assert)
200      (expand "##")
201      (assert)
202      (typepred "X!1")
203      (hide-all-but (-1 1 -5))
204      (split)
205      (("1"
206        (assert)
207        (case "lb([|m!1, M!1|])=m!1")
208        (("1"
209          (case "ub([|m!1, M!1|])=M!1")
210          (("1" (rewrite -1) (expand 'strictinterval? '-2)
211           (propax))
212           ("2" (expand 'strictinterval? '-2) (propax))))
213         ("2" (expand 'strictinterval? '-1) (propax))))
214       ("2" (propax))))
215     ("2" (hide-all-but (-1 1)) (assert))))
216   ("2" (typepred "X!1") (expand 'strictinterval? '-1)
217         (ground))))
218
219
220 Riemann_integ_interval_approx.F_inclusion_cos: proved -
221 complete [shostak](72.05 s)
222
223 (""
224  (lemma "Cos_inclusion")
```

```
225  (skosimp*)
226  (assert)
227  (skolem!)
228  (inst?)
229  (assert)
230  (lazy-grind))
231
232
233 Riemann_integ_interval_approx.F_inclusion1_cos: proved -
234 complete [shostak](1.81 s)
235
236 (""
237  (lemma "Mult_inclusion")
238  (lemma "F_inclusion_cos")
239  (skosimp*)
240  (assert)
241  (skolem!)
242  (inst? -1)
243  (inst -1 "x!1")
244  (inst -1 "i!1")
245  (lemma "r2i_inclusion")
246  (inst -1 "(ub(X!1) - lb(X!1)) / 2 ^ m!1")
247  (inst -3
248   "Cos(n!1)([|eq_partition
249             (lb(X!1), ub(X!1), 1 + 2 ^ m!1)`seq(i!1 - 1),
250              eq_partition(lb(X!1),
251              ub(X!1), 1 + 2 ^ m!1)`seq(i!1)|])"
252   "[|(ub(X!1) - lb(X!1)) / 2 ^ m!1|]" "cos(x!1)"
253   "(ub(X!1) - lb(X!1) )/ 2 ^ m!1")
254  (assert)
255  (case "cos(x!1) * ((ub(X!1) -  lb(X!1)) / 2 ^ m!1)
256        = (sin(x!1) * ub(X!1) - cos(x!1) * lb(X!1))/ 2 ^ m!1 ")
257  (("1"
258    (assert)
259    (hide -2)
260    (hide -2)
261    (field)
262    (musimp)
263    (("1" (assert))
264     ("2"
265      (field)
266      (use "r2i_inclusion")
267      (lemma "F_inclusion_cos")
268      (inst?)
269      (inst -1 "n!1" "x!1")
270      (assert)
271      (inst -1 "i!1")
272      (assert))))
273   ("2" (field))))
```

```
274
275
276 Riemann_integ_interval_approx.F_inclusion_cos_TCC1: proved -
277  complete [shostak](n/a s)
278
279 ("" (lazy-grind))
280
281
282 Riemann_integ_interval_approx.F_inclusion_cos_TCC2: proved -
283 complete [shostak](n/a s)
284
285 (""
286  (lemma "F_integrable_cos")
287  (skosimp*)
288  (inst? -1)
289  (case "lb(X!1) < ub(X!1)")
290  (("1"
291    (ground)
292    (reveal -1)
293    (inst -1 "[|P!1`seq(i!1 - 1), P!1`seq(i!1)|]")
294    (case "P!1`seq(i!1 - 1)< P!1`seq(i!1)")
295    (("1" (ground) (expand 'integrable? '-2) (propax))
296     ("2" (ground) (expand 'strictinterval? '-5) (ground))))
297   ("2" (typepred "X!1") (expand 'strictinterval? '-1) (propax))))
298
299
300 Riemann_integ_interval_approx.F_inclusion_cos: proved -
301  complete [shostak](4.20 s)
302
303 (""
304  (lemma "integ_r2i_bound")
305  (skosimp*)
306  (assert)
307  (skosimp*)
308  (name-replace "F_cos_i"
309   "Cos(n!1)([|eq_partition(lb(X!1),
310                            ub(X!1),
311                            1 + 2 ^ m!1)`seq(i!1 - 1),
312               eq_partition(lb(X!1),
313                            ub(X!1),
314                            1 + 2 ^ m!1)`seq(i!1)|])")
315  (name-replace "P_i_1"
316   "eq_partition(lb(X!1),
317                 ub(X!1),
318                 1 + 2 ^ m!1)`seq(i!1 - 1)")
319  (name-replace "P_i"
320   "eq_partition(lb(X!1),
321                 ub(X!1),
322                  1 + 2 ^ m!1)`seq(i!1 )")
```

```
(inst -1 "ub(F_cos_i)"
         "[|P_i_1, P_i|]"
         "cos"
         "lb(F_cos_i)")
(assert)
(musimp)
(("1"
  (case "[|P_i - P_i_1|]=[|(ub(X!1) - lb(X!1)) / 2 ^ m!1|] ")
  (("1"
    (rewrite -1 1)
    (assert)
    (expand "F_cos_i")
    (case "[|lb(Cos(n!1)([|eq_partition(lb(X!1),
             ub(X!1), 1 + 2 ^ m!1)`seq(i!1 - 1),
             eq_partition(lb(X!1),
             ub(X!1), 1 + 2 ^ m!1)`seq(i!1)|])),
             ub(Cos(n!1)([|eq_partition(lb(X!1),
                          ub(X!1), 1 + 2 ^ m!1)`seq(i!1 - 1),
                          eq_partition(lb(X!1), ub(X!1),
                          1 + 2^m!1)`seq(i!1)|]))|]
                          =Cos(n!1)
                          ([|eq_partition(lb(X!1),
                          ub(X!1), 1 + 2 ^ m!1)`seq(i!1 - 1),
                          eq_partition(lb(X!1),
                          ub(X!1),
                          1 + 2 ^ m!1)`seq(i!1)|])")
    (("1" (rewrite -1 1))
     ("2"
      (expand "[||]")
      (hide-all-but (1 -3))
      (musimp)
      (("1" (propax)) ("2" (propax))))))
   ("2"
    (lemma "eq_part_width")
    (inst? -1)
    (assert)
    (inst -1 "i!1")
    (expand "P_i")
    (expand "P_i_1")
    (assert))))
 ("2"
  (lemma "F_inclusion_cos")
  (expand "P_i")
  (skosimp*)
  (typepred "x!2")
  (inst -3 "X!1" "m!1" "n!1" "x!2")
  (assert)
  (inst -3 "i!1")
  (expand "F_cos_i")
```

```
372     (musimp)
373     (("1"
374       (case "[|lb(Cos(n!1)
375       ([|eq_partition(lb(X!1),
376                       ub(X!1), 1 + 2 ^ m!1)`seq(i!1 - 1),
377                       eq_partition(lb(X!1),
378                                    ub(X!1),
379                                     1 + 2 ^ m!1)`seq(i!1)|])),
380                       ub(Cos(n!1)
381                             ([|eq_partition
382                               (lb(X!1),
383                                ub(X!1),
384                                1 + 2 ^ m!1)`seq(i!1 - 1),
385                              eq_partition
386                              (lb(X!1),
387                               ub(X!1),
388                               1 +2^m!1)`seq(i!1)|]))|]
389                               =Cos(n!1)
390                               ([|eq_partition(lb(X!1),
391                                   ub(X!1),
392                                   1 + 2 ^ m!1)`seq(i!1 - 1),
393                                 eq_partition
394                                 (lb(X!1),
395                                  ub(X!1),
396                                  1 + 2 ^ m!1)`seq(i!1)|]) ")
397       (("1" (rewrite -1 1))
398        ("2"
399         (hide-all-but (-1 -2 -3 -4 1))
400         (assert)
401         (expand "[||]")
402         (musimp)
403         (("1" (propax)) ("2" (propax))))))
404      ("2"
405       (expand "P_i_1")
406       (expand "P_i")
407       (expand "##")
408       (musimp)
409       (("1" (assert)) ("2" (assert)) ("3" (assert))
410        ("4" (assert))
411        ("5" (assert)) ("6" (assert)) ("7" (assert))
412        ("8" (assert))))
413      ("3" (expand "P_i_1") (propax))))
414    ("3"
415     (lemma "F_integrable_cos")
416     (inst?)
417     (expand "P_i_1")
418     (expand "P_i")
419     (assert)
420     (hide-all-but (-1 -2 1))
```

