CONTRACTS FOR MESSAGE-PASSING PROGRAMS

R5204

by

Ziqing Luo

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

Fall 2019

CONTRACTS FOR MESSAGE-PASSING PROGRAMS

R5204

by

Ziqing Luo

Approved: _____

Kathleen F. McCoy, Doctor of Philosophy
Chair of the Department of Computer and Information Sciences

Approved: _____

Levi T. Thompson, Doctor of Philosophy
Dean of the College of Engineering

Approved: _____

Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

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

Signed: _____

Stephen F. Siegel, Doctor of Philosophy
Professor in charge of dissertation

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

Signed: _____

James Clause, Doctor of Philosophy
Member of dissertation committee

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

Signed: _____

Sunita Chandrasekaran, Doctor of Philosophy
Member of dissertation committee

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

Signed: _____

Matthew B. Dwyer, Doctor of Philosophy
Member of dissertation committee

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

Signed: _____

William D. Gropp, Doctor of Philosophy
Member of dissertation committee

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Message-passing parallelism is widely used in High Performance Computing (HPC), and will continue to be used in the foreseeable future. HPC applications perform complex and expensive computations that are of fundamental importance for science, engineering, and society. It is therefore critical that these programs are *correct*, and effective methods for verifying the correctness of message-passing programs are needed.

This dissertation presents a new method for specifying and verifying correctness of message-passing programs. The method is based on procedure contracts. These provide a natural and intuitive way to decompose the specification and verification tasks.

While procedure contracts have been widely-studied for sequential programs, it is not obvious how to generalize them for message-passing parallel programs. The approach presented here is based on the notion of *collective contract*, which is analogous to the notion of *collective function* in message-passing programming.

The first contribution of this dissertation is a rigorous theory of collective contracts, presented using a "toy" message-passing language. Second, we present an automated approach for verifying such contracts, given a bound on the number of processes, using model checking and symbolic execution techniques. Our third contribution is a specification language for C/MPI programs, which deals with many of the complexities of those languages. Finally, we have implemented a verifier for specified C/MPI programs using a general verification framework, CIVL. This system is evaluated with a number of simple, but realistic, C/MPI programs.

Part I

PRELIMINARIES

Chapter 1

INTRODUCTION

1.1 Motivation

Message-passing is used pervasively in high-performance computing (HPC). This is expected to continue for the foreseeable future. For example, a recent survey [16] reported that among 28 applications and 49 software technology projects under the U.S. Department of Energy's Exascale Computing Project, 100% of the applications and many (28/49) software technology projects are currently using the Message Passing Interface (MPI, [85]).[1] Furthermore, 93% of those projects plan to continue to use MPI for the exascale version of their applications in the future.[2]

These applications are run on the world's most powerful supercomputers—machines which are extremely expensive to build, operate, and maintain. Time on these machines is very valuable, and the machines consume an enormous amount of energy. Time spent debugging at scale, or re-running programs after a defect was discovered at scale, wastes these resources.

Moreover, HPC projects deal with issues of fundamental importance to science, engineering, and society more generally. They include applications to predict earthquake damage [58], model the global climate [34], perform atomic-level simulations of chemical and biological systems [110], and to investigate the electronic structure

[1] Table 2, The "Using MPI" column, a breakdown of the numbers of ECP projects reporting that they are actually using MPI.

[2] Table 4, Question 20, "Do you expect the exascale version of your program to use MPI?". Based on the 77 projects which responded to the survey, out of a total of 97 active projects. For the reported result, "application" includes application and software technology projects.

of matter [81], among many others. These programs inform both profound scientific conclusions and decisions of the utmost importance to society.

For all of these reasons, it is imperative to develop effective methods for verifying correctness of message-passing HPC programs. Indeed, there is a growing consensus in the HPC community that better methods are needed to verify the correctness of HPC applications [27, 28, 42, 45–47, 84].

There are many aspects to the correctness of a program. A programming language, such as C, imposes certain *generic* correctness requirements that apply to all programs in that language. For example, any correct C program should not use an array index that lies outside the bounds of the array, dereference a null pointer, or invoke `free` on a pointer that was not returned by an earlier call to `malloc`. The use of concurrency APIs, such as MPI, imposes further generic requirements; for example, a correct MPI program should not deadlock, or have processes in a communicator invoke collective operations in different orders.

While the work of this dissertation does apply to these generic properties, the focus is more on *program-specific* correctness properties. These properties assert that a program—or an individual procedure within a program—should behave in the intended way, and should compute what the developer expects.

By their very nature, program-specific properties require the developer to *specify* intended behavior. While there are many different approaches to specifying programs, one of the most influential is the *code contract*. This approach was introduced by Bertrand Meyer [87] and realized in the Eiffel programming language [86], and has since spread to many contexts. Larch [112] was one of the early specification languages and was applicable to multiple programming languages, the Java Model Langauge [71] was developed to specify Java programs and has been extremely influential, Spark [3] is a highly successful specification language for Ada [2], and ACSL [1] is a specification language for C that is used with the Frama-C analysis platform.

The most fundamental kind of code contract is the *procedure contract*. A procedure contract consists primarily of a pair of *pre-* and *post-conditions*. These conditions specify a relation between the input and the output of the procedure. The contract asserts that *if* the pre-condition holds when the procedure is called, *then* the post-condition will hold when the procedure returns. This fundamental semantic idea originated in Hoare logic [50], but is extended in several ways to deal with procedure calls, side effects, and other complexities that arise in real programming languages.

The contract approach enables modular and composite verification of programs. For every procedure in a program, one verifies whether it satisfies its contract under the assumption that all other procedures satisfy their contracts. By doing so, the verification problem is decomposed into a number of smaller and independent sub-problems. This sort of decomposition, or something like it, is the only way that verification has any chance of scaling to software of the considerable size and complexity seen in modern HPC.

There are many different ways to "verify" that a procedure satisfies its contracts. These include the automatic insertion of assertions for runtime checks [5,93], and automatic generation of test cases [67,119]. Such approaches cannot verify with certainty, but only provide varying degrees of evidence for the likelihood that a contract holds.

The most rigorous approach to contract verification is *deductive reasoning*, which works by generating verification conditions that are individual theorems, which, if proved, imply a contract holds. Advances in automated theorem proving [12,26,33,65] have eliminated much of the human effort previously required to discharge the proof obligations, but even the best tools still require significant manual guidance in the form of invariants, lemmas, and various "hints" for the theorem provers. This methodology has been realized in many contract verification tools, e.g., [8,10,11,13,31,32,38,43,72,74].

A common point of the majority of these tools is that they are only suitable for sequential programs. There has been some work extending the ideas for shared-memory concurrent programs (e.g., [4,18,25]), but almost nothing for message-passing

parallelism.

In fact, it is not obvious that procedure contracts are suitable for reasoning about message-passing programs. First, in a message-passing parallel program, the behavior of a procedure depends not only on its inputs but also on the behaviors of other processes that run in parallel. Second, there is not a clear notion of pre- or post-state because processes can enter and exit a given procedure at different times. Nevertheless, the need to "divide and conquer" the verification task is as evident for message-passing programs as it is for sequential programs.

1.2 Contributions

1.2.1 A contract theory for message-passing programs

The first contribution of this dissertation is to cope with the problems above by developing a *theory of contracts* for message-passing programs. The basic idea of this theory is to use a contract to specify the collective behavior of a procedure that is executed by all processes collectively. We call such procedures the *collective-style procedures*.

This theory is applicable to programs containing non-collective-style procedures, it simply does not provide a way to decompose the specification/verification task for such procedures. In other words, if a collective-style procedure f calls some non-collective-style procedure g, then f and g will be specified and verified as a unit: the user will write a contract for f (but not write one for g) and the verification of f's contract will require the definitions of both f and g.

Obviously, this approach is only useful to the extent that there are many collective-style procedures in real message-passing programs. We contend that such procedures are ubiquitous. For starters, all of the official MPI collective functions are collective-style, and the MPI collectives are widely used. As the ECP teams responded in [16], 80% of them use MPI collective functions in their current applications and this percentage increases to 82% for exascale version of their applications.

Furthermore, we have examined source code in two large, state-of-the-art MPI applications: the Monte Carlo particle transport code OpenMC [95], and a module (`parcsr_ls`) in the algebraic multigrid solver AMG [114]. OpenMC consists of over 24 thousand lines of C++ code, and the AMG module is over 35 thousand lines of C code. Through a combination of static analysis and manual inspection, we confirmed that every function in these codes either involves no MPI communication (in which case the usual sequential contract techniques can be applied) or is collective-style.

1.2.2 A method for verifying collective contracts

The second contribution of this dissertation is a method for automatically verifying that a collective-style function satisfies its contract. As explained above, deductive reasoning is typically used for this sort of verification in the case of sequential programs. The main advantage of deductive reasoning is that it can perform *complete verification* for an arbitrary configuration of a program. But it costs the user's effort in specifying every single function and loop in the program with a contract and a *loop invariant*, respectively, and possibly additional annotations specifying theories, lemmas, and so on.

Instead of deductive reasoning, our method is based on model checking [23, 82] and symbolic execution [24, 63]. The main advantage of this method is that it requires no invariants or annotations beyond the procedure contracts, and given those it is completely automated. Furthermore, in order to verify that a procedure f satisfies its contract, it is not necessary to have contracts for all procedures called by f. The called procedures without contracts will effectively be executed when verifying f. This enables a very incremental approach to the specification and verification work.

The cost is that our method performs *bounded verification*—it requires small bounds to be placed on the number of processes, and possibly on other parameters. If the verification succeeds, it is theoretically possible that a contract could be violated with a larger number of processes. The effectiveness of this approach relies on the

Small Scope Hypothesis [17, 55, 68], which posits that almost all defects can be found by exhaustively exploring all possible behaviors in all small configurations of a system.

Both the contract theory and the verification method will be formulated in this dissertation using a "toy" message-passing language, which we call MiniMP. Using a simple language with a mathematically precise semantics is the best way to elucidate the essential ideas, and also allows us to state and prove precise statements about the soundness of the verification method.

1.2.3 An MPI Specification Language

The simple MiniMP language leaves out many of the "accidental" complexities of C and MPI. These includes pointers and pointer arithmetic, specifying buffers through `void*` pointers and byte-counts, use of multiple communicators, representing types and reduction operations as values, and so on. A practical specification language for message-passing programs must deal with all these issues.

As the third contribution of this dissertation, we present such a specification language for C/MPI programs. The language extends the existing ACSL language for sequential C. It adds new "MPI-aware" primitives which realize many of the central concepts that arise informally in the MPI Standard. These include the notion of a *region* of memory specified by a `void*` pointer and size (i.e., a buffer); a predicate for asserting that two buffers contain the same typed values; predicates that compare memory across different processes, and so on. The contract system supports a significant subset of MPI, including standard mode blocking point-to-point communication, the wildcards `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, all blocking collective communication, all kinds of pre-defined MPI data types and operations, and multiple communicators.

We believe this new specification language has significant practical potential, even without a corresponding verification tool. For example, it could be used to make formally precise the semantics of the MPI collective functions, which are currently described using only natural language in the MPI Standard. It could also be used by programmers simply to document the intended behavior of their code. The language

is sufficiently intuitive and readable that it can be easily understood by programmers who are not experts in formal verification.

1.2.4 A Prototype MPI Contract-Based Verification Tool

The fourth contribution of this dissertation is an actual verification tool for C/MPI programs specified using the language described above. The prototype tool implements the theoretical verification approach, based on model checking and symbolic execution, described in §9.3.

The tool was developed using a general verification framework, CIVL [101, 117, 118]. The framework is based on an intermediate verification language, CIVL-C, which provides a small set of generic concurrency primitives on top of standard C. We have added an automatic transformer which consumes a C/MPI program with contracts and produces a CIVL-C program. The CIVL-C program contains assertions and assumptions, in such a way that if all assertions hold on all executions, then the original contract must hold.

This prototype implementation was evaluated with a number of simple C/MPI programs, including MPI collective implementations used by CIVL itself, advanced MPI collective algorithms and typical MPI computational applications.

1.3 Outline

The dissertation is composed of three parts.

Part I includes chapters for preliminary information. Chapter 2 introduces background knowledge and related work. Chapter 3 presents general notation that will be used throughout the document.

In Part II, we describe a theory of message-passing contracts and their verification using the simple MiniMP language. The syntax and semantics of the language proper are formally described in Chapter 4. In Chapter 5, we present a specification language for MiniMP and show how the correctness of a MiniMP program can be expressed using this specification language. Chapter 6 presents an inference system,

which consists of four rules, for reasoning about MiniMP programs with specifications. Chapter 7 presents a system based on that inference system for verifying MiniMP programs, using model checking and symbolic execution techniques.

Part III focuses on bringing the theoretical approach to practice. Chapter 8 gives an overview of the CIVL framework, on which a contract system for verifying C/MPI programs will be built. In Chapter 9, we introduce our *MPI contract language* and the implementation of an C/MPI contract system. Chapter 10 presents our evaluation of the contract language and system. It also includes a discussion of the experimental results.

Finally, we conclude this dissertation in Chapter 11.

Chapter 2

BACKGROUND & RELATED WORK

In this chapter, we give a brief summary on background and related research work. The background knowledge includes introduction to classical formal verification approaches and the MPI standard (§2.1, §2.2, §2.3, §2.4 & §2.5). The state-of-the-art verification tools for message-passing parallel programs, as well as their relations to this dissertation, are described in the second part (§2.6, §2.7 & §2.8).

2.1 Hoare Logic

Hoare logic [50] provides a proof system for deriving safety properties of sequential programs. It is the basis for deductive verification.

The key notion in Hoare logic is the *Hoare triple*, $\{P\}\ S\ \{Q\}$, where S is a program statement, and P and Q are logic formulas. The formula P is called a *pre-condition*, and Q is called a *post-condition*. The Hoare triple is *valid* if the following holds: whenever S is executed from a state in which P holds, if S terminates then Q will hold in the resulting state.

Hoare's original paper uses a simple "while" programming language which has assignments, while loops, if and if-else statements, and sequential composition (;). A set of inference rules are defined for reasoning about Hoare triples over this language. This rule set is called the *Hoare calculus*. Each inference rule has a set of premises—Hoare triples assumed to hold—and a conclusion—the Hoare triple which one can conclude holds given that the premises do.

Fig. 2.1 shows the inference rules in Hoare calculus. The rules are presented in the standard syntax where the premises appear above the horizontal line and the conclusion below the line.

The *skip* rule shows that a pre-condition will be preserved by a no-op.

The *assignment* rule has no premise. It states that the post-condition Q of the assignment a `:=` e can be ensured, if the pre-condition is obtained by substituting a with expression e in Q.

The *sequence* rule states that if there is a condition that can serve as the post-condition of a statement S_0 as well as the pre-condition of another statement S_1, a Hoare triple for the composite statement $S_0\,;S_1$ can be inferred.

The *consequence* rule introduces a *side condition*, which is a logical formula over the pre- and post-conditions of two Hoare triples. The rule states that for a statement S, one triple can be inferred from the other if the side condition is valid.

The *conditional* rule expresses that a Hoare triple of a conditional statement can be inferred from Hoare triples corresponding to the two branches.

The *loop* rule states that a Hoare triple of a while-loop can be inferred with a condition I, if I is an invariant of the loop body. This is the rule that makes proving programs with Hoare logic hard. The problem is that there are an infinity of loop invariants I (i.e., formulas I for which the premise of the loop rule is valid), but there is no algorithm to find just the right loop invariant to make a proof go through. Most program verification systems based on deductive reasoning require the user to provide the loop invariants. This requires intuition about the program and experience.

Given appropriate loop invariants for each loop, the proof process of applying Hoare calculus to Hoare triples can be automated with the *verification condition generation* (VCG) [10, 43, 74] technique. Given a Hoare triple, the VCG algorithm automatically computes a set of formulas in the underlying logic with the property that the validity of the formulas implies the existence of a Hoare logic derivation of the triple. The key step in the VCG algorithm is the computation of the *weakest pre-condition* (WP) of the statement with respect to the post-condition. Most of the deductive verification tools for sequential programs are based on VCG.

Example 2.1. Figure 2.2 shows a derivation for a Hoare triple of a *while* program using Hoare calculus. In this figure, auxiliary variables X and Y are used to represent

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}} \quad \text{(skip)}$$

$$\frac{}{\{Q[e/a]\}\ a\ \texttt{:=}\ e\ \{Q\}} \quad \text{(assignment)}$$

$$\frac{\{P\}\ S_0\ \{R\},\ \{R\}\ S_1\ \{Q\}}{\{P\}\ S_0 ; S_1\ \{Q\}} \quad \text{(sequence)}$$

$$\frac{\{P'\}\ S\ \{Q'\}}{\{P\}\ S\ \{Q\}} \quad \text{(consequence)} \quad \text{if } P \Rightarrow P' \wedge Q' \Rightarrow Q$$

$$\frac{\{P \wedge C\}\ S_0\ \{Q\},\ \{P \wedge \neg C\}\ S_1\ \{Q\}}{\{P\}\ \texttt{if}\ C\ \texttt{then}\ S_0\ \texttt{else}\ S_1\ \{Q\}} \quad \text{(cond)}$$

$$\frac{\{C \wedge I\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ C\ \texttt{do}\ S\ \{\neg C \wedge I\}} \quad \text{(loop)}$$

Figure 2.1: Five inference rules from Hoare calculus.

the values of `x` and `y` at the pre-state, respectively. Auxiliary variables are not part of a program, hence they will retain their values in the post-state. The VCG computation for this example is given in Fig. 2.3. Note that the term $abs(\texttt{x})$ in the code stands for the absolute value of `x`. By VCG, the validity of the Hoare triple is represented by the following condition:

$$(\texttt{x} = X \wedge \texttt{y} = Y) \Rightarrow \mathrm{WP}(\texttt{if x >= 0 then y = x else y = -x},\ \ \texttt{y} = abs(\texttt{x}))$$

i.e., $(\texttt{x} = X \wedge \texttt{y} = Y) \Rightarrow \mathit{true}$

This verification condition is obviously *true* hence the Hoare triple is valid. □

Design by Contract™ [87] (DbC) is a methodology developed by Bertrand Meyer for software correctness[1]. The idea of an important part of DbC is to use pre- and post-conditions to document, as well as check the correctness of, program procedures. A pair of a pre- and post-conditions for a procedure is called a *procedure contract*. Procedure contracts are built-in constructs in the Eiffel [86] programming language. They can be checked as assertions during runtime executions.

[1] The term was trademarked and the trademark is now owned by Eiffel Software.

	$\{\texttt{x} = X \wedge \texttt{y} = Y\}$ `if x>=0 then y = x else y = -x` $\{\texttt{y} = abs(\texttt{x})\}$	(cond 1, 2)
1	$\{\texttt{x} = X \wedge \texttt{y} = Y \wedge X \geq 0\}$ `y = x` $\{\texttt{y} = abs(\texttt{x})\}$	(conseq 1.1)
1.1	$\{\texttt{x} = abs(\texttt{x})\}$ `y = x` $\{\texttt{y} = abs(\texttt{x})\}$	(assign)
2	$\{\texttt{x} = X \wedge \texttt{y} = Y \wedge X < 0\}$ `y = -x` $\{\texttt{y} = abs(\texttt{x})\}$	(conseq 1.2)
1.2	$\{-\texttt{x} = abs(\texttt{x})\}$ `y = -x` $\{\texttt{y} = abs(\texttt{x})\}$	(assign)

Figure 2.2: The derivation of a Hoare triple. X and Y are auxiliary variables holding the values of `x` and `y`, respectively, at pre-state. The symbol abs represents the absolute value function in the underlying first order logic.

WP(`if x>=0 then y = x else y = -x`, $\texttt{x} = abs(\texttt{y})$)		(cond)
$=$	$\texttt{x} \geq 0 \Rightarrow \text{WP}(\texttt{y = x},\ \texttt{x} = abs(\texttt{y})) \ \wedge$	(assign)
	$\texttt{x} < 0 \Rightarrow \text{WP}(\texttt{y = -x},\ \texttt{x} = abs(\texttt{y}))$	(assign)
$=$	$(\texttt{x} \geq 0 \Rightarrow \texttt{x} = abs(\texttt{x})) \ \wedge \ (\texttt{x} < 0 \Rightarrow \texttt{x} = abs(\texttt{-x}))$	
$=$	$true$	

Figure 2.3: The computation of the weakest pre-condition, $\text{WP}(S, Q)$, for a given statement S and desired post-condition Q.

Procedure contracts is also absorbed by many programming languages for playing the role of specifications. Why3 [38] and Boogie [10] are verification programming languages with rich built-in specification constructs, including procedure contracts. Both of them at the same time also refer to the deductive verifiers that can prove the functional correctness of Why3 and Boogie programs, respectively, against procedure contracts and other kinds of specifications. There are other programming languages adopt the idea of procedure contracts. For example, ACSL, JML [44] and Spec# [11] are specification languages for C, Java and C#, respectively. Spark [9] is a dialect of Ada for describing specifications. Dafny [72] is a programming language with built-in primitives for specification.

Programs can be formally verified by different tools with respect to procedure contracts and other kinds of specifications written in the aforementioned languages. Such as the static verifiers powered by Boogie, including HAVOC [8, 69], the Spec#

verifier and the Dafny verifier. Spark-2014 [3] and Frama-C [30] with its plug-in WP [31] are the verification tools using Why3.

2.2 Formal Methods For Concurrency

Hoare's original logic is not suitable for specifying and proving concurrent programs. A Hoare triple describes a statement in terms of its input and output. However, in a concurrent program, the behavior of a process executing a statement depends on not only the input but also the behaviors of other processes that run in parallel. Besides, Hoare logic cannot describe *liveness properties*, which are crucial for concurrent programs.

Owicki and Gries [90], Lamport [70] and Takaoka [106] introduced extensions to the original Hoare logic for proving properties of concurrent programs.

Owicki and Gries introduced a new technique in [90] based on Hoare logic for proving *safety properties* of concurrent programs. In their work, concurrency is enabled by two general statements: (1) `cobegin` S_0 `//` ... `//` S_n `coend` means that each statement S_i, $0 \leq i \leq n$, is executed by a separate process and all these processes are run in parallel; (2) `await` B `then` S means that the process has to stop at S unless B is true and once B is true, the process executes S in an atomic step. Formal semantics for these two statements are defined with new Hoare-style rules. Then for a concurrent program S that consists of the extended set of statements, the validity of $\{P\}\ S\ \{Q\}$ can be derived using the rules.

The key idea in this work is the semantics of the `cobegin ... coend` statement,

$$\frac{\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}, \ldots \text{ are interference-free}}{\{P_1 \wedge P_2 \wedge \ldots\}\ \texttt{cobegin}\ S_1\ \texttt{//}\ S_2\ \texttt{//}\ \ldots\ \texttt{coend}\ \{Q_1 \wedge Q_2 \wedge \ldots\}}$$

which is straightforward except for the term "interference-free."

Given a `cobegin` S_1 `//` ... `//` S_m `coend` statement. Suppose there are Hoare logic derivations for $\{P_i\}\ S_i\ \{Q_i\}$, for $1 \leq i \leq m$. For each sub-statement S' of S_i, there is an assertion immediately precedes S' in the derivation, denoted $\mathsf{pre}(S')$. A sub-statement T of S_j, $1 \leq j \leq m$, is said to not *interfere* $\{P_i\}\ S_i\ \{Q_i\}$, if both of the following hold:

1. T preserves $\mathsf{pre}(S')$ for every sub-statement S' in S_i.
2. T preserves the post-condition Q_i of S_i.

To prove the correctness of the `cobegin` statement, one must show that every atomic sub-statement of S_i cannot interfere S_j, for $1 \leq i \neq j \leq m$. Usually, in order to prevent a triple $\{P\}\ S\ \{Q\}$ from being interfered, the user needs to weaken P to accept the behaviors of other parallel statements.

Lamport [70] generalizes Hoare logic by refining the meaning of the triple $\{P\}\ S\ \{Q\}$ so that (1) P must be strong enough to guarantee that Q will hold upon termination even if the execution starts from any location inside S; and (2) P must remain true as long as the control is in S, where S is a concurrent program composed of sequential statements as well as the `cobegin` statement used by Owicki and Gries. There are three predicates that refer to program locations with respect to a statement S: immediately before S, inside S and immediately after S. For a sub-statement S in a `cobegin` structure, a strong enough pre-condition usually takes program locations of different processes into consideration.

Takaoka [106] proves a triple $\{P\}\ S\ \{Q\}$ for a parallel program S by analyzing a program graph representing S, where vertices are sets of locations of running processes and arrows bridging vertices are labeled by program statements. In addition, each vertex has an associated assertion. Initial locations are attached with pre-conditions and final locations are attached with post-conditions. Then for every arrow, the assertion of the destination vertex can be derived from the assertion of the source vertex with respect to the statement labeling the arrow. The rules for the assertion derivation are defined in Hoare style. The triple $\{P\}\ S\ \{Q\}$ is valid iff such a graph is successfully constructed. In Takaoka's work, parallel programs contain synchronizing statements. A process p can "wait for" another process q by executing `call` until q executes `accept`. A similar program-graph-based analysis was done by Newton [88] earlier. In [88], program locations can be used to express properties in assertions though the parallel programs have no synchronizing statement.

A more intuitive approach [7] was investigated earlier by Ashcroft. Compared to the Hoare-style based approaches described above, many researchers, including Lamport[2], believe that Ashcroft's approach is a better way to prove correctness of concurrent programs. In [7], safety properties of a concurrent program are expressed as a *global invariant*, i.e., an assertion that is supposed to hold at any location of the program. The proof is constructed by showing that every atomic step in the program preserves the invariant.

A liveness property claims that a desired event must eventually occur during an infinite execution of a program. Liveness properties involve the concept "eventually". *Temporal logic*, which extends the ordinary logic with temporal operators such as "henceforth" and "eventually", is widely used to express liveness properties. There are different techniques for proving liveness properties. The *well-founded set* approach was originally applied by Floyd [39] and later adapted by Manna and Puneli in [80]. Other examples include the *proof lattice* approach [91] and the *rely-guarantee* approach [104].

Modern tools for verifying concurrent programs in an axiomatic way include VCC [25], COMPLEX [4], KeY [13], Chalice [73] and VeriFast [56]. VCC is a static verifier for concurrent C programs. The user of VCC needs to annotate invariants and other specifications for programs. COMPLEX is a verification framework that targets general imperative programming languages. It is based on the aforementioned proof system proposed by Owicki and Gries. KeY verifies Java programs against their specifications written in JML [44]. It generalizes Hoare logic for Java to the *first order Java Dynamic Logic*. Chalice uses a permission based approach to reason about multi-threaded concurrent programs. VeriFast utilizes separation logic [94] for verifying multi-threaded C and Java programs.

---

[2] Leslie Lamport comments on his publication number 23: "Ashcroft got it right. Owicki and Gries and I just messed things up. It took me quite a while to figure this out." He comments on his publication number 40: "I think Ashcroft was right; one should simply reason about a single global invariant, and not do this kind of decomposition based on program structure." from `http://lamport.azurewebsites.net/pubs/pubs.html#proving`

2.3 Model Checking

Unlike the deductive verification techniques described in the previous two sections that can verify infinite state systems, model checking [23] is a technique for verifying finite state systems. Although the technique is restricted to finite state systems, model checking has the benefit of being performed automatically. In addition, model checking is widely used for hardware verification because many hardware systems have a finite number of states. Furthermore, one can convert an infinite state system to a finite state system through a combination of abstraction and restriction. For example, to apply model checking to a message-passing concurrent system, one can (1) abstract the actual complex communication protocols to simple message channels [98–100], and (2) bound the number of processes or the size of the message channels. In many cases, errors in systems can be found by model checkers with small bounds on program parameters.

Transforming a system to a *finite state model* is the first step of using model checking. The second step is to give a specification, which is commonly in the form of temporal logic assertions.

A model checker verifies the satisfaction of a finite state model to a specification by searching all reachable states in the finite state space of the model. A state space is usually structured as a directed graph, where a node is a state and an edge is a transition which alters a state in an atomic step.

Model checking is effective for verifying properties of concurrent systems. A concurrent systems can be abstracted to an *interleaving model*. In an interleaving model, the real-world asynchronous executions are represented by nondeterministic choices on atomic transitions. For example, the parallel execution of two atomic commands a and b can be modeled as a nondeterministic choice over two transition sequences: a followed by b or b followed by a.

A challenge in model checking is the state explosion problem. Typically, the number of states grows exponentially when the number of components in a concurrent system increases. Hence, many model checkers can only scale to programs whose inputs

are bounded with small concrete values. But model checking is still widely used to solve real problems for two reasons. First, there is a belief in the *Small Scope Hypothesis* [55], which states that most bugs can be found by exploring all possible program executions with inputs, the number of processes/threads, and other program parameters being bounded in small scopes. Second, many sophisticated *reduction* techniques allow model checkers to only search in a sub-space of the full state space without loss of soundness. For example, using the *partial-order reduction* (POR) technique [41, 60, 111], a model checker usually only needs to explore one among a number of commutative transitions.

2.4 Symbolic Execution

Symbolic execution is an approach for testing and analyzing programs [63]. It replaces the concrete program inputs with symbols, each of which represents a set of concrete values. During the symbolic execution, a boolean formula, the *path condition* (PC), is maintained. PC denotes the condition under which an execution path is feasible. Symbols are also known as *symbolic constants*. Expressions over constants and symbolic constants are called *symbolic expressions*. In symbolic execution, PC and variable values are all symbolic expressions.

Symbolic execution is able to explore all possible paths of a program that a concrete execution may take unless there are an infinite number of them. Different paths are generated from the different choices that a concrete execution can make at branches. During the exploration of each path, PC initially is assigned the *true* value and gets updated at each branch. The updated value is a conjunction of the old PC value and the branch condition corresponding to the choice made in the path. Once PC becomes unsatisfiable, the path being explored is infeasible and will be ignored immediately. Automated theorem provers [12,26,33] are used to reason about symbolic expressions during the exploration.

Example 2.2. Considering the following C code:

```
if (x >= 0)
  y = x;
else
  y = -x;
```

Suppose the variable `x` holds a symbolic expression X. A symbolic execution engine will explore two paths for this code snippet: one path leads to a result where the variable `y` has value X and its PC becomes $X \geq 0$; the other path leads to a result where the variable `y` has value $-X$ and its PC becomes $X < 0$. □

2.5 MPI

The Message-Passing Interface (MPI) standard specifies a library for writing message-passing parallel programs in C/C++ and Fortran. This library includes a large number of functions, constants, and data types. For an MPI program, every process has its private storage but shares nothing with others. Communication among processes is carried out by transferring data using different MPI functions. By far the most common way that it is done, an MPI program is written, compiled, and linked to generate an executable file. The program is executed by instantiating a number of processes, each of which runs a copy of the executable.

In MPI programs, a *communicator* is an abstraction of a "communication universe" for processes. A communicator comprises an ordered group of n ($1 \leq n$) processes. Processes are identified by $0, 1, ..., n-1$. The numerical identifier of a process in a communicator is called the *rank* of the process. A process may belong to multiple communicators and is assigned different identifiers in each of them. But there is a default communicator, named `MPI_COMM_WORLD`, which contains all processes that are invoked when a program is launched. The rank of a process in the default communicator can be used as the unique ID. By branching on process ranks, different processes may behave differently even though they run the same program.

Communication operations are performed through communicators. Conceptually, a communicator includes a set of message channels. Every message channel can be

identified by an ordered pair (i, j) $(0 \leq i, j < n)$. In general, a channel behaves like a First-In-First-Out (FIFO) queue for bufferring messages that were sent from a process i but have not been received by a process j. Specially, when receiving an message with a specific *message tag*, instead of the actual first message, the first message matched the tag in the channel will be pull out. If a wildcard message tag (`MPI_ANY_TAG`) is used for the receive operation, the message channel behaves in exact FIFO order.

In MPI, communication can be nondeterministic. A process can perform a receive operation without specifying an exact sender but an `MPI_ANY_SOURCE`. Such a receive operation is called a wildcard receive. In concept, a wildcard receive performed by process i nondeterministically picks up a message from the set $\{(j, i)|0 \leq j < n\}$ of message channels that contain appropriate messages.

Example 2.3. Figure 2.4 is a simple C/MPI program. Constructs defined in the MPI library all start with a "`MPI_`" prefix. `MPI_COMM_WORLD` is the constant that refers to the default communicator. The `MPI_Init` function initiates the message-passing environment. Every process can obtain its rank in a communicator by calling `MPI_Comm_rank`. The `MPI_Finalize` function terminates the environment.

In this example, process of rank 0 sends a message to process of rank 1. The point-to-point (P2P) communication is carried out by a pair of standard P2P send and receive functions:

1. the `MPI_Send` function (at line 7) sends the data, which is stored in `dat` and of the size of one `MPI_INT`, to the process of rank 1 in the default communicator with a tag 0.
2. the `MPI_Recv` function (at line 9) receives the message tagged by 0, in which the data is expected to be no larger than the size of one `MPI_INT` and will be saved in `dat`, from the process of rank 0 in the default communicator.

No communication action is performed by any other process. □

2.6 Model Checkers for MPI

Many automated formal verification tools for MPI are based on model checking. ISP [109] and DAMPI [113] are dynamic MPI model checkers. They actually

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
  int rank, dat = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //gets the rank of this process
  if (rank == 0)
    MPI_Send(&dat, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  else if (rank == 1)
    MPI_Recv(&dat, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Finalize();
}
```

Figure 2.4: A simple C/MPI program. Process 0 sends and process 1 receives a message. The remaining processes perform no operation.

execute the MPI program being verified but exhaustively rearrange different schedules so that eventually all possible schedules will be covered. These schedules differ from the matches that the MPI wildcard receive operations can make. ISP and DAMPI support a wide range of MPI primitives, including blocking and non-blocking point-to-point communications as well as collective communications. They are able to check deadlock freedom and other safety properties for MPI programs.

The advantage of such dynamic model checking approaches is that each schedule can be explored almost as fast as running the parallel program in a sequential way. Nevertheless, these model checkers can only verify a program for a concrete setting at a time, i.e., both the number of processes and the program inputs have to be concrete.

The model checking technique can be combined with symbolic execution. With symbolic execution, a model checker can verify an MPI program for arbitrary program inputs. MPI-SV [116] uses a symbolic execution engine to explore different paths of an MPI program. Once a violation-free path is explored, a *Communication Sequential Process* (CSP) [51] model will be generated from the path. The model is then fed to a CSP model checker, which explores all alternative interleaves and nondeterministic choices that could be made along the path. MPI-SV verifies MPI programs for deadlock freedom and user-customized properties expressed in temporal logic.

MOPPER [40] and HERMES [62] are similar to MPI-SV. They generate SAT

formulas from an execution of an MPI program. The SAT formulas encode the deadlock freedom problem. If all the SAT formulas are proved valid, the corresponding path is free of deadlock regardless of the possible interleaves or MPI communication matches that can happen along itself. MOPPER cannot deal with multi-path programs. A multi-path program contains branches that depend on program inputs. It means that multiple runs of a same multi-path program with different inputs may result in different paths. MOPPER generates SAT formulas from only one execution so it does not guarantee to cover all possible paths. HERMES generates SAT formulas for all the paths that are explored by a symbolic execution engine hence it is sound for multi-path programs.

Other tools such as MPI-Spin [99], TASS [102], and CIVL [79] combine model checking and symbolic execution by having symbolic expressions as values in states. In addition, a path condition is associated to each state, which is a boolean symbolic expression. The model checker ignores a state if its path condition is unsatisfiable. Automated theorem provers are used to reason about the symbolic expressions. MPI-Spin provides libraries in the *Promela* language for MPI primitives and symbolic operations. To use MPI-Spin, the user needs to write a model for an MPI program in Promela with the provided libraries. Properties of the program can be customized in temporal logic. The model is eventually checked by the Spin model checker [52]. CIVL uses an intermediate verification language, CIVL-C, to model various concurrency APIs, including MPI. MPI programs are automatically translated to CIVL-C programs and then sent to a CIVL-C verifier. TASS is the predecessor of CIVL.

The Spin model checker has been highly optimized to the Promela language. So for an MPI program, MPI-Spin is able to explore much more states than CIVL within the same amount of time. There is less chance for the CIVL verifier to be optimized for a specific concurrency API due to its generality to multiple such APIs. Although CIVL can hardly run as fast as MPI-Spin, it can deal with programs using not only different concurrency APIs but also the combinations of them.

A common challenge that model checkers confront is state explosion. Typically,

the number of states in the state space of a program grows exponentially with the number of processes. In addition, model checkers require the state space to be finite. So the number of processes as well as some of the program inputs have to be bounded. Therefore, model checking based tools for monolithic verification cannot be scaled to verify large codebases with realistic configurations.

2.7 Static Analysis and Runtime Verification for MPI

2.7.1 MPI Static Analyzers

Compared to model checkers, static analysis tools reason about programs for arbitrary inputs and usually have better scalability. A common limitation to static analysis approaches is that the analyzed result is not always exact because of lack of dynamic information. Moreover, static analysis tool developers often face the tradeoff between precision and efficiency.

MPI-Checker [35] is a static analyzer for MPI programs that was built on the Clang Static Analysis [22] framework. It can check various MPI-specific errors with a close to zero false positive rate, such as deadlock caused by communication mismatch, invalid type of arguments to MPI calls and incorrect message buffer referencing. The analysis is performed on an *Abstract Syntax Tree* (AST) of an MPI program. MPI-Checker was applied to MPI applications, some of which have tens of thousands of lines of code.

In addition to the AST, the *Control-Flow Graph* (CFG) is a typical abstraction of a program where static analyzers perform *data-flow analysis*. In [21], the authors propose an analysis approach on CFG for Erlang [6] message-passing programs. The message-passing model in Erlang is very similar to the conceptual model of MPI used in this dissertation. The analyzer corresponds CFGs of functions to a *communication graph*, where a vertex is a process and a directed edge corresponds to a message channel for the two connected processes. This analyzer is able to detect message-passing related errors such as deadlock caused by a receive without a matching send, or potential deadlock caused by a send without a matching receive. The analysis is approximated.

CFG has been extended to represent concurrent programs. The PARCOACH [53] static analyzer relies on Parallel Program Control Flow Graphs to detect *collective errors* of parallel programs using concurrency APIs such as MPI. A collective error in MPI programs is a mismatch of MPI collective functions. PARCOACH is efficient in analyzing large HPC applications but may produce false positives and false negatives. To avoid false positive or false negative, studies [20, 105] have been made toward improving the precision of static analysis for message-passing via extending CFGs.

Authors of [115] combines the static analysis and symbolic execution approaches for detecting *anomalies* [19]. An anomaly is a suspicious code that does not meet the expected regularities. Most anomalies are true program defects. Other cases include coding-style violations or inelegant code. For MPI programs, anomalies can reveal true MPI defects including deadlock caused by communication call mismatch, message buffer type mismatch and message buffer data race. This approach first performs static analysis to a whole MPI program and collects MPI-related control flow information. Then, symbolic execution is only performed to program segments specified by users. The biggest advantage of this approach is that the *partially* applied symbolic execution largely increases the precision of the analysis while mitigates the *path explosion problem*. Another advantage is that the symbolic execution task on each code segment is independent hence there is a potential to parallelize these tasks. One disadvantage of this approach is that an anomaly is not equivalent to a defect so that false alarms will be reported.

2.7.2 MPI Runtime Verification

There are runtime verification tools, such as Marmot [66] and its successor MUST [49]. MUST intercepts MPI calls during the execution of an MPI program and collects the state of MPI communication operations. After the execution, the tool studies the *wait-for* dependencies among processes and determines if MPI-related error, such as deadlock or message type mismatch, can happen in alternative executions. Compared to the actually observed execution, alternative executions may have different

interleaves, different matches for MPI wildcard receives or different results returned from `MPI_Test`. Runtime verification tools, such as MUST, is scalable. Furthermore, it is benefitted from dynamic information hence does not report false alarm. However, runtime verification may miss a defect if it is not revealed in the executions to which it is applied.

2.8 Deductive Verification for MPI

The related work described in previous sections all have a certain degree of automation: no human effort is required during the verification and little knowledge about verification is needed. On the other hand, these approaches face limitations. First, they have to balance between precision and scalability. In addition, these approaches verify programs only against a set of pre-defined properties (e.g., deadlock or runtime error freedom). Some tools accept customized assertions or temporal logic formulas. But in general, verification against a user-defined specification is not well supported.

In contrast, deductive verification approaches can formally verify the satisfaction of a program to a specification for arbitrary program inputs. The user is responsible for providing a specification and usually is required to interact with the verification tools to guide the verification.

ParTypes [76] verifies an MPI program with a *protocol*. A protocol consists of vectors of *term types*. A subset of MPI operations, such as standard send and receive functions and collective functions, can be represented by term types. *Well-formed* term types are deadlock-free. Whether a protocol is well-formed is *decidable* and will be proved by SMT solvers. To verify the satisfaction of an MPI program to a protocol, ParTypes translates the protocol to VCC [25] contracts and then feeds both the program and the contracts to the VCC verifier. In addition to a protocol, the user needs to annotate branches in the program to inform the verifier about the control flow.

ParTypes's protocol language cannot express the functional correctness of a program. Furthermore, ParTypes cannot deal with `MPI_ANY_SOURCE`. The subset of MPI supported in ParTypes is deterministic, i.e., given the same inputs, the way how MPI calls match is always the same.

In [89], Oortwijn et al. present their work for specifying and verifying MPI programs modularly. They use process algebra [14] terms, which are named *futures*, to model MPI operations. The user can use futures to "predict" (or specify) the communication behavior of each procedure in an MPI program for arbitrary inputs. Whether a procedure conforms to its futures is proved by Hoare style reasoning. The global communication correctness is carried out by a model checker in the mCRL2 [29] tool set, which reasons about all the futures for arbitrary number of processes. The current state of this work is preliminary: only a limited set of MPI primitives are supported, including standard send and receive operations and few collective operations. Receiving from "any" source is supported.

A different preliminary work for message-passing deductive verification is done by Luo et al. [78]. The authors verify a simple message-passing program with contracts and a global invariant. The approach used in this work was originally proposed by Ashcroft in [7]. The verification is complete, i.e., both termination and functional correctness of the program is proved for arbitrary inputs and number of processes. Such completeness costs a decent amount of effort from the user side. The user has to provide a global invariant, function contracts and loop invariants to the program. Figuring out a proper global invariant is hard. The deductive reasoning, which shows the global invariant is preserved by every program statement, is done by Frama-C/WP. Although the tool is automated, the author spent much time in manually guiding the proof with intermediate assertions.

Chapter 3

NOTATION

Let X and Y be sets. $X \to Y$ denotes the set of functions from X to Y. $X \times Y$ denotes the *Cartesian product* of X and Y, i.e., the set of ordered pairs (x, y), with $x \in X$ and $y \in Y$.

Let $\mathbb{Z}$ be the set of integers and $\mathbb{N} = \{0, 1, \ldots\}$ be the set of natural numbers. For $n \in \mathbb{N}$, we use $[n]$ to denote the set of natural numbers $\{0, 1, \ldots, n-1\}$.

Let S be a set. The power set of S, i.e., the set of all subsets of S, is denoted $\wp(S)$. The set of all finite sequences of elements of S is denoted S^*. Formally,

$$S^* = \bigcup_{n \geq 0} ([n] \to S).$$

However, we will often use the standard notation $s_0 s_1 \cdots s_{n-1}$ to denote the sequence $\xi \colon [n] \to S$ such that $\xi(i) = s_i$ for $0 \leq i < n$.

Let $\xi = s_0 s_1 \cdots s_{n-1}$ be an element of S^*. The length of ξ, denoted as $|\xi|$, is n. The empty sequence, i.e., the unique element in S^* of length 0, is denoted ϵ. If $\zeta = s'_0 \cdots s'_{m-1}$ is also an element of S^*, then the *concatenation* of ξ and ζ, denoted as $\xi \circ \zeta$, is the sequence $s_0 \cdots s_{n-1} s'_0 \cdots s'_{m-1}$. Note $|\xi \circ \zeta| = n+m$. If $|\xi| > 0'$, $\mathsf{head}(\xi) = s_0$ and $\mathsf{tail}(\xi) = s_{n-1}$. (Note $\mathsf{head}(\epsilon)$ and $\mathsf{tail}(\epsilon)$ are undefined.)

There are various sturctures, e.g., sequences, tuples and expressions. Let t be some structure. A *substitution* notation, $t[a'/a]$, denotes the structure of the same type as t that is obtained by substituting all appearances of a with a' in t.

X and Y are two sets. With $f \in X \to Y$, $a \in X$ and $b \in Y$, a *patching* notation, $f[a \mapsto b]$, denotes a function that is defined as :

$$f[a \mapsto b](c) = \begin{cases} b & \text{if } a = c \\ f(c) & \text{otherwise} \end{cases}$$

Let A, B and C be formulas in a logic. The *inference rule* $\frac{A, B, \ldots}{C}$ means that if the *premises* A, B, ... are derived, the *conclusion* C can be derived.

Propositional and first order logic operators are written as follows: $\wedge$ denotes *and*, $\vee$ denotes *or*, $\neg$ denotes *negation*, $\Rightarrow$ denotes *implies*, $\forall$ denotes *universal quantifier* and $\exists$ denotes *existential quantifier*.

Let l be a variable, s, e be mathematical expressions. We use "let $l = s$ in e" to denote the expression resulting from replacing every appearance s with l in e.

Part II

THEORY

Chapter 4

THE MINIMP PROGRAMMING LANGUAGE

In this chapter, we introduce a "toy" message-passing programming language—MiniMP. Because of its simplicity, MiniMP will be used in the rest of this part to formulate our approaches of contracts for message-passing. The syntax of MiniMP is given in §4.1. We describe an abstract representation, called the "MiniMP model", of this language in §4.2. The semantics of MiniMP is then defined using the MiniMP model in §4.3.

4.1 Syntax

```
List-of-T      := T (, T)* |
Constant       := true | false | IntegerLiteral | ...
UnOp           := - | ! | ...
BinOp          := + | - | * | / | == | != | && | || | ==> | ...
Expr           := PID | NPROCS | Constant | ( Expr ) | LExpr | len ( LExpr )
                | UnOp Expr | Expr BinOp Expr | new [ Expr ] | { List-of-Expr }
LExpr          := Identifier | LExpr [ Expr ]
Statement      := LExpr = Expr ;
                | if ( Expr ) Statement (else Statement)?
                | (LExpr =)? Identifier ( List-of-Expr ) ;
                | while ( Expr ) Statement
                | send Expr to Expr ;
                | recv LExpr from Expr ;
                | recv LExpr from any LExpr ;
                | return Expr ;
                | { Statement* }
Procedure      := fun Identifier ( List-of-Identifier )
                  { (var List-of-Identifier)? ; Statement* }
Program        := (var List-of-Identifier)? ; Procedure+
```

Figure 4.1: The grammar of MiniMP. An *Identifier* is a character sequences. An *IntegerLiteral* is an integer literal. An { *List-of-Constant* } is an array literal.

Figure 4.1 shows the grammar of MiniMP. We use the word *procedure* for the MiniMP construct, and the word *function* for a mathematical function. A MiniMP procedure can return a value, and may modify global variables. A MiniMP program consists of procedures and sets of variables. There must be an "entry" `main` procedure. Variables declared outside of any procedure are called *global variables*. Variables declared inside the procedures are called *local variables*.

A MiniMP program is executed by specifying a number of processes, each of which executes a separate copy of the program starting from the `main` procedure. (This is similar to the usual usage of MPI.) There is no variable that is shared by multiple processes. Each process is assigned a unique ID. The IDs are consecutive integers $0, \ldots, n-1$, where n is the number of processes. A process can access its own ID and the total number of processes through pre-defined constants `PID` and `NPROCS`, respectively. Processes interact through the *communication* statements: the **`send`** *expr* **`to`** *dest* and **`recv`** *expr* **`from`** *src* is a pair of basic statements for sending a value to or receiving a value from a specific process; in addition, MiniMP provides a wildcard receive statement **`recv`** *expr* **`from any`** *src*, which receives a value from "**`any`**" possible sender in a nondeterministic way. The actual sender's ID will be assigned to *src*.

In MiniMP, an array of length m can be created with the **`new`** `[`m`]` expression. Array elements can be accessed through the subscript expression a`[`i`]`, with an array a and an index i. The expression **`len`**`(`a`)` represents the length of an array a. The binary operation `==>` stands for logical implication.

Other kinds of expressions and statements in Fig. 4.1 are commom in most imperative languages.

Example 4.1. Figure 4.2 shows a MiniMP `bcast` procedure. In this procedure, process `root` sends an array `dat` to all other processes. Every other process receives the array from process `root`. The received array is returned. □

Example 4.2. Figure 4.3 shows a MiniMP `gather` proceudre. In this procedure, process `root` receives integers from other processes in a nondeterministic way. The

```
1  fun bcast(dat, root) {
2    var i;
3    if (PID == root) {
4      i = 0;
5      while (i < NPROCS) {
6        if (i != root)
7          send dat to i;
8        i = i + 1;
9      }
10   } else
11     recv dat from root;
12   return dat;
13 }
```

Figure 4.2: A MiniMP `bcast` procedure

received integer will be stored in an element of the array `dat` with the index being equal to sender's ID. Every other process simply sends an integer to process `root`. The array `dat` will be returned eventually. □

Example 4.3. Figure 4.4 shows a MiniMP program. The `main` procedure (right) uses a `scatter` procedure (left). In the `scatter` procedure, process `root` scatters elements of an array to each process separately. Every other process simply receives a message from process `root`. Eventually, every process returns the scattered element. □

4.2 The MiniMP Model

In this section, we define an *abstract representation* for a generic MiniMP program. We call such an abstract representation a MiniMP *model*. A MiniMP model accurately represents a MiniMP program while minimizes the information that is needed for execution.

Definition 4.4. A MiniMP *vocabularly* is a tuple $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, where

- Procedure is a set of *procedure names*,
- Var is a set of *variables*,

```
1  fun gather(val, root) {
2    var i, y, dat;
3    if (PID == root) {
4      i = 0;
5      dat = new [NPROCS];
6      dat[root] = val;
7      while (i < NPROCS) {
8        if (i != root) {
9          recv val from any y;
10         dat[y] = val;
11       }
12       i = i + 1;
13     }
14   } else
15     send val to root;
16   return dat;
17 }
```

Figure 4.3: A MiniMP `gather` procedure

```
1  fun scatter(buf, root) {
2    var i, result;
3    if (PID == root) {
4      i = 0;
5      while (i < NPROCS) {
6        if (i == PID)
7          result = buf[i];
8        else
9          send buf[i] to i;
10       i = i + 1;
11     }
12   } else
13     recv result from root;
14   return result;
15 }
```

```
15 fun main() {
16   var buf, result, i;
17   if (PID == 0) {
18     buf = new [NPROCS];
19     i = 0;
20     while (i < NPROCS) {
21       buf[i] = i;
22       i = i + 1;
23     }
24   }
25   result = scatter(a, 0);
26   return result;
27 }
```

Figure 4.4: A MiniMP program where the `main` procedure (right) uses a `scatter` procedure (left).

- $\mathsf{Global} \subseteq \mathsf{Var}$ is the set of *global variables*, and
- $\mathsf{locvar} : \mathsf{Procedure} \to \wp(\mathsf{Var})$ is a function which, given a procedure name, returns the set of *local variables* for that procedure

such that

- $\forall f \in \mathsf{Procedure}, \mathsf{locvar}(f) \cap \mathsf{Global} = \emptyset$, and
- $\forall f, g \in \mathsf{Procedure}, f \neq g \Rightarrow \mathsf{locvar}(f) \cap \mathsf{locvar}(g) = \emptyset$.

□

Definition 4.5. Given a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, the set of program expressions over Var is denoted $\mathsf{Expr}_{\mathcal{V}}$. The set of program expressions of boolean type over Var is denoted $\mathsf{BExpr}_{\mathcal{V}}$. □

We next define the atomic statements that will be used to label transitions in a program graph representation.

Definition 4.6. Given a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, the set of *atomic statements*, denoted $\mathsf{Stmt}_{\mathcal{V}}$, is

1. `skip`
2. `assign return to` x
3. x `=` e
4. **`return`** e
5. **`send`** e_0 **`to`** e_1
6. **`recv`** e_0 **`from`** e_1
7. **`recv`** e_0 **`from any`** e_1
8. id `(`e`, ...)`

with $x, e, e_0, e_1 \in \mathsf{Expr}_{\mathcal{V}}$ and $id \in \mathsf{Procedure}$. The `skip` statement is simply a no-op. The `assign return to` x statement assigns the returned value from the last call to x. Atomic statements 3–8 are MiniMP program statements. □

Procedures in a MiniMP program is modeled as *procedure graphs*.

Definition 4.7. Given a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, and a procedure $f \in \mathsf{Procedure}$. A procedure graph for f is a tuple

$$g_f = (\mathsf{Loc}, l_0, T),$$

where

- Loc is a set of *program locations*,
- l_0 is the *start location*, and
- $T \subseteq \mathsf{Loc} \times \mathsf{BExpr}_{\mathcal{V}} \times \mathsf{Loc} \times \mathsf{Stmt}_{\mathcal{V}}$ is a set of *local transitions*.

Let $(l, \phi, l', a) \in T$. The location l is called the *origin* of the transition, l' is the *target*. The boolean expression ϕ is the *guard* of the transition, and a is the atomic statement of the transition. The variables that occur in ϕ and a must belong to $\mathsf{Global} \cup \mathsf{locvar}(f)$. □

A local transition (l, ϕ, l', a) denotes that, during an execution, if the control is at location l, it can execute the atomic statement in one step and arrives at location l', if the guard ϕ is true. Local transitions that model conditional statements or `recv` statements have non-trivial guards. For example, a `recv` statement shall block the execution if there is no message that has been sent to but not received by the receiver. In addition to the expressions defined in Fig. 4.1, we define two expressions only for guards of local transitions:

1. `empty(`*src*`)` means that there is no message that has been sent from process *src* but not received by the process associated to the local transition.
2. `allempty` means that there is no message that has been sent from any process but not received by the process associated to the local transition.

We next describe the process that translates a procedure in a MiniMP program to a procedure graph.

Definition 4.8. Given a vocabulary (Procedure, Var, Global, locvar). A *translation* process transStmt(l, S) takes a start location l and a *Statement* S, which belongs to a procedure in Procedure. The process returns an ordered pair (l', T), where l' is the exit location and T is a set of local transitions. The process transStmt(l, S) guarantees that (1) T represents a sub-graph corresponding to S; (2) l and l' are the unique entry and exit locations, respectively, of T. We define transStmt(l, S) using a pseudocode in Fig. 4.5. □

Note a `recv` v `from` e statement is guarded by `!empty`(e), and a `recv` v_0 `from any` v_1 statement is guarded by `!allempty`.

Given a MiniMP procedure f, a procedure graph of f can be constructed as (Loc, l_0, transStmt$(l_0, \{S_0 S_1 \dots\})$), where $S_0 S_1 \dots$ is the body of f, Loc is the set of the locations used in transStmt(l_0, S).

Example 4.9. We present the procedure graph $g_{\texttt{bcast}}$ for the program in Fig. 4.2 as an example here:

$$g_{\texttt{bcast}} = (\{l_0, \cdots l_{12}\}, l_0, T), \text{ where}$$

$T = \{$
$(l_0,$ `PID == root`, `skip`, $l_1)$,
$(l_2,$ `i < NPROCS`, `skip`, $l_3)$,
$(l_4,$ `true`, `send dat to i`, $l_5)$,
$(l_3,$ `i == root`, `skip`, $l_6)$,
$(l_7,$ `true`, `i = i + 1`, $l_8)$,
$(l_2,$ `i >= NPROCS`, `skip`, $l_9)$,
$(l_{10},$ `!empty(root)`, `recv dat from root`, $l_{11})$,
$(l_{11},$ `true`, `skip`, $l_{12})$,
$(l_1,$ `true`, `i = 0`, $l_2)$,
$(l_3,$ `i != root`, `skip`, $l_4)$,
$(l_5,$ `true`, `skip`, $l_7)$,
$(l_6,$ `true`, `skip`, $l_7)$,
$(l_8,$ `true`, `skip`, $l_2)$,
$(l_0,$ `PID != root`, `skip`, $l_{10})$,
$(l_9,$ `true`, `skip`, $l_{12})$,
$\}$

Figure 4.6 visualizes this procedure graph. □

Definition 4.10. A MiniMP model $\mathcal{M}$ over a vocabulary (Procedure, Var, Global, locvar) is a tuple:

$$\mathcal{M} = (\texttt{main}, \texttt{pg}),$$

Input: integer l, *Statement* S
Output: ordered pair (integer l, transition-set R)
begin transStmt(l, S)
  transition-set $R2, R \leftarrow \emptyset$;
  if S is an assignment, a return, or a `send` statement **then**
    $R \leftarrow \{(l, \texttt{true}, S, l+1)\}$;
    $l \leftarrow l + 1$
  else if S is `recv` e_0 `from` e_1 **then**
    $R \leftarrow \{(l, \texttt{!empty(}e_1\texttt{)}, S, l+1)\}$;
    $l \leftarrow l + 1$
  else if S is `recv` e_0 `from any` e_1 **then**
    $R \leftarrow \{(l, \texttt{!allempty}, S, l+1)\}$;
    $l \leftarrow l + 1$
  else if S is $x = f(\dots)$ **then**
    $R \leftarrow$
     $\{(l, \texttt{true}, f(\dots), l+1), (l+1, \texttt{true}, \texttt{assign return to x}, l+2)\}$;
    $l \leftarrow l + 2$
  else if S is `{`$S_0 \dots S_{n-1}$`}` **then**
    foreach $i : 0$ `..` $n-1$ **do**
      $(l, R2) \leftarrow$ transStmt(l, S_i);
      $R \leftarrow R2 \cup R$
    end
  else if S is `if (`e`)` S_0 `else` S_1 **then**
    integer exit0, exit1, entry $\leftarrow l$;
    $R \leftarrow \{(\text{entry}, e, \texttt{skip}, l+1)\}$;
    $(l, R2) \leftarrow$ transStmt$(l+1, S_0)$;
    $R \leftarrow R2 \cup R$;
    exit0 $\leftarrow l$;
    $R \leftarrow R \cup (\text{entry}, \texttt{!}e, \texttt{skip}, l+1)$;
    $(l, R2) \leftarrow$ transStmt$(l+1, S_1)$;
    $R \leftarrow R2 \cup R$;
    exit1 $\leftarrow l$;
    $R \leftarrow R \cup \{(\text{exit0}, \texttt{true}, \texttt{skip}, l+1), (\text{exit1}, \texttt{true}, \texttt{skip}, l+1)\}$;
    $l \leftarrow l + 1$
  else if S is `while (`e`)` S_0 **then**
    integer entry $\leftarrow l$;
    $R \leftarrow \{(\text{entry}, e, \texttt{skip}, l+1)\}$;
    $(l, R2) \leftarrow$ transStmt$(l+1, S_0)$;
    $R \leftarrow R2 \cup R$;
    $R \leftarrow R \cup \{(l, \texttt{true}, \texttt{skip}, \text{entry}), (\text{entry}, \texttt{!}e, \texttt{skip}, l+1)\}$;
    $l \leftarrow l + 1$
  end
  return (l, R)
end

Figure 4.5: The pseudocode definition of the translation process.

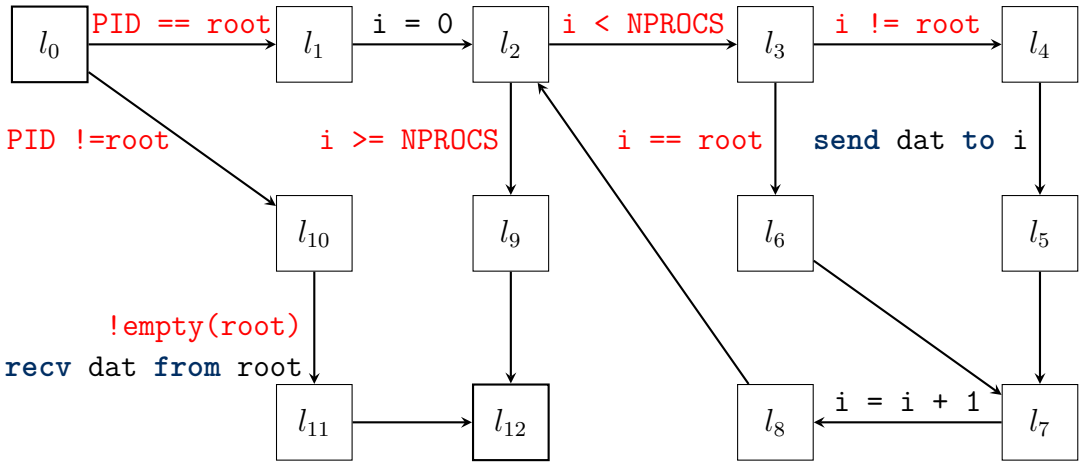

Figure 4.6: Procedure graph of the `bcast` procedure in Fig. 4.2. Boxes are program locations. Arrows are labeled by an atomic statement and a guard (in red). Trivial statement `skip` and trivial guard `true` are omitted. Two boxes, including the arrow that connects them, represent a local transition. The bold boxes l_0 and l_{12} are the sole entry and exit, respectively.

where

- `main` $\in$ Procedure: the `main` procedure
- pg is a function maps from Procedure to procedure graphs, e.g., pg(`main`) is the procedure graph of the `main` procedure

□

4.3 Semantics

4.3.1 Process States and Global States

In a state of a MiniMP program, variables hold values corresponding to a domain.

Definition 4.11. The *domain* Val that contains a special value $\mathbb{UNDEF}$ that stands for undefined values, integers, boolean values and sequences are defined recursively by

$$\mathsf{Val}_0 = \{\mathbb{UNDEF}\} \cup \mathbb{Z} \cup \mathbb{B} \cup \ldots,$$

$$\mathsf{Val}_{i+1} = \mathsf{Val}_i^* \text{ for } i \geq 0, \text{ and}$$

$$\mathsf{Val} = \mathsf{Val}_0 \cup \mathsf{Val}_1 \cup \mathsf{Val}_2 \cup \ldots .$$

□

A *frame* is an entry in a process's call stack. It stores information about the current procedure for that process: the location within the procedure's program graph, and the values of the procedure's local variables.

Definition 4.12. Given a vocabulary $(\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a domain Val, and a MiniMP model $\mathcal{M} = (\mathtt{main}, \mathsf{pg})$. A frame of a procedure f in $\mathcal{M}$ is a tuple:

$$\mathsf{frame}_f = (l, \mathsf{mem}_f),$$

where

- l is a location in $\mathsf{pg}(f)$.
- $\mathsf{mem}_f \colon \mathsf{locvar}(f) \to \mathsf{Val}$ is a *local memory*.

Let $\mathsf{Frame}_{f,\mathcal{M}}$ be the set of frames of a procedure f in $\mathcal{M}$. Let $\mathsf{Frame}_{\mathcal{M}}$ be the union of all such $\mathsf{Frame}_{f,\mathcal{M}}$ that f is in $\mathcal{M}$. □

The state of a single process is called a *process state*.

Definition 4.13. Given a vocabulary $(\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a domain Val, and a MiniMP model $\mathcal{M}$. A *process state* is a pair:

$$\mathsf{procState} = (\mathsf{stack}, \mathsf{gmem}),$$

where

1. $\mathsf{stack} \in \mathsf{Frame}^*_{\mathcal{M}}$ is the call stack, which is a sequence of frames.
2. $\mathsf{gmem} \colon \mathsf{Global} \to \mathsf{Val}$ is a *global memory*.

□

In a process state, the last element of its *stack* denotes the procedure, at which the process currently is. The location of the last *stack* element denotes the program location, at which the process is.

Let n be a positive integer. Recall that $[n]$ is the set of all integers between 0 and $n-1$, inclusive. Now we define a *state* of a MiniMP program with n processes, each of which is identified by a unique integer in $[n]$.

Definition 4.14. Given a vocabulary $(\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a domain Val, and a MiniMP model. A *state* with n processes is an $(n+1)$-ary tuple:

$$\mathsf{state}_n = (\mathsf{procState}_0, ..., \mathsf{procState}_{n-1}, \mathsf{chan}),$$

where

- $\mathsf{procState}_i$ is the process state of process i.
- $\mathsf{chan}\colon [n] \times [n] \to \mathsf{Val}^*$ is the channel function that maps ordered pairs of integers to sequences of values.

Let State_n be the set of states with n processes. We may simplify State_n to State when n is clear in the context. $\square$

Given an ordered pair of integers (i, j), the value sequence $\mathsf{chan}(i, j)$ represents the *message channel* from process i to j.

We introduce the following shortcut notations for the sake of brevity in the rest of this dissertation:

- $\mathsf{procState}_p(s)$: the process state of process p of a state s
- $\mathsf{stack}(s, p)$: the stack on the process state of the process p in a state s
- $\mathsf{top}(s, p)$: the *top frame* in $\mathsf{stack}(s, p)$, i.e., $\mathsf{tail}(\mathsf{stack}(s, p))$
- $\mathsf{toploc}(s, p)$: the location of the frame $\mathsf{top}(s, p)$
- $\mathsf{mem}(s, p)$: the union of the local memory of $\mathsf{top}(s, p)$ and the global memory of $\mathsf{procState}_p(s)$
- $\mathsf{chan}(s)$: the channel function in state s
- $s[x'/x]_p$: substitutes all appearances of a mathematical expression x with a mathematical expression x' in $\mathsf{top}(s, p)$
- $s \downarrow_p f$ (**push frame**) denotes a state obtained by appending a new frame f to $\mathsf{stack}(s, p)$, i.e.,

$$\begin{aligned} s \downarrow_p f = \ & \text{let } a = \mathsf{stack}(s, p) \text{ in} \\ & \text{let } b = \mathsf{procState}_p(s)[a \circ f/a] \text{ in} \\ & s[b/\mathsf{procState}_p(s)] \end{aligned}$$

- $s \uparrow_p$ (**pop frame**) denotes a state obtained by removing the last frame in $\mathsf{stack}(s, p)$, i.e.,

$$\begin{aligned}\exists \mathcal{M} \exists \xi \in \mathsf{Frame}^*_{\mathcal{M}}\, \exists f \in \mathsf{Frame}_{\mathcal{M}}.\ & \text{let } a = \mathsf{stack}(s,p) \text{ in} \\ & \text{let } b = \mathsf{procState}_p(s)[\xi/a] \text{ in} \\ & a = \xi \circ f \wedge s \uparrow_p = s[b/\mathsf{procState}_p(s)]\end{aligned}$$

- $s[\mathsf{var} \leftarrow \mathsf{val}]_p$ (**assign**) denotes a state obtained by assigning a value val to a variable var at a state s for a process p. It is either the case that only the local memory of $\mathsf{stack}(s, p)$ or only the global memory in $\mathsf{procState}_p(s)$ is changed from s to $s[\mathsf{var} \leftarrow \mathsf{val}]_p$ such that

$$\mathsf{mem}(s[\mathsf{var} \leftarrow \mathsf{val}]_p, p)(\mathsf{var}) = \mathsf{val}$$

- $s[(p, q)\,!\,\mathsf{val}]$ (**enqueue**) denotes a state obtained by appending a value val on the channel identified by the ordered pair (p, q), i.e.,

$$\begin{aligned}s[(p,q)\,!\,\mathsf{val}] = \ & \text{let } a = \mathsf{chan}(s)(p,q) \text{ in} \\ & \text{let } b = \mathsf{chan}(s)[(p,q) \mapsto a \circ \mathsf{val}] \text{ in} \\ & s[b/\mathsf{chan}(s)]\end{aligned}$$

- $s[(p, q)?\mathsf{var}]$ (**dequeue**) denotes a state obtained by removing the first element from the channel identified by the ordered pair (p, q), and then saving the removed element in var. That is,

$$\begin{aligned}\exists \xi \in \mathsf{Val}^*\, \exists \mathsf{val} \in \mathsf{Val}.\ & \text{let } a = \mathsf{chan}(s)(p,q) \text{ in} \\ & \text{let } b = \mathsf{chan}(s)[(p,q) \mapsto \xi] \text{ in} \\ & \text{let } c = s[\mathsf{var} \leftarrow \mathsf{val}]_q \text{ in} \\ & a = \mathsf{val} \circ \xi \wedge s[(p,q)?\mathsf{var}] = c[b/a]\end{aligned}$$

The notations `!` and `?` are borrowed from CSP [51].

4.3.2 Interpretation

To describe behaviors of a MiniMP program, one needs to describe how is an expression evaluated and how does a local transition alter a state.

$$
\begin{array}{rcl}
[\![c]\!]_n(s,p) & = & c, \text{ where } c \in \mathsf{Val} \\
[\![\texttt{PID}]\!]_n(s,p) & = & p \\
[\![\texttt{NPROCS}]\!]_n(s,p) & = & n \\
[\![v]\!]_n(s,p) & = & \mathsf{mem}(s,p)(v) \\
[\![v\texttt{[}e\texttt{]}]\!]_n(s,p) & = & [\![v]\!]_n(s,p)([\![e]\!]_n(s,p)) \\
[\![\texttt{len(}e\texttt{)}]\!]_n(s,p) & = & |[\![e]\!]_n(s,p)| \\
[\![\texttt{new [}e\texttt{]}]\!]_n(s,p) & = & \xi \text{ s.t. let } m = [\![e]\!]_n(s,p) \text{ in } |\xi| = m \wedge \\
 & & \xi[i] = \mathbb{UNDEF} \text{ for } 0 \le i < m \\
[\![e_0 \ \mathsf{bop} \ e_1]\!]_n(s,p) & = & [\![e_0]\!]_n(s,p) \ \mathsf{bop} \ [\![e_1]\!]_n(s,p) \\
[\![\mathsf{uop} \ e]\!]_n(s,p) & = & \mathsf{uop} \ [\![e]\!]_n(s,p) \\
[\![\texttt{empty(}e\texttt{)}]\!]_n(s,p) & = & \mathsf{chan}(s)(p,[\![e]\!]_n(s,p)) = \epsilon \\
\mathbb{UNDEF}, & & \text{if any undefined case}
\end{array}
$$

Figure 4.7: MiniMP Expression Evaluation. The bop stands for a binary operator (e.g. `+`, `-`, ...). The uop stands for a unary operator (e.g. `!`, `-`, ...).

Definition 4.15. Given a vocabulary $\mathcal{V}$ and a domain Val. A *n-processes evaluation function* $[\![e]\!]_n : \mathsf{State}_n \times [n] \to \mathsf{Val}$ evaluates an expression $e \in \mathsf{Expr}_\mathcal{V}$ for a process at a state. The definition of $[\![e]\!]_n$ is given in Fig. 4.7. □

The evaluation function is complete. For undefined cases, such as reading uninitialized variable, array out-of-bound access, division by zero, the function returns $\mathbb{UNDEF}$. For brevity, we may simplify $[\![\ ]\!]_n$ to $[\![\ ]\!]$ when the number of processes is clear in a context.

Definition 4.16. Given a vocabulary and a n-processes evaluation function $[\![\ ]\!]$. An *n-processes interpretation function* $I_n : \mathsf{State} \times [n] \times T \to 2^{\mathsf{State}}$, describes the states, each of which is a possible result of executing a local transition $(l, g, a, l') \in T$ from a state $s \in \mathsf{State}$ by a process $p \in [n]$. The definition of $I_n(s, p, (l, g, a, l'))$ is given in Fig. 4.8. □

When the number of processes in clear in a context, we may simplify I_n to I.

We briefly explain Fig. 4.8,

$I_n(s, p, (l, g, a, l')) =$

$\emptyset$, if $[\![g]\!](s,p) = \mathbf{F} \vee \mathsf{toploc}(s,p) \neq l$

or, $\{s[l'/l]_p\}$, if a is `skip`

or, let $s' = s[l'/l]_p$ in
let $\mathsf{val} = [\![e]\!](s,p)$ in $\{s'[v \leftarrow \mathsf{val}]_p\}$, if a is v `=` e

or, let $s' = s[l'/l]_p$ in
let $\mathsf{idx} = [\![e_0]\!](s,p)$ in
let $\mathsf{val} = [\![e_1]\!](s,p)$ in
let $\mathsf{arrval} = [\![v]\!](s,p)[\mathsf{idx} \mapsto \mathsf{val}]$ in
$\{s'[v \leftarrow \mathsf{arrval}]_p\}$, if a is v`[`e_0`]` `=` e_1

or, Suppose $\alpha_0, \ldots, \alpha_{n-1}$ are parameters of f, and l_0 is the start location of f,
let $\beta_i = [\![e_i]\!](s,p)$ for $0 \leq i < n$ in
let $\mathsf{fr} = (l_0, \mathsf{mem}_f)$ in
$\{(s \downarrow_p \mathsf{fr})[\ \alpha_0 \leftarrow \beta_0, \ldots, \alpha_{n-1} \leftarrow \beta_{n-1}]_p\}$, if a is $f(e_0, \ldots, e_{n-1})$

or, Suppose $[\![e]\!](s,p)$ has been saved as the latest returned value,
$\{(s \uparrow_p)\}$, if a is **`return`** e

or, Suppose retval is the latest returned value,
$\{(s[x \leftarrow \mathsf{retval}]_p\}$, if a is `assign return to` x

or, let $s' = s[l'/l]_p$ in
let $\mathsf{val} = [\![e_0]\!](s,p)$ in
let $\mathsf{dest} = [\![e_1]\!](s,p)$ in $\{s'[\ (p, \mathsf{dest})!\mathsf{val}]\}$, if $a =$ **`send`** e_0 **`to`** e_1

or, let $s' = s[l'/l]_p$ in
let $\mathsf{src} = [\![e]\!](s,p)$ in
$\{(s'[\ (\mathsf{src}, p)?v\ ])\}$, if a is **`recv`** v **`from`** e

or, let $s' = s[l'/l]_p$ in
$\{s'' \in \mathsf{State} \mid$
$\exists q \in [n].\ s'' = s'[(q,p)?v_0][v_1 \leftarrow q]_p \wedge$
$|\mathsf{chan}(s')(q,p)| > 0$ if $a =$ **`recv`** v_0 **`from any`** v_1
$\}$,

Figure 4.8: The definition of the n-processes interpretation function I_n that describes the results of executing a local transition from a state by a process.

- A local transition cannot be executed by a process from a state s, if the guard evaluates to `false` at s, or the location of the process at s does not match the origin of the local transition.
- An execution of a local transition alters the current location of the running process. If the local transition is associated with the `skip` statement, nothing else will be changed by the execution.
- If a local transition is associated with an assignment, either the local memory of the top entry in the call stack, or the global memory, of the running process will be updated after an execution.
- If a local transition is associated with a call, a new frame will be pushed onto the call stack of the running process, and actual parameters will be assigned to the formal parameters, after an execution.
- If a local transition is associated with a `return` e action, after an execution, the top entry in the call stack of the running process will be popped and the returned value will be saved.
- If a local transition is associated with an `assign return to` x action, the saved returned value will be assigned to x.
- If a local transition is associated with a **`send`** statement, after an execution, the sending value will be appended to the message channel identified by the ordered pair (p, q), where p is the sender and q is the receiver.
- It a local transition is associated with a **`recv`** v **`from`** src statement, after an execution, the first element in the message channel identified by (p, q) is removed and is assigned to v, where p is the sender and q is the receiver.
- If a local transition is associated with a wildcard **`recv`** v **`from any`** src statement, an execution nondeterministically selects a message channel among multiple of them that are non-empty and are identified by (p, q) for some sender $p \in [n]$ and the receiver q. With the selected message channel, the execution alters the state as if a deterministic `recv` is performed on that channel.

Definition 4.17. Given a n-processes interpretation I_n. We define $\mathsf{nexts}_n(S, T)$ to be the union of the I_n results of executing a local transition $t \in T$ from a state $s \in S$ by every process, i.e.,

$$\mathsf{nexts}_n(S, T) = \bigcup_{s \in S, t \in T, p \in [n]} I_n(s, p, t).$$

Denoted by $\mathsf{nexts}_n^i(S, T)$, the repeated i applications of nexts_n for $i \geq 0$ is defined by

$\mathsf{nexts}_n^0(S,T) = \mathsf{nexts}_n(S,T)$,

$\mathsf{nexts}_n^{i+1}(S,T) = \mathsf{nexts}_n(\mathsf{nexts}_n^i(S,T),T)$.

□

We say a state s is *reachable* from a set of states S over a set of local transitions T, if there is a non-negative i such that $s \in \mathsf{nexts}_n^i(S,\, T)$.

The bahaviors of a MiniMP model are described by a state space graph.

Definition 4.18. Given a vocabulary $\mathcal{V}$, a domain and a MiniMP model $\mathcal{M}$. Let T be the union of local transitions of procedures in $\mathcal{M}$. A finite *state space graph* G of $\mathcal{M}$ with n processes is a tuple:

$$G = (n,\, s_0,\, S,\, \rightsquigarrow),$$

where

- n is the number of processes
- s_0 is the initial state where all processes are about to call the `main` procedure
- S is the set of states that are reachable from s_0 over T
- $\rightsquigarrow_{\mathcal{M}} : \mathsf{State} \times [n] \times \mathsf{Stmt}_{\mathcal{V}} \times \mathsf{State}$ is a *global transition realtion* defined by

$$\{(s,p,\alpha,s') \mid s' \in I(s,p,\alpha), s \in S, p \in [n], \alpha \in T\}.$$

□

For a global transition $t = (s,p,\alpha,s')$, we call s the *source state*, denoted by $\mathsf{src}(t)$, and s' the *destination state*, denoted as $\mathsf{dest}(t)$. We use $\mathsf{proc}(t)$ to denote the process of t.

In the rest of this dissertation, for a global transition t, instead of referring to the atomic statement associated to t, we will refer to the action of t.

Definition 4.19. Given a n-processes evaluation function $[\![\ ]\!]$ and a n-processes interpretation function I. Denoted by $\mathsf{act}(t)$, the *action* of a transition $t = (s,p,\alpha,s')$ corresponds to the atomic statement associated with α, where expressions are evaluated with respect to the source state s and the process p. The following defines $\mathsf{act}(t)$ for few cases of t:

$$\text{let } g(x) = \begin{cases} x, & \text{if } x \in \mathsf{Var} \\ g(x')[[\![i]\!](s,p)], & \text{if } x \text{ matches } x'\,[i] \end{cases} \text{ in}$$

1. $\mathsf{act}(t) = g(x)$ `=` $[\![e]\!](s,p)$, if α is labeled x `=` e
2. $\mathsf{act}(t) = \mathsf{send}([\![e_0]\!](s,p), [\![e_1]\!](s,p))$, if α is labeled **`send`** e_0 **`to`** e_1
3. $\mathsf{act}(t) = \mathsf{recv}(g(x), [\![e]\!](s,p))$, if α is labeled **`recv`** x **`from`** e
4. $\mathsf{act}(t) = \mathsf{recv}(g(x), q)$, if α is labeled **`recv`** x **`from any`** x', where

q is the sender s.t. $\mathsf{chan}(s)(q,p)$ is the only message channel changed from s to s'.

We leave other cases of t that are obvious to readers out here. □

Note an action is deterministic.

If taking states as vertices and global transitions as directed edges, a state space graph G is a directed graph. A path ρ in G is a finite sequence of transitions $t_0 t_1 t_2 ... t_{m-1}$ such that for any pair $t_i t_{i+1}, 0 \leq i < m-1$, in ρ, $\mathsf{src}(t_{i+1}) = \mathsf{dest}(t_i)$. The notation $\mathsf{Tr}(\rho)$ denotes the set of transitions in ρ. $\mathsf{St}(\rho)$ denotes the set of states in a path ρ, i.e. $\mathsf{St}(\rho) = \bigcup_{t \in \mathsf{Tr}(\rho)} \{\mathsf{src}(t), \mathsf{dest}(t)\}$. An *execution* π in G is a path such that the the first state $\mathsf{src}(\mathsf{head}(\pi))$ is s_0 and there is no transition emanating from the last state of π, i.e., $\forall t \in \rightsquigarrow. \mathsf{src}(t) \neq \mathsf{dest}(\mathsf{tail}(\pi))$. Note we assume an execution is acyclic. Paths that contain cycles mean that there are infinite loops or recursions. We only focus on programs that will evetually terminate in this dissertation.

Chapter 5

THE SPECIFICATION LANGUAGE FOR MINIMP

In this section, we introduce a specification language MiniSpec for MiniMP programs. A MiniSpec specification specifies partial correctness properties for a MiniMP statement sequence. Inspired by Hoare logic, as well as Lamport's "Hoare Logic" for concurrent programs [70], we invented an abstract representation, called a *collective triple*, for a MiniMP statement sequence with its specification. When it is clear in the context, we may use "triple" to refer "collective triple" for brevity.

This chapter is structured as the follow. §5.1 describes an abstract representation of a MiniMP program segment. §5.2 talks about the notion of *path predicates*. Syntax of MiniSpec is given in §5.3. Semantics of MiniSpec with respect to a MiniMP program segment is described in §5.4. Finally, we define the meaning of that a MiniMP statement sequence satisfies a specification using collective triples in §5.5.

5.1 Program Segment

A *program segment* represents a partial MiniMP program. It is associated to a statement sequence of a complete MiniMP program.

Definition 5.1. Given a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a MiniMP program P, and the model $\mathcal{M} = (\mathtt{main}, \mathsf{pg})$ of P. Let C: *Statement** be a sequence of statements in P such that C contains no **return** sub-statement. Corresponding to $\mathcal{M}$, a *program segment* of C, denoted $l\colon C\ l'$, is a tuple

$$(\mathsf{Loc}, l, l', T),$$

where

- Loc is a set of program locations,
- $l \in \mathsf{Loc}$ is the sole entry location of C,
- $l' \in \mathsf{Loc}$ is the sole exit location of C, and
- T is a set of local transitions such that $(l', T) = \mathsf{transStmt}(l, C)$ and $T \subseteq \mathsf{pg}(f)$, for $f \in \mathsf{Procedure}$.

□

Example 5.2. A program segment associated to the loop *Statement* of the `bcast` procedure (line 5-9) in Fig. 4.2 is given by the follow

$$(\{l_2, \ldots, l_9\},\ l_2,\ l_9,\ T),$$

where $T = \{$

$(l_2,$ `i < NPROCS`, `skip`, $l_3)$, $(l_3,$ `i != root`, `skip`, $l_4)$,
$(l_4,$ `true`, `send dat to i`, $l_5)$, $(l_5,$ `true`, `skip`, $l_7)$,
$(l_3,$ `i == root`, `skip`, $l_6)$, $(l_6,$ `true`, `skip`, $l_7)$,
$(l_7,$ `true`, `i = i + 1`, $l_8)$, $(l_8,$ `true`, `skip`, $l_2)$,
$(l_2,$ `i >= NPROCS`, `skip`, $l_9)$,
$\}$

Figure 5.1 visualizes the program segment. The visualization makes it clear that the program segment is a sub-graph of the one of `bcast` given in Fig. 4.6. □

Definition 5.3. Given a vocabulary $\mathcal{V}$, a domain, a MiniMP model $\mathcal{M}$ and a program segment $l\colon C\ l' = (\mathsf{Loc}, l, l', T)$ corresponding to $\mathcal{M}$. The *state space graph* of $l\colon C\ l'$ with n processes is a tuple:

$$(n, S_0, S, \rightsquigarrow_{l\colon C\ l'}),$$

where

- n is the number of processes,
- S_0 is a set of initial states, at each of which, a process is at the location l,

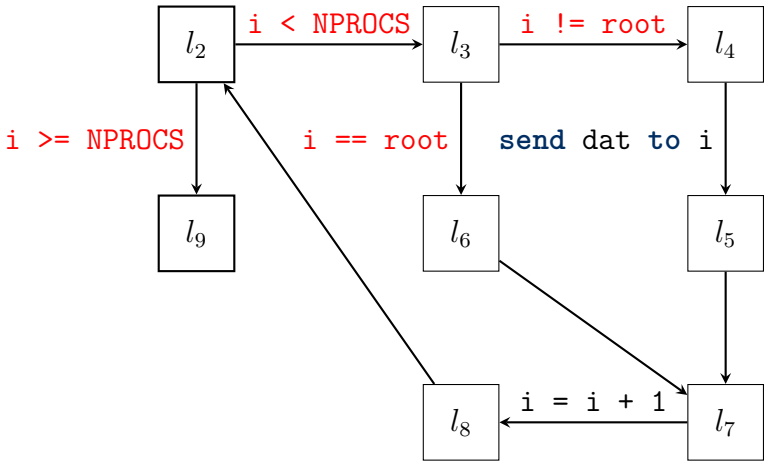

Figure 5.1: The program segment of the loop *Statement* in the `bcast` procedure showed in Fig. 4.2.

- S is the set of reachable states from S_0 over T
- $\rightsquigarrow_{l:\, C\ l'} : \mathsf{State} \times [n] \times \mathsf{Stmt}_{\mathcal{V}} \times \mathsf{State}$ is a set of global transitions defined by

$$\{(s, p, \alpha, s') \in \rightsquigarrow \mid s' \in I(s, p, \alpha), s \in S, p \in [n], \alpha \in T\}$$

□

Let G be a state space graph of a program segment $l\colon C\ l'$. A path in G is said a path of $l\colon C\ l'$. An execution π in G is said an execution of $l\colon C\ l'$. For brevity, we also say G is a state space graph of C and π is an execution of C when there is no possible ambiguity.

Given a program segment and a path ρ, there are two kinds of global transitions in ρ that we have special interests.

Definition 5.4. Given a program segment $l\colon C\ l'$ and a path ρ. A global transition $t \in \mathsf{Tr}(\rho)$ marks the *first entering* of C by a process p on ρ, denoted $\mathsf{enter}^C_{p,\rho}$, if t is the first such transition on ρ that

$$\mathsf{toploc}(\mathsf{src}(t), p) = l \ \wedge\ \mathsf{proc}(t) = p.$$

Given $\mathsf{enter}^C_p \in \mathsf{Tr}(\rho)$. A global transition $t \in \mathsf{Tr}(\rho)$ marks the *exiting* of C by process p corresponding to $\mathsf{enter}^C_{p,\rho}$, denoted $\mathsf{exit}^C_{p,\rho}$, if t is the first such transition appearing after $\mathsf{enter}^C_{p,\rho}$ on ρ that

1. $\mathsf{toploc}(\mathsf{dest}(t), p) = l' \ \wedge \ \mathsf{proc}(t) = p$, and
2. $|\mathsf{stack}(\mathsf{dest}(t), p)| = |\mathsf{stack}(\mathsf{src}(\mathsf{enter}^C_{p,\rho}), p)|$

Note that $\mathsf{exit}^C_{p,\rho}$ depends on the stack size of the source state of $\mathsf{enter}^C_{p,\rho}$ because the execution π may involve recursions. □

When the context is clear that there is only one unique path ρ, we simplify $\mathsf{enter}^C_{p,\rho}$ and $\mathsf{exit}^C_{p,\rho}$ to enter^C_p and enter^C_p, respectively. Remark that there is at most one enter^C_p and exit^C_p in a given path for C and $p \in [n]$.

5.2 Path Predicate

A path predicate consumes a path and returns either true or false.

Definition 5.5. Given a path ρ. Let $A, B, C \in \wp(\rightsquigarrow)$ be three global transition sets. A predicate $\mathsf{no}\ A\ \mathsf{after}\ B\ \mathsf{until}\ C$ is true for path ρ, iff

$$\forall i, j \in \mathbb{Z}.\ ((0 \leq i < j < |\rho|) \wedge \rho[i] \in B \wedge \rho[j] \in A) \Rightarrow \exists k \in \mathbb{Z}.\ i \leq k < j \wedge \rho[k] \in C$$

□

The intuitive meaning of $\mathsf{no}\ A\ \mathsf{after}\ B\ \mathsf{until}\ C$ for a path ρ is that no transition in A can happen after a transition in B until a transition in C appears on ρ [36][1].

Definition 5.6. Given a path ρ. Let $A, B, C \in \wp(\rightsquigarrow)$ be three global transition sets. The predicate $\mathsf{no}\ A\ \mathsf{after}\ B\ \mathsf{until}\ C$ is a *path predicate*. Moreover, if f, g are path predicates, $f \wedge g$, $\neg f$ and $f \vee g$ are also *path predicates*. □

Definition 5.7. Given a state space graph G. Let ρ be a path in G. Let s, s' be states in G. For a path predicate g, we use

- $\rho \models g$ to denote that g is true for ρ, and

[1] An absence assertion in fact can be expressed as an LTL formula, which falls into the exact "absence of" pattern with scope "after .. until" in the pattern system concluded in [36]. The pattern system can be also found at https://matthewbdwyer.github.io/psp/patterns/ltl.html (visited Nov 2019)

- $G, s \hookrightarrow s' \models g$ to denote that g is true for every such path ζ in G that $\mathsf{src}(\mathsf{head}(\zeta)) = s$ and $\mathsf{dest}(\mathsf{tail}(\zeta)) = s'$.

□

5.3 Syntax of The MiniMP Specification Language

The grammar of the specification language MiniSpec for MiniMP is given in Fig. 5.2. The *SpecExpr* is an extension of *Expr*. The overlap of *SpecExpr* and *Expr* is omitted.