```
421     (expand "Integrable?")
422     (case "eq_partition(lb(X!1),
423                         ub(X!1),
424                         1 + 2 ^ m!1)`seq(i!1)
425             <
426              eq_partition(lb(X!1),
427                           ub(X!1),
428                           1 + 2 ^ m!1)`seq(i!1 - 1)")
429     (("1" (assert) (expand "StrictInterval?") (ground))
430      ("2" (expand "StrictInterval?") (ground))))))
431
432
433 Riemann_integ_interval_approx.integral_bounds_cos_TCC1: proved -
434 complete [shostak](n/a s)
435
436 (""
437  (lemma "F_integrable_cos")
438  (skosimp*)
439  (inst? -1)
440  (case "lb(X!1)<ub(X!1)")
441  (("1"
442    (ground)
443    (expand 'strictinterval? '-3)
444    (expand 'integrable? '-2)
445    (propax))
446   ("2" (expand 'strictinterval? '-2) (propax))))
447
448
449 Riemann_integ_interval_approx.integral_bounds_cos_TCC2: proved -
450  complete [shostak](n/a s)
451
452 ("" (subtype-tcc))
453
454
455 Riemann_integ_interval_approx.cos_test1_TCC1: proved -
456 complete [shostak](n/a s)
457
458 (""
459  (lemma "F_integrable_cos")
460  (ground)
461  (inst -1 "[| 0 , 1 |]")
462  (("1" (ground) (expand 'integrable? '-1) (propax))
463   ("2" (ground) (expand 'strictinterval? '1) (ground))))
464
465
466 Riemann_integ_interval_approx.cos_test1: proved -
467 complete [shostak](0.53 s)
468
469 (""
```

```
470   (lemma "integral_bounds_cos")
471   (inst -1 "[|0,1|]" "12" "2")
472   (("1"
473     (case "RiemannSum_r2i( lb([|0, 1|]),
474                            ub([|0, 1|]),
475                            12,
476                            2 ^ 12,
477                            F_1,
478                             2) << [| 0.841,0.852|]")
479     (("1"
480       (case "StrictInterval?([|0,1|])")
481       (("1"
482         (assert)
483         (lemma "Incl_Member")
484         (inst -1 " RiemannSum_r2i(0, 1, 12, 2 ^ 12, F_1, 2)"
485          "[|841/1000, 213/250|]")
486         (assert)
487         (inst -1 "integral(0, 1, cos)")
488         (ground))
489        ("2" (expand 'strictinterval? '1) (ground))))
490      ("2" (hide-all-but 1) (eval-formula))))
491    ("2" (expand 'strictinterval? '1) (ground))))
492
493
494 Riemann_integ_interval_approx.d_arith_Sum_TCC1: proved -
495  complete [shostak](n/a s)
496
497 ("" (subtype-tcc))
498
499
500 Riemann_integ_interval_approx.d_arith_Sum_TCC2: proved -
501  complete [shostak](n/a s)
502
503 ("" (termination-tcc))
504
505
506 Riemann_integ_interval_approx.d_arith_Sum_TCC3: proved -
507 complete [shostak](n/a s)
508
509 ("" (subtype-tcc))
510
511
512 Riemann_integ_interval_approx.list_sub_integ_TCC1: proved -
513 complete [shostak](n/a s)
514
515 ("
516  (skosimp*)
517  (case "j!1/=0")
518  (("1"
```

```
519     (typepred "j!1")
520     (case "length(PâĆĆ!1)=dd!1")
521     (("1" (assert))
522      ("2"
523       (assert)
524       (case "j!1/=0 and dd!1/=0")
525       (("1" (expand 'eq_partition '-4) (musimp))
526        ("2" (assert))))))
527    ("2" (assert))))
528
529
530  Riemann_integ_interval_approx.list_sub_integ_TCC2: proved -
531   complete [shostak](n/a s)
532
533  (""
534   (skosimp*)
535   (typepred "j!1")
536   (expand 'eq_partition '-2)
537   (musimp)
538   (grind))
539
540
541  Riemann_integ_interval_approx.list_sub_integ_TCC3: proved -
542  complete [shostak](n/a s)
543
544  (""
545   (skosimp*)
546   (lemma "parts_order[real]")
547   (inst -1 "a!1" "b!1" "PâĆĆ!1" "j!1-1" "j!1")
548   (("1" (assert)) ("2" (case "j!1/=0") (("1" (assert))
549    ("2" (assert))))))
550
551
552  Riemann_integ_interval_approx.list_sub_integ_TCC4: proved -
553  complete [shostak](n/a s)
554
555  (""
556   (skosimp)
557   (case "j!1/=0")
558   (("1"
559     (lemma "parts_order[real]")
560     (skolem! 2)
561     (inst -1 "a!1" "b!1" "PâĆĆ!1" "j!1-1" "j!1")
562     (("1"
563       (musimp)
564       (("1" (assert)) ("2" (assert)) ("3" (typepred "j!1")
565       (assert))
566        ("4" (assert))))
567      ("2" (assert)) ("3" (typepred "j!1") (assert))))
```

```
568    ("2" (assert))))
569
570
571  Riemann_integ_interval_approx.list_sub_integ_TCC5: proved -
572  complete [shostak](n/a s)
573
574  (""
575   (skosimp)
576   (typepred "j!1")
577   (case "j!1/=0")
578   (("1" (grind)) ("2" (grind))))
579
580
581  Riemann_integ_interval_approx.list_sub_integ_TCC6: proved -
582  complete [shostak](n/a s)
583
584  ("" (subtype-tcc))
585
586
587  Riemann_integ_interval_approx.listd_arith_sum_TCC1: proved -
588   complete [shostak](n/a s)
589
590  ("" (skosimp*) (typepred "z!1") (rewrite -2) (grind))
591
592
593  Riemann_integ_interval_approx.listd_arith_sum_TCC2: proved -
594   complete [shostak](n/a s)
595
596  ("" (skosimp*) (typepred "z!1") (rewrite -2) (grind))
597
598
599  Riemann_integ_interval_approx.listd_arith_sum_TCC3: proved -
600   complete [shostak](n/a s)
601
602  (""
603   (skosimp*)
604   (lemma "parts_order[real]")
605   (inst -1 "a!1" "b!1" "PâĆČ!1" "z!1-1" "z!1")
606   (("1" (assert)) ("2" (case "z!1/=0") (("1" (assert))
607   ("2" (assert))))))
608
609
610  Riemann_integ_interval_approx.listd_arith_sum_TCC4: proved -
611   complete [shostak](n/a s)
612
613  ("" (subtype-tcc))
614
615
616  Riemann_integ_interval_approx.listd_arith_sum_TCC5: proved -
```

```
617 complete [shostak](n/a s)
618
619 (""
620  (skosimp*)
621  (typepred "z!1")
622  (rewrite -1)
623  (typepred "z!1")
624  (rewrite -2)
625  (grind))
626
627
628 Riemann_integ_interval_approx.listd_arith_sum_TCC6: proved -
629 complete [shostak](n/a s)
630
631 ("" (termination-tcc))
632
633
634 Riemann_integ_interval_approx.listd_arith_sum_TCC7: proved -
635 complete [shostak](n/a s)
636
637 ("" (subtype-tcc))
```

Appendix C

Proof Scripts Soundness Theorem

THIS appendix presents the mechanical proof scripts of Theorem (9.3.1). This fundamental theory represent the corner stone of the parameterized automatic proof strategy conts-numerical-Riemann.