```
List-of-T   := T (, T)* |
Spec        := Clause*
Clause      := requires SpecExpr ; | ensures SpecExpr ;
             | requires AbsentSet ; | ensures AbsentSet ;
             | assigns List-of-LSpecExpr ;
SpecExpr    := ...| LSpecExpr | \old ( SpecExpr ) | SpecExpr .. SpecExpr
             | \on ( SpecExpr , SpecExpr )
             | \forall Identifier ; SpecExpr
             | \exists Identifier ; SpecExpr
LSpecExpr   := LExpr | LSpecExpr [ SpecExpr ]
AbsentSet   := (Expr ==>)?  Absent (&& Absent)*
Absent      := \no Event \after Event \until Event
Event       := \send( Expr , Expr ) | \exit( Expr ) | \enter( Expr )
```

Figure 5.2: The syntax of MiniMP specification language

A specification consists of a set of *Clause*s. There are five kinds of clauses: the `requires` *SpecExpr* clause specifies a *state requirement*; the `ensures` *SpecExpr* clause specifies a *state guarantee*; the `requires` *AbsentSet* clause specifies a *path requirement*; the `ensures` *AbsentSet* clause specifies a *path guarantee*; the `assigns` *SpecExpr* clause specifies a *frame condition*, which is essentially a state-guarantee.

State-requirement corresponds to pre-condition in Hoare logic. Similarly, state-guarantee corresponds to post-condition. A state-requirement is an assertion holds at a pre-state for all processes. A state-guarantee is an assertion holds at a post-state for all processes.

The path requirement and path guarantee are designed for specifying communication correctness. We will explain and give precise definition for them later.

Definition 5.8. Given a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$. Denoted by $\mathsf{SExpr}_{\mathcal{V}}$, the set of *SpecExpr*s over Var. Denoted by $\mathsf{SEvent}_{\mathcal{V}}$, the set of *Event*s over Var. Denoted by $\mathsf{SAbsent}_{\mathcal{V}}$, the set of *Absent* assertions over Var. □

A few constructs were borrowed from ACSL:

1. A range $e_0 \,..\, e_1$ is the finite set of integers that are bounded between e_0 and e_1, both inclusive.
2. The **\old** expression bridges the pre- and post-states. Expression **\old**(e) refers to the value of e as if e is evaluated at a corresponding pre-state. The **\old** construct can only be used in a state-guarantee.
3. First order logic quantifiers: **\forall** and **\exists**

In addition, we designed two new constructs for concurrency:

1. The **\on** construct bridegs different processes. The expression **\on**(e_0,e_1) refers to the value of expression e_0 on the process whose PID equals to the value of e_1.
2. The *absence assertion* that allows users to specify the no A after B until C path predicate. It is the basic element of path requirements or path guarantees. An absence assertion consists of *events*. There are three kinds of events: **\send**, **\enter** and **\exit**, each of which represents a specific kind of transitions associated with a path. For a path, the assertion **\no** θ_0 **\after** θ_1 **\until** θ_2 means that no event θ_0 can happen after event θ_1 until event θ_2 on the path.

Informally, a **\send**$(p,\ q)$ event represents the set of transitions associated with send$(p,\ q)$ actions; an **\enter**(p) event represents the set of transitions marking the first entering of an associated statement sequence by process p; an **\exit**(p) event represents the set of transitions marking the exiting, that corresponds to the first entering, of an associated statement sequence by process p.

These three kinds of events can be combined in many ways in an absence assertion to express various behaviors. In fact, to specifying properties for MiniMP programs, only a few combinations of the events are needed. So for simplicity, we impose Restriction 5.9 on the use of events in absence assertions.

Restriction 5.9. Let p, q be *Expr*s, an absence assertion,

1. can only have the following form if it is a path requirement:

   `\no \send(`p`, PID) \after \exit(`p`) \until \exit(PID)`

2. can only have one of the following forms if it is a path guarantee:

   (a) `\no \send(PID,` p`) \after \enter(PID) \until \enter(`q`)`

   (b) `\no \exit(PID) \after \enter(PID) \until \enter(`p`)`

□

The syntax of the path requirement or path guarantee clauses implies that they can conceptually be represented as the conjunction of a set of absence assertions.

A MiniSpec specification is assigned to a statement sequence. We call a specification of a program segment C a *contract* of C. When there is no ambiguity, we also call it a *contract* of C.

5.4 Semantics of The MiniMP Contracts

Similar to Hoare logic, the semantics of the MiniMP specification language is based on the notion of the *pre-state* and *post-state*.

We fix the following structures for the discussion in the rest of this chapter: a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a domain Val, a MiniMP model $\mathcal{M}$ and a program segment $l\colon C\ l'$ corresponding to $\mathcal{M}$.

Definition 5.10. Let G be the state space graph of $l\colon C\ l'$ with n processes. A state s in G is a *pre-state* of C, if all n processes are at location l in s. Given a pre-state s of C and a path ρ containing s in G, a state s' is a *post-state* of C in G corresponding to s, if

1. all n processes are at the location l', and
2. $\forall p \in [n].\ |\mathsf{stack}(s', p)| = |\mathsf{stack}(s, p)|$

Denoted by $\mathsf{pre}(s', \rho, l\colon C\ l')$, the pre-state of C, to which the given post-state s' of C corresponds, in path ρ. □

$$
\begin{aligned}
[\![e_0 \ \texttt{..}\ e_1]\!]^{\mathsf{spec}}_n(s,p) &= \{i \in \mathbb{Z} \mid [\![e_0]\!]^{\mathsf{spec}}_n(s,p) \le i < [\![e_1]\!]^{\mathsf{spec}}_n(s,p)\} \\
[\![\texttt{a[}e_0 \ \texttt{..}\ e_1\texttt{]}]\!]^{\mathsf{spec}}_n(s,p) &= \{[\![\texttt{a[}i\texttt{]}]\!]^{\mathsf{spec}}_n(s,p) \mid [\![e_0]\!]^{\mathsf{spec}}_n(s,p) \le i < [\![e_1]\!]^{\mathsf{spec}}_n(s,p)\} \\
[\![\texttt{\textbackslash on(}e_0\texttt{, }e_1\texttt{)}]\!]^{\mathsf{spec}}_n(s,p) &= \begin{cases} [\![e_0]\!]^{\mathsf{spec}}_n(s, [\![e_1]\!]^{\mathsf{spec}}_n(s,p)), & \text{if } [\![e_1]\!]^{\mathsf{spec}}_n(s,p) \in [n] \\ \mathbb{UNDEF}, & \text{otherwise} \end{cases} \\
[\![\texttt{\textbackslash old(}e\texttt{)}]\!]^{\mathsf{spec}}_n(s,p) &= [\![e]\!]^{\mathsf{spec}}_n(\mathsf{pre}(s,\rho,l\colon C\ l'),p) \\
[\![\texttt{\textbackslash forall}\ v\texttt{;}\ e]\!]^{\mathsf{spec}}_n(s,p) &= \begin{cases} \mathbf{T}, & \text{if } [\![e]\!]^{\mathsf{spec}}_n(s,p) = \mathbf{T} \text{ for every } v \in \mathsf{Val} \\ \mathbf{F}, & \text{otherwise} \end{cases} \\
[\![\texttt{\textbackslash exists}\ v\texttt{;}\ e]\!]^{\mathsf{spec}}_n(s,p) &= \begin{cases} \mathbf{T}, & \text{if } [\![e]\!]^{\mathsf{spec}}_n(s,p) = \mathbf{T} \text{ for some } v \in \mathsf{Val} \\ \mathbf{F}, & \text{otherwise} \end{cases} \\
[\![e]\!]^{\mathsf{spec}}_n(s,p) &= [\![e]\!]_n(s,p)
\end{aligned}
$$

Figure 5.3: The definition for the n-processes specification evaluation function $[\![e]\!]^{\mathsf{spec}}_n$, which evaluates a contract expression e for a process p at a pre- or a post-state s of a program segment $l\colon C\ l'$ in a path ρ.

Conceptually, a pre-state and post-state of a statement C is defined as if there is a pair of "bulk-synchronous" barriers at the beginning and end of C. The pre-state is the state where all processes are blocked at the first barrier. The post-state is the state where all processes are blocked at the second barrier. Considering the fact that C may contain recursions, the post-state must have a consistent call stack size with the pre-state.

The semantics of a contract expression is given by an evaluation function extended from the evaluation function for MiniMP expressions.

Definition 5.11. Let G be the state space graph of $l\colon C\ l'$ with n processes. Let ρ be a path in G that contains a pre-state $\mathsf{pre}(s,\rho,l\colon C\ l')$ of C and a post-state s of C. The n-processes *specification evaluation function* $[\![e]\!]^{\mathsf{spec}}_n : \mathsf{State} \times [n] \to \mathsf{Val}$ evaluates a contract expression $e \in \mathsf{SExpr}_{\mathcal{V}}$ for a process p at either the pre- or the post-state. $[\![e]\!]^{\mathsf{spec}}_n$ is defined in Fig. 5.3. □

When the number of processes is clear in the context, we simplify $[\![e]\!]^{\mathsf{spec}}_n$ to $[\![e]\!]^{\mathsf{spec}}$.

$$
\begin{aligned}
I^{\mathsf{event}}_{C,\rho}(\texttt{\textbackslash enter(}e\texttt{)}, s, p) &= \{\mathsf{enter}^{C}_{[\![e]\!](s,p)}\} \\
I^{\mathsf{event}}_{C,\rho}(\texttt{\textbackslash exit(}e\texttt{)}, s, p) &= \{\mathsf{exit}^{C}_{[\![e]\!](s,p)}\} \\
I^{\mathsf{event}}_{C,\rho}(\texttt{\textbackslash send(}e_0\texttt{, }e_1\texttt{)}, s, p) &= \{t \in \rightsquigarrow \mid \mathsf{proc}(t) = [\![e_0]\!](s,p) \wedge \\
&\qquad \mathsf{act}(t) = \texttt{send(}v\texttt{, }[\![e_1]\!](s,p)\texttt{)}, v \in \mathsf{Var}\}
\end{aligned}
$$

Figure 5.4: The definition of the event interpretation function $I^{\mathsf{event}}_{C,\rho}$ with respect to a path ρ and a program segment $l\colon C\ l'$

A frame condition states what is unchanged by a program segment. The `assigns` clause specifies a list of *LSpecExpr*s and encodes the meaning that any object that is not listed will never be changed by the program segment. For a programming language as simple as MiniMP, a frame condition can always be transformed to a state-guarantee statically. Because one can always assert that a variable at a post-state preserves its value in the pre-state using the `\old` expression. All visible variables to a location are known statically. The transformation process is omitted. In the rest of this dissertation, we will not pay special attention to frame conditions.

An *Event* is interpreted to a set of global transitions with respect to a path of a program segment.

Definition 5.12. Let G be the state space graph of C with n processes. Let ρ be a path in G. The *event interpretation* function $I^{\mathsf{event}}_{C,\rho} : \mathsf{SEvent}_{\mathcal{V}} \times \mathsf{State} \times [n] \to \wp(\rightsquigarrow)$ depends on C and ρ. It interprets an event $\theta \in \mathsf{SEvent}_{\mathcal{V}}$ to a set of global transitions at a state s for a process p. $I^{\mathsf{event}}_{C,\rho}(\theta, s, p)$ is defined in Fig. 5.4. □

An absence assertion will be interpreted to a path predicate.

Definition 5.13. Given the state space graph G of the program segment C with n processes. Let ρ be a path in G. Let $I^{\mathsf{event}}_{C,\rho}$ be the event interpretation function depending on C and ρ. The n-processes *absence assertion evaluation* function $[\![e]\!]^{\mathsf{absent}}_{C,\rho}$:

$\mathsf{State} \times [n] \to \{\mathbf{T}, \mathbf{F}\}$ depends on $I^{\mathsf{event}}_{C,\rho}$. It evaluates an absence assertion $e \in \mathsf{SAbsent}_{\mathcal{V}}$ to either true or false at a state s for a process p. We define

$$[\![\texttt{\textbackslash no}\ \theta_0\ \texttt{\textbackslash after}\ \theta_1\ \texttt{\textbackslash until}\ \theta_2]\!]^{\mathsf{absent}}_{C,\rho}(s,p) =$$
$$\text{let } A = I^{\mathsf{event}}_{C,\rho}(\theta_0, s, p) \text{ in}$$
$$\text{let } B = I^{\mathsf{event}}_{C,\rho}(\theta_1, s, p) \text{ in}$$
$$\text{let } C = I^{\mathsf{event}}_{C,\rho}(\theta_2, s, p) \text{ in } \rho \models \mathsf{no}\ A\ \mathsf{after}\ B\ \mathsf{until}\ C.$$

Recall we also write $\rho \models [\![e]\!]^{\mathsf{absent}}_{C}(s,p)$ for $[\![e]\!]^{\mathsf{absent}}_{C,\rho}(s,p) = \mathbf{T}$. □

When it is clear that there is a unique program segment in the context, we simplify $[\![e]\!]^{\mathsf{absent}}_{C,\rho}(s,p)$ to $[\![e]\!]^{\mathsf{absent}}_{\rho}(s,p)$.

Writing Contracts for MiniMP Statements. A MiniMP program will be run by a group of processes collectively. The program is written with the idea that it will be run by a generic process. A contract of a MiniMP statement sequence is designed with the same idea. A contract should be assigned to a statement sequence that is intended to be run by all processes collectively. The contract specifies the collective behavior of the code. Intuitively, a contract can be understood by the user in the perspective of a generic process. For example, an absence assertion `\no \send(`p`, PID) \after \exit(`p`) \until \exit(PID)` in a contract of a statement sequence C can be understood as " process p shall not send a message to me after it exits C until I exit C too.", where "I" refers to a generic process.

We use the following two MiniSpec specifications as examples to give readers a taste of what does "a statement satisfies a contract in a collective way" mean. The formal description of the contract satisfaction will be described in §5.5.

Example 5.14. Figure 5.5 is a contract for the branch statement (at line 3-11) of the `bcast` procedure presented in the left of Fig. 4.2. In the contract, the state-requirement states that all processes must have the same value for their `root`. In addition, the value of `root` shall be bounded between 0 and `NPROCS`. The statement may modify variables `dat` and `i` so they are listed in an `assigns` clause. The state-guarantee expresses

```
1  var i, root, dat;
2  // contract:
3  requires \forall t; \on(root, t) == root;
4  requires 0 <= root && root < NPROCS;
5  assigns dat, i;
6  ensures PID == root ==> dat == \old(dat);
7  ensures dat == \on(dat, root);
8  ensures \no \exit(PID) \after \enter(PID) \until \enter(root);
9  // code:
10 if (PID == root) {
11   i = 0;
12   while (i < NPROCS) {
13     if (i != root)
14       send dat to i;
15     i = i + 1;
16   }
17 } else
18   recv dat from root;
```

Figure 5.5: A contract of a branch statement that "broadcasts" a value.

that, for every process, if it is `root`, `dat` remains unchanged. Otherwise, `dat` will be modified to have the same value as the one on process `root` at the post-state. The contract finally guarantees that no process can exit the statement until process `root` enters it. □

Example 5.15. Figure 5.6 is a contract for the branch statement (at line 3-15) in the `gather` procedure presented in the right of Fig. 4.3. The contract requires that all processes must have the same value for `root`, which must refer to a valid `PID` value. The statement may modify variables `i`, `y` and `dat`, which are therefore listed in the `assigns` clause. The state-guarantee states that the value of `dat`$[i]$ on the `root` process equals to the value of `val` on process i, for every process i. Finally, the contract guarantees that, for the `root` process, it will not exit the statement until all processes have entered it. □

```
1  var i, y, dat, root, val;
2  // contract:
3  requires \forall t; \on(root, t) == root;
4  requires 0 <= root && root < NPROCS;
5  assigns dat, i, y;
6  ensures PID == root ==>
7            \forall int t; 0 <= t && t < NPROCS ==>
8              dat[t] == \on(val, t);
9  ensures PID == root ==>
10   \no \exit(PID) \after \enter(PID) \until \enter(0 .. NPROCS-1);
11 // code:
12 if (PID == root) {
13   i = 0;
14   dat = new [NPROCS];
15   dat[root] = val;
16   while (i < NPROCS) {
17     if (i != root) {
18       recv val from any y;
19       dat[y] = val;
20     }
21     i = i + 1;
22   }
23 } else
24   send val to root;
```

Figure 5.6: A contract for the branch statement that "gathers" values from all processes.

5.5 The Collective Triple

In this section, we first define an abstract representation for a program segment with a contract, called *collective triple*. Then we discuss about the intuitive meaning of a collective triple. Finally, we formally define the meaning of the satisfaction of a program segment to a contract in terms of a corresponding collective triple.

Recall that we have fixed the context of our discussion with these general structures: a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a domain Val, a MiniMP model $\mathcal{M}$ and a program segment $l\colon C\ l'$ corresponding to $\mathcal{M}$.

In Fig. 5.2, the grammar only allows a path requirement or guarantee to have such a form: $\big($*Expr* ==>$\big)$? *Absent* && *Absent* && .... Hence a path requirement or guarantee can be represented by the conjunction of a set of absence assertions. The set contains all absence assertions connected by && in a clause, if *Expr* evaluates to **T**. Otherwise, the set is an empty.

Definition 5.16. Suppose the program segment $l\colon C\ l'$ is assigned a contract. A *collective triple* representing the program segment with the contract has such a form:

$$\langle \psi, \Gamma \rangle\ l\colon C\ l'\ \langle \phi, \Upsilon \rangle,$$

where

- $\psi, \phi \in \mathsf{SExpr}_{\mathcal{V}}$ are the state-requirement and -guarantee of C, respectively, and
- $\Gamma, \Upsilon \in \wp(\mathsf{SAbsent}_{\mathcal{V}})$ are the sets of absence assertions, the conjunctions of which represent the path-requirement and -guarantee, respectively.

□

When no ambiguity is possible in the context, and the location information is not needed, we may simplify $\langle \psi, \Gamma \rangle\ l\colon C\ l'\ \langle \phi, \Upsilon \rangle$ to $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$.

5.5.1 The Intuition in Collective Triple

Definition 5.17. The statement sequence C satisfies a contract iff the collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$, which represents C and the contract, is *valid*. □

We informally discuss about the meaning of that a triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ is valid in this subsection.

Intuitively, a MiniMP program can be considered as a "collective" statement sequence in that

1. An execution of the program starts from a pre-state of the body of the `main` procedure, and will end at a corresponding post-state of the body, if every process eventually terminates.
2. Every message that was sent by a process p when p was in the statement sequence will eventually be received by a process q when q is in the statement sequence.

Such a "collective" statement sequence C has two features that we are interested in. First, C does not depend on the environment to ensure its communication correctness. Second, the functional correctness of C can be expressed via asserting the inputs and outputs of C with respect to its pre- and post-states. Ideally, if a MiniMP program can be divided into a set of such "collective" statement sequences, we can apply Hoare logic, without much extension, to reason about the program in a composite and modular way. We use Example 5.18 to briefly illustrate the idea.

Example 5.18. Figure 5.7 shows a MiniMP program in pseudocode. In the program, a bulk-synchronous ***`barrier`*** divides the body of the `main` procedure into two "collective" statement sequences: one, C_0, is at line 3–9 and the other, C_1, is at line 10–11. Functional correctness of C_0 and C_1 can be specified using assertions over their inputs and outputs with respect to their pre- and post-states. The reasoning, informally, will be something like the following:

- For every process, C_0 ensures that `dat[9]` will be modified to have the same value as `dat[1]` of its `right` neighbor.
- For every process, C_1 ensures that `dat[0]` will be modified to have the same value as `dat[8]` of its `left` neighbor. Nothing else will be modified by C_1.
- Concluding from the above, the body of the `main` procedure ensures that, for every process, its `dat[0]` and `dat[9]` will be modified to have the same values as `dat[8]` and `dat[0]` of its `left` and `right` neighbors, respectively.

```
1  fun main() {
2    var dat, left, right;
3    dat = new [10];
4    left = (PID + NPROCS - 1) % NPROCS;
5    right = (PID + 1) % NPROCS;
6    f(PID, dat);
7    send dat[1] to left;
8    recv dat[9] from right;
9    barrier;
10   send dat[8] to right;
11   recv dat[0] from left;
12 }
```

Figure 5.7: A MiniMP program in pseudocode where f is a procedure that depends on `PID` and initializes `dat` elements. Suppose ***barrier*** is a bulk-synchronous barrier.

□

However, most of the MiniMP programs cannot be divided into multiple such ideal statement sequences. There are two reasons. First, bulk-synchronous ***barrier*** is not frequently used in real message-passing applications because of the pursuit of performance. So there is not necessarily a pair of pre- and post-states of some statement sequence in an execution of a program. Second, without ***barrier***s, the communication correctness of a statement sequence may depend on the environment. In such cases, one can hardly specify the communication correctness of a statement sequence by only constraining the inputs and outputs. We show why the second reason matters by revisiting Example 5.15.

Considering the contract (Fig. 5.6) of the statement S. If one reasons about S with respect to the contract only over the executions in state space graphs of S, S satisfies the contract. But it is insufficient. In fact, the existence of S may make a program invalid. The satisfaction of S to a contract must guarantee the validity of the use of S in any case as long as the requirement of the contract is met. Thus, S acutally fails to satisfy the contract. The communication correctness of S indeed depends on its context.

Imagine that there is an execution of a program containing S. Suppose there is

a suffix ρ of the execution that starts from a pre-state of S. On ρ, a non-`root` process p completes S and then sends a message again to the `root` process q before q done all the receiving in S. Since q receives messages with the wildcard `recv` statement, q can receive twice from p before completing S. Such an execution is undesired. Because it can possibly lead to a deadlock. More importantly, there is no way one can prove the satisfaction of S to its contract because the message coming from the outside of S is unspecified. Therefore, S must have an assumption on the contexts in order to ensure its communication correctness. In general, we call the situation, in which a process q receives a message sent by a another one p after p completes S, S *is interfered*. *Interference* is unwelcome. We discuss about this problem in the next subsection.

Path requirements and path guarantees are designed to address the interference problem with the idea of minimizing the communication detail reflected off a contract.

As aforementioned, the state space graphs of a program segment is not a sufficient scope for the problem of contract satisfaction. Therefore, we define a more general notion, *extended executions*.

Definition 5.19. Let T be the union of local transition sets of procedures in the model $\mathcal{M}$. Let $l\colon C\ l'$ be a program segment corresponding to $\mathcal{M}$. A global transition sequence $\zeta \in \rightsquigarrow^*$ is called an *extended execution* of $l\colon C\ l'$, if

1. ζ starts from a pre-state of C,
2. $\forall (s, p, \alpha, s') \in \mathsf{Tr}(\zeta).\, s' \in I(s, p, \alpha)\ \wedge\ \alpha \in T$, and
3. for every sub-path $t_i t_{i+1}$ of ζ, $\mathsf{dest}(t_i) = \mathsf{src}(t_{i+1})$.

□

We also call ζ is an extended execution of C when it is clear in the context that there is a unique program segment of C. An intuitive explanation for Def. 5.19 is that an extended execution of C is a path emanating from a pre-state of C, where processes may continue to execute after completing C.

With the notion of extended executions, we informally generalize the validity of a collective triple $\langle \psi,\, \Gamma \rangle\ C\ \langle \phi,\, \Upsilon \rangle$ to the following:

- for every execution π of C, if ψ and `allempty` hold on the initial pre-state of C for all processes, then ϕ and `allempty` will hold on the corresponding post-state of C for all processes, as long as every process completes C eventually, and
- for every same execution π as above, path predicates interpreted at the initial state of π for all processes are true for π, and
- for every extended execution ζ of C, where ψ and `allempty` hold on the initial pre-state s of C for all processes, if interference happens on ζ, ζ must violate at least one path predicate among those interpreted from elements in Γ at s for all processes.

We call the statement sequence C that is specified by a MiniSpec contract a *collective style* statement sequence. The name implies the intention that C shall be executed by all processes collectively and its communication correctness shall be independent with the environment as long as the requirement is met.

5.5.2 Interference

On an extended execution π of $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$, we call the situation, in which a process p sends a message after completing C and a process q receives the message before completing C, an *interference*. We also say that C is *interfered* by the statement following it.

Note that the symmetry case that a process in C receives a message sent by a process that has not entered C can be ignored. Suppose a MiniMP program is divided into a set of statement sequences $C_0 C_1 \ldots C_{m-1}$, if one proves that C_i can never be interfered by C_j, for $j > i$, one also proves the freedom of the symmetry case. If a process in C_i receives a message sent by a another process that has not entered C_i, there must be a statement C_k ahead of C_i (i.e., $k < i$) in which a message sent by a process in C_k is never received by a process in C_k. Without loss of generality, we can assume that C_k is the first statement in a program. C_k is suppose to satisfy a contract, according to the decomposition assumption. However, since there is a execution of C_k, on which a message sent by a process is never received by any process in C_k, the message channels cannot all be empty at the post-state of C_k. Therefore, there is no collective triple of C_k can be proved valid.

We now given the interference a precise description in terms of the number of messages that were sent and received by a process.

Definition 5.20. Given a path ρ. We use $\mathsf{nrecvs}_\rho(p, p')$ to denote the number of receive actions executed by process p' from process p in ρ. Dually, we use $\mathsf{nsends}_\rho(p, p')$ to denote the number of send actions executed by process p to process p' in ρ. Formally,

$$\mathsf{nrecvs}_\epsilon(p, p') = 0$$

$$\mathsf{nrecvs}_{\rho \circ t}(p, p') = \begin{cases} \mathsf{nrecvs}_\rho(p, p') + 1, \text{ if } \mathsf{proc}(t) = p' \wedge \mathsf{act}(t) = \mathsf{recv}(v,\ p) \text{ for some } v \\ \mathsf{nrecvs}_\rho(p, p'), \text{ otherwise} \end{cases}$$

$$\mathsf{nsends}_\epsilon(p, p') = 0$$

$$\mathsf{nsends}_{\rho \circ t}(p, p') = \begin{cases} \mathsf{nsends}_\rho(p, p') + 1, \text{ if } \mathsf{proc}(t) = p \wedge \mathsf{act}(t) = \mathsf{send}(v,\ p') \text{ for some } v \\ \mathsf{nsends}_\rho(p, p'), \text{ otherwise} \end{cases}$$

□

Definition 5.21. Given an extended execution ζ of the program segment $l\colon C\ l'$. Let ρ_p be a prefix of ζ such that $\mathsf{tail}(\rho_p) = \mathsf{exit}_p^C$. An *interference happens* on ζ if $\mathsf{nsends}_{\rho_p}(p, q) < \mathsf{nrecvs}_{\rho_q}(p, q)$, denoted as $\mathsf{interfere}_\zeta^C(p, q)$. □

5.5.3 The Validity of A Collective Triple

Finally, we give a precise definition for the validity of a collective triple.

Definition 5.22. For every execution π and extended execution ζ of C with n processes. Let $s = \mathsf{src}(\mathsf{head}(\pi))$, $s' = \mathsf{dest}(\mathsf{tail}(\pi))$ and $s'' = \mathsf{src}(\mathsf{head}(\zeta))$. Let $\langle \psi,\ \Gamma \rangle\ C\ \langle \phi,\ \Upsilon \rangle$ be a collective triple corresponding to the program segment. The triple $\langle \psi,\ \Gamma \rangle\ C\ \langle \phi,\ \Upsilon \rangle$ is said *deterministically valid*, denoted $\mathcal{M} \models^D \langle \psi,\ \Gamma \rangle\ C\ \langle \phi,\ \Upsilon \rangle$, iff

1. if

$$\forall p \in [n].\ [\![\psi \wedge \texttt{allempty}]\!]^{\mathsf{spec}}(s, p) = \mathbf{T},$$

and s' is a post-state of C corresponding to s, then

$$\forall p \in [n].\, [\![\phi \wedge \texttt{allempty}]\!]^{\mathsf{spec}}(s', p) = \mathbf{T},$$

and

2. $\pi \models \bigwedge_{\upsilon \in \Upsilon, p \in [n]} [\![\upsilon]\!]^{\mathsf{absent}}(s, p)$.

The triple is said *valid*, denoted $\mathcal{M} \models \langle \psi,\, \Gamma \rangle\ C\ \langle \phi,\, \Upsilon \rangle$, iff

1. $\mathcal{M} \models^{D} \langle \psi,\, \Gamma \rangle\ C\ \langle \phi,\, \Upsilon \rangle$, and
2. $\forall p, q \in [n].\ \mathsf{interfere}^{C}_{\zeta}(p, q) \Rightarrow \zeta \models \neg \bigwedge_{\gamma \in \Gamma, k \in [n]} [\![\gamma]\!]^{\mathsf{absent}}(s'', k)$.

□

Corollary 5.23. For the model $\mathcal{M}$, If $\mathcal{M} \models^{\Delta} \langle \psi,\, \Gamma \rangle\ C\ \langle \phi,\, \Upsilon \rangle$, and there is no wildcard receive ever used in C, we have $\mathcal{M} \models \langle \psi,\, \Gamma \rangle\ C\ \langle \phi,\, \Upsilon \rangle$. □

Chapter 6

A INFERENCE SYSTEM FOR MINIMP

Let the context of our discussion in this chapter be fixed with a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a domian Val and a MiniMP model $\mathcal{M}$. Given a procedure $f \in \mathsf{Procedure}$, we use $\mathsf{body}(f)$ to denote the body statement of f.

The partial correctness of a MiniMP program can be specified as a collective triple $\langle \psi, \Gamma \rangle\ \mathsf{body}(\texttt{main})\ \langle \phi, \Upsilon \rangle$. To prove the validity of this triple in a composite and modular way, we introduce an inference system, denoted $\mathcal{R}$, in this chapter. By applying $\mathcal{R}$ to $\langle \psi, \Gamma \rangle\ \mathsf{body}(\texttt{main})\ \langle \phi, \Upsilon \rangle$, the validity of the triple can be proved by proving a set of collective triples, each of which specifies a smaller statement than $\mathsf{body}(\texttt{main})$.

The inference system $\mathcal{R}$ is defined with a set of rules in §6.1. §6.2 discusses about the *adaptation problem* arising in practice of the rules. We show a derivation example of $\mathcal{R}$ in §6.3. Finally, we define notations in §6.4, which are used in the soundness proof of $\mathcal{R}$ in §6.5.

6.1 Rules for Decomposing Collective Triples

In this section, we introduce an inference system for composite and modular verification of MiniMP programs. We use $\mathcal{R}$ to denote this inference system. $\mathcal{R}$ consists of four rules for reasoning about collective triples. With $\mathcal{R}$, one is able to decompose a collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ into a set of collective triples Tri such that $\mathcal{M} \models \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$, if $\forall \mathsf{tri} \in \mathsf{Tri}.\, \mathcal{M} \models \mathsf{tri}$. Normally, proving the validity of a $\mathsf{tri} \in \mathsf{Tri}$ is a much smaller problem than proving the validity of the original triple.

The rules in $\mathcal{R}$ involve assertions that will be explained below. These assertions, as well as the rules, constraint absence assertions in terms of mathematical set operations. These operations shall be interpreted under a certain context—a path, where there is exact one state for evaluating an absence assertion involved in an operation.

Definition 6.1. A *partial evaluation* $\mathsf{pe}\colon \mathsf{SAbsent}_{\mathcal{V}} \times \mathsf{State} \times [n] \to \mathsf{SAbsent}_{\mathcal{V}}$ is a function that maps an absence assertion to another absence assertion at a state for a process. Given an absence assertion λ, a state s and a process p, any expression appearing in $\mathsf{pe}(\lambda, s, p)$ must be a constant expression. We define $\mathsf{pe}(\lambda, s, p)$ as below. For brevity, we do not show the full definition. The omitted definition should be obvious to readers.

$$
\begin{aligned}
\mathsf{pe}(\lambda, s, p) = \ & \text{let } p = [\![e_0]\!](s, p) \text{ in} \\
& \text{let } q = [\![e_1]\!](s, p) \text{ in} \\
& \text{let } r = [\![e_2]\!](s, p) \text{ in} \\
& \texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r), \\
& \text{if } \lambda = \texttt{\textbackslash no \textbackslash send}(e_0,\ e_1)\ \texttt{\textbackslash after \textbackslash exit}(e_0)\ \texttt{\textbackslash until \textbackslash exit}(e_2) \\
& \ldots
\end{aligned}
$$

□

Definition 6.2. Let λ be an absence assertion, Λ_0, Λ_1 be two sets of absence assertions. Given a path ρ. Let $s, s_0, s_1 \in \mathsf{St}(\rho)$ be the only states, where λ, Λ_0 and Λ_1 shall be evaluated, respectivey. We define

$$\lambda \in_\pi \Lambda_0 \text{ iff } \exists \lambda' \in \Lambda_0 \exists p, q \in [n].\ \mathsf{pe}(\lambda, s, p) = \mathsf{pe}(\lambda', s_0, q)$$

$$\Lambda_0 \subseteq_\pi \Lambda_1 \text{ iff } \forall \lambda_0 \in \Lambda_0 \exists \lambda_1 \in \Lambda_1 \exists p, q \in [n].\ \mathsf{pe}(\lambda_0, s_0, p) = \mathsf{pe}(\lambda_1, s_1, q)$$

□

Now we describe the assertions. Let C_0 and C_1 be two MiniMP statement sequences. Given three triples $\langle \psi,\ \Gamma \rangle\ C_0 C_1\ \langle \phi,\ \Upsilon \rangle$, $\langle \psi_0,\ \Gamma_0 \rangle\ C_0\ \langle \phi_0,\ \Upsilon_0 \rangle$ and $\langle \psi_1,\ \Gamma_1 \rangle\ C_1\ \langle \phi_1,\ \Upsilon_1 \rangle$.

Let $\rho = \pi_0 \circ \rho_1$ be a path such that π_0 is an execution of C_0 and ρ_1 is a prefix of an execution of C_1. Obviously, according to the semantics of collective triples defined in Def. 5.22, $\Gamma, \Upsilon, \Gamma_0, \Upsilon_0$ shall be paritally evaluated at $\mathsf{src}(\mathsf{head}(\rho))$, and Γ_1, Υ_1 shall be partially evaluated at $\mathsf{src}(\mathsf{head}(\rho_1))$.

- Υ_1 $\mathsf{guarantees}_\rho$ Γ_0 iff for every

  $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\rho \Gamma_0,$$

  there is

  $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\rho \Upsilon_1.$$

- $\mathsf{noSend}(\Upsilon)$ denotes the maximum subset of Υ that only contains the absence assertions of the form:

  $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r).$$

- Υ_0 and Υ_1 infer_ρ $\mathsf{noSend}(\Upsilon)$ iff for every

  $$\upsilon = \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\rho \mathsf{noSend}(\Upsilon),$$

  we have $\upsilon \in_\rho \Upsilon_0$, and

  - $\texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\rho \Upsilon_0$, or
  - $\upsilon[x/r] \in_\rho \Upsilon_1$, if there is such $x \in [n]$ that
    $\texttt{\textbackslash no \textbackslash exit}(x)\ \texttt{\textbackslash after \textbackslash enter}(x)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\rho \Upsilon_0$, or
  - $\upsilon \in_\rho \Upsilon_1 \ \vee\ \upsilon[q/r] \in_\rho \Upsilon_1$.

  We explain this assertion. An element $\upsilon \in_\rho \mathsf{noSend}(\Upsilon)$ asserts that process p shall not send a message to process q during in C_0C_1 before process r entering C_0C_1. We have Υ_0 and Υ_1 of C_0 and C_1, respectively, infer υ, if

  1 Υ_0 also contains υ. If so, p cannot send a message to q during in C_0 before r entering C_0.

  2 Moreover, p shall also not send a message to q during in C_1 before r entering C_0. Hence, one of the following conditions must hold.

  2.1 C_0 guarantees that the sender p will not enter the following C_1 until r enters itself;

  2.2 If C_0 guarantees that some process x will not enter the following C_1 until r enters itself, C_1 shall guarantee that p will not send a message to q until x enters itself.

2.3 C_1 guarantees that process p will not send a message to q until process q or r enters itself.

- Let

$$\gamma = \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\rho \Gamma_0,$$

we say Υ_1 $\mathsf{cancels}_\rho$ γ iff

$$\texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \in_\rho \Upsilon_1,$$

where $r' = r \vee r' = q$. The intuitive meaning for the $\mathsf{cancels}$ assertion is that for a $\gamma \in_\rho \Gamma_0$, if Υ_1 $\mathsf{cancels}_\rho$ γ, C_0C_1 no longer requires γ.

Rules in Hoare logic is defined with respect to kinds of program statements, such as branch or loop. Here we define a premise of a rule in $\mathcal{R}$ that depends on the semantics instead of the structure of MiniMP programs. We call this premise the "elaboration condition" because the complexity of proving this premise for a statement sequence C will be no less than elaborating all executions in state space graphs of C.

Definition 6.3. Given two statement sequences C and C'. We say C' *contains behaviors of* C *under* ψ, denoted $\langle \psi, \emptyset \rangle\ \overrightarrow{C} \subseteq \overrightarrow{C'}$, if for every extended execution ζ of C such that ψ holds on $\mathsf{src}(\mathsf{head}(\zeta))$ for all processes, there is an *equivalent* extended execution ζ' of C'. The equivalence of paths is defined in Def. 6.4. □

Definition 6.4. Given two paths ρ_0 and ρ_1. Let ξ_0 and ξ_1 be two transition sequences that are the projections of ρ_0 and ρ_1, respectively, onto transitions associated with non-`skip` actions. We say ρ_0 and ρ_1 are *equivalent* if $\xi_0 = \xi_1$. □

Now we define the rules in $\mathcal{R}$.

Definition 6.5. The *inference system* $\mathcal{R}$ for collective triples is defined in Fig. 6.1. □

The *sequence*, *consequence* and *collective loop* rules in $\mathcal{R}$ were inspired by the *sequence*, *consequence* and *loop* rules in Hoare logic. Rules in $\mathcal{R}$ over state-requirements

$$\frac{\langle \psi,\, \Gamma_0\rangle\ C_0\ \langle \mu,\, \Upsilon_0\rangle,\ \langle \mu,\, \Gamma_1\rangle\ C_1\ \langle \phi,\, \Upsilon_1\rangle}{\langle \psi,\, \Gamma\rangle\ C_0C_1\ \langle \phi,\, \Upsilon\rangle} \quad \textit{(sequence)}$$

if for every execution $\pi_0 \circ \pi_1$ of C_0C_1 s.t.
π_0 is an execution of C_0 and π_1 is an execution of C_1,

1. Υ_1 $\mathsf{guarantees}_{\pi_0\circ\pi_1}$ Γ_0,
2. Υ_0 and Υ_1 $\mathsf{infer}_{\pi_0\circ\pi_1}$ $\mathsf{noSend}(\Upsilon)$,
3. $\Upsilon \subseteq_{\pi_0\circ\pi_1} \Upsilon_0 \cup \Upsilon_1$, and
4. $\{\gamma \in \Gamma_0 \mid \neg(\Upsilon_1\ \mathsf{cancels}_{\pi_0\circ\pi_1}\ \gamma)\} \cup \Gamma_1\ \subseteq_{\pi_0\circ\pi_1}\ \Gamma$.

$$\frac{\langle \psi', \Gamma'\rangle\ C\ \langle \phi', \Upsilon'\rangle}{\langle \psi, \Gamma\rangle\ C\ \langle \phi, \Upsilon\rangle} \quad \textit{(consequence)}$$

if for every execution π of C, $\Gamma' \subseteq_\pi \Gamma\ \wedge\ \Upsilon \subseteq_\pi \Upsilon'\ \wedge\ \psi \Rightarrow \psi'\ \wedge\ \phi' \Rightarrow \phi$

$$\frac{\langle \mathit{inv} \wedge c,\, \Gamma\rangle\ C\ \langle \mathit{inv},\, \Upsilon\rangle}{\langle \mathit{inv},\, \Gamma\rangle\ \texttt{while(}c\texttt{)}\ C\ \langle \neg c \wedge \mathit{inv},\, \Upsilon\rangle} \quad \textit{(collective loop)}$$

if for every execution π of C, Υ $\mathsf{guarantees}_\pi$ Γ

$$\frac{\langle \psi,\, \emptyset\rangle\ \overrightarrow{C} \subseteq \overrightarrow{C_0C_1}\ ,\ \langle \psi,\, \Gamma\rangle\ C_0C_1\ \langle \phi,\, \Upsilon\rangle}{\langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle} \quad \textit{(elaboration)}$$

Figure 6.1: Rules for decomposing collective triples

and -guarantees are similar to pre- and post-conditions in Hoare logic. Rules over path-requirements and -guarantees are explained by the follows.

The *sequence* rule is the basis of the decomposition. For a a sequence of two statement sequences C_0C_1, in order to prove $\mathcal{M} \models \langle \psi,\, \Gamma\rangle\ C_0C_1\ \langle \phi,\, \Upsilon\rangle$, one proves $\mathcal{M} \models \langle \psi,\, \Gamma_0\rangle\ C_0\ \langle \mu,\, \Upsilon_0\rangle$ and $\mathcal{M} \models \langle \mu,\, \Gamma_1\rangle\ C_1\ \langle \phi,\, \Upsilon_1\rangle$ separately instead. The decomposition is feasible if all the side conditions are satisfied.

Side Condition 1 enforces that C_0 will never be interfered by C_1. The meaning of Side Condition 2 has been explained with the "... and ... infer ..." assertion. Side

Condition 3 expresses that the path guarantees of C_0 and C_1 together shall be no weaker than the one of C_0C_1. Stated by Side Condition 4, since Υ_1 may cancels some elements in Γ_0, the path requirement of C_0 and C_1 together may be weaker but shall not be stronger than the one of C_0C_1.

The *consequence* rule is more obvious: if a collective triple is valid, it is still valid after strengthening its state- or path-requirement, and it is still valid after weakening its state- or path-guarantee.

The *collective loop* rule requires the user to provide a loop invariant *inv*. This rule can only be applied to the loops whose body C can be specified by a collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$. In other words, the loop body must be collective style. This rule can be seen as a sequence of applications of the *sequence* rule to the unrolled loop body $CC\ldots$.

The *elaboration* rule is designed for the cases where a collective triple cannot be decomposed by the three rules above due to the existence of branches, such as the following statement C (at line 3–6):

```
1 var dat;
2 ...
3 if (PID == 0)
4 {bcast(dat, 0); gather(dat, 0);}
5 else
6 {bcast(dat, 0); gather(dat, 0);}
```

where `bcast` and `gather` are procedures defined in Fig. 4.2 and 4.3, respectively. One can tell that the code above is equivalent to such a sequence of calls

```
bcast(dat, PID); gather(dat, PID);.
```

However, the sequence rule is not suitable for the original branch. With the elaboration rule, if one can prove the "elaboration condition",

$$\langle \psi, \emptyset \rangle\ \overrightarrow{C} \subseteq \overrightarrow{\texttt{bcast(dat, PID);gather(dat, PID);}}$$

for some condition ψ, is valid, one can infer a collective triple of the original branch from a triple of the sequence of calls. The later triple can be further decomposed by

an application of the sequence rule. Proving the elaboration condition requires some other techniques, such as model checking. It is out of the scope of this chapter.

Concluding this section, $\mathcal{R}$ is incomplete. It means that it is not always the case that a collective triple can be derived by $\mathcal{R}$. As we argued in Chapter 1, the effectiveness of $\mathcal{R}$ relies on the collective programming style, which dominates the message-passing HPC programming.

6.2 The Adaptation Problem

There is an *adaptation problem* that arises from re-using verified collective triples. Example 6.6 shows such a problem.

Example 6.6. Let C be the left and C' be the right statements below:

```
1 if (PID == 0) {
2   send(x, y);
3 } else if (PID == 1) {
4   recv(x, 0);
5 }
```

```
1 if (PID == 0) {
2   send(x, z);
3 } else if (PID == 1) {
4   recv(x, 0);
5 }
```

The only difference between C and C' is that variable `y` in C is replaced by `z` in C'. Now suppose that we have verified a collective triple of C, $\langle \mathtt{y} = 1, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$, and is going to verify a collective triple of C', $\langle \mathtt{z} = 1, \Gamma \rangle\ C'\ \langle \phi, \Upsilon \rangle$. Intuitively, the validity of the triple of C' can be concluded from the former one. However, no rule in $\mathcal{R}$ can be applied here to re-use the verified triple of C without a variable name substitution from `y` to `z` happens first. □

To generalize the solution to the problem of Example 6.6, one shall allow free substitution from existing variable names to fresh new variable names over collective triples. Such substitutions are common transformations in first order logic that cannot affect the validity of collective triples. In this dissertation, we will not discuss about details in solving adaptation problem for two reasons.

First, the adaptation problem has been well studied and tackled by earlier works [50, 64]. There is nothing new in this problem for our unique inference system for MiniMP.

More importantly, in later chapters, we will focus on applying collective triples to procedures only. This is one of the typical ways to implement a contract based verification system. For MiniMP procedures, the adaptation problem is naturally solved. Recall the semantics of MiniMP in §4.3, a mapping from actual parameters to formal parameters happens automatically. In addition, global variables are visible everywhere in a program. No substitution is ever needed for global variables.

6.3 An Derivation Example

We now show an example of the application of $\mathcal{R}$.

Figure 6.2 shows a contract for the sequential combination $C = C_0C_1$ of two statements: C_0 and its contract given in Fig. 5.5, and C_1 and its contract given in Fig. 5.6. Recall that we have explained in §5.5 that the contract of C_1 misses a path requirement.

Figure 6.3 shows a $\mathcal{R}$ derivation of the collective triple representing C and its contract. In the derivation, we write id for the value of `PID`, n for the value of `NPROCS`, v for the value of `val`, and r for the value of `root`. We write x_p for a variable or value x on process p. Let $X_p, Y_p, Z_p \in \mathsf{Val}$ be values of `dat` on a process $p \in [n]$ at different states. In this derivation, irrelavant conditions are simplified, e.g., we ignored the values of variables `root` and `val` in the state-guarantee of Triple 1.

At a generic pre-state of C, variable `dat` on process i holds value X_i. We simplified the condition that all processes have an agreement on their values of `root` by letting all processes have the same value r for `root`. Note the state-guarantee in Triple 1 involves the use of the `\old` construct to refer the value X_i of `dat` at the pre-state on process i. Triple 1 represents C and its contract. It can be proved valid if Triple 1.1 and 1.2 are valid, according to the sequence rule.

Triple 1.1 relates to the contract of C_0. Note we added the path requirement here that was missing in Fig. 5.6. Besides, since `dat` is modified by C_0, the value of `dat` at the post-state of C_0 needs to be represented by a different symbol Y. According