```
2  Proof scripts for file Riemann_integ_interval_approx.pvs:


5 Riemann_integ_interval_approx.IMP_integral_def_TCC1: proved -
      complete [shostak](n/a s)

7 ("" (assuming-tcc))


10 Riemann_integ_interval_approx.IMP_integral_def_TCC2: proved -
      complete [shostak](n/a s)

12 ("" (expand "not_one_element?") (skosimp*) (inst 1 "x!1+1") (grind)
      )


15 Riemann_integ_interval_approx.R_TCC1: proved - complete [shostak](n
      /a s)

17 ("" (subtype-tcc))


20 Riemann_integ_interval_approx.R_TCC2: proved - complete [shostak](n
      /a s)
```

```
22 ("" (subtype-tcc))

25 Riemann_integ_interval_approx.R_TCC3: proved - complete [shostak](n
       /a s)

27 ("" (subtype-tcc))

30 Riemann_integ_interval_approx.RiemannSum_R2I_TCC1: proved -
       complete [shostak](n/a s)

32 ("" (subtype-tcc))

35 Riemann_integ_interval_approx.RiemannSum_R2I_TCC2: proved -
       complete [shostak](n/a s)

37 ("" (subtype-tcc))

40 Riemann_integ_interval_approx.RiemannSum_R2I_TCC3: proved -
       complete [shostak](n/a s)

42 ("" (termination-tcc))

45 Riemann_integ_interval_approx.F_integrable_cos: proved - complete [
       shostak](n/a s)

47 (""
48  (lemma "fundamental_indef[real]")
49  (("1"
50    (skosimp*)
51    (inst -1 "lb(X!1)" "ub(X!1)" "cos")
52    (assert)
53    (lemma "cos_cont_fun")
54    (propax))
55   ("2" (assert) (expand 'connected? '1) (propax))))

58 Riemann_integ_interval_approx.F_integrable_cos2_TCC1: proved -
       complete [shostak](n/a s)

60 ("" (subtype-tcc))

63 Riemann_integ_interval_approx.F_integrable_cos2: proved - complete
       [shostak](n/a s)
```

```
65  ("" 
66   (lemma "F_integrable_cos")
67   (skosimp*)
68   (inst?)
69   (ground)
70   (expand "Integrable?")
71   (typepred "X!1")
72   (expand "StrictInterval?")
73   (ground))


76  Riemann_integ_interval_approx.eq_part_width_TCC1: proved - complete
        [shostak](n/a s)

78  ("" (subtype-tcc))


81  Riemann_integ_interval_approx.eq_part_width_TCC2: proved - complete
        [shostak](n/a s)

83  ("" (subtype-tcc))


86  Riemann_integ_interval_approx.eq_part_width_TCC3: proved - complete
        [shostak](n/a s)

88  ("" (subtype-tcc))


91  Riemann_integ_interval_approx.eq_part_width: proved - complete [
       shostak](n/a s)

93  ("" 
94   (lemma "width_eq_part")
95   (skosimp*)
96   (assert)
97   (skosimp*)
98   (name-replace "P_1" " eq_partition(lb(X!1), ub(X!1), 2 ^ m!1 + 1)
        ")
99   (inst? -1)
100  (assert)
101  (typepred
102   "width(lb(X!1), ub(X!1), eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m
        !1))")
103  (expand "width")
104  (typepred
105   "max({l: real |
106                  EXISTS (ii: below(length(eq_partition(lb(X!1), ub(
```

```
                        X!1), 1 + 2 ^ m!1)) - 1)):
107                     l = seq(eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m
                           !1))(1 + ii) - seq(eq_partition(lb(X!1), ub(X
                           !1), 1 + 2 ^ m!1))(ii)})
108           ")
109   (("1"
110     (skosimp*)
111     (replace -1)
112     (reveal -1)
113     (expand "P_1")
114     (expand "eq_partition")
115     (field))
116    ("2"
117     (assert)
118     (expand "P_1")
119     (field)
120     (lemma "analysis@integral_def.width_TCC3")
121     (name-replace "PP"
122      "eq_partition[real](lb(X!1), ub(X!1), 1 + (2 ^ m!1))")
123     (inst -1 "lb(X!1)" "ub(X!1)" "PP" "seq(PP)")
124     (("1"
125       (inst?)
126       (("1"
127         (assert)
128         (hide-all-but (-1 1))
129         (split)
130         (("1" (propax)) ("2" (expand "PP") (propax))))
131        ("2" (typepred "X!1") (assert))))
132      ("2" (typepred "X!1") (assert))))
133    ("3" (typepred "X!1") (assert) (skosimp*) (assert))))


136 Riemann_integ_interval_approx.Mult_r2i_dist: proved - complete [
       shostak](n/a s)

138 ("" (grind))


141 Riemann_integ_interval_approx.integ_r2i_bound: proved - complete [
       shostak](n/a s)

143 ("" 
144  (skosimp*)
145  (lemma "integral_bound[real]")
146  (("1" (inst?) (expand "##") (musimp) (("1" (assert)) ("2" (assert)
        )))
147   ("2" (assert) (expand 'connected? '1) (propax))))
```

```
150 Riemann_integ_interval_approx.simple_one: proved - complete [
       shostak](n/a s)

152 ("" (skosimp*) (expand "##") (typepred "x!1") (ground))


155 Riemann_integ_interval_approx.integ_r2i_boundâĆĆ: proved - complete
        [shostak](n/a s)

157 (""
158  (lemma "integ_r2i_bound")
159  (skosimp*)
160  (inst? -1)
161  (inst -1 " MâĆĂ!1" "mâĆĂ!1")
162  (lemma "Mult_r2i_dist")
163  (inst -1 "[|mâĆĂ!1, MâĆĂ!1|]" "ub(X!1) - lb(X!1)")
164  (("1"
165    (case "[|mâĆĂ!1 * (ub(X!1) - lb(X!1)), MâĆĂ!1 * (ub(X!1) - lb(X
          !1))|]= Mult([|lb([|mâĆĂ!1, MâĆĂ!1|]), ub([|mâĆĂ!1, MâĆĂ!1|])
          |], [|ub(X!1) - lb(X!1)|])")
166    (("1"
167      (assert)
168      (expand "##")
169      (assert)
170      (typepred "X!1")
171      (hide-all-but (-1 1 -5))
172      (split)
173      (("1"
174        (assert)
175        (case "lb([|mâĆĂ!1, MâĆĂ!1|])=mâĆĂ!1")
176        (("1"
177          (case "ub([|mâĆĂ!1, MâĆĂ!1|])=MâĆĂ!1")
178          (("1" (rewrite -1) (expand 'strictinterval? '-2) (propax))
179           ("2" (expand 'strictinterval? '-2) (propax))))
180         ("2" (expand 'strictinterval? '-1) (propax))))
181       ("2" (propax))))
182     ("2" (hide-all-but (-1 1)) (assert))))
183   ("2" (typepred "X!1") (expand 'strictinterval? '-1) (ground))))


186 Riemann_integ_interval_approx.Integ_Inclusion_fun_TCC1: proved -
       complete [shostak](n/a s)

188 (""
189  (inst 1 "cos")
190  (skosimp*)
191  (expand 'integ_inclus_fun? '1)
192  (lemma "F_integrable_cos")
193  (inst? -1)
```

```
194   (split)
195   (("1"
196     (expand "Integrable?")
197     (typepred "X!1")
198     (assert)
199     (expand "StrictInterval?")
200     (assert))
201    ("2"
202     (lemma "Cos_inclusion")
203     (inst? -1)
204     (lemma "Cos_inclusion")
205     (inst?)
206     (inst 1 "Cos")
207     (inst?)
208     (ground)
209     (skosimp*)
210     (inst -1 "x!1")
211     (ground)
212     (typepred "x!1")
213     (ground)
214     (expand "##")
215     (propax))))