```
1 var dat, val, root, i, y;
2 ...
3 // contract:
4 requires \forall t; \on(root, t) == root;
5 requires 0 <= root && root < NPROCS && val == dat[PID];
6 assigns dat, i, y;
7 ensures \forall t; 0 <= t && t < NPROCS ==>
8           dat[t] == \old(\on(dat[t], t));
9 ensures \no \exit(PID) \after \enter(PID) \until \enter(root);
```

Figure 6.2: A contract for the sequential combination of the two branch statements given by Fig. 5.6 and 5.5.

to the contract of C_0, elements in `dat` on process `root` are "gathered" from other processes, i.e.,

$$\forall t \in [n]. Y_r = X_t[t].$$

Triple 1.2 relates to the contract of C_1. In addition to the contract showed in Fig. 5.5, we added an extra path guarantee

$$\texttt{\textbackslash no \textbackslash send(id, root) \textbackslash after \textbackslash enter(id) \textbackslash until \textbackslash enter(root)}.$$

Adding such a path guarantee strengthens the guarantee of the original contract. It is not hard for the reader to find out that C_1 still satisfies the strengthened contract. By the consequence rule, if the strengthened the triple is valid, the original triple of C_1 is also valid. The extra path guarantee is required in meeting Side Condition 1 of the sequence rule. The rest of the side conditions are valid with obviousness. At the post-state of C_1, the values of `dat` are represented with the symbol Z for the aforementioned reason. The state-guarantee can be further simplified to be exact the same as the state-guarantee in Triple 1.

6.4 Extending the Entering/Exiting Notation

In §5.1, we defined two notations $\mathsf{enter}^{C}_{p,\rho}$ and $\mathsf{exit}^{C}_{p,\rho}$ for marking the first entering and the corresponding exiting of a statement sequence C by a process p on a path ρ. Here we extend such notations for a sequence of statement sequences $C_0 C_1 \ldots C_{m-1}$.

1. $\langle\ \mathsf{dat}_{\mathsf{id}} = X_{\mathsf{id}} \wedge \mathtt{root}_{\mathsf{id}} = r \wedge \mathtt{val}_{\mathsf{id}} = v_{\mathsf{id}} \wedge$ (seq 1.1, 1.2)
$X_{\mathsf{id}}[\mathsf{id}] = v_{\mathsf{id}} \wedge 0 \leq r < n,\ \emptyset$
$\rangle$
$C_0 C_1$
$\langle\ \forall t \in [n].\, Z_{\mathsf{id}}[t] = X_t[t]\ \wedge\ \mathsf{dat}_{\mathsf{id}} = Z_{\mathsf{id}},$
$\{$**\no \exit**(id) **\after \enter**(id) **\until \enter**$(r)\}$
$\rangle$

1.1. $\langle\ \mathsf{dat}_{\mathsf{id}} = X_{\mathsf{id}} \wedge \mathtt{root}_{\mathsf{id}} = r \wedge \mathtt{val}_{\mathsf{id}} = v_{\mathsf{id}} \wedge X_{\mathsf{id}}[\mathsf{id}] = v_{\mathsf{id}} \wedge 0 \leq r < n,$
$\{$**\no \send**(id, r) **\after \exit**(id) **\until \exit**$(r)\}$
$\rangle$
C_0
$\langle\ (\forall t \in [n].\, Y_r[t] = X_t[t]) \wedge \mathsf{dat}_{\mathsf{id}} = Y_{\mathsf{id}} \wedge \mathtt{root}_{\mathsf{id}} = r \wedge 0 \leq r < n,\ \emptyset\ \rangle$

1.2. $\langle\ (\forall t \in [n].\, Y_r[t] = X_t[t]) \wedge \mathsf{dat}_{\mathsf{id}} = Y_{\mathsf{id}} \wedge \mathtt{root}_{\mathsf{id}} = r \wedge 0 \leq r < n,\ \emptyset\ \rangle$
C_1
$\langle\ (\forall t \in [n].\, Y_r[t] = X_t[t])\ \wedge\ \mathsf{dat}_{\mathsf{id}} = Z_{\mathsf{id}}\ \wedge\ Z_{\mathsf{id}} = Z_r\ \wedge\ Z_r = Y_r,$
$\{$**\no \send**(id, r) **\after \enter**(id) **\until \enter**(r),
\no \exit(id) **\after \enter**(id) **\until \enter**$(r)\}$
$\rangle$

Figure 6.3: A $\mathcal{R}$ derivation of a collective triple that specifies a sequential combination of two collective style MiniMP statements.

Definition 6.7. Given a program segment $l_0 \colon C_0\ l_1 \colon C_1\ l_2 \ldots l_{m-1} \colon C_{m-1}\ l_m$ and a path ρ. Let C be $C_0 C_1 \ldots C_{m-1}$. Let $t \in \mathsf{Tr}(\rho)$ be a global transition of the path ρ. We define $\mathsf{enter}^{C_i}_{p,\rho}$ and $\mathsf{exit}^{C_i}_{p,\rho}$ for $0 \leq i < m$ by

- $\mathsf{enter}^{C_0}_{p,\rho} = \mathsf{enter}^{C}_{p,\rho}$,
- $t = \mathsf{enter}^{C_i}_{p,\rho}$, for $i > 0$, if t is the first such transition appearing after $\mathsf{exit}^{C_{i-1}}_{p,\rho}$ on ρ that
$$\mathsf{toploc}(\mathsf{src}(t), p) = l_i\ \wedge\ \mathsf{proc}(t) = p$$
- $t = \mathsf{exit}^{C_i}_{p,\rho}$, for $i \geq 0$, if t is the first such transition appearing after $\mathsf{enter}^{C_i}_{p,\rho}$ on ρ that

$$\mathsf{toploc}(\mathsf{dest}(t), p) = l_{i+1} \wedge \mathsf{proc}(t) = p \wedge |\mathsf{stack}(\mathsf{dest}(t), p)| = |\mathsf{stack}(\mathsf{src}(\mathsf{enter}^{C_i}_{p,\rho}), p)|$$

□

Intuitively, $\mathsf{enter}_{p,\rho}^{C_i}$ and $\mathsf{exit}_{p,\rho}^{C_i}$ represent the entering and exiting, respectively, of each sub-statement C_i of C by the process p on ρ. Again, when ρ is clear in the context, we may simplify the notations to $\mathsf{enter}_{p}^{C_i}$ and $\mathsf{exit}_{p}^{C_i}$.

Remark that, for an execution or an extended execution with n processes of $C_0C_1 \dots C_{m-1}$, there is a unique $\mathsf{enter}_{p}^{C_i}$, or $\mathsf{exit}_{p}^{C_i}$, for $p \in [n], 0 \leq i < m$.

6.5 Soundness

This section presents a proof for the soundness of $\mathcal{R}$.

Definition 6.8. Given a path $t \circ t'$. Let $\mathsf{act}(t) = p \wedge \mathsf{act}(t') = q$ such that $p \neq q$. The actions of t and t' are said to be *swappable* if it is not the case that

$$\mathsf{act}(t) = \mathsf{send}(v,\ q)\ \wedge\ \mathsf{act}(t') = \mathsf{recv}(v',\ p)\ \wedge\ \mathsf{chan}(\mathsf{src}(t))(p, q) = \epsilon$$

□

For convenience, we also say two global transitions t and t' are swappable if their actions are swappable. Intuitively, t and t' are swappable if they are associated with different processes and they are independent, i.e., $\mathsf{proc}(t)$ can execute $\mathsf{act}(t)$ regardless of t' being executed first or not, and vice versa; t and t' cannot access the process state of a same process or the same message in a message channel.

Lemma 6.9. Let $s \xrightarrow[p]{a} s'$ represent a global transition $t = (s, p, \alpha, s')$ such that $\mathsf{act}(t) = a$. A path then can be written as $\dots s_0 \xrightarrow[p]{a} s_1 \xrightarrow[q]{b} s_2 \dots$. For $s_0 \xrightarrow[p]{a} s_1 \xrightarrow[q]{b} s_2$, if the two global transitions are swappable, we have

$$s_0 \xrightarrow[q]{b} s_1' \xrightarrow[p]{a} s_2$$

Proof A proof can be constructed by induction on atomic actions of the two global transitions. We omit this proof for two reasons (1) it is in fact a more specific case of the general *reduction* approach proposed by Lipton in [75]; and (2) the lemma is

intuitive and is well-known in the field of verification of message-passing programs. □

Given a path ρ: $s_0 \xrightarrow[p]{a} s_1 \xrightarrow[q]{b} s_2$ with two swappable transitions. The path $s_0 \xrightarrow[q]{b} s_1' \xrightarrow[p]{a} s_2$ is said to be obtained via *swapping transitions* of ρ. And, by Lemma 6.9, we know these two paths share their initial and the final states.

Lemma 6.10. Given a path ρ in the state space graph of C_0C_1 with n processes. If C_1 does not interfere C_0, then for every sub-path $t \circ t'$ of ρ such that (1) $\mathsf{proc}(t) \neq \mathsf{proc}(t')$, and (2) t appears before $\mathsf{exit}^{C_0}_{\mathsf{proc}(t),\rho}$ and t' appears after $\mathsf{exit}^{C_0}_{\mathsf{proc}(t'),\rho}$ on ρ, formally,

$$\exists i, j, x, y \in \mathbb{Z}.\ i < j \ \wedge\ x < y \ \wedge\ t = \rho[i] \ \wedge\ t' = \rho[y] \ \wedge$$
$$\mathsf{exit}^{C_0}_{\mathsf{proc}(t),\rho} = \rho[j] \wedge \mathsf{exit}^{C_0}_{\mathsf{proc}(t'),\rho} = \rho[x],$$

t and t' are swappable.

Proof We prove by contradiction. Suppose that there is a sub-path $t \circ t'$ of ρ satisfying Condition 1 and 2 given in the lemma above, but t, t' are not swappable. Let $\mathsf{proc}(t) = p$, $\mathsf{proc}(t') = q$, and ρ_p, ρ_q be two prefixes of ρ such that ρ_p ends with $\mathsf{exit}^{C_0}_p$ and ρ_q ends with $\mathsf{exit}^{C_0}_q$, respectively. Since C_0 is not interfered by C_1 on ρ,

$$\mathsf{nsends}_{\rho_p}(p,q) = \mathsf{nrecvs}_{\rho_q}(p,q) \ \wedge\ \mathsf{nsends}_{\rho_q}(q,p) = \mathsf{nrecvs}_{\rho_p}(q,p) \tag{3}$$

Since t and t' are not swappable, $\mathsf{chan}(\mathsf{src}(t))(p,q) = \epsilon$. Let ρ_t be the prefix of ρ_p that ends with t. We have

$$\mathsf{nrecvs}_{\rho_t}(p,q) = \mathsf{nsends}_{\rho_t}(p,q). \tag{4}$$

Since $\mathsf{act}(t) = \mathsf{send}(\mathsf{p}, \mathsf{q})$,

$$\mathsf{nsends}_{\rho_p}(p,q) > \mathsf{nsends}_{\rho_t}(p,q). \tag{5}$$

Simplifying (3), (4), (5) to

$$\mathsf{nrecvs}_{\rho_q}(p,q) > \mathsf{nrecvs}_{\rho_t}(p,q). \tag{6}$$

However, by our assumption, $t \circ t'$ appears after $\mathsf{exit}_q^{C_0}$ on ρ, we have

$$\mathsf{nrecvs}_{\rho_q}(p,q) \leq \mathsf{nrecvs}_{\rho_t}(p,q). \tag{7}$$

There is a contradiction between (6) and (7). $\square$

Corollary 6.11. Given an execution π in a state space graph of C_0C_1 that ends with the termination of all processes. If C_1 cannot interfere C_0, Lemma 6.10 allows one to keep swapping swappable transitions on π to produce an alternative path π' in the same state space graph such that

1. π and π' share their initial and the final states.
2. $\pi' = \pi_0 \circ \pi_1$, where π_0 is an execution of C_0 and π_1 is an execution of C_1.

$\square$

Lemma 6.12. Given an extended execution ζ of C_0C_1 with n processes. Let P be a set of processes such that

$$p \in P \text{ iff } \mathsf{exit}_{p,\zeta}^{C_0} \in \mathsf{Tr}(\zeta).$$

Now if

$$\forall q \in [n].\, \forall p \in P.\ \mathsf{chan}(\mathsf{dest}(\mathsf{exit}_{p,\zeta}^{C_0}))(q,p) = \epsilon,$$

then there is another extended execution ζ' of C_0C_1 with n processes such that $\zeta' = \pi_0' \circ \zeta_1'$, where

1. π_0' is an execution of C_0 with n processes,
2. ζ_1' is an extended execution of C_1 with n processes,
3. $\forall t \in \mathsf{Tr}(\zeta_1').\, \mathsf{proc}(t) \in P$, and
4. every global transition $t \in \mathsf{Tr}(\zeta_1')$ corresponds to a unique one appearing after $\mathsf{exit}_{\mathsf{proc}(t),\zeta}^{C_0}$ on ζ, formally,

$$\begin{aligned}
&\text{let } N = \{i \in \mathbb{Z} \mid 0 \leq i < |\zeta|\} \text{ in} \\
&\forall t' \in \mathsf{Tr}(\zeta_1').\, \exists t \in \mathsf{Tr}(\zeta).\, \exists i,j \in N.\ i < j \ \wedge\ \zeta[i] = \mathsf{exit}_{\mathsf{proc}(t)}^{C_0} \ \wedge \\
&\qquad\qquad \zeta[j] = t \ \wedge\ \mathsf{act}(t) = \mathsf{act}(t') \ \wedge\ \mathsf{proc}(t) = \mathsf{proc}(t').
\end{aligned}$$

Proof Considering the same path ζ and process set P in this lemma. By Lemma 6.9 and 6.10, we have an alternative path ρ to ζ with n processes such that

1. ρ shares the initial state with ζ
2. $\rho = \rho_0 \circ \rho_1$ such that

$$p \in P \text{ iff } \mathsf{exit}^{C_0}_{p,\rho} \in \mathsf{Tr}(\rho_0) \;\text{ and }\; p \in P \text{ iff } \mathsf{enter}^{C_1}_{p,\rho} \in \mathsf{Tr}(\rho_1) \tag{2.1}$$

Condition 2.1 above implies that (1) all processes in P are at the exit of C_0 (or entry of C_1) at the final state of ρ_0 while other processes are still in C_0, and (2) ρ_1 contains no transition associated with processes not in P.

Furthermore, according to the assumption over message channels, we have

$$\forall q \in [n].\, \forall p \in P.\, \mathsf{chan}(\mathsf{src}(\mathsf{head}(\rho_1)))(q, p) = \epsilon.$$

Therefore, no transition in ρ_1 depends on any process in $[n] \backslash P$ or the message channels from $[n] \backslash P$ to P at the initial state of ρ_1.

Now we first construct the execution π'_0 of C_0 to $\pi'_0 = \rho_0 \circ \rho'_0$ via letting all processes in $[n] \backslash P$ continue to execute from $\mathsf{dest}(\mathsf{tail}(\rho_0))$ until they are all at the exit of C_0. Process states of processes in P stay the same along ρ'_0 because none of them ever executes. Note we are not interested in non-termination problem here.

Second, we construct the ζ' to $\zeta' = \pi'_0 \circ \zeta'_1$ via making $|\rho_1| = |\zeta'_1|$ and

$$\forall i \in \mathbb{Z}.\, 0 \leq i < |\rho_1| \Rightarrow \mathsf{proc}(\rho_1[i]) = \mathsf{proc}(\zeta'_1[i]) \;\wedge\; \mathsf{act}(\rho_1[i]) = \mathsf{act}(\zeta'_1[i]).$$

Although there is no guarantee that $\mathsf{chan}(\mathsf{dest}(\mathsf{tail}(\pi'_0))(q, p) = \epsilon$ for some $q \in [n]$ and $p \in P$, transitions in ρ_1 only depend on the process states of processes in P. So the construction is feasible.

Remark that transitions in ρ_1 are obtained by swapping transitions in ζ, so Condition 4 in this lemma is naturally implied. $\square$

Theorem 6.13 (soundness of the sequence rule)**.** Assuming all the side conditions of the sequence rule hold,

$$\mathcal{M} \models \langle \psi, \Gamma \rangle\ C_0 C_1\ \langle \phi, \Upsilon \rangle, \text{ if}$$
$$\mathcal{M} \models \langle \psi, \Gamma_0 \rangle\ C_0\ \langle \mu, \Upsilon_0 \rangle \text{ and } \mathcal{M} \models \langle \mu, \Gamma_1 \rangle\ C_1\ \langle \phi, \Upsilon_1 \rangle.$$

Proof The proof of Theorem 6.13 is carried out by Lemma 6.14, 6.15, 6.16 and 6.17. □

Lemma 6.14. Under the assumption made in Theorem 6.13, for an execution π in the state space graph of C_0C_1 with n processes, if ψ and $\mathsf{allempty}$ hold on the initial state for all processes, C_1 does not interfere C_0 on π.

Proof We prove by contradiction. Assuming $\mathsf{interfere}_{\pi}^{C_0}(p, q)$ for an ordered pair of unique processes $p, q \in [n]$. Without loss of generality, we also assume that

(a) (p, q) is the only such ordered pair of processes that $\mathsf{interfere}_{\pi}^{C_0}(p, q)$, and

(b) let ρ_p and ρ_q be the prefixes of π that end with $\mathsf{exit}_{p,\pi}^{C_0}$ and $\mathsf{exit}_{q,\pi}^{C_0}$, respectively, $\mathsf{nsends}_{\rho_p}(p, q) + 1 = \mathsf{nrecvs}_{\rho_q}(p, q)$.

Then π must have such a form: $\ldots \circ t_p \circ \ \ldots \circ t_q \circ \ldots$, where

1. $\mathsf{proc}(t_p) = p$, $\mathsf{act}(t_p) = \mathsf{send}(v, q)$, $\mathsf{proc}(t_q) = q$ and $\mathsf{act}(t_q) = \mathsf{recv}(v', p)$, for some v and v', and
2. $\mathsf{chan}(\mathsf{src}(t_p))(p, q) = \epsilon$, and
3. t_p appears after $\mathsf{exit}_{p,\pi}^{C_0}$ and t_q appears before $\mathsf{exit}_{q,\pi}^{C_0}$ on π.

Since C_0 is interfered by C_1 as above, and $\mathcal{M} \models \langle \psi, \Gamma_0 \rangle\ C_0\ \langle \mu, \Upsilon_0 \rangle$, it must be the case that

$$\texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_{\pi} \Gamma_0,$$

for some processes $r \in [n]$.

Considering the prefix ρ of π that ends with t_p. Let P be the set of processes such that $x \in P$ iff $\mathsf{exit}_x^{C_0}$ is in $\mathsf{Tr}(\rho)$. Obviously $p \in P$ and $q, r \notin P$. By the assumption (a) above, we have

$$\forall y \in [n] \forall x \in P.\, \mathsf{chan}(\mathsf{src}(\mathsf{head}(\mathsf{exit}_x^{C_0})))(y, x) = \epsilon.$$

So we can apply Lemma 6.12. It results in the existence of an extended execution $\zeta = \pi_0 \circ \rho_1$ of C_0 such that

- the execution π_0 of C_0 shares the initial state with π,
- ρ_1 is a prefix of an execution of C_1, and
- $\forall t \in \mathsf{Tr}(\rho_1).\ \mathsf{proc}(t) \in P$ and $\exists t \in \mathsf{Tr}(\rho_1).\ \mathsf{act}(t) = \mathsf{send}(v,\ q)$ for some v.

Note ζ is also in the same state space graph as where π is in.

Since π_0 is an execution of C_0, which shares the initial state with π, and the triple of C_0 is valid, we have μ and $\mathsf{allempty}$ hold at the final state $\mathsf{dest}(\mathsf{tail}(\pi'))$ (equal to $\mathsf{src}(\mathsf{head}(\rho'))$) for all processes.

Since $\mathcal{M} \models \langle \mu,\ \Gamma_1 \rangle\ C_1\ \langle \phi,\ \Upsilon_1 \rangle$, considering the ρ_1, there is no

$$\texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \in_{\rho_1} \Upsilon_1$$

such that $r' = r \vee r' = q$.

Thus, for ζ, the assertion $\Upsilon_1\ \mathsf{guarantees}_\zeta\ \Gamma_0$ fails to hold. It contradicts the assumption that Side Condition 1 is satisfied. $\square$

Lemma 6.15. Under the assumption made in Theorem 6.13, if an execution π of C_0C_1 emanates from a pre-state of C_0 where ψ and $\mathsf{allempty}$ hold for all processes, and π ends with a post-state of C_1, then ϕ and $\mathsf{allempty}$ hold on the post-state for all processes.

Proof According to Lemma 6.14, C_0 cannot be interfered by C_1 on π. Then we can apply Corollary 6.11. It results in an alternative π' to π in the same state space graph such that

1. π and π' share the initial and the final states, and
2. $\pi' = \pi_0 \circ \pi_1$ such that π_0 is an execution of C_0 and π_1 is an execution of C_1.

Because of $\mathcal{M} \models \langle \psi,\ \Gamma_0 \rangle\ C_0\ \langle \mu,\ \Upsilon_0 \rangle$ and $\mathcal{M} \models \langle \mu,\ \Gamma_1 \rangle\ C_1\ \langle \phi,\ \Upsilon_1 \rangle$, we have ϕ and $\mathsf{allempty}$ hold on $\mathsf{dest}(\mathsf{tail}(\pi'))$. Thus, ϕ and $\mathsf{allempty}$ also hold on $\mathsf{dest}(\mathsf{tail}(\pi))$. $\square$

Lemma 6.16. Under the assumption made in Theorem 6.13, for an execution π with n processes of C_0C_1 that emanates from a pre-state s of C_0, where ψ and $\mathsf{allempty}$ hold for all processes, $\pi \models \bigwedge_{v \in \Upsilon, p \in [n]} [\![v]\!]^{\mathsf{absent}}_{C_0C_1}(s, p)$.

Proof By Lemma 6.14, C_1 cannot interfere C_0. We prove this lemma by contradiction. Let $p, q, r, x \in [n]$ be some processes. Let $s = \mathsf{src}(\mathsf{head}(\pi))$. Suppose that there is an absence assertion $\upsilon \in_\pi \Upsilon$ such that

$$\pi \models \neg[\![\upsilon]\!]^{\mathsf{absent}}_{C_0 C_1}(s, k), \tag{a}$$

for any $k \in [n]$. There are two cases according to the possible forms of υ.

Case 1, we have

$$\upsilon = \texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(q). \tag{b}$$

By Side Condition 3 of the sequence rule, there are two sub-cases.

(1.1) $\upsilon \in_\pi \Upsilon_0$. By Corollary 6.11, there is an alternative execution π' to π such that $\pi' = \pi_0 \circ \pi_1$, where

- π and π_0 share the initial state s,
- π_0 is an execution of C_0, and
- π_1 is an execution of C_1.

The transition swappings performed on π for obtaining π' preserves the order that $\mathsf{exit}^{C_0}_{p,\pi}$ appears before $\mathsf{enter}^{C_0}_{q,\pi}$ on π (by Assumption a). So $\pi_0 \models \neg[\![\upsilon]\!]^{\mathsf{absent}}_{C_0}(s, k)$ as long as $\pi \models \neg[\![\upsilon]\!]^{\mathsf{absent}}_{C_0 C_1}(s, k)$. It contradicts the assumption that the triple of C_0 is valid.

(1.2) Since C_0 cannot be interfered by C_1, we can apply Lemma 6.12 to a prefix of π that ends with $\mathsf{exit}^{C_1}_{p,\pi}$. It results in an extended execution ζ of C_0 such that $\zeta = \pi_0 \circ \xi_1$, where

- π_0 is an execution of C_0,
- ξ_1 is a prefix of an execution of C_1, and μ and $\mathsf{allempty}$ hold for all processes on $\mathsf{src}(\mathsf{head}(\xi_1))$, according to Lemma 6.15, and
- $\upsilon \in_{\xi_1} \Upsilon_1$.

We show a contradiction to the validity of the triple of C_1 by showing that υ is violated by ξ_1. By Assumption a, process q has not entered C_0 while process p exits C_1, i.e., $\mathsf{enter}^{C_0}_{q,\pi}$ appears after $\mathsf{exit}^{C_1}_{p,\pi}$ on π. And by fact, $\mathsf{enter}^{C_1}_{q,\pi}$ has to appear after $\mathsf{enter}^{C_0}_{q,\pi}$ on π. Thus, no $\mathsf{enter}^{C_1}_{q,\zeta}$ in ξ_1, while by Lemma 6.12, ξ_1 includes $\mathsf{exit}^{C_1}_{p,\zeta}$.

Now considering Case 2. We have

$$\upsilon = \texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r). \tag{c}$$

By the $\mathsf{noSend}(\Upsilon) \subseteq_\pi \Upsilon_0$ part of Side Condition 2 of the sequence rule, we have $\upsilon \in_\pi \Upsilon_0$. We now have two sub-cases regarding to the order of $\mathsf{exit}^{C_0}_{p,\pi}$ and a trasition t_p on π, where $\mathsf{proc}(t_p) = p$ and $\mathsf{act}(t_p) = \mathsf{send}(v,\ q)$ for some v. Note such t_p must exist on π because υ is assumed to be violated on π.

2.1 $$t_p \text{ appears before } \mathsf{enter}^{C_0}_{r,\pi} \text{ and } \mathsf{exit}^{C_0}_{p,\pi} \text{ on } \pi \tag{d}$$

Let us continue to consider the execution $\pi' = \pi_0 \circ \pi_1$ mentioned in Case 1.1. Again, the transition swappings performed on π to obtain π' preserve the order that t_p appears before $\mathsf{enter}^{C_0}_{r,\pi}$ on π (by Condition d). We have $\pi_0 \models \neg[\![\upsilon]\!]^{\mathsf{absent}}_{C_0}(s,k)$ as long as $\pi \models \neg[\![\upsilon]\!]^{\mathsf{absent}}_{C_0C_1}(s,k)$ (Assumption a). Contradicting the validity of the triple of C_0.

2.2 $$t_p \text{ appears before } \mathsf{enter}^{C_0}_{r,\pi} \text{ and after } \mathsf{exit}^{C_0}_{p,\pi} \text{ on } \pi \tag{e}$$

There are three further sub-cases according to Side Condition 2 of the sequence rule.

2.2.1 There is

$$\texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\pi \Upsilon_0$$

This case can be proved contradicting to Assumption e with the same idea as how was Case 2.1 proved.

2.2.2 Considering the alternative $\pi' = \pi_0 \circ \pi_1$ mentioned in Case 1.1. There is

$$\texttt{\textbackslash no \textbackslash exit}(x)\ \texttt{\textbackslash after \textbackslash enter}(x)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\pi \Upsilon_0$$

and there is

$$\texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(x) \in_{\pi_1} \Upsilon_1$$

Since both the triples of C_0 and C_1 are valid, we have

$\mathsf{enter}^{C_0}_{r,\pi_0}$ appears before $\mathsf{exit}^{C_0}_{x,\pi_0}$ on π_0, (f)

$\mathsf{enter}^{C_1}_{x,\pi_1}$ appears before every such t'_p on π_1 that $\mathsf{proc}(t'_p) = p \wedge \mathsf{act}(t'_p) = \mathsf{act}(t_p)$ (g)

Transition swappings performed on π for obtaining π' preserves the orders of the transitions given in Condition f and g. Thus, we have $\mathsf{enter}^{C_0}_{r,\pi}$ appears before t_p on π. Contradicting Assumption a under Case 2.

2.2.3 Considering the alternative $\pi' = \pi_0 \circ \pi_1$ mentioned in Case 1.1. There is

$$\texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \in_{\pi_1} \Upsilon_1,$$

where $r' = r \vee r' = q$. A contradiction to Assumption a under Case 2 can be proved with the same idea used in Case 2.2.2.

□

Lemma 6.17. Under the assumption made in Theorem 6.13, for an extended execution ζ with n processes of C_0C_1, which starts from a pre-state of C_0 where ψ and $\mathsf{allempty}$ hold for all processes, we have

$$\begin{aligned}&\text{let } s = \mathsf{src}(\mathsf{head}(\zeta)) \text{ in}\\ &\forall p, q \in [n].\ \mathsf{interfere}_{\zeta}^{C_0C_1}(p,q) \Rightarrow (\zeta \models \neg \bigwedge_{\gamma\in\Gamma, k\in[n]} [\![\gamma]\!]_{C_0C_1}^{\mathsf{absent}}(s,k)).\end{aligned}$$

Proof Let $p, q, r \in [n]$ be some processes. Let $s = \mathsf{src}(\mathsf{head}(\zeta))$. To prove by contradiction, we assume that

$$\mathsf{interfere}_{\zeta}^{C_0C_1}(p,q)\ \wedge\ \zeta \models \bigwedge_{\gamma\in\Gamma, k\in[n]} [\![\gamma]\!]_{C_0C_1}^{\mathsf{absent}}(s,k), \tag{a}$$

for an ordered pair of processes (p, q). Without loss of generality, we further assume that

- (p, q) is the only such order pair of processes that $\mathsf{interfere}_{\zeta}^{C_0C_1}(p,q)$, and
- let ρ_p and ρ_q be the prefixes of ζ such that end with $\mathsf{exit}_{p,\zeta}^{C_1}$ and $\mathsf{exit}_{q,\zeta}^{C_1}$, respectively, we have $\mathsf{nsends}_{\rho_p}(p,q) + 1 = \mathsf{nrecvs}_{\rho_q}(p,q)$.

Then ζ must have the form $\ldots \circ t_p \circ \ldots \circ t_q \circ \ldots$, where

- $\mathsf{proc}(t_p) = p$, $\mathsf{proc}(t_q) = q$, $\mathsf{act}(t_p) = \mathsf{send}(v,\ q)$ and $\mathsf{act}(t_q) = \mathsf{recv}(v',\ p)$ for some v, v', and
- $\mathsf{chan}(\mathsf{src}(t_p))(p,q) = \epsilon$, and
- t_p appears after $\mathsf{exit}_{p,\zeta}^{C_1}$ and t_q appears before $\mathsf{exit}_{q,\zeta}^{C_1}$ on ζ.

There are two cases for ζ with respect to the order of t_q and $\mathsf{exit}^{C_0}_{q,\zeta}$ on ζ:

1. t_q appears before $\mathsf{exit}^{C_0}_{q,\zeta}$ on ζ, and
2. t_q appears after $\mathsf{exit}^{C_0}_{q,\zeta}$ and before $\mathsf{exit}^{C_1}_{q,\zeta}$ on ζ.

For Case 1, we have $\mathsf{interfere}^{C_0}_{\zeta}(p, q)$ because ζ is also a extended execution of C_0. Then, by the assumption that the triple of C_0 is valid, we have

$$\gamma = \texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_{\zeta} \Gamma_0 \tag{b}$$

such that ζ violates γ for C_0. By Side Condition 4 of the sequence rule, there are two sub-cases.

(1.1) We have $\gamma \in_{\zeta} \Gamma$. According to our construction of ζ. It contradicts Assumption a.

(1.2) By our construction of ζ, we can apply Lemma 6.12 to ζ to get an extended execution ζ' of C_0 such that $\zeta' = \pi'_0 \circ \xi'_1$, where π'_0 is an execution of C_0 and ξ'_1 is an extended execution of C_1. In this case, Υ_1 $\mathsf{cancels}_{\zeta'}$ γ_0. Consequently,

$$\upsilon = \texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(x) \in_{\zeta'} \Upsilon_1,$$

where $x = r \vee x = q$. We need to show that υ is violated by ξ_1 for a contradiction to the validity of the collective triple of C_1.

Recall in our construction of ζ,

$$t_p \text{ appears after } \mathsf{exit}^{C_1}_{p,\zeta} \text{ and before } \mathsf{exit}^{C_0}_{q,\zeta} \text{ on } \zeta.$$

Consequently,

$$\mathsf{exit}^{C_1}_{p,\zeta} \text{ appears before } \mathsf{enter}^{C_1}_{q,\zeta} \text{ on } \zeta. \tag{c}$$

Since the existence of t_p leads to the violation of γ by ζ for C_0 (Assumption b), t_p appears before $\mathsf{enter}^{C_1}_{r,\zeta}$ on ζ. Consequently,

$$\mathsf{exit}^{C_1}_{p,\zeta} \text{ appears before } \mathsf{enter}^{C_1}_{r,\zeta} \text{ on } \zeta. \tag{d}$$

Note that the order of $\mathsf{exit}^{C_1}_{p,\zeta}$, $\mathsf{enter}^{C_1}_{q,\zeta}$ and $\mathsf{enter}^{C_1}_{r,\zeta}$ is preserved during the transition swappings performed on ζ. Then, by Condition c and d, we have that υ is violated by ξ'_1 for C_1.

Now we prove for Case 2. Let us continue to consider the extended execution $\zeta' = \pi'_0 \circ \xi'_1$ used in Case 1.2.

Under Case 2, $\mathsf{interfere}^{C_1}_{\xi'_1}(p, q)$, because the order of t_p, t_q, $\mathsf{exit}^{C_1}_{p,\zeta}$ and $\mathsf{exit}^{C_1}_{q,\zeta}$ is preserved by the transition swappings performed on ζ. Then, since the triple of C_1 is valid, there must be

$$\gamma = \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_{\xi'_1} \Gamma_1 \tag{e}$$

s.t. γ is violated by ξ'_1 for C_1.

By Side Condition 4 of the sequence rule, we have

$$\gamma \in_{\zeta'} \Gamma. \tag{f}$$

Again, the order of t_p, t_q, $\mathsf{exit}^{C_1}_{p,\zeta}$, $\mathsf{exit}^{C_1}_{q,\zeta}$ and $\mathsf{exit}^{C_1}_{r,\zeta}$ is preserved by transition swappings performed on ζ for obtaining ζ', we conclude that

$$\zeta \text{ has the form } \rho_0 \circ t_p \circ \rho_1 \circ t_q \circ \rho_2$$

s.t.

- $\mathsf{exit}^{C_1}_{p,\zeta} \in \mathsf{Tr}(\rho_0)$, by construction of ζ, and
- $\mathsf{exit}^{C_1}_{r,\zeta} \in \mathsf{Tr}(\rho_1) \cup \mathsf{Tr}(\rho'')$, by Condition e, and
- $\mathsf{exit}^{C_1}_{q,\zeta} \in \mathsf{Tr}(\rho_2)$, by construction of ζ.

Thus, together with Condition f, υ is violated by ζ. Contradicting Assumption a. □

Theorem 6.18 (soundness of the consequence rule)**.** Assuming the side conditions of the consequence rule over $\psi, \psi', \phi, \phi', \Gamma, \Gamma', \Upsilon$ and Υ' hold,

$$\mathcal{M} \models \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, \text{ if } \mathcal{M} \models \langle \psi', \Gamma' \rangle\ C\ \langle \phi', \Upsilon' \rangle.$$

Proof Strengthening the state- or the path-requirement cannot invalidate a valid collective triple. Dually, weakening the state- or the path-guarantee cannot invalidate a valid collective triple. □

Theorem 6.19 (soundness of the collective loop rule)**.** Assuming the side condition of the collective loop rule holds,

$$\mathcal{M} \models \langle inv, \Gamma\rangle \texttt{ while(}c\texttt{) } C \ \langle \neg c \wedge inv, \Upsilon\rangle, \text{ if } \mathcal{M} \models \langle inv \wedge c, \Gamma\rangle \ C \ \langle inv, \Upsilon\rangle.$$

Proof Imagine unrolling the loop to a finite sequence $CC \ldots C$ of length m. The validity of the original triple of the loop can be proved by proving the following two tasks,

1. $\mathcal{M} \models \langle inv \wedge c, \Gamma\rangle \ CC \ldots C \ \langle inv \wedge c, \Upsilon\rangle$, with the strengthened assumption $\mathcal{M} \models \langle inv \wedge c, \Gamma\rangle \ C \ \langle inv \wedge c, \Upsilon\rangle$, for the prefix $CC \ldots C$ of length $m-1$, and
2. $\mathcal{M} \models \langle inv \wedge c, \Gamma\rangle \ C \ \langle inv, \Upsilon\rangle$.

Task 1 can be proved by repeatedly applying the sequence rule. Task 2 is directly inferred by the assumption. □

Theorem 6.20 (soundness of the elaboration rule)**.**

$$\mathcal{M} \models \langle \psi, \Gamma\rangle \ C \ \langle \phi, \Upsilon\rangle$$

if $\langle \psi\rangle \ \overrightarrow{C} \subseteq \overrightarrow{C_0 C_1}$ and $\mathcal{M} \models \langle \psi, \Gamma\rangle \ C_0 C_1 \ \langle \phi, \Upsilon\rangle$.

Proof According to the elaboration condition, for every execution π of C, there is an equivalent execution π' of $C_0 C_1$ such that that both executions start from a same pre-state where ψ holds. The only difference between π and π' is that they have different transitions in them that are all labeled by `skip`, which has no effect on neither the memory in state nor the order of process executions. So whenever π' satisfies a specification, π satisfies the same specification. □

Chapter 7

A VERIFICATION SYSTEM FOR MINIMP

In this chapter, we describe a contract based MiniMP verification system. It verifies an MiniMP program against a specification in a composite and modular way.

We first introduce the basic model checking and symbolic execution algorithm for monolithic verification in §7.1. In §7.2, we improve the basic algorithm with respect to $\mathcal{R}$ for composite and modular verification. Around the improved the algorithm, we discuss about a procedure contract based verification system in §7.3. Finally, we give proofs for the soundness of the algorithms in §7.4.

We fix the context of our discussion in this chapter with a vocabulary $\mathcal{V} = (\mathsf{Procedure}, \mathsf{Var}, \mathsf{Global}, \mathsf{locvar})$, a domain Val and a MiniMP model $\mathcal{M}$.

7.1 Symbolic Execution and Model Checking for MiniMP

In this section, we introduce a basic algorithm for monolithic verification of functional correctness of a MiniMP program segment against its contract. This algorithm uses model checking and symbolic execution techniques.

In order to do symbolic execution, we must allow variables in MiniMP to hold *symbolic expressions*. In addition, states of MiniMP programs must be associated with *path conditions* (PCs).

Definition 7.1. Let Sym be a set of symbols. Let $\mathsf{Expr}_{\mathsf{Val}\cup\mathsf{Sym}}$ be the set of expressions over Val and Sym. An expression in $\mathsf{Expr}_{\mathsf{Val}\cup\mathsf{Sym}}$ is said a *symbolic expression*. Specially, an element in Sym is also said a *symbolic constant*, and an element in Val is also said a *concrete value*. □

$$
\begin{array}{l}
I_n^{\mathsf{Sym}}((s, \mathsf{pc}), p, (l, g, a, l')) = \\
\text{let } \mathsf{pc}' = \mathsf{pc} \wedge [\![g]\!]_n(s, p) \text{ in} \\
\{\omega'[\mathsf{pc}'/\mathsf{pc}] \in \mathsf{State}^{\mathsf{Sym}} \mid \omega' \in I(s, p, (l, \mathtt{true}, a, l'))\}
\end{array}
$$

Figure 7.1: The definition of the interpretation for symbolic execution with n processes.

Definition 7.2. Given a symbolic expression set $\mathsf{Expr}_{\mathsf{Val} \cup \mathsf{Sym}}$. A *state for symbolic execution* of $\mathcal{M}$ is a pair:

$$(s, \mathsf{pc}),$$

where $s \colon \mathsf{State}$ is a state of $\mathcal{M}$ and $\mathsf{pc} \colon \mathsf{Expr}_{\mathsf{Val} \cup \mathsf{Sym}}$ is the boolean path condition. Let $\mathsf{State}^{\mathsf{Sym}}$ be the set of states for symbolic execution. □

Given a state $\omega \in \mathsf{State}^{\mathsf{Sym}}$, by $\mathsf{PC}(\omega)$, we denote the path condition of ω.

From now on, for the rest of this chapter, we call elements in $\mathsf{State}^{\mathsf{Sym}}$ *states*. Given a state $\omega = (s, \mathsf{pc})$, we refer any component c of s as "c of ω", and we refer pc as "the PC of ω", which may also be said "the path condition of ω".

The shortcut notations introduced in §4.3 involving elements in State will be naturally extended with respect to elements in $\mathsf{State}^{\mathsf{Sym}}$. For example, the call stack of a process p in a state $\omega \in \mathsf{State}^{\mathsf{Sym}}$ can be denoted $\mathsf{stack}(\omega, p)$. Correspondingly, the evaluation functions with n processes: $[\![\ ]\!]_n$, $[\![\ ]\!]_n^{\mathsf{spec}}$, and $[\![\ ]\!]_n^{\mathsf{absent}}$, are also extended in a natural way for symbolic expressions and $\mathsf{State}^{\mathsf{Sym}}$.

Definition 7.3. Given the symbolic expression set $\mathsf{Expr}_{\mathsf{Val} \cup \mathsf{Sym}}$ and an evaluation function $[\![\ ]\!]_n$. Let T be the set of local transitions. A n-processes *interpretation for symbolic execution* $I_n^{\mathsf{Sym}} \colon \mathsf{State}^{\mathsf{Sym}} \times [n] \times T \to \wp(\mathsf{State}^{\mathsf{Sym}})$ is a function describes the states, each of which is a possible result of executing a local transition $(l, g, a, l') \in T$ from a state $\omega \in \mathsf{State}^{\mathsf{Sym}}$ by a process $p \in [n]$. We define I^{Sym} in Fig. 7.1. □

When the number of processes n is clear in the context, we may simplify I_n^{Sym} to I^{Sym}.

Given a boolean symbolic expression e, e is said *satisfiable* iff there is at least an assignment from symbols in e to concrete values that makes e true. If e is true for any assignment, e is said valid. If e is false for any assignment, e is said invalid, or *unsatisfiable*. The intuition in I^{Sym} is that, given a source state, a process and a local transition, I^{Sym} always compute the target states regardless of whether the guard is valid or not. Besides, it adds the guard to the path conditions of the target states. If the path condition of a state is unsatisfiable, paths containing the state are *infeasible*.

Let Mc be the monolithic algorithm that verifies the validity of a collective triple for a fixed number processes. Mc consumes a model, a collective triple $\langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle$, and a integer number n, and returns $\mathbf{T}$ iff $\mathcal{M} \models \langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle$ for n processes.

$\textsc{Mc}(\mathcal{M}, \langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle, n)$ verifies the validity of the triple through searching states and paths that violate the specification in a *state space graph for symbolic execution* of C.

Definition 7.4. Given a collective triple $\langle \psi,\, \Gamma\rangle\ l\colon C\ l'\ \langle \phi,\, \Upsilon\rangle$. Let T be the set of local transitions translated from $l\colon C\ l'$. The *state space graph for symbolic execution* G of C with n processes is a tuple:

$$G = (n, \omega_0, \Omega, \rightsquigarrow_C^{\mathsf{Sym}}),$$

where

- n is the number of processes
- ω_0 is the initial state that satisfies the following conditions
  1. ω_0 is a pre-state of C,
  2. for every $p \in [n]$ and every $v \in \mathsf{Var}$, $\mathsf{mem}(\omega_0, p)(v)$ is a unique symbolic constant, and
  3. $\mathsf{PC}(\omega_0) = \bigwedge_{p\in[n]} [\![\psi]\!](\omega_0, p)$ and $\mathsf{allempty}$ holds for all processes on ω_0.
- Ω is the set of states that are reachable from ω_0 over T with the interpretation I^{Sym}.
- $\rightsquigarrow_C^{\mathsf{Sym}} \colon \mathsf{State}^{\mathsf{Sym}} \times [n] \times T \times \mathsf{State}^{\mathsf{Sym}}$ is a set of global transitions defined as

$$\{(\omega, p, \alpha, \omega') \mid \omega' \in I^{\mathsf{Sym}}(\omega, p, \alpha), p \in [n], \omega \in \Omega, \alpha \in T\}$$

We also call G the state space graph searched by $\textsc{Mc}(\mathcal{M}, \langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle, n)$. $\square$

The definition of $\mathsf{act}(t)$ is also extended for the global transition $t \in \mathsf{State}^{\mathsf{Sym}} \times [n] \times T \times \mathsf{State}^{\mathsf{Sym}}$ in a natural way.

Given a state space graph G searched by $\textsc{Mc}(\mathcal{M}, \langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle, n)$. For any execution in G, processes terminate once they complete C. So it is not possible for a process to send messages after it exits C. In other words, G excludes the extended executions of C that may cause interference. Then does it mean that it is not sufficient to prove the validity of the triple by exhaustively exploring G? In fact, it is sufficient to verify the triple only within G, if one always make the worst case assumption for interference, i.e., for any process p that has terminated, p may send a message to any other process at any time. We now define a predicate $\mathsf{mayInterfere}$ for this worst case assumption.

Definition 7.5. Given a state space graph $G = (n, \omega_0, \Omega, \rightsquigarrow_C^{\mathsf{Sym}})$ searched by the algorithm $\textsc{Mc}(\mathcal{M}, \langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle, n)$. Given a state $\omega \in \Omega$. By $\mathsf{emanate}_G(\omega)$, we denote the set of global transitions that emanate from ω in G, i.e.,

$$t \in \mathsf{emanate}_G(\omega) \text{ iff } t \in \rightsquigarrow_C^{\mathsf{Sym}} \ \wedge\ \mathsf{src}(t) = \omega.$$

Note for any $t \in \mathsf{emanate}_G(\omega)$, $\mathsf{PC}(\mathsf{dest}(t))$ may be unsatisfiable. $\square$

Definition 7.6. Given a state space graph G searched by $\textsc{Mc}(\mathcal{M}, \langle \psi,\, \Gamma\rangle\ C\ \langle \phi,\, \Upsilon\rangle, n)$. Let ω be a state in a feasible path in G. Let $p, q \in [n]$ be some processes. By $\mathsf{mayInterfere}_G(\omega, p, q)$, we denote that the state ω can lead to an interference if process p sends a message to q in the next step. Let $v \in \mathsf{Expr}_{\mathcal{V}}$ be some expression. Formally, we have

$\mathsf{mayInterfere}(G, \omega, p, q, r)$ iff

$\exists t_q \in \mathsf{emanate}_G(\omega).\ \neg\exists t_p \in \mathsf{emanate}_G(\omega).$

$\mathsf{proc}(t_q) = q \wedge \mathsf{proc}(t_p) = p \wedge \mathsf{act}(t_q) = \mathsf{recv}(v,\, p) \wedge \mathsf{chan}(\omega)(p, q) = \epsilon.$

$\square$

The intuition in Def. 7.6 is that if we assume the worst case, for any state ω in a feasible path in G, no transition $t \in \mathsf{emanate}_G(\omega)$ shall be labeled by a receive action over an empty channel from a terminated process p to $\mathsf{proc}(t)$. Otherwise, by the worst case assumption, p will send a message to $\mathsf{proc}(t)$ in the next step. Then executing t becomes feasible and will cause an interference. Now considering the collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ associated with G. For an execution π in G such that $\omega \in \mathsf{St}(\pi)$, if

$$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma,$$

we know that p will not send a message to q until a process r exits C. Then the worst case assumption can be constrained to that process p will send a message to q after exiting C at any time after r exits C.

Definition 7.7. A prefix ρ of a feasible execution in the state space graph G searched by the algorithm $\textsc{Mc}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n)$ violates the contract or contains a deadlock, denoted $\rho \not\models \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$, if it belongs to one of the cases defined in Fig. 7.2. We have $\textsc{Mc}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n) = \mathbf{T}$, if for every prefix ρ in G, ρ contains no deadlock and does not violate the contract. □

If $\textsc{Mc}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n) = \mathbf{T}$, we also informally say that the state space graph searched by $\textsc{Mc}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n)$ is *error free*.

We explain Fig. 7.2 briefly. Given a state space graph G searched by the algorithm $\textsc{Mc}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n)$. **DL** defines the condition of deadlock, i.e., the final state of a feasible execution is not a post-state of C; **SGV** defines the case where the state guarantee ϕ fails to hold on a post-state of C; **ITF** defines the case where the path requirement Γ fails to prevent possible interferences; finally, **PGV** defines the case where a feasible execution of C fails to satisfy the path guarantee Υ.

The soundness of Mc is proved in §7.4.1

7.2 Composite and Modular Verification for MiniMP

We have introduced a monolithic verification algorithm Mc in §7.1 that is able to prove the validity of a collective triple for a fixed number of processes. In this section,

DL, deadlock:
let $\omega = \mathsf{dest}(\mathsf{tail}(\rho))$ in
ω is not a post-state of C, and
$\forall t \in \mathsf{emanate}_G(\omega).\, \mathsf{PC}(\mathsf{dest}(t)) = \mathbf{F}$

SGV, state guarantee violation
let $\omega = \mathsf{dest}(\mathsf{tail}(\rho))$ in
if ω is a post-state of C, then $\bigwedge_{p \in [n]} [\![\mathsf{allempty}\ \&\&\ \phi]\!]^{\mathsf{spec}}(\omega, p) = \mathbf{F}$

ITF, interference
let $\omega \in \mathsf{St}(\rho)$ in
$\exists p, q, r \in [n].\, \mathsf{mayInterfere}_G(\omega, p, q) \wedge$
$\neg\exists$`\no \send(`p`, `q`) \after \exit(`p`) \until \exit(`r`)` $\in_\rho \Gamma$.
ω appears before $\mathsf{exit}^C_{r,\rho}$ on ρ

PGV, path guarantee violation
let $\omega = \mathsf{src}(\mathsf{head}(\rho))$ in
$\rho \models \neg \bigwedge_{\upsilon \in \Upsilon, p \in [n]} [\![\upsilon]\!]^{\mathsf{absent}}(\omega, p)$

Figure 7.2: The definition of contract violation and deadlock for a prefix ρ of an execution in the state space graph G searched by $\textsc{Mc}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n)$.

we improve Mc to $\textsc{Mc}^\Delta$, which is a composite and modular verification algorithm taking advantages from $\mathcal{R}$.

Given a collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ such that $C = C_0 C_1 \ldots C_{m-1}$ or $\langle \psi \rangle \overrightarrow{C} \subseteq \overrightarrow{C_0 C_1 \ldots C_{m-1}}$ for m statement sequences C_0, C_1, $\ldots$, C_{m-1}, $m > 0$. We call a set of collective triples a *set of sequentially-decomposed triples* of C, if the set contains a unique collective triple $\langle \psi_i, \Gamma_i \rangle\ C_i\ \langle \phi_i, \Upsilon_i \rangle$ for each C_i, $0 \leq i < m$.

Let Tri be a subset of a sequentially-decomposed triple set of C. The algorithm $\textsc{Mc}^\Delta$ verifies the validity of $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ by exploring feasible paths in a state space graph of C with the assumption that all triples in Tri are valid. During the verification, for each $\langle \psi_i, \Gamma_i \rangle\ C_i\ \langle \phi_i, \Upsilon_i \rangle \in \mathsf{Tri}, 0 \leq i < m$, only the contract of C_i is used to derive

behaviors of C_i.

Intuitively, there is a gap between contract evaluation and semantics of collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$: for an execution π of C, there is not necessarily a pair of pre- and post-states of a $C_i, 0 \leq i < m$, exist in π. In such cases, where is the contract of C_i suppose to be evaluated? We propose a solution to this problem using *collective states* in §7.2.1. The algorithm MC^{Δ} is based on collective states and is described in §7.2.2.

In this section, we fix our context with the collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ and the subset Tri of a sequentially-decomposed triple set of C.

7.2.1 Collective States

A *collective state* is a state obtained by merging process states from different states in an execution. It is associated with a path ρ and plays the role of a specific state, which may not exist in ρ, where all processes simultaneously arrive at a set of locations as if there are "bulk-synchronous" barriers placed at each of these locations.

Our interest actually focuses on certain kinds of collective states, the *collective pre-* and *post-states*. They are the collective states where processes are at entry and exit, respectively, of a particular statement sequence.

Given an execution π of C. We are interested in the pre- and post-states of C_i, $(0 \leq i < m)$, associated with π, if $\langle \psi_i, \Gamma_i \rangle\ C_i\ \langle \phi_i, \Upsilon_i \rangle \in \mathsf{Tri}$. The process state of a process p in the collective pre-state of C_i will be equivalent to $\mathsf{procState}(\mathsf{src}(\mathsf{enter}^{C_i}_{p,\pi}))$. Similarly, the process state of p in the collective post-state of C_i will be equivalent to $\mathsf{procState}(\mathsf{dest}(\mathsf{exit}^{C_i}_{p,\pi}))$. Recall enter and exit are defined in §6.4.

The value of the message channels in the collective pre-state (resp. post-state) of C_i associated with π follows an observation. Suppose that there is a "bulk-synchronous" barrier at the entry (resp. exit) of C_i. Let π' be an execution of such C that contains the barrier and the sub-statement C_i. Let ω be the state in π' where all processes are blocked by the barrier. The message channels in ω depends and only depends on the transitions that appear before ω and are labeled by a send or a receive action.

Such an observation will be generalized by a precise definition for collective states. Before that, we introduce a few notations.

- Given an execution π of C. By $\mathsf{preCS}(\pi, C_i)$, we denote the collective pre-state of C_i associated with π; by $\mathsf{postCS}(\pi, C_i)$, we denote the collective post-state of C_i associated with π.
- Given an execution π of C. A transition t is said to *be depended on by the message channels* of the collective state $\mathsf{preCS}(\pi, C_i)$ (resp. $\mathsf{postCS}(\pi, C_i)$), if
  1. $t \in \mathsf{Tr}(\pi)$,
  2. $\mathsf{act}(t)$ is either a send or receive action, and
  3. t appears before $\mathsf{enter}^{C_i}_{\mathsf{proc}(t),\pi}$ (resp. $\mathsf{exit}^{C_i}_{\mathsf{proc}(t),\pi}$) on π.
- Given an execution π of C. Let ρ be a subpath of π. We use the procedure $\mathsf{preCoChan}(\omega, \rho, \pi, C_i)$ to describe the state that is obtained by updating the message channel function in the given state ω with respect to the transitions in ρ, on which $\mathsf{preCS}(\pi, C_i)$ depends:

$$
\begin{aligned}
&\text{let } t = \mathsf{head}(\rho) \text{ in}\\
&\text{let } p = \mathsf{proc}(t) \text{ in}\\
&\text{let } \rho = t \circ \rho', \text{if } \rho \neq \epsilon, \text{ in}\\
&\mathsf{preCoChan}(\omega, \rho, \pi, C_i) =\\
&\begin{cases}
\omega, & \text{if } \rho = \epsilon\\
\mathsf{preCoChan}(\omega, \rho', \pi, C_i), & \text{if } \mathsf{preCS}(\pi, C_i) \text{ not depends on } t\\
\mathsf{preCoChan}(\omega[(p, q)\,!\,v], \rho', \pi, C_i), & \text{if } \mathsf{act}(t) = \mathsf{send}(v,\, q)\\
\mathsf{preCoChan}(\omega[(q, p)?v], \rho', \pi, C_i) & \text{if } \mathsf{act}(t) = \mathsf{recv}(v,\, q)
\end{cases}
\end{aligned}
$$

  Similar for $\mathsf{postCoChan}$, whose definition is omitted here for brevity.

Definition 7.8. Given an execution π with n processes of C. A collective pre-state $\mathsf{preCS}(\pi, C_i)$ of C_i associated with π is defined by

process states:

$$\forall p \in [n].\, \mathsf{procState}(\mathsf{preCS}(\pi, C_i), p) = \mathsf{procState}(\mathsf{src}(\mathsf{enter}^{C_i}_{p,\pi}), p)$$

message channels:

$$\mathsf{chan}(\mathsf{preCS}(\pi, C_i)) = \mathsf{chan}(\mathsf{preCoChan}(\mathsf{src}(\mathsf{head}(\pi)), \pi, \pi, C_i))$$

path condition:

$$\mathsf{PC}(\mathsf{preCS}(\pi, C_i)) = \bigwedge_{p \in [n]} \mathsf{PC}(\mathsf{src}(\mathsf{enter}^{C_i}_{p,\pi}))$$

The definition of the collective post-state $\mathsf{postCS}(\pi, C_i)$ is similar to above. Hence we omit it for brevity. $\square$

Message channels of the collective pre-state $\mathsf{preCS}(\pi, C_i)$ actually can be represented in many different ways since for any state ω that appears before the first $\mathsf{enter}^{C_i}_{p,\pi}$ for some $p \in [n]$ on π, the message channels of ω depend on all the transitions labeled by communication actions. Hence one can still obtain the correct message channels by applying the $\mathsf{preCoChan}$ procedure on ω and the suffix of π emanating from ω. In practice, it is the most efficient to do

$$\mathsf{preCoChan}(\mathsf{src}(\mathsf{head}(\rho)), \rho, \pi, C_i),$$

where ρ is defined as the minimal sub-path of π such that $\forall p \in [n].\, \mathsf{enter}^{C_i}_{p,\pi} \in \mathsf{Tr}(\rho)$.

Example 7.9. Figure 7.3 shows a collective pre-state (upper right) of a statement C_1 associated with an execution (below) of a MiniMP code C_0C_1 (upper left), where C_0 is a statement sequence at line 4-5 and C_1 is a statement sequence at line 6-7. The collective state depends on the process states in state 4, 6 and 7. The value of the message channels in the collective state is initialized by the one in state 4, where the message channels from 0 to 1 and from 1 to 2 are non-empty. After state 4, communication actions performed by process 1 and 2 can still affect the collective state. From state 5, process 1 `recv`s a message from process 0. It causes the message channel from 0 to 1 in collective state to be updated to empty. After state 6, only process 2 can affect the collective state. From state 6, process 2 `recv`s a message from process 1. The message channels in the collective state eventually are updated to all empty. $\square$

7.2.2 The Composite and Modular Verification Algorithm

By $\textsc{Mc}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi, \Gamma\rangle\ C\ \langle\phi, \Upsilon\rangle}, n) = \mathbf{T}$, we denote that the algorithm $\textsc{Mc}^{\Delta}$ verifies the validity of $\langle\psi, \Gamma\rangle\ C\ \langle\phi, \Upsilon\rangle$ for n processes in a composite and modular way with the assumption that all triples in Tri are valid.

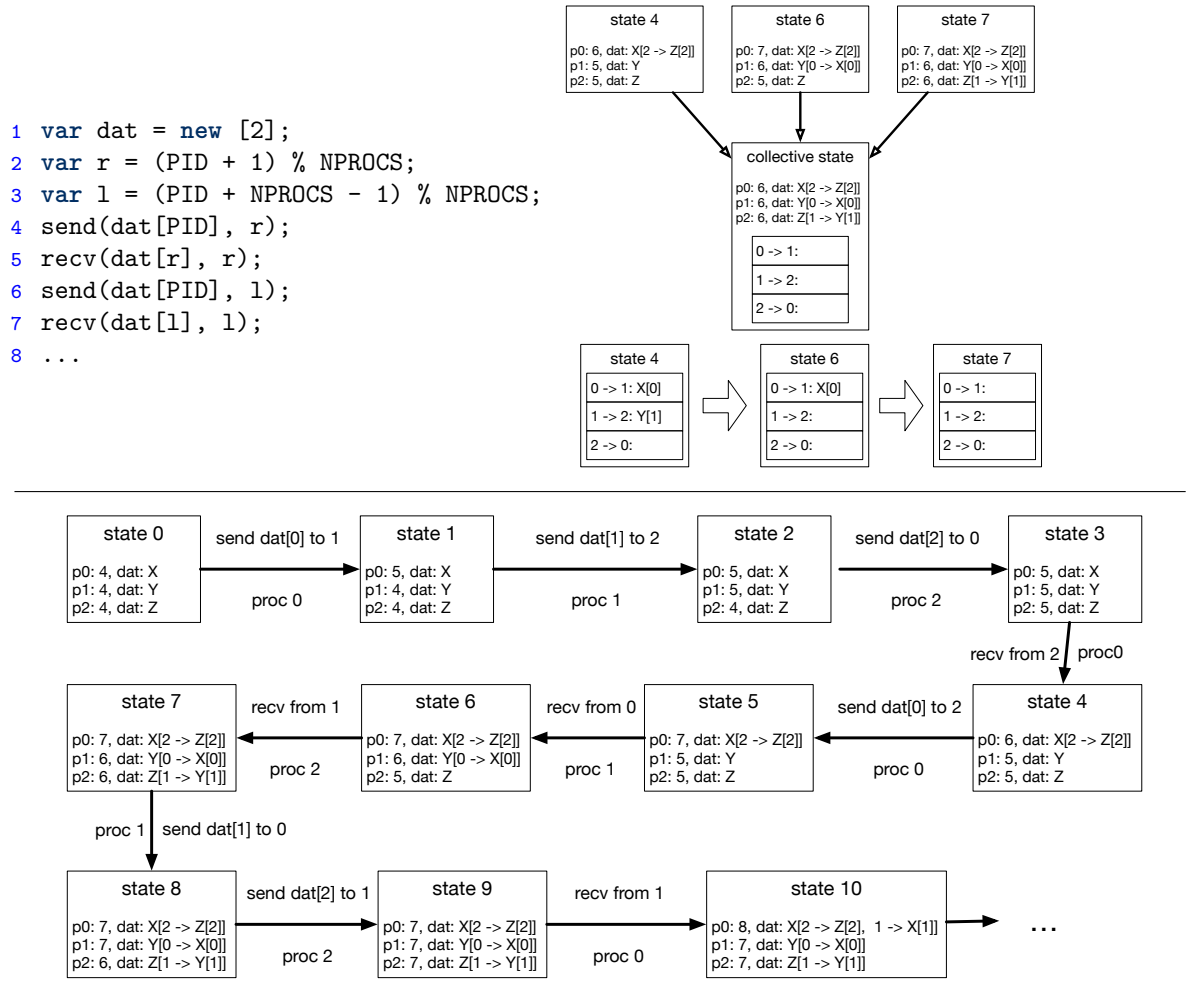

Figure 7.3: A collective state (upper right) associated with an execution (below) of a MiniMP program (upper left). All processes are collectively at line 6 in the collective state.

We describe MC$^{\Delta}$ by illustrating the differences between MC and itself.

First, besides the atomic statements defined in §4.3, we add a new atomic statement refresh for MC$^{\Delta}$. The statement means to assign "fresh new" (i.e., unique and

unconstrained) symbols to all *visible* variables to a process. Formally, it is

$$
\begin{aligned}
&I^{\mathsf{Sym}}(\omega, p, (l, g, \mathsf{refresh}, l')) = \\
&\text{let } f \text{ be the procedure associated with } \mathsf{top}(\omega, p) \text{ in} \\
&\text{let } \{X_0, X_1, \dots\} \text{ be an infinite set of symbols that are fresh new to } \omega \text{ in} \\
&\text{let } \{v_0, v_1, \dots, v_{m-1}\} \text{ be union of } \mathsf{Global} \cup \mathsf{locvar}(f) \text{ in} \\
&\text{let } \mathsf{pc} = \mathsf{PC}(\omega) \text{ in} \\
&\{\omega[\mathsf{pc} := \mathsf{pc} \wedge [\![g]\!](\omega, p)][v_0 \leftarrow X_0, v_1 \leftarrow X_1, \dots, v_{m-1} \leftarrow X_{m-1}]_p\}
\end{aligned}
$$

Let $\mathsf{refresh}$ also be an atomic action. So we have $\mathsf{act}(t) = \mathsf{refresh}$, if given a gloabl transition $t = (\omega, p, (l, g, \mathsf{refresh}, l'), \omega)$.

We then modify the translation procedure $\mathsf{transStmt}$ to $\mathsf{transStmt}_{\mathsf{Tri}}$, which translates C to a set of local transitions such that, for any $\langle \psi_i, \Gamma_i \rangle\ C_i\ \langle \phi_i, \Upsilon_i \rangle \in \mathsf{Tri}$, the translation result excludes the detail of C_i. Formally,

$$
\mathsf{transStmt}_{\mathsf{Tri}}(l, C_i) = \begin{cases} (l+1, (l, \mathtt{true}, \mathsf{refresh}, l+1)), & \text{if } \exists \langle \psi_i, \Gamma_i \rangle\ C_i\ \langle \phi_i, \Upsilon_i \rangle \in \mathsf{Tri} \\ \mathsf{transStmt}(l, C_i), & \text{otherwise} \end{cases}
$$

Intuitively, the `refresh` statement can be seen as an operation for making the weakest over-approximation of the behaviors of a sub-statement of C.

While the weakest over-approximation can barely be useful, we want the appoximation of C_i to reflect the contract. We shall assume that ϕ_i and $\mathsf{allempty}$ hold on the collective post-state $\mathsf{postCS}(\pi, C_i)$, if ϕ_i and $\mathsf{allempty}$ hold on the collective pre-state $\mathsf{preCS}(\pi, C_i)$. In addition, we assume that the path-guarantee Υ_i is satisfied by every execution of C. Technically, an execution is infeasible if these assumptions fail to hold on it.

Definition 7.10. For the collective triple $\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ and the triple set Tri, the state space graph G searched by $\textsc{Mc}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle}, n)$ is a tuple:

$$
(n, \omega_0, \Omega, \rightsquigarrow_C^{\mathsf{Sym},\Delta}),
$$

where

- n is the number of processes,
- ω_0 is the initial state that satisfies the following conditions
  1. ω_0 is a pre-state of C,
  2. for every $p \in [n]$ and every $v \in \mathsf{Var}$, $\mathsf{mem}(\omega_0, p)(v)$ is a unique symbolic constant, and
  3. $\mathsf{PC}(\omega_0) = \bigwedge_{p \in [n]} [\![\psi]\!](\omega_0, p)$ and $\mathsf{allempty}$ holds for all processes on ω_0.
- Ω is the set of states that are reachable from ω_0 over $\mathsf{transStmt}_{\mathsf{Tri}}(l, C)$ for some location l, and
- $\rightsquigarrow_C^{\mathsf{Sym},\Delta}$ is the set of global transitions

$$\{(\omega, p, a, \omega') \mid \omega' \in I^{\mathsf{Sym}}(\omega, p, \alpha), p \in [n], \omega \in \Omega, \alpha \in \mathsf{transStmt}_{\mathsf{Tri}}(l, C)\}$$

□

We define whether an execution explored by $\mathrm{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi, \Gamma\rangle\ C\ \langle\phi, \Upsilon\rangle}, n)$ is feasible with respect to the contracts of triples in Tri.

Definition 7.11. Given the state space graph G searched by $\mathrm{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi, \Gamma\rangle\ C\ \langle\phi, \Upsilon\rangle}, n)$. An execution π in G is *infeasible*, if one of the following conditions holds.

1. There is a state $\omega \in \mathsf{St}(\pi)$ such that $\mathsf{PC}(\omega) = \mathbf{F}$
2. There is $\langle\psi_i, \Gamma_i\rangle\ C_i\ \langle\phi_i, \Upsilon_i\rangle \in \mathsf{Tri}$ such that
   (a) $\exists p \in [n].\ [\![\phi_i]\!]^{\mathsf{spec}}(\mathsf{postCS}(\pi, C_i), p) = \mathbf{F}$, or
   (b) $\pi \models \neg \bigwedge_{v \in \Upsilon, p \in [n]} [\![v]\!]^{\mathsf{absent}}_{C_i}(\mathsf{preCS}(\pi, C_i), p)$

The execution π is *feasible* if it is not *infeasible*. □

In fact, Condition 2.b in Def. 7.11 can be strengthened to

$$\pi \models \neg \bigwedge_{v \in \Lambda, p \in [n]} [\![v]\!]^{\mathsf{absent}}_{C_i}(\mathsf{preCS}(\pi, C_i), p), \text{ where } \Lambda = \Upsilon \backslash \mathsf{noSend}(\Upsilon).$$

Because details of C_i has been abstracted to $\mathsf{refresh}$. Consequently, for any $v \in \mathsf{noSend}(\Upsilon)$, $[\![\ ]\!]^{\mathsf{absent}}_{C_i,\pi}$ can only evaluate it to true.

For an execution in the state space graph searched by $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C\ \langle\phi,\Upsilon\rangle}, n)$, there is at most one collective pre-state (resp. collective post-state) of C_i, which is specified by a triple in Tri. Because recursions of C_i have been abstracted away in the execution.

Therefore, we can extend the set-operation $\in_\rho$ associated with a path ρ over absence assertions that are suppose to be evaluated on collective states.

Definition 7.12. Given a feasible execution π in the state space graph searched by $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C\ \langle\phi,\Upsilon\rangle}, n)$. Let C_i be a sub-statement of C such that $\langle\psi_i,\ \Gamma_i\rangle\ C_i\ \langle\phi_i,\ \Upsilon_i\rangle \in \mathsf{Tri}$. For an absence assertion λ,

$$\lambda \in_\pi \Gamma_i \text{ iff } \exists\gamma \in \Gamma_i.\ \exists p, q \in [n].\ \mathsf{pe}(\lambda, \mathsf{preCS}(\pi, C_i), p) = \mathsf{pe}(\gamma, \mathsf{preCS}(\pi, C_i), q).$$

Similar for $\lambda \in_\pi \Upsilon_i$. $\square$

The algorithm MC^{Δ} returns **F** if there is a feasible execution in the searching space that violates a contract or contains a deadlock.

Definition 7.13. Given a feasible execution π in the state space graph searched by $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C\ \langle\phi,\Upsilon\rangle}, n)$. π violates a contract or contains a deadlock, denoted $\pi \not\models^{\Delta} \langle\psi,\ \Gamma\rangle\ C\ \langle\phi,\ \Upsilon\rangle$, if $\pi \not\models \langle\psi,\ \Gamma\rangle\ C\ \langle\phi,\ \Upsilon\rangle$ or one of the cases defined in Fig. 7.4 describes π. $\square$

We explain the conditions in Fig. 7.4 for Def. 7.13. Given $C = C_0 \ldots C_i \ldots C_j \ldots$ or $\overrightarrow{C} \subseteq \overrightarrow{C_0 \ldots C_i \ldots C_j \ldots}$ such that there are collective triples $\langle\psi_i,\ \Gamma_i\rangle\ C_i\ \langle\phi_i,\ \Upsilon_i\rangle$ and $\langle\psi_j,\ \Gamma_j\rangle\ C_j\ \langle\phi_j,\ \Upsilon_j\rangle$ in Tri.

- The **SRV** describes the violation that the state-requirement of C_i fails to hold on a collective pre-state of C_i.
- The **PRV** describes the violation of the path-requirement Γ_i of C_i.
- The **PRV-II** describes the violation of the path-requirement Γ_i of C_i in an execution **in the state space graph of** C due to weakness of the path-requirement Γ of C. The weakness can cause C_i be interfered by a statement following C. Remark that this is a violation that can happen in the state space graph of C instead of where MC^{Δ} searches.

suppose $i < j$ and $\langle \psi_i, \Gamma_i \rangle\ C_i\ \langle \phi_i, \Upsilon_i \rangle, \langle \psi_j, \Gamma_j \rangle\ C_j\ \langle \phi_j, \Upsilon_j \rangle \in \mathsf{Tri}$,

SRV,(**s**tate **r**equirement **v**iolation) :

$$\bigwedge_{p \in [n]} [\![\mathsf{allempty}\ \texttt{\&\&}\ \psi_i]\!]^{\mathsf{spec}}(\mathsf{preCS}(\pi, C_i), p) = \mathbf{F}$$

PRV,(**p**ath **r**equirement **v**iolation) :

$$\pi \models \neg \bigwedge_{\gamma \in \Gamma_i, p \in [n]} [\![\gamma]\!]^{\mathsf{absent}}_{C_i}(\mathsf{preCS}(\pi, C_i), p)$$

PRV-II :

$\exists p, q, r, r' \in [n]$.

$\exists$ `\no \send(`p`, `q`) \after \exit(`p`) \until \exit(`r`)` $\in_\pi \Gamma_i$.

$\neg\exists$ `\no \send(`p`, `q`) \after \exit(`p`) \until \exit(`r'`)` $\in_\pi \Gamma$.

$\mathsf{exit}^{C}_{p,\pi}$ appears before $\mathsf{exit}^{C_i}_{r,\pi}$ on $\pi\ \wedge\ \mathsf{exit}^{C_i}_{r,\pi}$ appears before $\mathsf{exit}^{C}_{r',\pi}$ on π

PRV-III :

$\exists p, q, r, r' \in [n]$.

$\exists$ `\no \send(`p`, `q`) \after \exit(`p`) \until \exit(`r`)` $\in_\pi \Gamma_i$.

$\neg\exists$ `\no \send(`p`, `q`) \after \enter(`p`) \until \enter(`r'`)` $\in_\pi \Upsilon_j$.

$\mathsf{enter}^{C_j}_{p,\pi}$ appears before $\mathsf{exit}^{C_i}_{r,\pi}$ on $\pi\ \wedge\ \mathsf{exit}^{C_i}_{r,\pi}$ appears before $\mathsf{enter}^{C_j}_{r',\pi}$ on π

ITF-II :

$\exists t \in \mathsf{Tr}(\pi).\ \exists p, q, r \in [n]$.

$\neg\exists$ `\no \send(`p`, `q`) \after \enter(`p`) \until \enter(`r`)` $\in_\pi \Upsilon_i$.

$\mathsf{mayInterfere}^{\Delta}_{G}(\mathsf{src}(t), p, q)\ \wedge\ \mathsf{enter}^{C_i}_{p,\pi}$ appears before t on $\pi \wedge$

t appears before $\mathsf{enter}^{C_i}_{r,\pi}$ on π,

where for some $v, \mathsf{mayInterfere}^{\Delta}_{G}(\omega, p, q) = \mathsf{chan}(\omega)(p, q) = \epsilon \wedge$

$\exists t_q \in \mathsf{emanate}_G(\omega).\ \mathsf{act}(t_q) =$ `recv(`v`, `p`)` $\wedge\ \mathsf{proc}(t_q) = q$

PGV-II :

$\exists p, q, r, r' \in [n]$.

$\exists$ `\no \send(`p`, `q`) \after \enter(`p`) \until \enter(`r`)` $\in_\pi \Upsilon$.

$\neg\exists$ `\no \send(`p`, `q`) \after \enter(`p`) \until \enter(`r'`)` $\in_\pi \Upsilon_i$.

$\mathsf{enter}^{C_i}_{p,\pi}$ appears before $\mathsf{enter}^{C}_{r,\pi}$ on $\pi\ \wedge\ \mathsf{enter}^{C}_{r,\pi}$ appears before $\mathsf{enter}^{C_i}_{r',\pi}$ on π

Figure 7.4: Conditions of a contract violation of an execution π explored by $\mathrm{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle}, n)$.

- The **PRV-III** describes the violation of the path-requirement Γ_i of C_i in an execution **in the state space graph of** C due to the weakness of the path guarantee Υ_j of C_j. The weakness can cause C_i be interfered by C_j.
- The **ITF-II** describes the violation that a sub-statement C_k of C such that $k < i$ can be interfered by C_i due to the weakness of the path guarantee of C_i. Remark that $\mathsf{mayInterfere}^{\Delta}_{G}$ is in fact a weaker predicate than $\mathsf{mayInterfere}_{G}$.
- Finally, the **PGV-II** describes the violation of the path guarantee of C in an execution **in the state space graph of** C due to the weakness of the path guarantee of C_i.

We remark that the algorithm $\textsc{Mc}^{\Delta}$ assumes that every collective-style statement sequence specified by a triple in Tri is invoked collectively. Of course, a sound verification system should be able to detect invalid invocations of collective style statement sequences. This is actually one of the reasons why we implemented a contract system on the basis of procedure contracts. It is natural to detect unmatched collective procedure calls. We talk about building such a contract system in §7.3.

Finally, the soundness of $\textsc{Mc}^{\Delta}$ is proved in §7.4.2.

7.3 Procedure Contract System for MiniMP

In this section, we discuss about the challenges in building a verification system based on procedure contracts for MiniMP programs. Instead of general program segments, the system allows the user to only specify contracts to procedures. Those contracts are called *procedure contracts*.

Recall that a program segment has no **`retrun`** sub-statement. But **`retrun`**s are important to procedures. In addition, different procedure calls have different actual parameters. It comes to the adaptation problem. We explain solutions to these problems in §7.3.1. We describe how to integrate collective states to *on the fly* model checking in §7.3.2. In §7.3.3, we briefly talk about the verification of absence assertions and how does the verification affect the partial order reduction problem. Finally, we discuss about the infinite state space problem of model checking in §7.3.4.

7.3.1 Formal Parameters and Return Value

In order to let the syntax of MiniMP procedures accept contracts, we modify the grammar of MiniMP to:

Procedure := **fun** *Identifier* (*List-of-Identifiers*) *Spec?*
{ (*List-of-Identifier*)? ; *Statements* }

Whether a procedure has a contract is optional. Lexically, a contract of a procedure follows the formal parameter declaration. It gives a hint that formal parameters are visible to the contract.

Formal Parameters. For a procedure contract, variables that are visible to it are global variables and formal parameters. Note the local variables of the procedure are invisible to the contract.

However, formal parameters of a procedure f are invisible to the location where f is called. For example, given the following procedure definition with a contract:

```
int f(int x)
requires x > 0;
...
```

Recall the adaptation problem. Suppose we need to infer the behavior of the call `f(a)`, we have to substitute the formal parameter `x` with the actual parameter `a` in the contract of `f`.

A natural solution to this problem follows the semantics of procedure calls. Whenever the contract of a procedure call needs to be evaluated, a frame associated with the procedure will be pushed onto the stack of the corresponding process, just like how is a call executed. Then the contract will be evaluated in the context where formal parameters are visible and have been assigned proper values corresponding to the actual parameters. Finally, after executing the refresh statement, the process pops the frame and continues its execution.

Returned Values. In many cases, procedures return values. Recall that program segments are defined to have no **return** statement. Hence there is no construct

defined in MiniSpec for accessing returned values. For a procedure, returned value carries the most important result delivered by the functionality of the procedure. A contract language without construt for returned values is useless for procedures.

Therefore, we extended MiniSpec by adding `\result` as a specification expression to *SpecExpr*. This expression is borrowed from ACSL. It can only be used in the state-guarantee of a procedure contract to refer the returned value.

Whenever the behavior of a procedure call has to be inferred from its contract, `\result` refers to a fresh new symbol representing the arbitrary returned value. The returned value usually is constrained by the contract. The symbol is then saved as the returning value which will be assigned to a variable, if the call is the right-hand side of an assignment.

Example 7.14. Figure 7.5 shows a MiniMP procedure `exchange` with a contract.

In the procedure, every process sends `a[1]` and `a[n-2]` to its `left` and `right` neighbor, respectively. And, every process receives values from its `left` and `right` neighbors in `a[0]` and `a[n-1]`, respectively.

In the contract, the state-requirement expresses that the "neighbor-relation" is well-defined, i.e., (1) a neighbor of a process must be one of the running processes; and (2) neighbors are mutual, i.e., every process is the `left` (resp. `right`) neighbor of its `right` (resp. `left`) neighbor. Besides, the state-requirement requires that the length of the array `a` must be at least 4. The state-guarantee states that the messages were sent and received by every process correctly, i.e., the returned value of every process will store the value of `a[1]` of its `right` neighbor in the `n-1`-th element, and stores the value of `a[n-2]` of its `left` neighbor in the `0`-th element. The path guarantee states that a process cannot leave the procedure until both of its neighbors enter the procedure.

By the contract, only two elements of `a` are modified by the procedure. Since the returned value is actually `a`, the elements in the returned value excluding `\result[0]` or `\result[n-1]` remain their values in the pre-state. Since the returned value is an arbitrary fresh new symbol constrained by the state-guarantee, without the condition