218 Riemann_integ_interval_approx.Integ_inclus_f: proved - complete [
        shostak](n/a s)

220 (""
221   (skosimp*)
222   (expand "F_Bound?")
223   (lemma "integ_r2i_boundâĆĆ")
224   (inst? -1)
225   (musimp)
226   (hide-all-but (-2 1))
227   (skosimp*)
228   (case "[|lb(F!1(n!1)(X!1)), ub(F!1(n!1)(X!1))|]= F!1(n!1)(X!1)")
229   (("1" (assert) (rewrite -1) (inst -1 "x!1"))
230    ("2"
231     (hide-all-but 1)
232     (ground)
233     (expand "[||]")
234     (ground)
235     (musimp)
236     (("1" (propax)) ("2" (propax))))))


239 Riemann_integ_interval_approx.Integ_inclusion_f: proved - complete
        [shostak](n/a s)
```

```
241 ("" 
242  (lemma "Integ_inclus_f")
243  (skosimp*)
244  (inst? -1)
245  (assert)
246  (case "F!1(n!1)(X!1)= [|lb(F!1(n!1)(X!1)), ub(F!1(n!1)(X!1))|]")
247  (("1" (rewrite -1) (assert))
248   ("2"
249    (hide-all-but 1)
250    (expand "[||]")
251    (musimp)
252    (("1" (propax)) ("2" (propax))))))


255 Riemann_integ_interval_approx.Integ_inclusionâĆĆ_f_TCC1: proved -
       complete [shostak](n/a s)

257 ("" 
258  (skosimp*)
259  (expand 'strictinterval? '1)
260  (ground)
261  (musimp)
262  (ground)
263  (case "P!1`length > 1")
264  (("1" (use "parts_order") (musimp) (ground)) ("2" (ground))))


267 Riemann_integ_interval_approx.Integ_inclusionâĆĆ_f: proved -
       complete [shostak](n/a s)

269 ("" 
270  (lemma "Mult_inclusion")
271  (skosimp*)
272  (ground)
273  (skosimp*)
274  (case "g!1(x!1) * (ub(X!1) - lb(X!1)) / 2 ^ m!1 = (ub(X!1) * g!1(x
        !1) - lb(X!1) * g!1(x!1)) / 2 ^ m!1")
275  (("1"
276    (case-replace
277     "  (ub(X!1) * g!1(x!1) - lb(X!1) * g!1(x!1)) / 2 ^ m!1 = g!1(x
           !1) * (ub(X!1) - lb(X!1)) / 2 ^ m!1 ")
278    (("1"
279      (inst -2
280       "Eval!1(n!1)([|eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)`
             seq(i!1 - 1), eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)`
             seq(i!1)|]) "
281       " [|(ub(X!1) - lb(X!1)) / 2 ^ m!1|]" "g!1(x!1)"
282       "(ub(X!1) - lb(X!1)) / 2 ^ m!1")
283      (case "g!1(x!1) ## Eval!1(n!1)([|eq_partition(lb(X!1), ub(X!1)
```

```
           , 1 + 2 ^ m!1)`seq(i!1 - 1), eq_partition(lb(X!1), ub(X!1),
            1 + 2 ^ m!1)`seq(i!1)|]) AND (ub(X!1) - lb(X!1)) / 2 ^ m!1
            ## [|(ub(X!1) - lb(X!1)) / 2 ^ m!1|]")
284       (("1" (assert))
285        ("2"
286         (typepred "Eval!1")
287         (expand "F_Bound?")
288         (inst -1 "x!1")
289         (("1"
290           (assert)
291           (ground)
292           (expand "##" 1)
293           (musimp)
294           (("1" (ground)) ("2" (ground))))
295          ("2"
296           (typepred "x!1")
297           (hide-all-but (1 -4))
298           (expand "##")
299           (case "eq_partition[real](lb(X!1), ub(X!1), 1 + 2 ^ m!1)`
                 seq(i!1 - 1)=lb([|eq_partition(lb(X!1), ub(X!1), 1 + 2
                 ^ m!1)`seq(i!1 - 1), eq_partition(lb(X!1), ub(X!1), 1 +
                  2 ^ m!1)`seq(i!1)|])")
300           (("1" (rewrite -1) (ground)) ("2" (ground)))))))))
301      ("2" (field))))
302    ("2" (field))))


305 Riemann_integ_interval_approx.Integ_inclusionâĆČ_f_TCC1: proved -
       complete [shostak](n/a s)

307 (""
308  (lemma "parts_order")
309  (skosimp*)
310  (inst -1 "lb(X!1)" "ub(X!1)" "eq_partition(lb(X!1),ub(X!1), 2^m
        !1+1)"
311   "i!1-1" "i!1")
312  (("1" (ground))
313   ("2" (expand "eq_partition" 1) (typepred "i!1") (field 1))
314   ("3" (expand "eq_partition" 1) (typepred "i!1") (field 1))))


317 Riemann_integ_interval_approx.Integ_inclusionâĆČ_f_TCC2: proved -
       complete [shostak](n/a s)

319 (""
320  (skosimp*)
321  (typepred "g!1")
322  (inst -1
323   "             ([|finseq_appl[closed_interval[real](lb(X!1), ub(X
```

```
        !1))]
324                          (P!1)(i!1 - 1),
325                       finseq_appl[closed_interval[real](lb(X!1), ub(X
                             !1))]
326                          (P!1)(i!1)|])"
327    "1")
328   (("1" (expand "integ_inclus_fun?") (assert))
329    ("2"
330     (expand "StrictInterval?")
331     (ground)
332     (lemma "parts_order")
333     (inst?)
334     (ground))))


337  Riemann_integ_interval_approx.Integ_inclusionâĆČ_f: proved -
       complete [shostak](n/a s)

339  (""
340   (skosimp*)
341   (ground)
342   (lemma "Integ_inclusion_f")
343   (skosimp*)
344   (ground)
345   (inst? -1)
346   (("1"
347     (inst -1 "g!1")
348     (case " integrable?(lb([|eq_partition(lb(X!1), ub(X!1), 1 + 2 ^
            m!1)`seq(i!1 - 1), eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m
            !1)`seq(i!1)|]),
349                                       ub([|eq_partition(lb(X!1), ub(X
                                             !1), 1 + 2 ^ m!1)`seq(i!1 - 1)
                                             , eq_partition(lb(X!1), ub(X
                                             !1), 1 + 2 ^ m!1)`seq(i!1)|]),
                                              g!1)
350                            AND F_Bound?([|eq_partition(lb(X!1), ub(X!1)
                                  , 1 + 2 ^ m!1)`seq(i!1 - 1), eq_partition
                                  (lb(X!1), ub(X!1), 1 + 2 ^ m!1)`seq(i!1)
                                  |], g!1, n!1, Eval!1)")
351     (("1"
352       (ground)
353       (assert)
354       (case " [|(ub(X!1) - lb(X!1)) / 2 ^ m!1|]=[|eq_partition(lb(X
             !1), ub(X!1), 1 + 2 ^ m!1)`seq(i!1) - eq_partition(lb(X!1),
              ub(X!1), 1 + 2 ^ m!1)`seq(i!1 - 1)|] ")
355       (("1" (rewrite -1))
356        ("2"
357         (hide-all-but 1)
358         (lemma " eq_part_width")
```