```
1  fun exchange(var a, n, left, right)
2    requires 0 <= left && left < NPROCS;
3    requires 0 <= right && right < NPROCS;
4    requires \on(right, left) == PID && \on(right, left) == PID;
5    requires 4 <= n && n == len(a);
6    assigns a[0], a[n-2];
7    ensures \on(a[n-2], left) = \result[0] &&
8            \on(a[1], right) = \result[n-1];
9    ensures \forall i; 1 <= i && i < n-1 ==> \result[i] = a[i];
10   ensures \no \exit(PID) \after \enter(PID) \until \enter(left);
11   ensures \no \exit(PID) \after \enter(PID) \until \enter(right);
12 {
13   send(a[1], left);
14   recv(a[n-1], right);
15   send(a[n-2], right);
16   recv(a[0], left);
17   return a;
18 }
```

Figure 7.5: A MiniMP procedure `exchange` and its contract.

at line 9, one can never infer certain values in `\result[1 .. n-2]` from this contract. □

7.3.2 On The Fly Model Checking with Collective States

In §7.1 and §7.2, we described the model checking algorithms Mc and Mc$^{\Delta}$ with the assumption that the full state space graphs have been constructed. However, in reality, many model checkers do not construct the full state space before searching. Instead, they construct the graph along with the searching algorithm. This is called the *on the fly* model checking.

There are two reasons why on the fly model checking usually is more efficient than searching in a constructed full state space. First, the user normally prefers a model checker to stop immediately once it finds an error. Second, with the state-of-art reduction techniques, such as *partial order reduction* (POR), a model checker only

needs to explore a sub-graph of the full state space graph at most of the time. In both cases above, it is ideal to avoid the construction of the full state space graph.

During on the fly model checking, a model checker starts from an initial state and performs *depth first search* (or *breath first search*) along with the transitions. For every newly explored *current state*, properpties of the state will be checked and transitions emanating from the state will be computed. The current state then can be saved as a seen state.

In order to realize a MC^{Δ} based on the fly model checking algorithm, the notion of collective states must be incorporated into the algorithm. The construction of a collective state depends on a path. So the model checker may have to keep track of the relevant informations along with the state exploration. In addition, whether a path contains an error or is infeasible can depend on the contract that is evaluated on the corresponding collective state. Therefore, a collective state shall be constructed as early as possible.

We introduce a special data structure, *collate*, for aggregating informations, on which a collective state depends, along with state exploration. By saving collates in states, a collate is naturally associated with the proper path. Once a collate collects all necessary information for a collective state, the collective state will be constructed immediately. The structure of a collate is showed in Fig. 7.6.

A collate corresponds to a unique collective pre- or post-state of a procedure call. A collate is labeled by an identifier id of a procedure and a tag indicating pre or post. A collate for n processes has n slots, each of which is for holding a process state of a specific process. In addition, a collate maintains a copy of message channels of a state. The message channels will keep being updated since being created until the collate becomes *complete*. A collate is complete if all of its slots are filled.

When a process enters (or exits) a call to a procedure with a contract, it is either the case that this process is the first one that enters (or exits) the instance of the call or it is not the first one. For the former case, a collate will be created to start to collect information of a collective state of the called procedure. For the latter case,

id	tag: pre/post	
slot_0	...	slot_{n-1}
channels		

Figure 7.6: The structure of a collate type object for n processes. id denotes the identifier of the associated procedure, tag indicates whether the collate serves for a collective pre- or post-state. Every slot_i will hold a process state of a process i. The channels stores a copy of message channels, which keeps being updated.

a collate associated with the collective state must have been created. The process will update the existing collate.

On a path, a process can run ahead to execute several procedure calls without another process getting executed. To deal with such cases, we use a *collate queue* to maintain collates during the state exploration. The head of the queue stores the earliest enqueued collate.

We now describe the algorithm of collate queue management. Let f be a procedure annotated with a contract.

- When a process p reaches (resp. leaves) a call to f, iterating the collate queue (from the head). The iteration

  1. stops at a collate c where the slot_p is empty, or,
  2. creates and enqueues a new collate c, if no existsing collate in the queue has its slot_p empty. For c, id is initialized to f, the tag is initialized to pre (resp. post) and the channels is initialized by copying the message channels from the current state.

  Then for c, if its id of c mismatches f, report an error for mismatch of calls to a collective style procedure. Finally, process p saves a copy of its own process state in the current state to slot_p of c. Such a copy is called a *snapshot* of p at the current state.

- When a process p executes a send(v, q) (resp. recv(v, q)) action, the same action will be performed to every collate c in the collate queue that slot_p of c is still empty.

- Once a collate is complete. It is dequeued from the collate queue. A collective state can be constructed from the contents of a complete collate.

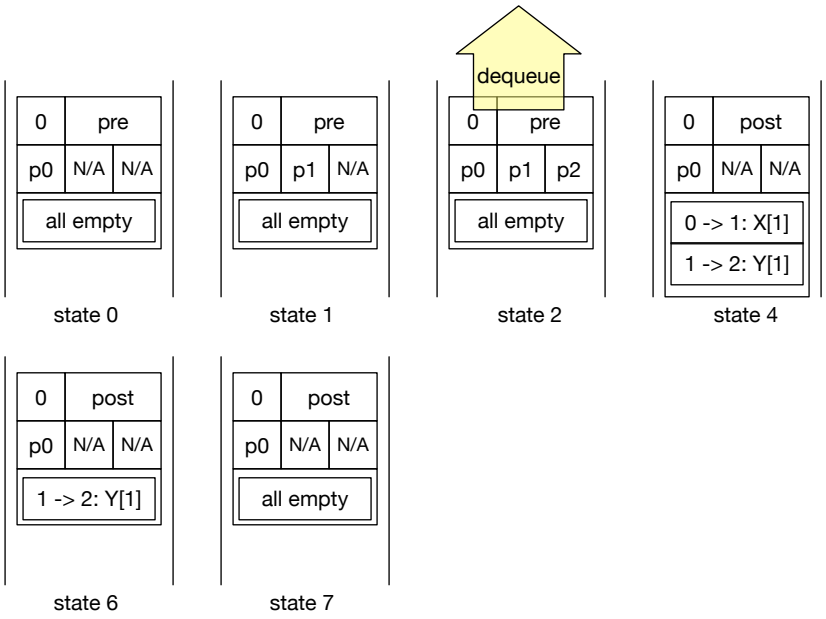

Figure 7.7: Collates in states along the on-the-fly exploration of the execution showed in Fig. 7.3.

This algorithm guarantees that the collective states are constructed as early as possible during the state exploration.

Example 7.15. Figure 7.7 shows the collates, which are associated with the collective pre- and post-states of C_0 of the MiniMP code C_0C_1 given in Example 7.9. Along the on-the-fly exploration of the execution given in Fig. 7.3 (below), process 0 first creates a collate for the collective pre-state of C_0 after state 0, the rest of the processes follow up and complete this collate after state 2. The completed collate is dequeued. Process 0 then runs ahead and creates a collate for the collective post-state of C_0 after state 4. The rest of the processes continue to modified the message channels on the collate until they have copied their snapshots to the collate respectively after state 6 and 7.

7.3.3 Absence Assertion Verification and Partial Order Reduction

Absence Assertion Verification. With collective states, state-requirements and guarantees of procedure calls can be checked on these states. But path-requirements

and -guarantees of procedure calls need to be checked in a different way since their correctness depends on not a single state but a path.

In this subsection, we present a basic algorithm for checking the correctness of path-requirements and -guarantees. The algorithm is a straightforward realization for detecting absence assertion violations defined in Fig. 7.2 and Fig. 7.4.

Given a collate queue. Let c_i be a collate in the i-th position of the queue. c_0 is the head of the queue. Let $\mathsf{pos}(p)$ be the largest position of a collate in the queue where slot_p is non-empty. It implies that slot_p of $c_{\mathsf{pos}(p)+1}$ must be empty, if $c_{\mathsf{pos}(p)+1}$ exists in the queue. Let $\mathsf{arrived}(c)$ denote the process set P such that $p \in P$ iff slot_p in c is non-empty.

Without breaking the properties provided by the collate queue management algorithm given in §7.3.2, we postpone the dequeue operation on a completed collate c, if c is associated with a collective pre-state, until the completion of the collate that is associated with the corresponding collective post-state. Remark that the construction of collective states from completed collates is never postponed.

A procedure annotated with a contract can be said a *contracted* procedure. Since definitions of contracted procedures have been abstracted away. No procedure call can be reached in between a pair of enter and exit of a contracted procedure in an execution. Consequently, if a collate c_i in a queue is associated with a collective pre-state, the collate c_{i+1}, if it exists, in the queue must be associated with the corresponding collective post-state.

Considering the on the fly $\mathrm{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi_g, \Gamma_g\rangle\ \mathsf{body}(g)\ \langle\phi_g, \Upsilon_g\rangle}, n)$ algorithm. Let G be the searching state space graph. Let f, h be some procedures such that

$$\langle\psi_f,\, \Gamma_f\rangle\ \mathsf{body}(f)\ \langle\phi_f,\, \Upsilon_f\rangle,\ \langle\psi_h,\, \Gamma_h\rangle\ \mathsf{body}(h)\ \langle\phi_h,\, \Upsilon_h\rangle \in \mathsf{Tri}.$$

Let π be the execution that is currently being explored and ω the current state. For brevity, we define the following shortcuts

$$\mathsf{arrived}(c - c') \colon \mathsf{arrived}(c) \backslash \mathsf{arrived}(c')$$

$$\mathsf{arrived}(-c) \colon [n] \backslash \mathsf{arrived}(c')$$

The free of path-requirement or -guarantee violation is checked by the following conditions.

- If exists $t \in \mathsf{emanate}_G(\omega)$ such that $\mathsf{proc}(t) = q$ and $\mathsf{act}(t) = \mathsf{recv}(v, p)$, and

  1. $\mathsf{mayInterfere}_G(\omega, p, q)$, and the collate c for the post-state of g exists in the queue in ω, check

     $$\forall p \in \mathsf{arrived}(c).\, \exists r \in \mathsf{arrived}(-c).$$
     $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma_g,$$

     and

  2. $\mathsf{mayInterfere}^{\Delta}_G(\omega, p, q)$, then for every collate c_i in the queue in ω such that $i > \mathsf{pos}(q)$, $\mathsf{id} = f$ and $\mathsf{tag} = \mathsf{pre}$, check

     $$\forall p \in \mathsf{arrived}(c_i - c_{i+1}).\, \exists r \in \mathsf{arrived}(-c_i).$$
     $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\pi \Upsilon_f$$

- If exists $t \in \mathsf{emanate}_G(\omega)$ such that $\mathsf{proc}(t) = p$ and $\mathsf{act}(t) = \mathsf{send}(v, q)$ for some v,

  1. for the first collate c_0 (if exists) in the queue in ω, check

     $$\forall r \in \mathsf{arrived}(-c_0).$$
     $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \notin_\pi \Upsilon_g,$$

     and

  2. for every c_i in the queue in ω such that $i \leq \mathsf{pos}(p)$, $\mathsf{id} = f$ and $\mathsf{tag} = \mathsf{pre}$, check

     $$\forall q \in \mathsf{arrived}(c_i - c_{i+1}),\, r \in \mathsf{arrived}(-c_{i+1}).$$
     $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \notin_\pi \Gamma_f$$

- if $\mathsf{enter}^{\mathsf{body}(f)}_{p,\pi} \in \mathsf{emanate}_G(\omega)$ and exists collate c_i in the queue in ω such that $i = \mathsf{pos}(p) + 1$,

  1. for every collate c_j in the queue in ω such that $j + 1 < i$, $\mathsf{id} = h$ and $\mathsf{tag} = \mathsf{pre}$, check

     $$\forall q \in \mathsf{arrived}(c_j - c_{j+1}),\, r \in \mathsf{arrived}(-c_{j+1}).$$
     $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma_h \Rightarrow$$
     $$(\exists r' \in \mathsf{arrived}(-c_i).$$
     $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \in_\pi \Upsilon_f),$$

     and

2. $\forall q \in [n]$, $r \in \mathsf{arrived}(-c_0)$.

   $\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\pi \Upsilon_g \Rightarrow$
   $(\exists r' \in \mathsf{arrived}(-c_i).$
   $\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \in_\pi \Upsilon_f)$

- If $\mathsf{exit}_{p,\pi}^{\mathsf{body}(g)} \in \mathsf{emanate}_G(\omega)$, let c be the collate for the post-state of g, check

  1. $\forall q \in \mathsf{arrived}(-c_0)$.

     $$\texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(q) \notin_\pi \Upsilon_f$$

  2. for every collate c_i in the queue in ω such that $\mathsf{id} = f$ and $\mathsf{tag} = \mathsf{pre}$,

     $\forall q \in \mathsf{arrived}(c_i - c_{i+1})$, $r \in \mathsf{arrived}(-c_{i+1})$.
     $\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma_f \Rightarrow$
     $\exists r' \in \mathsf{arrived}(-c).$
     $\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r') \in_\pi \Gamma_g$

Partial Order Reduction. The *partial order reduction* (POR) technique is one of the important reasons that the on the fly model checking can be more efficient than searching a constructed state space graph. The basic idea of POR is to let a model checker only search a sub-graph of the complete state space graph by ignoring commutative executions.

Example 7.16. Suppose there are twp processes executing the statement `x = 1;` from a pre-state ω of it. Then there are two executions in the state space graph of `x = 1;` with two processes.

$$\omega \xrightarrow[p_0]{\texttt{x = 1}} \omega_1 \xrightarrow[p_1]{\texttt{x = 1}} \omega' \text{ and}$$
$$\omega \xrightarrow[p_1]{\texttt{x = 1}} \omega_2 \xrightarrow[p_0]{\texttt{x = 1}} \omega'$$

Let assume that the order of transitions in a path cannot affect the properties being verified. A typical POR algorithm determines that one of the executions can be ignored. The ignoring will be achieved by only exploring one of the transitions emanating from ω. □

In general, a POR algorithm is to safely approximate 1) the dependency relation among transitions; 2) the visibility of the transitions to the checking properties. Briefly, a transition $\omega \xrightarrow[p_0]{a_0} \omega_0$ is independent of a transition $\omega \xrightarrow[p_1]{a_1} \omega_1$ if both the following two paths are feasible: $\omega \xrightarrow[p_0]{a_0} \omega_0 \xrightarrow[p_1]{a_1} \omega_2$ and $\omega \xrightarrow[p_1]{a_1} \omega_1 \xrightarrow[p_0]{a_0} \omega_2$. A transition is invisible if the execution of it has no effect on any path predicate that express a checking property.

POR algorithms for message-passing concurrent systems have been studied by many work [92, 98, 100, 109]. For a simple message passing system such as MiniMP, a transition t depends on another transition t' iff t is labeled by a `recv` action, t' is labeled by a `send` action and both transitions operate on a same message. For properties that are expressed as state predicates, no transition is visible.

However, our contract system checks path-requirements and -guarantees, which are expressed by path predicates. To check path predicates, the visibility of transitions must be taken into consideration. Typical POR algorithms determines that for a set of transitions emanating from a state, if there is a transition labeled by a non-communication action, only the transition needs to be explored. However, such a POR algorithm can cause the miss of absence assertion violations. For example, given a procedure f and two processes p and q, from a state where both exiting f by p and entering f by q can be emanated, only one of the transitions needs to be explored. If the model checker chooses the former one, it is fine. But if the model checker chooses the latter one, it misses an execution where process p exits f before the entering of f by q. If f is specified by a path-guarantee

$$\texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(q),$$

a violation of this assertion can be missed.

As aforementioned, POR algorithm performs over-approximation. A safe over-approximation can be as simple as blindly assuming that any transition is visible. But this will cause that the model checker has to explore the full state space. Without loss of safety, we designed the following algorithm for determine whether a transition is visible with improved precision.

Let g be the procedure that is being verified. Let f be a contracted procedure called inside g. Let Γ_g and Υ_g be path requirement and path guarantee of g, respectively. Ditto for Γ_f and Υ_f. Let π be the execution that is currently being explored by a model checker.

- $\mathsf{enter}^{\mathsf{body}(g)}_{p,\pi}$ is visible, if the event `enter(`p`)` appears in any absence assertion in Γ_g.
- $\mathsf{exit}^{\mathsf{body}(g)}_{p,\pi}$ is visible unless all other processes than p have exited g.
- $\mathsf{enter}^{\mathsf{body}(f)}_{p,\pi}$ is visible unless all other processes than p have copied their snapshots to the collate c associated with the collective pre-state of f at state $\mathsf{src}(\mathsf{enter}^{\mathsf{body}(f)}_{p,\pi})$.
- $\mathsf{exit}^{\mathsf{body}(f)}_{p,\pi}$ is visible, if the event `\exit(`p`)` appears in any absence assertion in Γ_f.
- A transition t such that $\mathsf{proc}(t) = p$ and $\mathsf{act}(t) = \mathsf{send}(v, q)$ is visible unless that for every collate c tagged by post in the queue in $\mathsf{src}(t)$, $p \in \mathsf{arrived}(c) \Rightarrow \forall q \in [n].\, \mathsf{arrived}(c)$.

7.3.4 Infinite Reachable States

Finally, we talk about one of the well-known limitations of the model checking and symbolic execution approach: the searching state space must have a finite number of states.

Recall that variables may hold unconstrained symbolic expressions, the validity of branch and loop conditions may not always can be determined. Thus, a model checker may infinitely discover new states for a loop or recursions.

To keep the searching state space stay finite, one solution is to add bounds on some of the symbols that are involved in the branch and loop conditions. In our contract system, symbols are introduced by the $\mathsf{refresh}$ actions and are constrained by the procedure contracts. Hence one can add bounds to symbols via strengthening the procedure contracts. The drawback of this solution is that the contracts may be too strong to be general.

Summarizing the behavior of a loop using a loop invariant can be another solution. A loop invariant can be seen as a specification of a loop. Loop invariants can

be generated automatically during the symbolic execution [61,83,97], or annotated by the user [48].

7.4 Soundness

7.4.1 Soundness of The Monolithic Algorithm

The soundness of MC is expressed by the following theorem.

Theorem 7.17. $\mathcal{M} \models \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ iff for any positive n such that $\text{MC}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n) = \mathbf{T}$

Proof Theorem 7.17 is a consequence of lemma 7.18 and 7.19. □

Let $\mathsf{concretize} : \mathsf{Sym} \to \mathsf{Val}$ be a function that maps symbols to concrete values. Besides, we abuse $\mathsf{concretize}$ by

- letting $\mathsf{concretize}(e)$ denote a concrete value that is obtained by replacing every symbol X in a symbolic expression e with $\mathsf{concretize}(X)$;
- letting $\mathsf{concretize}(\omega)$ denote a state that is obtained by 1) replacing every symbolic expression e in a state ω by $\mathsf{concretize}(e)$; and then 2) getting rid of the path condition;
- letting $\mathsf{concretize}(t)$ denote a transition that is obtained by 1) replacing the states of t with $\mathsf{concretize}(\mathsf{src}(t))$ and $\mathsf{concretize}(\mathsf{dest}(t))$, respectively; and then 2) replacing every symbolic expression e in $\mathsf{act}(t)$ with $\mathsf{concretize}(e)$;
- letting $\mathsf{concretize}(\rho)$ denote a path that is obtained by replacing every transition t in ρ with $\mathsf{concretize}(t)$.

Lemma 7.18. An execution π searched by $\text{MC}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n)$ is feasible iff there is a function $\mathsf{concretize} : \mathsf{Sym} \to \mathsf{Val}$ such that $\mathsf{concretize}(\pi)$ is an execution in the state space graph G of C with n processes, and ψ and $\mathsf{allempty}$ hold on $\mathsf{src}(\mathsf{concretize}(\pi))$.

Proof. We first show that if π is feasible, $\mathsf{concretize}(\pi)$ is in the state space graph of C for some function $\mathsf{concretize}$. Since π is feasible, for any state $\omega \in \mathsf{St}(\pi)$, $\mathsf{PC}(\pi)$ is satisfiable. So for any such ω, $\mathsf{concretize}(\omega)$ is a state in G. Then we need to show that for

any $t \in \mathsf{Tr}(\pi)$, $\mathsf{dest}(t) \in I(\mathsf{src}(t), \mathsf{proc}(t), \alpha)$, where α is the local transition associated with $\mathsf{concretize}(t)$. This can be done by induction on I. We skip the induction.

Now we show the other direction. Let π' be an execution in G such that ψ holds on $\mathsf{src}(\pi')$. We construct a state $\omega_0 = (s, \mathsf{pc})$ by letting s be obtained by assigning every variable v in $\mathsf{src}(\pi')$ with a fresh new symbol X. At the same time, let $\mathsf{concretize}$ be a function that maps every such X to the value of such v in $\mathsf{src}(\pi')$. Let $\mathsf{pc} = \bigwedge_{p \in [n]} [\![\psi]\!](s, p)$. Given a transition $t \in \mathsf{Tr}(\pi)$ and a state ω in the searching space of $\textsc{Mc}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n)$. Let α be the local transition associated with t. We need to show that if $\mathsf{concretize}(\omega) = \mathsf{src}(t)$, then there is $\omega' \in I^{\mathsf{Sym}}(\omega, \mathsf{proc}(t), \alpha)$ such that $\mathsf{PC}(\omega')$ is satisfiable. This can be done by induction on I^{Sym}. We skip the induction for brevity. $\square$

Lemma 7.19. Given a function $\mathsf{concretize} \colon \mathsf{Sym} \to \mathsf{Val}$. Let π and $\mathsf{concretize}(\pi)$ be two executions in the searching space of $\textsc{Mc}^{\Delta}(\mathcal{M}, \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle, n)$ and the state space graph G of C with n processes, respectively. The execution π is feasible. Let λ be an absence assertion, $\pi \models \lambda$ iff $\mathsf{concretize}(\pi) \models \lambda$.

Proof. The interpretation of an absence assertion is about the order of some specific transitions representing the three kinds of events. We show that the order of these specific transitions in π is preserved by $\mathsf{concretize}(\pi)$. let g be a procedure called inside C. For a transition $t \in \mathsf{Tr}(\pi)$,

- if t is labeled by $\mathsf{send}(v, p)$, $\mathsf{concretize}(t)$ is labeled by $\mathsf{send}(\mathsf{concretize}(v), \mathsf{concretize}(p))$;
- if t is $\mathsf{enter}_{p,\pi}^{\mathsf{body}(g)}$ (resp. $\mathsf{exit}_{p,\pi}^{\mathsf{body}(g)}$), obviously $\mathsf{concretize}(t)$ is $\mathsf{enter}_{p,\mathsf{concretize}(\pi)}^{\mathsf{body}(g)}$ (resp. $\mathsf{exit}_{p,\mathsf{concretize}(\pi)}^{\mathsf{body}(g)}$).

Therefore, the order of the specific transitions in π is preserved by $\mathsf{concretize}(\pi)$. $\square$

7.4.2 Soundness of The Composite And Modular Algorithm

The soundness of the algorihtm $\textsc{Mc}^{\Delta}$ can be expressed by the following theorem:

Theorem 7.20. $\mathcal{M} \models \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$, iff for any number n of processes, we have $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle}, n) = \mathbf{T}$.

Proof Let $\{C_0, C_1, \ldots, C_{m-1}\}$ be the set of call statements. For each C_i, $0 \leq i < m$, the procedure f_i of C_i is specified by $\langle \psi_i, \Gamma_i \rangle\ \mathsf{body}(f_i)\ \langle \phi_i, \Upsilon_i \rangle \in \mathsf{Tri}$.

Let $\{S_0, S_1, \ldots\}$ be the set of statement sequences such that no triple in Tri specifies S_j or a sub-statement sequence of S_j, $j \geq 0$. Note S_j can be nothing (i.e., ϵ). Since collective style procedures must be called collectively, we have

$$C = S_0 C_0 S_1 C_1 S_2 \ldots S_{m-1} C_{m-1} S_m$$

or

$$\langle \psi \rangle \overrightarrow{C} \subseteq \overrightarrow{S_0 C_0 S_1 C_1 S_2 \ldots S_{m-1} C_{m-1} S_m}.$$

For either case, we have $\mathcal{M} \models \langle \psi, \Gamma \rangle\ C\ \langle \phi, \Upsilon \rangle$ iff for any number n of processes,

$$\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma \rangle\ S_0 C_0 \ldots S_{m-1} C_{m-1} S_m\ \langle \phi, \Upsilon \rangle}, n) = \mathbf{T}.$$

Then, this theorem is a consequence of the repeated applications of Lemma 7.21 and 7.22. □

Lemma 7.21. Assuming that $\langle \psi_1, \Gamma_1 \rangle\ C_1\ \langle \phi_1, \Upsilon_1 \rangle \in \mathsf{Tri}$, for some statement sequence C_0, we have $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma \rangle\ C_0 C_1\ \langle \phi, \Upsilon \rangle}, n) = \mathbf{T}$ iff

1. exists a contract of C_0 s.t. $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi_0, \Gamma_0 \rangle\ C_0\ \langle \phi_0, \Upsilon_0 \rangle}, n) = \mathbf{T}$, and
2. all four side conditions of the sequence rule

$$\frac{\langle \psi_0, \Gamma_0 \rangle\ C_0\ \langle \phi_0, \Upsilon_0 \rangle, \langle \psi_1, \Gamma_1 \rangle\ C_1\ \langle \phi_1, \Upsilon_1 \rangle}{\langle \psi, \Gamma \rangle\ C_0 C_1\ \langle \phi, \Upsilon \rangle}$$

hold.

Proof We first show that $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma \rangle\ C_0 C_1\ \langle \phi, \Upsilon \rangle}, n) = \mathbf{T}$ implies Condition 1 and 2.

We construct the contract of C_0 as the follow:

- $\psi_0 = \psi$ and $\phi_0 = \psi_1$,

- Γ_0 is the maximum subset of Γ s.t. Υ_1 $\mathsf{guarantees}_\pi$ Γ_0 for every feasible execution π searched by $\text{MC}^\Delta(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C_0C_1\ \langle\phi,\Upsilon\rangle}, n)$,
- $\Upsilon_0 = \mathsf{noSend}(\Upsilon)$.

Such a contract of C_0 makes all the side conditions of the sequence rule for C_0C_1 be satisfied, hence Condition 2 is proved.

We prove Condition 1 by first showing that $\text{MC}^\Delta(\mathcal{M}, \frac{Tri}{\langle\psi,\Gamma_0\rangle\ C_0\ \langle\psi_1,\Upsilon_0\rangle}, n)$ searches in a sub-graph G' of the graph G searched by $\text{MC}^\Delta(\mathcal{M}, \frac{Tri}{\langle\psi,\Gamma\rangle\ C_0C_1\ \langle\phi,\Upsilon\rangle}, n)$.

The G and G' share the same initial state, which is a pre-states of C_0 where ψ and $\mathsf{allempty}$ hold for all processes. Obviously $\mathsf{transStmt}_{\mathsf{Tri}}(C_0, l)$ is a subset of $\mathsf{transStmt}_{\mathsf{Tri}}(C_0C_1, l)$. Then, for any state in G' that is reachable from the initial state over $\mathsf{transStmt}_{\mathsf{Tri}}(C_0, l)$, it must also be in G. Therefore, G' is a sub-graph of G. As a consequence, for an execution π' in G', there must be an execution π in G such that π' is a prefix of π.

Now we show that π' is error-free with the fact that π is error-free by elaborating the kinds of violations checked by MC^Δ.

DL Since π' is a prefix of π and π is free of **deadlock**, π' is free of **deadlock** as well.

SGV $\mathsf{dest}(\mathsf{tail}(\pi'))$ is also the pre-state of C_1 in π. Since π is error free, ψ_1 and $\mathsf{allempty}$ hold for all processes on $\mathsf{dest}(\mathsf{tail}(\pi'))$. Therefore, there is no **SGV** on π'.

ITF Assuming that there is a state $\omega' \in \mathsf{St}(\pi')$ s.t. for some processes $p, q \in [n]$,

> $\mathsf{mayInterfere}_{G'}(\omega', p, q)$ and
> $\forall r \in [n].$`\no \send(`p`, `q`) \after \exit(`p`) \until \exit(`r`)` $\notin_{\pi'} \Gamma_0$,
> and ω' appears before $\mathsf{exit}^{C_0}_{r,\pi'}$ on π'.

Since π' is a prefix of π, ω' is also in π. Note C_1 follows C_0, hence $\mathsf{enter}^{C_1}_{p,\pi} \in \mathsf{emanate}_G(\omega')$. Consequently, $\mathsf{mayInterfere}^\Delta_\pi(\omega', p, q)$. Then there must be

$$\texttt{\textbackslash no \textbackslash send(}p\texttt{, }q\texttt{) \textbackslash after \textbackslash enter(}p\texttt{) \textbackslash until \textbackslash enter(}r\texttt{)} \in_\pi \Upsilon_1.$$

Otherwise, π contains **ITF-II**. This contradicts the assumption Υ_1 $\mathsf{guarantees}_\pi$ Γ_0.

PGV According to the construction of Υ_0, there is only one kind of absence assertions that can be in Υ_0. Given

$$\upsilon = \texttt{\textbackslash no \textbackslash send}(e_0, e_1)\ \texttt{\textbackslash after \textbackslash enter}(e_0)\ \texttt{\textbackslash until \textbackslash enter}(e_2).$$

Note $\upsilon \in \Upsilon_0 \cap \Upsilon$. Assuming $\pi' \models \neg[\![\upsilon]\!]^{\mathsf{absent}}_{C_0}(\mathsf{src}(\mathsf{head}(\pi')), k)$, for some $k \in [n]$. Consequently, $\pi \models \neg[\![\upsilon]\!]^{\mathsf{absent}}_{C_0}(\mathsf{src}(\mathsf{head}(\pi)), k)$, for the same process k. Note for the form of υ, $[\![\ ]\!]^{\mathsf{absent}}_{C_0,\pi} = [\![\ ]\!]^{\mathsf{absent}}_{C_0C_1,\pi}$. It contradicts the assumption that π is error-free.

SRV Assume that there is a sub-statement sequence C_2 of C_0 such that the state-requirement fails to hold on $\mathsf{preCS}(\pi', C_2)$. Since π' is a prefix of π, the same violation happens at $\mathsf{preCS}(\pi, C_2)$ as well. Contradicting the assumption that π is error free.

Similar to the proof of the free of **SRV** in π' above, the free of **PRV**, **PRV-II**, **PRV-III** , **ITF-II** and **PGV-II** in π' can be proved with the same idea. Hence we skip the proof for these kinds of violations.

Now we prove for the other direction, i.e., if Condition 1 and 2 are satisfied, we have $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C_0C_1\ \langle\phi,\Upsilon\rangle}, n) = \mathbf{T}$. We show that a feasible execution π in G is error free via elaborating all kinds of violations.

Since all side conditions of the sequence rule application are satisfied, C_0 cannot be interfered by C_1. Rcall Lemma 6.10, for π, there is an extended execution ζ of C_0 in G such that $\zeta = \pi' \circ \xi_1$, where π' is in G' and ξ_1 is a path emanating from a pre-state of C_1. In addition, by Condition 1, π' is error free.

DL Since C_1 is specified by a triple in Tri, C_1 is translated to a $\mathsf{refresh}$ statement. Thus $C_0C_1 = C_0\mathsf{refresh};$. Since $\mathsf{refresh}$ is independent with any communication statement, it cannot introduce new deadlock.

SGV The final state of π is a post-state of C_1, which is equivalent to $\mathsf{postCS}(\pi, C_1)$. Recall that if ϕ or $\mathsf{allempty}$ fail to hold on $\mathsf{postCS}(\pi, C_1)$, π is infeasible. It contradicts the assumption that π is feasible.

ITF Assuming that there is a state $\omega \in \mathsf{St}(\pi)$ s.t. for some $p, q \in [n]$,

$\mathsf{mayInterfere}_G(\omega, p, q)$, and
$\neg\exists r \in [n].\ \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma$
and ω appears before $\mathsf{exit}^{C_1}_{r,\pi}$ on π.

Transition swapping cannot falsify $\mathsf{mayInterfere}_G$, and C_1 is translated to $\mathsf{refresh}$, so there must be a state $\omega' \in \mathsf{St}(\pi')$ s.t. $\mathsf{mayInterfere}_{G'}(\omega', p, q)$. And we know G' is error free, there must be

$$\gamma_0 = \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_{\pi'} \Gamma_0$$
and ω' appears before $\mathsf{exit}^{C_0}_{r,\pi'}$ on π'. (a)

By our construction of the contract of C_0, $\gamma_0 \in \Gamma$ as well. For the transition $t \in \mathsf{emanate}_G(\omega)$ s.t. $\mathsf{proc}(t) = q$ and $\mathsf{act}(t) = \mathsf{recv}(v, p)$, the order of $\mathsf{exit}^{C_0}_r$ and t is preserved from π to ζ. Thus such $\mathsf{exit}^{C_0}_{r,\pi'}$ can only appear before ω' on π'. Contradicting Condition a.

PGV There are two cases depending on the form of $\upsilon \in \Upsilon$.

1. For some processes $p, q, r \in [n]$,

$$\upsilon = \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\pi \Upsilon,$$

assuming υ is violated on π. It means that

$$t \text{ appears before } \mathsf{enter}^{C_0}_{r,\pi} \text{ on } \pi,$$

where $\mathsf{act}(t) = \mathsf{send}(v, q)$ and $\mathsf{proc}(t) = p$. Both the transitions t and $\mathsf{enter}^{C_0}_{r,\pi}$ can only appear before $\mathsf{exit}^{C_0}_{p,\pi}$ and $\mathsf{exit}^{C_0}_{r,\pi}$, respectively. Hence their order is preserved from π to π'. In addition, by our construction of Υ_0 that $\Upsilon_0 = \mathsf{noSend}(\Upsilon)$, we have that π' fails to satisfy Υ since π and π' share the initial state, where Υ evaluates. Contradicting the assumption that G' is error free.

2. For

$$\upsilon = \texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(q) \in_\pi \Upsilon,$$

assuming π violates υ. By Side Condition 4 of the sequence rule, $\Upsilon \subseteq {}_\pi\Upsilon_0 \cup \Upsilon_1$. Besides, $\Upsilon_0 = \mathsf{noSend}(\Upsilon)$, by our construction for Υ_0. We have $\upsilon \in_\pi \Upsilon_1$. Since π is feasible,

$$\pi \models \bigwedge_{k \in [n]} [\![\upsilon]\!]^{\mathsf{absent}}_{C_1}(\mathsf{preCS}(\pi, C_1), k).$$

It means that p cannot exit C_0C_1 before q entering C_1. It contradicts the assumption that π violates υ, which implies that p exits C_0C_1 even before q entering C_0.

SRV Assuming that there is a sub-statement C_2 of C_0C_1 s.t.

$$\langle \psi_2, \Gamma_2 \rangle\ C_2\ \langle \phi_2, \Upsilon_2 \rangle \in \mathsf{Tri} \text{ and } \psi_2 \text{ fails to hold on } \mathsf{preCS}(\pi, C_2).$$

There are two cases.

1. If C_2 is a sub-statement of C_0, π' inevitably contains **SRV** as well. It contradicts that assumption that G' is error free.

2. Otherwise, $C_2 = C_1$. Then $\psi_2 = \psi_1 = \phi_0$. Since $\mathsf{dest}(\mathsf{tail}(\pi')) = \mathsf{preCS}(\pi, C_2)$, π' contains **SGV**. It contradicts the assumption that G' is error free.

PRV Assuming that there is a sub-statement C_2 of C_0C_1 s.t. $\langle \psi_2, \Gamma_2 \rangle\ C_2\ \langle \phi_2, \Upsilon_2 \rangle \in \mathsf{Tri}$. Let $p, q, r \in [n]$. Suppose

$$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma_2,$$
and $\mathsf{exit}^{C_2}_{p,\pi}$ appears before t and t appears before $\mathsf{exit}^{C_2}_{r,\pi}$ on π,

where $\mathsf{act}(t) = \mathsf{send}(v, q)$ and $\mathsf{proc}(t) = p$. Then there are two cases.

1. If C_2 is a sub-statement of C_0, π' must also contain **PRV**.
2. Otherwse, $C_2 = C_1$. But no transition labeled by a send action can appear after $\mathsf{enter}^{C_1}_{p,\pi}$ on π, since the definition of C_1 has been abstracted away.

Both cases arrive at a contradiction to the assumption.

PRV-II Assuming that there is a sub-statement C_2 of C_0C_1 s.t. $\langle \psi_2, \Gamma_2 \rangle\ C_2\ \langle \phi_2, \Upsilon_2 \rangle \in \mathsf{Tri}$. Let $p, q, r, r' \in [n]$. Suppose

$$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma_2$$
and $\mathsf{exit}^{C_1}_{p,\pi}$ appears before $\mathsf{exit}^{C_2}_{k,\pi}$ and $\mathsf{exit}^{C_2}_{k,\pi}$ appears before $\mathsf{exit}^{C_1}_{r,\pi}$ on π.

But there is no such $r' \in [n]$ that

$$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r') \in_\pi \Gamma$$
and $\mathsf{exit}^{C_2}_{k,\pi}$ appears before $\mathsf{exit}^{C_1}_{r',\pi}$ on π.

There are two cases.

1. C_2 is a sub-statement of C_0. Obviously, the same violation can happen in π' if Γ_0 is not stronger than Γ. By our construction, $\Gamma_0 \subseteq_\pi \Gamma$, which implies $\Gamma_0 \subseteq_{\pi'} \Gamma$ since π and π' share the initial state. Thus π' contains **PRV-II**. It contradicts the assumption that G' is error free.
2. Otherwise, $C_2 = C_1$. But by Side Condition 5 of the sequence rule, $\Gamma_1 \subseteq {}_\pi\Gamma$. Our assumption itself becomes invalid.

PRV-III Assuming that there are sub-statements C_2 and C_3 of C_0C_1 s.t.

- C_2 lexically precedes C_3
- $\langle \psi_2, \Gamma_2 \rangle\ C_2\ \langle \phi_2, \Upsilon_2 \rangle, \langle \psi_3, \Gamma_3 \rangle\ C_3\ \langle \phi_3, \Upsilon_3 \rangle \in \mathsf{Tri}$, and
- Γ_2 may be violated due to the weakness of Υ_3 π.

There are two cases.

1. Both C_2 and C_3 are sub-statements of C_0. Then similar to the proofs above, the assumption will result in a contradiction that π' contains **PRV-III**.
2. Otherwise, C_2 is a sub-statement of C_0 and $C_3 = C_1$. Then, by our construction, $\Upsilon_3(\Upsilon_1)$ $\mathsf{guarantees}_\pi$ Γ_0, it goes back to the case that we haved proved for free of **PRV-II**.

ITF-II Assuming that there is a sub-statement C_2 of C_0C_1 s.t. $\langle \psi_2, \Gamma_2 \rangle$ C_2 $\langle \phi_2, \Upsilon_2 \rangle \in \mathsf{Tri}$. Now suppose that there is $\omega \in \mathsf{St}(\pi)$ s.t. for $p, q \in [n]$, $\mathsf{mayInterfere}^{\Delta}_{\pi}(\omega, p, q)$, and Υ_2 is not strong enough to prevent C_2 from sending a message from p to q that can be received by a transition emanating from ω in an actual execution of C_0C_1.

There are two cases.

1. C_2 is a sub-statement of C_0. With a similar idea to the proofs of previous cases, we can eventually arrive at a contradiction that π' contains **ITF-II**.
2. $C_2 = C_1$. Then, by $\Upsilon_2(\Upsilon_1)$ $\mathsf{guarantees}_\pi$ Γ_0, it goes back to the case that we have proved for the free of **ITF**.

PGV-II Assuming there is a sub-statement C_2 of C_0C_1 s.t. $\langle \psi_2, \Gamma_2 \rangle$ C_2 $\langle \phi_2, \Upsilon_2 \rangle \in \mathsf{Tri}$. Suppose Υ_2 is not strong enough to guarantee that an actual definition C_2 will not violate $\mathsf{noSend}(\Upsilon)$.

There are two cases.

1. C_2 is a sub-statement of C_0. Since $\Upsilon_0 = \mathsf{noSend}(\Upsilon)$, we can derive a contradiction that π' contains **PGV-II** too with a similar idea to the proofs of previous cases above.
2. Otherwise, $C_2 = C_1$. By Side Condition 3 of the sequence rule that Υ_0 and $\Upsilon_2(\Upsilon_1)$ infer_π Υ, the assumption itself is invalid.

□

Lemma 7.22. Given a statement sequence $C_0C_1C_2$ and a triple set Tri. Suppose that $\langle \psi_1, \Gamma_1 \rangle$ C_1 $\langle \phi_1, \Upsilon_1 \rangle \in \mathsf{Tri}$ and there is no triple for any sub-statement of C_2 in Tri. Then, we have

$\textsc{Mc}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma \rangle\ C_0C_1C_2\ \langle \phi, \Upsilon \rangle}, n) = \mathbf{T}$ iff there exists μ, $\Gamma_0, \Upsilon_0, \Gamma_2$ and Υ_2 such that

1. $\textsc{Mc}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle \psi, \Gamma_0 \rangle\ C_0C_1\ \langle \mu, \Upsilon_0 \rangle}, n) = \mathbf{T}$,
2. $\textsc{Mc}(\mathcal{M}, \langle \mu, \Gamma_2 \rangle\ C_2\ \langle \phi, \Upsilon_2 \rangle, n) = \mathbf{T}$, and

3. all the side conditions of the sequence rule $\frac{\langle\psi,\Gamma_0\rangle\ C_0C_1\ \langle\mu,\Upsilon_0\rangle,\langle\mu,\Gamma_2\rangle\ C_2\ \langle\phi,\Upsilon_2\rangle}{\langle\psi,\Gamma\rangle\ C_0C_1C_2\ \langle\phi,\Upsilon\rangle}$ are satisfied.

Proof. Let G be the state space graph searched by $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C_0C_1C_2\ \langle\phi,\Upsilon\rangle}, n)$, G' be the state space graph searched by $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma_0\rangle\ C_0C_1\ \langle\mu,\Upsilon_0\rangle}, n)$ and G'' be the state space graph searched by $\text{MC}(\mathcal{M}, \langle\mu,\Gamma_2\rangle\ C_2\ \langle\phi,\Upsilon_2\rangle, n)$. Similar to what the proof of Lemma 7.21 has showed, for every execution π' in G', there is an execution π in G s.t. π' is a prefix of π.

We first prove that $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C_0C_1C_2\ \langle\phi,\Upsilon\rangle}, n) = \mathbf{T}$ implies Condition 1, 2 and 3. We construct μ, Γ_0, Υ_0, Γ_2 and Υ_2 as the follows

- let μ be the strongest condition s.t. for any post-state ω' of C_0C_1, μ holds on ω' iff ω' is in G'.
- let Γ_0 be the minimal set s.t. for any subset $\Gamma_0' \subset \Gamma_0$, $\text{MC}\Delta(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma_0'\rangle\ C_0C_1\ \langle\mu,\Upsilon_0\rangle}, n) \neq \mathbf{T}$;
- let Υ_0 be the maximal set s.t. for any super set $\Upsilon_0' \supset \Upsilon_0$, $\text{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma_0\rangle\ C_0C_1\ \langle\delta,\Upsilon_0'\rangle}, n) \neq \mathbf{T}$;
- let Γ_2 be the minimal set such that for any $\Gamma_2' \subset \Gamma_2$, $\text{MC}(\mathcal{M}, \langle\mu,\Gamma_2'\rangle\ C_2\ \langle\phi,\Upsilon_2\rangle, n) \neq \mathbf{T}$;
- let Υ_2 be the maximal set such that for any super set $\Upsilon_2' \supset \Upsilon_2$, $\text{MC}(\mathcal{M}, \langle\mu,\Gamma_2\rangle\ C_2\ \langle\phi,\Upsilon_2'\rangle, n) \neq \mathbf{T}$.

We show the validity of Condition 1 by showing that an execution π' in G', which is a prefix of some execution π in G, is error free. Note that π' is naturally free of **SGV** due to the construction of μ. Similarly, it is free of **ITF**, **PGV**, **PRV-II** or **PGV-II** due to the construction of Γ_0 and Υ_0.

DL If π' results in a deadlock, π results in a deadlock too. It contradicts the assumption that π is error free.

SRV Assuming that there is a sub-statement C_3 of C_0C_1 s.t. exists $\langle\psi_3,\ \Gamma_3\rangle\ C_3\ \langle\phi_3,\ \Upsilon_3\rangle$ is in Tri. Suppose π' contains **SRV** for ψ_3, i.e., ψ_3 fails to hold on $\mathsf{preCS}(\pi', C_3)$. Since π' is a prefix of π, π must also contain **SRV** for ψ_3. It contradicts the assumption that π is error free.

The proof of that π' is free of **PRV**, **PRV-III** or **ITF-II** can be constructed with the same idea that if π' contains such error, so does π, which results in a contradiction. Detailed proof for them is omitted for brevity.

Next we show that Condition 2 is valid. A pre-state ω'' of C_2, where μ and $\mathsf{allempty}$ hold, it is equivalent to a post-state ω' of C_0C_1. By the construction of μ, ω' must be in G'. Now let π' be an execution in G' that ends with ω'. If letting all processes continue to execute C_2 from ω', it will results in an execution $\pi' \circ \pi''$ in G. Since $\pi' \circ \pi''$ is in G, if it ends with a post-state of C_2, ϕ and $\mathsf{allempty}$ must hold on that post-state. In addition, since obviously π'' is an execution in G'', π'' cannot contain any violation against Γ_2 and Υ_2 according to their construction. Therefore, a generic execution π'' in G'' must be free of violation with respect to the given contract.

Now we prove Condition 3 by induction on all the side conditions of the sequence rule.

1. We prove Side Condition 1 by contradiction. Since Γ_0 is minimal for G', if we remove $\gamma_0 \in \Gamma_0$ from Γ_0, there is at least an execution π' in G' that contains either **ITF** or **PRV-II**. Let π be a feasible execution in G. Assuming that Υ_2 $\mathsf{guarantees}_\pi$ $\{\gamma_0\}$ is not satisfied. For $p, q, r \in [n]$, γ_0 partially evaluates to

   $$\texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r),$$

   on π while

   $$\texttt{\textbackslash no \textbackslash send}(p,\ q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \notin_\pi \Upsilon_2,$$

   where $r' = r$ or $r' = q$.

   Since Υ_2 is maximal, there is at least one execution π'' in G'' s.t. a global transition t appears before $\mathsf{enter}^{C_2}_{r',\pi''}$, and $\mathsf{proc}(t) = p \wedge \mathsf{act}(t) = \mathsf{send}(v,\ q)$ for some expression v. It means that there is a path for p starting from the entry of C_2 that ends with such t and is independent with process r'.

   - If π' contains **ITF** without γ_0, there must be a state $\omega' \in \mathsf{St}(\pi')$ s.t. $\mathsf{mayInterfere}_{G'}(\omega', p, q)$, and ω' appears before $\mathsf{exit}^{C_0C_1}_{r,\pi'}$ on π'. Since process p can continue to execute and send a message to q without process r' ever executing, C_2 can actually interfere C_0C_1. Let π be an execution in G where the interference happens. Consequently, process q receives more messages from some other process k than p before $\mathsf{exit}^{C_0C_1}_{q,\pi}$ than the number of messages sent by k to q before $\mathsf{exit}^{C_0C_1}_{k,\pi}$ on π. It causes the message channel from

k to q be not empty at $\mathsf{preCS}(\pi, C_1)$. That is π contains **SRV**. Contradicting the assumption that G is error free.

- If π' contains **PRV-II** without γ_0, there must be a sub-statement C_3 of C_0C_1 s.t.

  (a) $\langle \psi_3, \Gamma_3 \rangle\ C_3\ \langle \phi_3, \Upsilon_3 \rangle \in \mathsf{Tri}$ and

  $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_{\pi'} \Gamma_3,$$

  (b) $\mathsf{enter}^{C_3}_{r,\pi'}$ appears before $\mathsf{exit}^{C_0C_1}_{p,\pi'}$ on π', and

  (c) $\mathsf{exit}^{C_0C_1}_{p,\pi'}$ appears before $\mathsf{exit}^{C_3}_{r,\pi'}$ on π'.

  Similarly, starting from $\mathsf{dest}(\mathsf{exit}^{C_0C_1}_{p,\pi'})$, process p can continue to execute and send a message to q without process r' ever executing. Let ρ be such a feasible path in G. Obviously, ρ contains **PRV**. Contradicting the assumption that G is error free.

2. We prove Side Condition 2 by contradiction. Let $\upsilon \in \mathsf{noSend}(\Upsilon)$. Given an execution π in G. Let $p, q, r, x \in [n]$. On the proper state in π, υ is partially evaluated to

   $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r).$$

   Suppose that Υ_0 and Υ_2 $\mathsf{infer}_\pi\ \{\upsilon\}$ is not satisfied, we have

   (1) $\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \notin_\pi \Upsilon_0$, and

   (2) $\texttt{\textbackslash no \textbackslash exit}(x)\ \texttt{\textbackslash after \textbackslash enter}(x)\ \texttt{\textbackslash until \textbackslash enter}(r') \notin_\pi \Upsilon_0$ or $\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(x) \notin_\pi \Upsilon_2$ where $r' = r \vee r' = q$, respectively.

   For Case 1, since Υ_0 is maximal, we have an execution π' in G' that violates υ. Since π' is a prefix of an execution π in G and $\upsilon \in \Upsilon$, π violates Υ. Contradicting the assumption that G is error free.

   For Case 2, since Υ_0 is maximal, there must be an execution π' in G' s.t. $\mathsf{exit}^{C_0C_1}_{x,\pi'}$ appears before $\mathsf{enter}^{C_0C_1}_{r',\pi'}$ on π'. Since Υ_2 is maximal, we know that process p can send a message to q without any dependence on r' in C_2. Therefore, we can let process p continue to execute and send a message to q from the state $\mathsf{dest}(\mathsf{exit}^{C_0C_1}_{x,\pi'})$ without r' executing. Let ρ be such a new feasible path in G. Obviously, ρ violates υ. Contradicting the assumption that G is error free.

3. By Side Condition 2, we know $\mathsf{noSend}(\Upsilon) \subseteq \Upsilon_0$. So we only need to prove by contradiction towards absence assertions in such form

   $$\texttt{\textbackslash no \textbackslash exit}(e_0)\ \texttt{\textbackslash after \textbackslash enter}(e_0)\ \texttt{\textbackslash until \textbackslash enter}(e_1).$$

Let $p, q \in [n]$. Suppose that there is an execution $\pi = \pi' \circ \pi''$ in G s.t. π' is in G' and π'' is in G''. Assuming that

$$\upsilon = \texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(q) \in_\pi \Upsilon,$$

but $\upsilon \notin_\pi \Upsilon_0 \cup \Upsilon_2$.

Since both Υ_0 and Υ_2 are maximal, (1) we have $\mathsf{exit}^{C_0C_1}_{p,\pi'}$ appears before $\mathsf{enter}^{C_0C_1}_{q,\pi'}$ on π'; and (2) process p can exit C_2 without any dependence on process q. If letting process p continue to execute from the state $\mathsf{dest}(\mathsf{exit}^{C_0C_1}_{p,\pi'})$ until it exits C_2 without q executing, it forms a feasible path ρ in G. Obviously, ρ violates υ. We arrive at a contradiction.

4. We now prove that Side Condition 4 is satisfied. Let $p, q, r \in [n]$. There are two cases.

   4.1 Let $\pi' \circ \pi''$ be an execution in G, where π' is an execution in G' and π'' is an execution in G''. We show that if there is an element $\gamma \in_{\pi' \circ \pi''} \Gamma_0$, and $\neg(\Upsilon_2\ \mathsf{cancels}_{\pi' \circ \pi''}\ \gamma)$, , we have have $\gamma \in_{\pi' \circ \pi''} \Gamma$.
   Let γ be partially evaluated on π', to

   $$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r).$$

   Since Γ_0 is minimal, without loss of generality, we assume that there is a state $\omega' \in \mathsf{St}(\pi')$ s.t. $\mathsf{mayInterfere}_{G'}(\omega', p, q)$ and ω' appears before $\mathsf{exit}^{C_0C_1}_{r,\pi'}$ on π'. In addition, we know that $\neg(\Upsilon_2\ \mathsf{cancels}_{\pi' \circ \pi''}\ \gamma)$, so

   $$\texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \notin_{\pi''} \Upsilon_2,$$

   where $r' = r \ \vee \ r' = q$. Since Υ_2 is maximal, we know that process p can exit C_2 without any dependency on process r'. Hence, similar to the proofs for previous cases, we can conclude a path ρ in G that forks from ω' by letting process p continue to execute until exiting C_2 without r' executing. Consequently, $\gamma \in \Gamma$, otherwise ρ contains **ITF**.

   4.2 Let $\gamma \in \Gamma_2$ and $\pi' \circ \pi''$ be an execution in G, where π' is an execution in G' and π'' is an execution in G''. Since Γ_2 is minimal, without loss of generality, π'' can be assumed to contain **ITF** if γ is removed from Γ_2. Then, obviously, if there is no element in Γ that is partially evaluated at the proper state on $\pi' \circ \pi''$ to the same assertion, $\pi' \circ \pi''$ contains **ITF** as well. It will lead to a contradiction.

Now we prove for the other direction, i.e., Condition 1, 2 and 3 imply $\mathrm{MC}^{\Delta}(\mathcal{M}, \frac{\mathsf{Tri}}{\langle\psi,\Gamma\rangle\ C_0C_1C_2\ \langle\phi,\Upsilon\rangle}, n) = \mathbf{T}$. We prove a feasible execution π in G is error free by induction on the cases of violations.

Note that C_0C_1 cannot be interfered by C_1. So we can always apply Lemma 6.10 and 6.12 to executions in G.

DL Assuming that π ends with a state where a **DL** happens. According to Lemma 6.10, there is another feasible execution $\rho' \circ \rho''$ in G, where ρ' and ρ'' are feasible paths in G' and G'' respectively, shares the final state with π. Since the final state is where deadlocked, it is either in π' or π''. In other words, either G' or G'' contains **DL**. It contradicts the assumed Condition 1 and 2.

SGV By Lemma 6.10, there is a feasible execution $\pi' \circ \pi''$ in G s.t. 1) it shares the initial and final state with π; and 2) π' is a feasible execution in G' and π'' is a feasible execution in G''. Since G' is error free, μ and $\mathsf{allempty}$ hold at $\mathsf{src}(\mathsf{head}(\pi''))$. And, G'' is error free, ϕ and $\mathsf{allempty}$ must hold at $\mathsf{dest}(\mathsf{tail}(\pi''))$. Since the final state is shared by π'' and π, no **SGV** in π.

ITF For the execution π in G, assuming that there is a state $\omega \in \mathsf{St}(\pi)$. For $p, q \in [n]$, we have $\mathsf{mayInterfere}_G(\omega, p, q)$ and

$$\neg\exists \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma.$$
$$\omega \text{ appears before } \mathsf{exit}_{r,\pi}^{C_0C_1C_2} \text{ on } \pi \text{ for some } r \in [n].$$

According to Lemma 6.10, there is another feasible execution $\pi' \circ \pi''$ in G sharing initial and final states with π s.t. π' is a feasible execution in G' and π'' is a feasible execution in G''. Swapping transitions cannot falsify $\mathsf{mayInterfere}_G$, so there are two cases.

(a) There is a state ω' in π' s.t. $\mathsf{mayInterfere}_{G'}(\omega', p, q)$. By Side Condition 4 of the sequence rule, there are two sub-cases.

i. $\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \notin_{\pi'} \Gamma_0$. It results in **ITF** on π'. Contradicting the assumption that G' is error free.

ii. Let $\gamma = \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r)$. We have $\gamma \notin_{\pi'} \Gamma_0$ and $\Upsilon_2\ \mathsf{cancels}_\pi\ \gamma$. Since $\omega' \in \mathsf{St}(\pi')$, we can conclude that process q and r have not exited C_0C_1 at state ω on π. By Lemma 6.12, there must be a feasible path ρ in G'' that ends with $\mathsf{exit}_{p,\rho}^{C_2}$ and contains no transition associated with q or r. Obviously, ρ violates Υ_2. Contradicting the assumption that G'' is error free.

(b) There is a state ω'' in π'' s.t. $\mathsf{mayInterfere}_{G''}(\omega'', p, q)$. By Side Condition 4 of the sequence rule, we have

$$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \notin_{\pi''} \Gamma_2$$

So π'' contains **ITF** as well. Contradicting the assumption that G'' is error free.

PGV For any $\upsilon \in_\pi \Upsilon$, by Side Condition 3 of the sequence rule, we have $\upsilon_0 \in_\pi \Upsilon_0$ or $\upsilon_2 \in_\pi \Upsilon_2$. If π violates υ, by Lemma 6.10, we will arrive at a contradiction by finding an alternative execution in G' or G'' that violates υ_0 or υ_2, respectively. Detailed proof is omitted for brevity.

Recall that C_1, as well as any sub-statement of C_1, is not specified by a triple in Tri. Therefore, for any collective state associated with π, there must be an execution π' in G' s.t. an equivalent collective state is also associated with π'. Hence the free of **SRV**, **PRV**, **PRV-III** or **ITF-II** can all be proved with the idea that if π contains such an error, so does π', which contradicts the assumption that G' is error free. I omit the proof for these cases.

PRV-II Let C_3 be a sub-statement of C_0C_1 s.t. $\langle \psi_3, \Gamma_3 \rangle\ C_3\ \langle \phi_3, \Upsilon_3 \rangle \in \mathsf{Tri}$. Suppose for $p, q, r \in [n]$,

$\mathsf{exit}^{C_0C_1C_2}_{p,\pi}$ appears before $\mathsf{exit}^{C_3}_{r,\pi}$ on π and
$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r) \in_\pi \Gamma_3$ and
there is no $\gamma = \texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash exit}(p)\ \texttt{\textbackslash until \textbackslash exit}(r') \in_\pi \Gamma$ s.t. $\mathsf{exit}^{C_3}_{r,\pi}$ appears before $\mathsf{exit}^{C_0C_1C_2}_{r',\pi}$ on π

By Side Condition 4 of the sequence rule, we have $\gamma \notin_\pi \Gamma_0$ or $\Upsilon_2\ \mathsf{cancels}_\pi\ \gamma$.

For the former case, there must exist a feasible execution π' in G' s.t. $\mathsf{exit}^{C_0C_1}_{p,\pi'}$ appears before $\mathsf{exit}^{C_3}_{r,\pi'}$ on π', by Lemma 6.10. Then π' contains **PRV-II**. Contradicting the assumption that G' is error free.

For the latter case, there is

$$\texttt{\textbackslash no \textbackslash exit}(p)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \in_\pi \Upsilon_2,$$

where $r' = r \vee r' = q$. Again, by Lemma 6.10, an execution π'' in G'' can be obtained via swapping transitions in π s.t. $\mathsf{exit}^{C_2}_{p,\pi''}$ appears before $\mathsf{enter}^{C_2}_{r',\pi''}$ on π''. Obviously π'' violates Υ_2. Contradicting the assumption that G'' is error free.

PGV-II Let C_3 be a sub-statement of C_0C_1 s.t. $\langle \psi_3, \Gamma_3 \rangle\ C_3\ \langle \phi_3, \Upsilon_3 \rangle \in \mathsf{Tri}$. For π, suppose $\mathsf{enter}^{C_3}_{p,\pi}$ appears before $\mathsf{enter}^{C_0C_1C_2}_{r,\pi}$, where $p, r \in [n]$. In addition, assuming that

$$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r) \in_\pi \Upsilon$$

but there is no

$$\texttt{\textbackslash no \textbackslash send}(p, q)\ \texttt{\textbackslash after \textbackslash enter}(p)\ \texttt{\textbackslash until \textbackslash enter}(r') \in_\pi \Upsilon_2$$

s.t. $\mathsf{enter}_{r,\pi}^{C_0C_1C_2}$ appears before $\mathsf{enter}_{r',\pi}^{C_3}$ on π, for $q, r' \in [n]$.

By Lemma 6.10, there is a feasible execution π' in G' s.t. π' has the form:

$$\pi' = \ldots \mathsf{enter}_{p,\pi'}^{C_3} \ldots \mathsf{enter}_{r,\pi'}^{C_0C_1C_2} \ldots \mathsf{enter}_{r',\pi'}^{C_0C_1C_2} \ldots$$

By Side Condition 2 of the sequence rule, $\mathsf{noSend}(\Upsilon) \subseteq \Upsilon_0$, so π' contains **PGV-II** as well, which contradicts the assumption that G' is error free.

□

Part III

PRACTICE

Chapter 8

MODELING C/MPI PROGRAMS IN THE CIVL FRAMEWORK

To examine our contract-based verification approach for message-passing, we implemented a prototype for real-world MPI programs.

Compare to MiniMP, MPI programs are much more complicated in that 1) there are hundreds of functions, data types and constants defined in MPI libraries; and 2) client programming languages of MPI libraries, C/C++ and Fortran, are error-prone.

Our implementation is based on a mature verification framework, CIVL, which was designed for verifying general concurrent programs. The CIVL framework is flexible enough for developers to customize it for dealing with specific kinds of concurrency APIs. The author of this dissertation is one of the main developers that built the support of C/MPI in CIVL. The contract system is then built on top of this MPI support for only C programs.

In this chapter, we give a brief introduction for CIVL and its' MPI support, which serve as the background of the contract system implementation. An overview of CIVL is given in §8.1 and a description of the MPI support is presented in §8.2. More details of CIVL can be found in [101]. The work of verifying MPI programs using CIVL in a monolithic way is published as [79].

8.1 CIVL: The Concurrent Intermediate Verification Language

The CIVL verification framework consists of a front-end, an intermediate verification language CIVL-C and a back-end verifier for CIVL-C programs. Figure 8.1 shows the layout of the CIVL framework. CIVL is designed with idea that using CIVL-C to represent various concurrent programming languages and letting the back-end

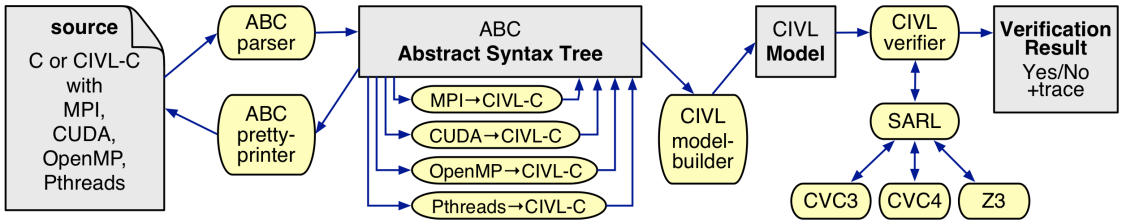

Figure 8.1: The layout of the CIVL framework

verifier focuses only on dealing with the CIVL-C language. The front-end automatically translates programs written in different concurrent programming languages to CIVL-C programs. The back-end verifier uses model checking and symbolic execution techniques for verifying CIVL-C programs. The reasoning of symbolic expressions is carried out by automated theorem provers [12, 33, 38].

The CIVL-C language is an extension of the sequential part of C11 [54] with a large number of primitives for concurrency and specification. In addition, CIVL-C allows nested function definitions, which plays an important role in representing different concurrency models.

We list a selected set of CIVL-C primitives in Fig. 8.2. Every CIVL-C primitive starts with a `$` sign. Fig. 8.3 shows a simple CIVL-C example that utilizes these primitives.

The program in Fig. 8.3 represents a hierarchical concurrency model. The `$parfor` at line 20 spawns 5 "processes", each of which executes the `proc` function with a unique `pid`. Each "process" spawns 5 "threads" to read the `$input` variable and `compute` in parallel (line 11). Each "thread" executes the `thread` function with a different `tid`. The `thread` function is defined inside the scope of the `proc` function so that "process"-local variables `pid` and `result` are also visible to each "thread" (line 9). Every "process" sums up the results of its "threads" to a global variable `sum` that is shared by all processes. The access to `sum` is protected by a `lock`. Each "process" attempts to acquire the `lock` by waiting until it is 0. The evaluation of the `$when` guard (i.e., `!lock`) and the execution of the first statement following `$when` (i.e., `lock`

- `$proc`: a type representing a process
- `$spawn` *`stmt`*: spawns a new process to execute a statement and returns immediately a `$proc` object representing the new process.
- `$parfor (int` *`i`* `:` *`domain`*`)` *`stmt`*: spawns a set of processes, each of which corresponds to an integer in an integral domain and executes the given statement, and waits until all processes terminate.
- `$when (`*`expr`*`)` *`stmt`*: executes the given statement once the guard *expr* evaluates to true; blocks the execution otherwise.
- `$choose_int (`*`n`*`)`: non-deterministically returns an integer in between 0 and *n*-1.
- `$input`: a type specifier that specifies a variable to be a program input, which is non-writable and will be initialized by an arbitrary value by default.
- `$output`: a type specifier that specifies a variable to be a program output, which shall not be read by the original program but will be compared against a specification.
- `$assume (`*`expr`*`)`: informs the verifier to ignore the current execution path unless *expr* evaluates to true.
- `$assert (`*`expr`*`)`: asserts that *expr* evaluates to true.

Figure 8.2: A selected set of CIVL-C primitives

`= 1`) is uninterruptible. The "process" eventually releases the `lock` by setting it back to 0.

Finally, the `$assert` statement (line 21) checks for the correctness of the computation. In CIVL-C, first order logic quantifiers, `$forall` and `$exists`, can be used in the boolean expressions in `$assert` or `$assume` primitives.

8.2 Modeling MPI with Transformation and Libraries

The support of MPI in CIVL is a combination of 1) customized MPI library implementation written in CIVL-C; and 2) a code transformer that takes in C/MPI programs and outputs pure CIVL-C programs.