```
359         (inst? -1)
360         (ground)
361         (inst -1 "i!1")
362         (case "StrictInterval?([|eq_partition(lb(X!1), ub(X!1), 1 +
               2 ^ m!1)`seq(i!1 - 1), eq_partition(lb(X!1), ub(X!1), 1 +
                2 ^ m!1)`seq(i!1)|])")
363         (("1" (ground))
364          ("2"
365           (typepred "X!1")
366           (hide-all-but -2 2)
367           (lemma "parts_order")
368           (expand "StrictInterval?")
369           (ground)
370           (inst? -1)
371           (ground))))))
372       ("2"
373        (musimp)
374        (("1" (typepred "Eval!1") (ground))
375         ("2"
376          (typepred "g!1")
377          (inst -1
378           "[|eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)`seq(i!1 - 1)
                , eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)`seq(i!1)
                |]"
379           "1")
380          (("1" (expand 'integ_inclus_fun? '-1) (musimp))
381           ("2"
382            (lemma "parts_order")
383            (expand "StrictInterval?")
384            (inst? -1)
385            (inst -1 "i!1")
386            (ground))))))
387       ("3"
388        (hide -1)
389        (ground)
390        (lemma "parts_order")
391        (expand "StrictInterval?")
392        (ground)
393        (inst?)
394        (ground))
395       ("4" (hide -1) (lemma "parts_order") (ground) (inst?) (ground))
          ))
396     ("2"
397      (expand "StrictInterval?")
398      (lemma "parts_order")
399      (inst?)
400      (inst -1 "i!1")
401      (ground))))
```

```
404 Riemann_integ_interval_approx.simple_two_TCC1: proved - complete [
       shostak](n/a s)

406 ("" (subtype-tcc))


409 Riemann_integ_interval_approx.simple_two_TCC2: proved - complete [
       shostak](n/a s)

411 ("" (subtype-tcc))


414 Riemann_integ_interval_approx.simple_two: proved - complete [
       shostak](n/a s)

416 ("" (skosimp*) (assert) (expand "eq_partition" 1) (field))


419 Riemann_integ_interval_approx.sub_integ_TCC1: proved - complete [
       shostak](n/a s)

421 ("" (subtype-tcc))


424 Riemann_integ_interval_approx.sub_integ_TCC2: proved - complete [
       shostak](n/a s)

426 ("" (subtype-tcc))


429 Riemann_integ_interval_approx.sub_integ_TCC3: proved - complete [
       shostak](n/a s)

431 ("" (subtype-tcc))


434 Riemann_integ_interval_approx.sub_integ_TCC4: proved - complete [
       shostak](n/a s)

436 (""
437  (skosimp*)
438  (lemma "parts_order")
439  (inst -1 " a!1" "b!1" "eq_partition(a!1,b!1,2^m!1+1)" "i!1-1" "i
        !1")
440  (("1" (ground)) ("2" (expand "eq_partition") (field 1))
441   ("3" (expand "eq_partition") (field 1))))
```

```
444 Riemann_integ_interval_approx.sub_integ_TCC5: proved - complete [
       shostak](n/a s)

446 (""
447  (skosimp*)
448  (typepred "g!1")
449  (inst -1
450   "([|finseq_appl[closed_interval[real](a!1, b!1)](P!1)(i!1 - 1),
451                   finseq_appl[closed_interval[real](a!1, b!1)](P!1)
                        (i!1)|])"
452   "1")
453  (("1" (expand "integ_inclus_fun?") (ground))
454   ("2"
455    (expand "StrictInterval?")
456    (lemma "parts_order")
457    (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,2^m!1+1)" "i!1-1" "i
          !1")
458    (("1" (ground))
459     ("2" (expand "eq_partition") (typepred "i!1") (field 1))
460     ("3" (expand "eq_partition") (typepred "i!1") (field 1))))))


463 Riemann_integ_interval_approx.sum_n_split_TCC1: proved - complete [
       shostak](n/a s)

465 ("" (subtype-tcc))


468 Riemann_integ_interval_approx.sum_n_split_TCC2: proved - complete [
       shostak](n/a s)

470 ("" (termination-tcc))


473 Riemann_integ_interval_approx.sum_n_split_TCC3: proved - complete [
       shostak](n/a s)

475 (""
476  (lemma "parts_order")
477  (skosimp*)
478  (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,2^m!1+1)" "0" "1")
479  (("1" (ground)) ("2" (expand "eq_partition") (field 1))
480   ("3" (ground))))


483 Riemann_integ_interval_approx.sum_n_split_TCC4: proved - complete [
       shostak](n/a s)

485 (""
```

```
486   (skosimp*)
487   (typepred "g!1")
488   (inst -1
489    "[|finseq_appl[closed_interval[real](a!1, b!1)](P!1)(0),
490                   finseq_appl[closed_interval[real](a!1, b!1)](P!1)
                          (1)|]"
491    "1")
492   (("1" (expand "integ_inclus_fun?") (ground))
493    ("2"
494     (ground)
495     (expand "StrictInterval?")
496     (ground)
497     (lemma "parts_order")
498     (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,2^m!1+1)" "0" "1")
499     (ground))))


502 Riemann_integ_interval_approx.sum_n_split_TCC5: proved - complete [
       shostak](n/a s)

504 ("" (skosimp*) (typepred "i!1") (field 1))


507 Riemann_integ_interval_approx.sum_n_split_TCC6: proved - complete [
       shostak](n/a s)

509 ("" (termination-tcc))


512 Riemann_integ_interval_approx.general_integ_split_TCC1: proved -
       complete [shostak](n/a s)

514 ("" (subtype-tcc))


517 Riemann_integ_interval_approx.general_integ_split_TCC2: proved -
       complete [shostak](n/a s)

519 ("" (subtype-tcc))


522 Riemann_integ_interval_approx.general_integ_split_TCC3: proved -
       complete [shostak](n/a s)

524 (""
525   (skosimp*)
526   (lemma "parts_order")
527   (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,2^m!1+1)" "0" "i!1")
528   (("1" (ground))
```

```
529    ("2" (typepred "i!1") (expand "eq_partition") (field 1))
530    ("3" (field 1))))


533  Riemann_integ_interval_approx.general_integ_split_TCC4: proved -
       complete [shostak](n/a s)

535  (""
536   (skosimp*)
537   (typepred "g!1")
538   (ground)
539   (inst -1 "[|P!1`seq(0), P!1`seq(i!1)|]" "1")
540   (("1" (expand "integ_inclus_fun?") (ground))
541    ("2"
542     (expand "StrictInterval?")
543     (lemma "parts_order")
544     (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,2^m!1+1)" "0" "i!1")
545     (("1" (ground))
546      ("2" (typepred "i!1") (expand "eq_partition") (field 1))))))


549  Riemann_integ_interval_approx.general_integ_split: proved -
       complete [shostak](n/a s)

551  (""
552   (assert)
553   (skolem!)
554   (flatten)
555   (induct "i")
556   (("1" (expand "sum_n_split") (ground))
557    ("2"
558     (skosimp*)
559     (name-replace "P_1" "eq_partition")
560     (case " integral(P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(0),
561                                                 P_1(a!1, b!1, 1 + 2 ^ m
                                                     !1)`seq(k!1 + 1), g!1)
                                                     = integral(P_1(a!1, b
                                                     !1, 1 + 2 ^ m!1)`seq
                                                     (0), P_1(a!1, b!1, 1 +
                                                      2 ^ m!1)`seq(k!1), g
                                                     !1)+ integral(P_1(a!1,
                                                      b!1, 1 + 2 ^ m!1)`seq
                                                     (k!1), P_1(a!1, b!1, 1
                                                      + 2 ^ m!1)`seq(k!1 +
                                                     1), g!1) ")
562     (("1"
563       (rewrite -3)
564       (case "sum_n_split(a!1, b!1, m!1, k!1 + 1, g!1)= sum_n_split(a
             !1, b!1, m!1, k!1, g!1) +integral(P_1(a!1, b!1, 1 + 2 ^ m
```

```
            !1)`seq(k!1), P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(k!1 + 1), g!1)
            ")
565       (("1" (assert))
566        ("2"
567         (case " integral(P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(0),
568                                                          P_1(a!1, b!1,
                                                               1 + 2 ^ m
                                                              !1)`seq(k
                                                              !1 + 1), g
                                                              !1)=
                                                              sum_n_split
                                                              (a!1, b!1,
                                                               m!1, k!1
                                                              + 1, g!1)
                                                              ")
569          (("1" (propax))
570           ("2"
571            (hide 1)
572            (both-sides -
573             "integral(P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(k!1),P_1(a!1, b
                 !1, 1 + 2 ^ m!1)`seq(k!1 + 1), g!1)"
574             -1)
575            (("1"
576              (field -1)
577              (hide 2)
578              (case-replace
579               " sum_n_split(a!1, b!1, m!1, k!1, g!1)=integral(P_1(a
                   !1, b!1, 1 + 2 ^ m!1)`seq(0), P_1(a!1,b!1, 1 + 2 ^ m
                   !1)`seq(1 + k!1), g!1) -integral(P_1(a!1, b!1, 1 + 2
                    ^ m!1)`seq(k!1),P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(1 +
                   k!1), g!1) ")
580              (("1"
581                (field 1)
582                (expand 'sum_n_split '1)
583                (case-replace
584                 " sum_n_split(a!1, b!1, m!1, k!1, g!1)=integral(P_1(a
                     !1, b!1, 1 + 2 ^ m!1)`seq(0), P_1(a!1,b!1, 1 + 2 ^
                      m!1)`seq(1 + k!1), g!1) -integral(P_1(a!1, b!1, 1
                      + 2 ^ m!1)`seq(k!1),P_1(a!1, b!1, 1 + 2 ^ m!1)`
                     seq(1 + k!1), g!1) ")
585                (field 1)
586                (field 1)
587                (assert)
588                (expand 'p_1 '1)
589                (propax))
590               ("2" (assert))))
591             ("2" (field 1))))))))
592       ("2"
593        (lemma "integral_split[real]")
```

```
594       (("1"
595         (inst -1 "P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(0)"
596          "P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(k!1)"
597          "P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(k!1 + 1)" "g!1")
598         (case "P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(0) <
599                                                 P_1(a!1, b!1, 1 + 2 ^
                                                     m!1)`seq(k!1)
600                                                 AND
601                                                 P_1(a!1, b!1, 1 + 2 ^
                                                     m!1)`seq(k!1) <
602                                                  P_1(a!1, b!1, 1 + 2
                                                     ^ m!1)`seq(k!1 +
                                                     1)
603                                                  AND
604                                                  integrable?(P_1(a!1,
                                                      b!1, 1 + 2 ^ m
                                                     !1)`seq(0),
605                                                              P_1(a!1,
                                                                  b!1,
                                                                  1 +
                                                                 2 ^ m
                                                                 !1)`
                                                                 seq(k
                                                                 !1),
                                                                 g!1)
606                                                   AND
607                                                  integrable?(P_1(a
                                                      !1, b!1, 1 + 2 ^
                                                      m!1)`seq(k!1),
608                                                              P_1(a
                                                                  !1,
                                                                  b!1,
                                                                   1 +
                                                                   2 ^
                                                                   m
                                                                  !1)`
                                                                  seq(
                                                                  k!1
                                                                  + 1)
                                                                  , g
                                                                  !1)
                                                                  ")
609         (("1" (assert))
610          ("2"
611           (lemma "parts_order")
612           (inst?)
613           (assert)
614           (lemma "parts_order")
615           (expand "P_1")
```

```
616           (inst -1 "a!1" "b!1" "eq_partition(a!1, b!1, 1 + 2 ^ m!1)"
617            "k!1" "k!1+1")
618           (("1"
619             (assert)
620             (hide-all-but 1)
621             (typepred "g!1")
622             (inst -1
623              "[|eq_partition(a!1,b!1,1+2^m!1)`seq(0),eq_partition(a
                    !1,b!1,1+2^m!1)`seq(k!1)|]"
624              "1")
625             (("1"
626               (expand 'integ_inclus_fun? '-1)
627               (assert)
628               (ground)
629               (hide -2)
630               (typepred "g!1")
631               (inst -1
632                "[|eq_partition(a!1,b!1,1+2^m!1)`seq(k!1),
                      eq_partition(a!1,b!1,1+2^m!1)`seq(1+k!1)|]"
633                "1")
634               (("1" (expand 'integ_inclus_fun? '-1) (assert))
635                ("2"
636                 (expand 'strictinterval? '1)
637                 (lemma "parts_order")
638                 (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)"
639                  "k!1" "k!1+1")
640                 (("1" (assert))
641                  ("2"
642                   (expand 'eq_partition '1)
643                   (typepred "k!1")
644                   (field 1))))
645                 ("3" (expand 'eq_partition '1) (field 1))
646                 ("4" (expand 'eq_partition '1) (field 1))))
647               ("2"
648                (expand 'strictinterval? '1)
649                (lemma "parts_order")
650                (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)"
                       "0"
651                 "k!1")
652                (assert))
653               ("3"
654                (expand 'eq_partition '1)
655                (typepred "k!1")
656                (field 1))))
657              ("2" (expand 'eq_partition '1) (field 1))
658              ("3" (expand 'eq_partition '1) (field 1))))))
659         ("2" (expand "connected?") (propax))))
660       ("3"
661        (typepred "g!1")
```

```
662       (inst -1
663        "[|P_1(a!1, b!1, 1 + 2 ^ m!1)`seq(k!1),P_1(a!1, b!1, 1 + 2 ^
              m!1)`seq(1+k!1) |]"
664        "1")
665       (("1" (expand "integ_inclus_fun?") (assert))
666        ("2"
667         (lemma "parts_order")
668         (expand 'strictinterval? '1)
669         (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)" "k!1"
670          "1+k!1")
671         (("1" (assert) (ground) (expand "P_1") (propax))
672          ("2" (expand 'eq_partition '1) (field 1))
673          ("3" (expand 'eq_partition '1) (field 1))))))
674      ("4"
675       (lemma "parts_order")
676       (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)" "k!1"
677        "1+k!1")
678       (("1" (expand "P_1") (assert))
679        ("2" (expand 'eq_partition '1) (field 1))
680        ("3" (expand 'eq_partition '1) (field 1))))
681      ("5" (expand 'p_1 '1) (expand 'eq_partition '1) (field 1))
682      ("6" (expand 'p_1 '1) (expand 'eq_partition '1) (field 1))
683      ("7" (expand 'p_1 '1) (expand 'eq_partition '1) (field 1))))
684    ("3"
685     (typepred "g!1")
686     (skosimp*)
687     (inst -1
688      "[|eq_partition[real](a!1, b!1, 1 + 2 ^ m!1)`seq(0),
             eq_partition[real](a!1, b!1, 1 + 2 ^ m!1)`seq(i!2)|]"
689      "1")
690     (("1" (expand "integ_inclus_fun?") (assert))
691      ("2"
692       (expand "StrictInterval?")
693       (lemma "parts_order")
694       (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)" "0" "i
              !2")
695       (assert))))
696    ("4"
697     (skosimp*)
698     (lemma "parts_order")
699     (inst -1 "a!1" "b!1" "eq_partition(a!1,b!1,1+2^m!1)" "0" "i!2")
700     (assert))
701    ("5" (skosimp*) (typepred "i!2") (expand "eq_partition") (field
          1))
702    ("6" (assert)) ("7" (expand 'eq_partition '1) (propax))
703    ("8" (field 1))))


706 Riemann_integ_interval_approx.general_integ_splitâĆA_TCC1: proved -
```

```
        complete [shostak](n/a s)

708  ("" 
709   (skosimp*)
710   (typepred "g!1")
711   (inst?)
712   (inst -1 "1")
713   (ground)
714   (expand "integ_inclus_fun?")
715   (propax))


718  Riemann_integ_interval_approx.general_integ_splitâĆA_TCC2: proved -
        complete [shostak](n/a s)

720  ("" 
721   (induct "m")
722   (("1" (typepred "m!1") (propax)) ("2" (typepred "m!1") (propax))
723    ("3" (ground))
724    ("4"
725     (skosimp*)
726     (ground)
727     (("1"
728       (field 1)
729       (typepred "j!1")
730       (field 1)
731       (case " 2^(1+j!1)= 2*2^(j!1)")
732       (("1" (rewrite -1) (lazy-grind)) ("2" (lazy-grind))))
733      ("2" (lazy-grind))))))


736  Riemann_integ_interval_approx.general_integ_splitâĆA: proved -
        complete [shostak](n/a s)

738  ("" 
739   (lemma "general_integ_split")
740   (skolem!)
741   (inst -1 "lb(X!1)" "ub(X!1)" "g!1" " m!1")
742   (assert)
743   (musimp)
744   (("1"
745     (inst -1 "2^m!1")
746     (lemma "simple_two")
747     (inst?)
748     (case "eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)`seq(0)= lb(X
           !1)")
749     (("1"
750       (case "eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)`seq(2^m!1)=
             ub(X!1)")
```

```
751       (("1"
752         (case " ub(X!1)=eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)`
                seq(2^m!1)")
753         (("1"
754           (case " lb(X!1)=eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m
                  !1)`seq(0)")
755           (("1" (hide -3) (hide -3) (rewrite -1) (rewrite -2) (
                  ground))
756            ("2" (ground))))
757          ("2" (ground))))
758        ("2" (ground))))
759      ("2" (ground))))
760    ("2" (ground) (typepred "X!1") (expand "StrictInterval?") (propax
         ))))


763  Riemann_integ_interval_approx.trivial: proved - complete [shostak](
       n/a s)

765  ("" (skosimp*) (lazy-grind))


768  Riemann_integ_interval_approx.Fundamental_Riemann_inclusionâĆA_TCC1
       : proved - complete [shostak](n/a s)

770  ("" (subtype-tcc))


773  Riemann_integ_interval_approx.Fundamental_Riemann_inclusionâĆA_TCC2
       : proved - complete [shostak](n/a s)

775  ("" (subtype-tcc))


778  Riemann_integ_interval_approx.Fundamental_Riemann_inclusionâĆA_TCC3
       : proved - complete [shostak](n/a s)

780  (""
781   (skosimp*)
782   (expand "StrictInterval?")
783   (lemma "parts_order")
784   (inst -1 "lb(X!1)" "ub(X!1)" "eq_partition(lb(X!1),ub(X!1),2^m
         !1+1)"
785    "i!1-1" "i!1")
786   (("1" (ground) (ground) (rewrite -2) (ground) (lazy-grind))
787    ("2" (expand "eq_partition") (typepred "i!1") (field 1))
788    ("3" (expand "eq_partition") (typepred "i!1") (field 1))))
```

```
Riemann_integ_interval_approx.Fundamental_Riemann_inclusionâĆĄ:
    proved - complete [shostak](n/a s)

("" 
 (skosimp*)
 (assert)
 (induct "i")
 (("1"
   (lemma "Integ_inclusionâĆČ_f")
   (skosimp*)
   (expand "sum_n_split")
   (expand "RiemannSum_R2I")
   (expand "R")
   (simplify)
   (assert)
   (assert)
   (inst?)
   (hide-all-but 1)
   (use "expt_ge1")
   (assert)
   (lemma "Integ_inclusionâĆČ_f")
   (simplify)
   (inst?)
   (inst -1 "g!1" "n!1" "lb(X!1)")
   (assert)
   (inst -1 "1")
   (musimp)
   (("1"
     (inst?)
     (("1" (ground)) ("2" (typepred "Eval!1") (inst -1 "1") (ground
        ))))
    ("2" (expand "##") (ground))))
  ("2"
   (skosimp*)
   (inst?)
   (expand "sum_n_split" 1)
   (expand "RiemannSum_R2I" 1)
   (case "integral(lb([|finseq_appl[closed_interval[real](lb(X!1),
      ub(X!1))](eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1))(k!1),
      finseq_appl[closed_interval[real](lb(X!1), ub(X!1))](
      eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1)) (1 + k!1)|]), ub
      ([|finseq_appl[closed_interval[real](lb(X!1), ub(X!1))](
      eq_partition(lb(X!1), ub(X!1), 1 + 2 ^ m!1))(k!1),finseq_appl
      [closed_interval[real](lb(X!1), ub(X!1))](eq_partition(lb(X
      !1), ub(X!1), 1 + 2 ^ m!1))(1 + k!1)|]),g!1) ## R(lb(X!1), ub
      (X!1), m!1, 1 + k!1, n!1, Eval!1)")
   (("1"
     (lemma "Add_inclusion")
     (inst -1 "R(lb(X!1), ub(X!1), m!1, 1 + k!1, n!1, Eval!1)"
```

```
830       "RiemannSum_R2I(lb(X!1), ub(X!1), m!1, k!1, n!1, Eval!1, R)"
831       "integral(lb
832                                 ([|finseq_appl[closed_interval[real](
                                        lb(X!1), ub(X!1))]
833                                     (eq_partition(lb(X!1), ub(X!1),
                                         1 + 2 ^ m!1))(k!1),
834                                    finseq_appl[closed_interval[real](
                                       lb(X!1), ub(X!1))]
835                                     (eq_partition(lb(X!1), ub(X!1),
                                         1 + 2 ^ m!1))
836                                     (1 + k!1)|]),
837                               ub
838                                 ([|finseq_appl[closed_interval[real](
                                        lb(X!1), ub(X!1))]
839                                     (eq_partition(lb(X!1), ub(X!1),
                                         1 + 2 ^ m!1))(k!1),
840                                    finseq_appl[closed_interval[real](
                                       lb(X!1), ub(X!1))]
841                                     (eq_partition(lb(X!1), ub(X!1),
                                         1 + 2 ^ m!1))
842                                     (1 + k!1)|]),
843                               g!1)"
844       "sum_n_split(lb(X!1), ub(X!1), m!1, k!1, g!1)")
845      (assert))
846     ("2"
847      (hide 2)
848      (expand "R")
849      (lemma "Integ_inclusionâĆČ_f")
850      (inst?)
851      (inst -1 "g!1" "n!1"
852       "lb([|finseq_appl[closed_interval[real](lb(X!1), ub(X!1))]
853                                       (eq_partition(lb(X!1), ub(X!1)
                                           , 1 + 2 ^ m!1))
854                                       (k!1),
855                                     finseq_appl[closed_interval[real](
                                        lb(X!1), ub(X!1))]
856                                       (eq_partition(lb(X!1), ub(X!1)
                                           , 1 + 2 ^ m!1))
857                                       (1 + k!1)|])")
858      (assert)
859      (inst -1 "k!1+1")
860      (("1"
861        (ground)
862        (("1"
863          (inst?)
864          (typepred "Eval!1")
865          (inst -1 "k!1+1")
866          (("1" (ground)) ("2" (lemma "trivial") (inst? -1) (ground)
              )))
```

```
867           ("2"
868            (hide 2)
869            (expand "##")
870            (ground)
871            (lemma "parts_order")
872            (lemma "parts_order")
873            (inst -1 "lb(X!1)" "ub(X!1)"
874             "eq_partition(lb(X!1),ub(X!1),2^m!1+1)" "k!1" "k!1+1")
875            (("1" (ground))
876             ("2" (typepred "k!1") (expand "eq_partition") (field 1)))
                 )))
877         ("2"
878          (field 1)
879          (ground)
880          (hide 2)
881          (hide -2)
882          (field)
883          (lemma "trivial")
884          (inst?)
885          (ground))))))
886    ("3" (typepred "X!1") (expand "StrictInterval?") (assert))
887    ("4"
888     (skosimp*)
889     (expand "StrictInterval?")
890     (lemma "parts_order")
891     (inst? -1)
892     (inst -1 "i!3")
893     (ground))
894    ("5" (skosimp*) (typepred "i!3") (expand "eq_partition") (field))
895    ("6" (typepred "X!1") (skosimp*) (expand "StrictInterval?") (
          propax))
896    ("7" (skosimp*) (expand "eq_partition") (typepred "i!3") (field))
897    ("8" (use "expt_ge1"))))


900  Riemann_integ_interval_approx.Simple_Riemann_Soundness_TCC1: proved
         - complete [shostak](n/a s)

902  (""
903   (skosimp*)
904   (lemma "parts_order")
905   (ground)
906   (inst?)
907   (ground)
908   (typepred "X!1")
909   (hide-all-but (-1 1))
910   (expand "StrictInterval?" -1)
911   (lemma "parts_order")
912   (ground)
```

```
913   (inst?)
914   (typepred "X!1")
915   (hide-all-but (-1 1))
916   (expand "StrictInterval?" -1)
917   (lemma "parts_order")
918   (inst?)
919   (inst?)
920   (expand "StrictInterval?")
921   (assert)
922   (lemma "parts_order")
923   (ground)
924   (inst?)
925   (ground))


928  Riemann_integ_interval_approx.Simple_Riemann_Soundness: proved -
        complete [shostak](n/a s)

930  (""
931   (skosimp*)
932   (assert)
933   (skosimp*)
934   (lemma "Fundamental_Riemann_inclusion1")
935   (inst?)
936   (inst -1 "X!1" "g!1" "n!1")
937   (assert)
938   (inst -1 "2^m!1")
939   (inst -1 "Eval!1")
940   (case "integral(lb(X!1), ub(X!1), g!1) = sum_n_split(lb(X!1), ub(X
         !1), m!1, 2 ^ m!1, g!1)")
941   (("1" (rewrite -1)) ("2" (lemma "general_integ_split1") (inst?))))
```

Appendix D

Binary agreement protocol

D.1 PSS Binary Agreement Rings

In this section, we use theorem proving techniques to generalize a small synthesized agreement protocol on a unidirectional ring (presented in [25]) that is weakly stabilizing for any $3 \leq n \leq 6$, where n is the number of processes. First, we introduce the agreement protocol.

Example D.1.1. *The agreement protocol, denoted AG(n), includes $n > 2$ processes located on a unidirectional ring. Each process p_j has a variable c_j with a domain $Dom = \{0, 1\}$. Thus, AG(n) has the set of variables $V_{AG(n)} = \{c_0, c_1, ..., c_{n-1}\}$. Each process can read but not write its left neighbor; i.e., $Read_p = \{c_{j\ominus 1}, c_j\}$ while $Write_p = \{c_j\}$. Each process has the following parameterized action:*

$$A_j : c_{j\ominus 1} < c_j \rightarrow c_j := c_{j\ominus 1} \tag{D.1}$$

If the variable c_j has a value greater than its predecessor then the process p_j sets the value of c_j to $c_{j\ominus 1}$ (which is equal to 0 due to the binary domain). A legitimate state of the AG(n) protocol is a state where all variables have the same value. Let I denote the set of legitimate states of AG(n). If the condition $(c_{j\ominus 1} < c_j)$ holds for a process j, then we call it a locally corrupted *process; otherwise, we say the protocol is silent at process j.*

D.1.1 PVS Specification of AG(n)

In our PVS specification of AG(n) there are n processes, where n is a theory parameter of type *positive natural* numbers, denoted *posnat*. We assume that $n > 2$. We define the type Bin: below[2] to capture the domain of binary variables. Moreover, we formalize a global state of AG(n) as a finite binary sequence of length n; i.e., STC:

NONEMPTY_TYPE {s:finseq (Bin) | s`length=n}. Furthermore, since each variable c_j has two pieces of information namely the process position and its value, we model its type by a tuple ndx_varb:TYPE+=[Bin,below[n]].

D.1.1.1 Specifying the Processes

We define the set of readable variables of a process p_j by the function $READ_p$.

Definition D.1.1. READ_p(s:STC, j:below[n]): set[ndx_varb]= {L:(ValPos(s, j $\ominus$ i)) | i=0,1}, where $\ominus$ denotes subtraction modulo 2.

Similarly, we define the set of writable variables of process j by the function $WRITE_p$.

Definition D.1.2. WRITE_p(s:STC,j:below[n]): set[ndx_varb]= {L:ndx_varb | L = ValPos(s, j)}

Finally, we define the set of transitions of each process p_j as the set of all possible transitions generated by the action function of p_j. The action function will not be activated on a process j unless the state was locally corrupted at process j. We define the function action(s, j) such that a locally corrupted state s at process j will be mapped non-deterministically to the state generated by action(s, j).

```
2  action(s:{ state:STC| not is_LEGT?(state)},j:below[n]): STC = TABLE
3  +-------------------------------------------------------------------------+
4  |not is_LEGT?(s)     |  (# length := n,seq := (LAMBDA (i:below[n]):
5                           IF (i=j and LocallyCorrupted?(s,i) )
6                            THEN 0 ELSE s`seq(i) ENDIF)  #)          ||
7  +-------------------------------------------------------------------------+
8  |is_LEGT?(s)         |                                                    ||
9  +-------------------------------------------------------------------------+
10 ENDTABLE
```

Listing D.1: action of binary agreement protocol at process j

DELTA_p(s, j) captures the set of transitions of a process j originated at a global state s.

Definition D.1.3. DELTA_p(s:S_ill, j:below[n]): set[Transition] = { tr: Transition | active_LocallyCorrupted?(s,j) $\wedge$ tr = (s, action(s,j)) }

For an arbitrary global state s, we now define a process of AG(n).

Definition D.1.4. PRS_p(s:STC,j:below[n])*:* p_process = (READ_p(s,j), WRITE_p(s,j), DELTA_p(s,j))

D.1.1.2 Specifying the Parameterized Protocol AG(n)

We parameterize the definition of the protocol AG(n) with an arbitrary illegitimate state s. To capture the set of processes of the AG(n) protocol, we define the function $PROC\_prt$ as follows:

Definition D.1.5. PROC_prt(s:STC): set[p_process]={p:p_process | ∃ (j:below[n]): p = PRS_p(s,j) }

The function VARB_prt returns the set of variables of the protocol AG(n).

Definition D.1.6. VARB_prt(s:S_ill): set[ndx_varb]= {v:ndx_varb | ∃ (j:below[n]):v ∈ WRITE_p(s,j))}

Likewise, the function DELTA_prt returns the set of transitions of the protocol AG(n).

Definition D.1.7. DELTA_prt(s:STC): set[Transition]= {tr:Transition | ∃ (j:below[n]): tr ∈ DELTA_p(s,j)}

Thus, starting from an initial state s the formalization of AG(n) is given by defining AG(n) as a function of type nd_Protocol with the images of READ_prt, WRITE_prt and DELTA_prt functions over the state s as the components of AG(n).

AG_n_s: nd_Protocol= (PROC_prt(s), VARB_prt(s), DELTA_prt(s))

D.1.2 Weak Stabilization of AG(n)

In this section, we show that the Add_Weak algorithm generates a weakly stabilizing version of the protocol AG(n) for any $n > 2$. To this end, we build a similar proof of coloring protocol by showing that AG(n) has a recursive constructor function.

This implicitly means that from any state outside the set of legitimate states I of the protocol AG(n) (for any $n > 2$), there exists a prefix that reaches I. (PVS specifications and proofs of binary agreement are available at `https://sites.google.com/a/mtu.edu/amertahatpvs/fault-tolerance-ss-protocols/Bin_Agreemen_n`)

Appendix E

Copyright Documentation

The animation in Chapter 8, Fig 8.1 was created by Dr.Kapil Sheth, Aviation System Division, NASA Ames research center, Mofett field, CA, USA. Using NASA's Future Air Traffic Management Tool (FACET), data from Federal Aviation Administration's Enhanced Traffic Management System (ETMS). Material in the public domain [1], includes material created by employees of the federal government, can be used or reproduced without the need for a permission.

[1] http://www.aviationsystemsdivision.arc.nasa.gov/research/modeling/facet.shtml