```
1  $input int IN[5];
2
3  int compute(int x, int y) { ... }
4
5  int sum, lock = 0;
6  void proc(int pid) {
7    int result[5];
8
9    void thread(int tid) {result[tid] = compute(IN[pid], tid);}
10
11   $parfor (int i : 0 .. 4) thread(i);
12   $when (!lock) {
13     lock = 1;
14     for (int i = 0; i < 5; i++) sum += result[i];
15     lock = 0;
16   }   ...
17 }
18
19 int main() {
20   $parfor(int i : 0 .. 4) proc(i);
21   $assert (sum == ... );
22 }
```

Figure 8.3: A CIVL-C representation of a hierarchical concurrency memory model.

8.2.1 The MPI Library

CIVL's MPI library implementation covers a subset of the MPI constructs, including standard point-to-point communication functions, all blocking collective functions and other constructs such as `MPI_Comm_dup` and `MPI_Init_thread`.

Every MPI construct was carefully modeled by CIVL-C code with the help of the core libraries in CIVL. The most important core library provided by CIVL for the MPI library is the `comm` library. It consists of a set of basic primitives for communication in a message-passing manner. Here we use the CIVL-C implementation of the `MPI_Recv` function as an example to demonstrate how does CIVL model MPI. Fig. 8.4 presents CIVL's definition of `MPI_Recv`, which is the standard point-to-point receive operation in MPI.

```
1  int $mpi_recv(void *buf, int count, MPI_Datatype datatype, int src,
2                int tag, MPI_Comm comm, MPI_Status *status) {
3    $message in;
4    int place = $comm_place(comm.p2p);
5    int nprocs = $comm_size(comm.p2p);
6
7    $assert((src >= 0 && src < nprocs) || src == MPI_ANY_SOURCE,
8            "Illegal MPI message receive source %d.\n", src)
9    $assert(tag == -2 || tag >= 0,
10           "Illegal MPI message receive tag %d.\n", tag);
11   $elaborate(src);
12   in = $comm_dequeue(comm.p2p, src, tag);
13
14   int size = count*sizeofDatatype(datatype);
15
16   $message_unpack(in, buf, size);
17   if (status != MPI_STATUS_IGNORE) {
18     status->size = $message_size(in);
19     status->MPI_SOURCE = $message_source(in);
20     status->MPI_TAG = $message_tag(in);
21     status->MPI_ERROR = 0;
22   }
23   return 0;
24 }
25
26 int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src,
27              int tag, MPI_Comm comm, MPI_Status *status) {
28   $assert(_mpi_state == _MPI_INIT, "MPI_Recv() cannot be invoked "
29     "without MPI_Init() being called before.\n");
30   return $mpi_recv(buf, count, datatype, src, tag, comm, status);
31 }
```

Figure 8.4: The definition of `MPI_Recv` in the MPI library in CIVL.

The function is defined at line 26–31. The argument `buf` points to a memory location where the received data will be saved when the function returns. The expected size of the message data is given by `count` and `datatype`. The argument `src` specifies the rank of the sender, `tag` specifies the message tag and `comm` specifies the MPI communicator. The `status` can be a pointer to a `MPI_Status` object which will be filled after the function returns or a constant `MPI_STATUS_IGNORE` for opting out the returned information.

In this function, the implementation first checks whether `MPI_Init` has been called then calls `$mpi_recv` to deliver the main functionality.

The `$mpi_recv` function is built upon the `comm` library:

- `$comm:` a type representing a reference to a communication universe. A communication universe mainly consists of a group of n processes and n^2 message channels. Each message channel is identified by an ordered pair (i, j), $0 \leq i, j < n$, which means that the message channel buffers messages sent from process i to j. An MPI communicator is implemented in CIVL as two communication universes, one for point-to-point communication and the other for collective communication. Therefore, the `MPI_Comm` type is implemented as a structure including two `$comm` fields, namely `p2p` and `col`, respectively.

- The pair of `$comm_place` and `$comm_size` functions are the "getters" that return the *pid* of the running process and the size of the communication universe, respectively, through a `$comm` reference.

- The function `$comm_dequeue($comm c, int src, int tag)` dequeues a message from the message channel of the communication universe referred by `c`. The message channel is identified by the ordered pair (*pid*, `src`), where *pid* is the ID the process in the communication universe. The message is tagged by `tag`. If there is no message matching the `tag` in the given channel, this function blocks the execution. The value of `tag` can be a constant pre-defined in MPI library, `MPI_ANY_TAG`, which matches any message tag.

- Messages are represented by a `$message` type. In addition to the data, a message includes meta information such as IDs of the sender and receiver, message tag and data size.

- A number of "getter" functions are provided for reading information from messages, e.g. `$message_unpack` reads the message data and `$message_source` reads the ID of the sender from a message.

With these infrastructures provided by the `comm` library, the `$mpi_recv` can be implemented with straightforward CIVL-C code as showed from line 1-24. Being different from real MPI implementations, CIVL's implementation is simple and deterministic.

Recall that CIVL is a symbolic execution tool, so the values of variables are not necessarily concrete. But the `$comm_dequeue` function requires the `src` to be concrete. To solve this problem, the `$elaborate(src)` statement was inserted at line 11 to inform the verifier to elaborate different concrete cases of the value of `src`.

8.2.2 AST-Level Code Transformation

In addition to the MPI library, CIVL applies an *Abstract Syntax Tree* (AST) level code transformer to automatically convert an C/MPI program to an equivalent CIVL-C program. The transformation approach can be generalized as a template, showed in Fig. 8.5.

For an MPI program, every process runs a copy of it and has no shared storage. Recall that processes in CIVL-C are defined as statements. And a variable is shared by two processes, if the variable is visible to the definitions of both the processes.

One of the most notable changes made by the transformer is that the whole original program, as well as the inlined libraries, is wrapped by a function that will be executed by a process. By doing so, there is no variable in the original program that is shared by processes. In our template in Fig. 8.5, the function is named `_mpi_process` (line 8-16) for the meaning of that it will be executed by every "MPI process".

The original `main` function is renamed to `_civl_main` (line 13). The transformer creates a new `main` function that is responsible for creating a set of processes to execute the `_mpi_process`.

Without affecting the message-passing concurrency model, the transformer creates a set of global variables, including `$inputs` and data structures representing the communication environment.

a generic MPI program:

```
int main(int argc, char * argv[]) {
  [body of main function]
}
```

---

the CIVL-C program after translation:

```
$input int _mpi_nprocs;
$input int _mpi_nprocs_lo = 1, _mpi_nprocs_hi, _civl_argc;
$input char _civl_argv[_civl_argc][];
$scope _mpi_root = $here;
$assume(_mpi_nprocs_lo <= _mpi_nprocs &&
        _mpi_nprocs <= _mpi_nprocs_hi);
$mpi_gcomm _mpi_gcomm_world, _mpi_gcomms[]; // global communicators:
void _mpi_process(int _mpi_rank) {
  MPI_Comm MPI_COMM_WORLD = $mpi_comm_create($here, _mpi_gcomm,
                                             _mpi_rank);
  [more definitions of MPI functions]
  [insert original source code here, but rename main _civl_main]
  _civl_main(_civl_argc,
             (char * [_civl_argc])$lambda (int i) _civl_argv[i]);
  $mpi_comm_destroy(MPI_COMM_WORLD);
}
int main() {
  _mpi_gcomm_world = $mpi_gcomm_create(_mpi_root,  _mpi_nprocs);
  $seq_init(&_mpi_gcomms, 1, &_mpi_gcomm_world);
  $parfor (int i: 0 .. _mpi_nprocs - 1) _mpi_process(i);
  $mpi_gcomm_destroy(_mpi_gcomm_world);
}
```

Figure 8.5: The general C/MPI to CIVL-C transformation template.

The number of processes, as well as its bounds, are defined as `$input` variables: `_mpi_nprocs`, `_mpi_nprocs_lo` and `_mpi_nprocs_hi`. Usually the user only needs to give a concrete value to the upper bound variable from the command line. The lower bound is by default one. The original program arguments are also transformed to `$input` variables: `_civl_argv` and `_civl_argc`.

Other than the `$input` variables, MPI communicators are declared globally. The `$mpi_gcomm` type represents an MPI communicator. The `_mpi_gcomm_world` variable represents the default communicator, which is referred by `MPI_COMM_WORLD`. In addition, there is a *sequence* `_mpi_gcomms` that manages dynamically created communicators.

A sequence is a CIVL-C primitive that performs as a dynamically resizable array. A sequence of T type variable is declared as an incomplete array of T. A sequence must be initialized by `$seq_init` (e.g., line 19) before use.

The creation of the default communicator is done by the new `main` function. Each process is responsible for creating its process-local `MPI_Comm` reference to the default communicator.

All these global variables are created by the transformer hence they are invisible to the original program. There is no direct access from the original code to any global variable. Indirect accesses to global data structures, such as buffering a sending message into a message channel, are all operated through specific handles. These handles guarantee that "MPI processes" cannot be aware of the existence of any global variable.

8.2.3 MPI Program Properties

We discuss about the quality of the MPI model in CIVL in terms of the properties of an original MPI program that are preserved by the transformed CIVL-C program.

These properties include a wide range of standard C properties that CIVL can verify for an MPI program such as assertion violation, improper pointer dereferencing or out-of-bound array access, division by zero, memory leak, etc.

One of the most common errors in MPI programs is deadlock. There are two kinds of deadlocks: *absolute deadlock* and *potential deadlock*. Absolute deadlocks are caused by the existence of unmatched send and receive operations. Potential deadlocks are caused by the short of message buffers. Potential deadlocks are hard to re-produce because buffer sizes can vary in different runs. CIVL can verify the freedom of both kinds of deadlocks.

Other MPI specific properties that can be verified by CIVL include: the freedom of receive buffer overflow, incompatible message and receive types, inconsistency of collective calls.

The functional correctness of an MPI program can be specified as either assertions or a separate sequential program, which is simple, trusted and functionally equivalent to the MPI program. For the latter manner, CIVL is able to check the functional equivalence between two programs, which is carried out by assuming the two programs have exact same inputs and verifying that they always produce the exact same outputs.

Chapter 9

VERIFYING C/MPI PROGRAMS WITH FUNCTION CONTRACTS

The function contract approach introduced in Part II for MiniMP is an instance of the "divide and conquer" methodology that has several advantages over monolithic model checking approaches. First, it divides a large problem into a number of smaller independent problems. Since each smaller problem is much easier to be solved, there is a potential that the "divide and conquer" approach can help mitigating the state explosion problem. Second, a function contract, which serves as a specification of a function, enables the verifier to cover the most general cases of the function with respect to the specification. Whereas whether a function can be throughly verified by a monolithic approach depends on how is the function called in a program. Finally, writing function contracts encourage developers to pay more attention to specifications and documents.

Based on the MPI model described in Chapter 8, our discussion in this chapter will focus on implementing a function contract system for MPI in CIVL. The rest of the chapter is structured as follows: §9.1 discusses about the challenges in applying the function contract approach to MPI programs and sketches the solutions; §9.2 introduces the contract language for C/MPI programs; §9.3 describes a source code transformation based implementation for the MPI contract system.

9.1 Bring Function Contracts From MiniMP To MPI

In Part II, we have discussed about the contract approach for message-passing programs around a toy language, MiniMP. Applying the approach to C/MPI programs would face many new challenges because of the complexity of both MPI and C. In the rest of this section, we briefly talk about these challenges and their solutions.

Multi-Communication Universes. A MiniMP program only involves a single communication universe that includes all the processes launched at the beginning. However, MPI programs can have multiple communication universes. A communicator conceptually contains two, one for point-to-point and the other for collective communication. Besides, communicators can be dynamically created. A process can participate in different communicators. Different communicators can have different sizes. A collective-style function in an MPI program may involve multiple communicators. Hence MiniMP's specification language cannot be proper for MPI since the constants `PID` and `NPROCS` have different values in different communication universes.

Consequently, the reasoning of the contracts must be sensitive to communication universes. For example, given two consecutive calls to collective-style functions $f\texttt{();}g\texttt{()}$, a guarantee

```
\no \send(p, q) \after \enter(p) \until \enter(r)
```

of g does not make g satisfy the requirement

```
\no \send(p, q) \after \exit(p) \until \exit(r)
```

of f, if the contracts are describing behaviors for two different communication universes.

The challenge of multiple communication universes demands that communications in different universes shall be reasoned independently. Within a single universe, the theory for MiniMP is still applicable to MPI.

Message Tags. In MPI, messages in point-to-point communication are always labeled with an integer tag. When a process sends or receives a message, it must specify a tag for the message. When receiving a message, one can specify `MPI_ANY_TAG` for the message tag. A receive operation on a specific message channel with `MPI_ANY_TAG` will get the earliest message in it (i.e., the head of the FIFO queue).

The existence of the message tags makes the communication in MPI more flexible while the reasoning of the communication more complicated. The MiniMP specification language has no notion for message tags hence it cannot specify absence assertion precisely in many cases.

```
1 void f() {
2   if (rank == 0)
3     for (int i = 1; i < size; i++)
4       MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD);
5   else
6     MPI_Send(&value, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
7 }
```

Figure 9.1: An MPI collective-style function that requires any following statement not to send any message with tag 1 to process 0 until process 0 exits it.

Therefore, to specify MPI programs, the absence assertions need to be extended with message tags.

Example 9.1. Considering the collective-style function in Fig. 9.1, process 0 uses the wildcard receive to gather messages from the rests. In order to prevent this function being interfered by following statements, the function contract must have a strong enough requirement stating that no process shall send a message to process 0 after it exits `f` until process 0 exits `f`. But this requirement is too strong. In fact, `f` only requires that no process shall send a message tagged by 1 to process 0. The MiniMP specification language has no notion of message tags hence cannot specify such a precise requirement for `f`. □

The complexity of the C/MPI APIs. The MPI library provides a large number of functions and datatypes. It will not be a surprise that MPI has more complicated APIs than the communication operations of MiniMP. This dissertation only focuses on a subset of them: blocking point-to-point and collective communication functions. For this subset, the complexity of the APIs is mainly attributed to MPI's unique type system and the relatively lower-level features in C (i.e., pointer and memory management). We use an MPI collective function `MPI_Bcast` as an example to illustrate these two aspects.

The `MPI_Bcast` has the following signature:

```
int MPI_Bcast(void * buf, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm).
```

`MPI_Bcast` must be called collectively by all processes, among which a process is designated to be the "root". The "root" process broadcasts a message to all other processes. Every non-"root" process receives the message. This function has five formal parameters:

- `buf` is a pointer to the memory location where, for the "root" process, the sending data occupies, and for the non-"root" processes, the receiving data will be saved.
- `count` and `datatype` together specify the expected *type signature* of the data in the message. All processes must agree on the type signature. A type signature determines the bytewise size of the data, i.e., the product of the `count` and the extent of the `datatype`. The `MPI_Datatype` is an enumeration, in which most enumerators refer to language specific data types, such as that `MPI_INT` refers to `int` in C.
- `root` specifies the rank of the process that will be the "root" process.
- `comm` specifies the communicator that is associated with this operation.

In C, one cannot access data through a `void` pointer such as the `buf` in our example. The `buf` must be converted to a pointer to a non-void T type before being used to access the data. The T can be determined by elaborating different cases of the `datatype`. However, elaboration is quite expensive when the value of the `datatype` is arbitrary. Moreover, under the weakest assumption, the data referred by `buf` is also arbitrary. One needs to find a representation for the arbitrary data of arbitrary type in CIVL.

To deal with such complexity, we use an uninterpreted function $\mathit{extent}(\mathit{datatype})$ to represent the bytewise extent of a *datatype*. A simple theory for the *extent* function is that for an `MPI_Datatype` d and a C type t, $\mathit{extent}(d) == \texttt{sizeof}(t)$ if and only if d corresponds to t.

Then an arbitrary data, of which the size is given by a pair of `MPI_Datatype` d and count c can be represented as an arbitrary `char` array of length $\mathit{extent}(d) \times c$.

The Complexity of C language. Finally, C language itself is a complicated language, which has various powerful but also error-prone features such as multi-level pointers, type casting, memory allocation, etc. A contract system for C/MPI programs must first be able to specify and verify C programs.

```
1  /*@ requires \valid(a) && \valid(b);
2      assigns \nothing;
3      behavior GT:
4        assumes *a > *b;
5        ensures \result == *a - *b;
6      behavior LTE:
7        assumes *a <= *b;
8        ensures \result == *b - *a;
9  */
10 int diff(int *a, int *b) {
11   if (*a > *b) return *a - *b;
12   else return *b - *a;
13 }
```

Figure 9.2: A sequential C function with an ACSL contract.

We introduce the contract language for C/MPI programs in the next section. The language is an extension of an existing specification language for sequential C programs. Hence instead of designing a contract language for C programs from scratch, we choose to add MPI features to a mature specification language.

9.2 The MPI Contract Language

We designed the contract language for C/MPI based on an existing specification language ACSL, which is for specifying sequential C programs. ACSL provides rich primitives for describing program behaviors and constructing logic proofs. An ACSL function contract is a pair of pre- and post-conditions that are in the form of a C program annotation. We extended ACSL with a few primitives for MPI and concurrency.

Example 9.2. Figure 9.2 presents a simple example of a C function with its ACSL function contract. The `requires`, `assigns`, `ensures` clauses also appear in MiniMP specification language and they remain their meanings in ACSL. Ditto for `\result`. The `\nothing` expression stands for an empty set of memory location. A `\valid(pts)` construct asserts that it is safe to dereference a pointer in the pointer set `pts`. A `behavior` that is identified by a *name* specifies a pair of pre- and post-conditions under

a specific assumption. The `assumes` clause in a behavior introduces the assumption. □

In this section, we mainly describe our extension for MPI to the ACSL. Original ACSL constructs will be explained when they are involved.

9.2.1 Syntax

The syntax of the MPI contract language is given in Fig. 9.3. An MPI function contract is a set of *behaviors*. There are two kinds of behaviors in ACSL, the *default behavior* and *named behaviors*. The default behavior is the set of *clauses* before any named behaviors. A named behavior starts with the keyword `behavior` and is identified by a *name*. A named behavior also consists of a set of clauses.

```
Contract            := Behaviors CollectiveBehaviors
Behaviors           := BehaviorBody NamedBehavior*
CollectiveBehaviors := \mpi_collective (Expr, CollectiveKind): Behaviors
NamedBehavior       := behavior Identifier : BehaviorBody
CollectiveKind      := P2P | COL
BehaviorBody        := Clause*
```

Figure 9.3: The syntax of the behaviors of the MPI contract language.

Our extension adds two higher-level structures to ACSL behaviors: *local behavior* and *collective behaviors*. The local behavior in a function contract is the set of ACSL behaviors before any collective behavior. A collective behavior starts with the keyword `\mpi_collective`, which is followed by two parameters: an expression of `MPI_Comm` type and a constant denoting the *collective kind*. The former one refers to an MPI communicator. The latter one can either be `P2P`, which stands for "point-to-point", or `COL`, which stands for "collective". These two parameters together identify a communication universe. A collective behavior consists of a set of ACSL behaviors as well.

Clauses in the MPI contract language are same as the ones in ACSL. For expressions, a few new constructs for MPI and concurrency are added to ACSL. The syntax of these new expressions is given in Fig. 9.4.

```
Expr       := ACSL-Expr | MPIExpr | \on(Expr, Expr)
MPIExpr    := \mpi_comm_rank | \mpi_comm_size | \mpi_extent(Expr)
            | \mpi_offset(Expr, Expr, Expr)
            | \mpi_region(Expr, Expr, Expr)
            | \mpi_valid(Expr, Expr, Expr)
            | \mpi_agree(Expr)
            | \mpi_equals(Expr, Expr)
            | \mpi_reduce(Event, Event, Event)
            | \absentof Event \after Event \until Event
Event      := \sendto(Expr, Expr) | \sendfrom(Expr, Expr)
            | \enter(Expr) | \enter
            | \exit(Expr) | \exit
```

Figure 9.4: The syntax of the expression of the MPI contract language. The *ACSL-Expr* stands for the expression syntax of ACSL.

Recall the syntax of MiniSpec in §5.3, **\on** is not allowed to be used in absence assertions. Following the same spirit, we remark that MPI contract language restricts the expressions in absence assertions to be process-local, i.e., no "\on" can appear in an absence assertion.

9.2.2 Semantics

A local behavior specifies the *process-local properties*. A collective behavior specifies *global properties* of a collective-style function. A global property may involve multiple processes and must be satisfied by all processes. Absence assertions in a collective behavior are independent with ones in another collective behavior if the two collective behaviors are associated with different communication universes.

Furthermore, there are two implicit assertions in every collective behavior: 1) a state-requirement asserting that the message channels in the associated communication universe are all empty and 2) a post-condition asserting the same property.

The interpretation of a local behavior is based on the process state of the process that reaches the entry or exit of a contracted function. Collective behaviors in theory shall be interpreted in a similar way as MiniMP contracts with respect to collective states. In fact, the interpretation of MPI contracts is simpler: a state-requirement

```
1  int rank, size, data, buf;
2   ...
3  /*@
4  \mpi_collective(MPI_COMM_WORLD, P2P):
5    requires rank == \mpi_comm_rank && size == \mpi_comm_size;
6    assigns buf;
7    ensures \on((rank + 1) % size, buf) == data &&
8            \absentof \exit \after \enter \until \enter((rank + 1) % size));
9  */
10 int ring() {
11   int left = (rank + size - 1) % size;
12   int right = (rank + 1) % size;
13   MPI_Sendrecv(&data, 1, MPI_INT, left, 0,
14                &buf, 1, MPI_INT, right, 0, MPI_COMM_WORLD);
15 }
```

Figure 9.5: A collective-style MPI function `ring` with a contract.

or -guarantee is still evaluated on corresponding collective pre- or post-state; a path-requirement or -guarantee is partially evaluated at the source state of a global transition representing the entering of the corresponding function by a process. This simplification is correct because only process-local expressions are allowed to be used in absence assertions.

Meanings of the constructs for MPI and concurrency are informally described as the follows:

- `\on(rank, expr)` represents the value of the expression `expr` that is evaluated by a process of rank `rank`.
- `\mpi_comm_rank` is a constant that stands for the rank of the running process.
- `\mpi_comm_size` is a constant that stands for the size of the communicator.
- `\mpi_extent(datatype)` represents the bytewise size of the `MPI_Datatype datatype`.
- `\mpi_offset(buf, count, datatype)` is such a pointer:

  ```
  (char *)buf + count * \mpi_extent(datatype)
  ```

- `\mpi_region(buf, count, datatype)` represents the value in a memory region that spans from the address `buf` to the length of `count * \mpi_extent(datatype)` bytes.
- `\mpi_agree(expr)` asserts that all processes must have the same value for `expr`.

- `\mpi_valid(buf, count, datatype)` asserts that the memory region spanning from the address `buf` to the length of `count * \mpi_extent(datatype)` bytes is accessable.
- `\mpi_equals(`r_0`, `r_1`)` asserts that values in the two given memory regions r_0 and r_1 are the same.
- `\mpi_reduce(buf, count, datatype, op)` represents the reduction value of the operation `op` over

  `\mpi_region(\on(0, buf), count, datatype)`,
  `\mpi_region(\on(1, buf), count, datatype)`,
  ...,
  `\mpi_region(\on(n-1, buf), count, datatype)`,

  where `op` has `MPI_Op` type and $n =$ `\mpi_comm_size`.
- `\absentof` θ_0 `\after` θ_1 `\until` θ_2 represents an absence assertion, where θ_0, θ_1 and θ_2 are events.
- `\sendto(dest, tag)` is an event representing the transitions that the running process sends a message with `tag` to a process of rank `dest`.
- `\sendfrom(src, tag)` is an event representing the transitions that a process of rank `src` sends a message with `tag` to the running process.
- `\enter(rank)` is an event representing the transitions that the process of `rank` enters the specified function. Optionally, `\enter` is a shortcut for `\enter(\mpi_comm_rank)`.
- `\exit(rank)` is an event representing the transitions that the process of `rank` exits the specified function. Optionally, `\exit` is a shortcut for `\exit(\mpi_comm_rank)`.

Names of clauses (i.e., state-requirement and -guarantee, path-requirement and -guarantee) in MPI contract language are inherited from MiniMP. The Restriction 5.9 on absence assertions in MiniMP is also inherited by MPI contract language with a natural extension.

Example 9.3. The function contract in Fig. 9.5 consists of a collective behavior that is associated with the point-to-point communication universe of the default MPI communicator. The collective behavior requires that for every process, the `rank` and `size`

shall hold the rank of the process in `MPI_COMM_WORLD` and the size of `MPI_COMM_WORLD` respectively. The contract states that no memory location of any process will be modified in `ring` except `buf`. The contract ensures that on the state where all processes collectively exit `ring`, for every process, the value of `buf` equals to the value of `data` on its right neighbor. Finally, `ring` guarantees that a process will not exit `ring` until its right neighbor enters `ring`. □

9.3 MPI Function Contract System Implementation

The implementation of the MPI function contract system in CIVL is a solution to the following 4 problems: 1) CIVL is designed for verifying complete programs monolithically hence cannot verify functions separately; 2) the semantics of the contract system is different from the one of CIVL-C, e.g., a call to a function with a contract shall be summarized only from the contract instead of the function definition; 3) there is no existing primitive in CIVL-C that is close to collate or collective states; 4) CIVL does not support path predicates.

Problem 1 and 2 are tackled by AST-level code transformation, and Problem 3 and 4 are solved by implementing CIVL-C libraries for collates and absence assertions. Both the two methods are in keeping with the philosophy of CIVL, which is to support a small verification kernel well, and do the rest of the work by customized libraries and source code transformation.

In the rest of this section, we first describe the collate library in §9.3.1, then explain the method of reasoning about absence assertions in §9.3.2. Finally, in §9.3.3, we show that how does source code transformation converts an MPI program to a CIVL-C program, which can be verified by the common CIVL verifier with respect to the contract system semantics.

9.3.1 Implementing Collates

One of the extensions the contract system made to CIVL is a `collate` library. This library provides data types and functions to realize the collate-based algorithm

for constructing collective states. Like the MPI library in CIVL, the `collate` library hides the details that a collate queue is in fact a shared object from the clients—MPI processes. Therefore, conceptually the `collate` library introduces no shared object.

In the library, a collate queue is implemented by a data structure, `$gcollator`. A collate queue is associated with a group of processes in a communicator. Each process owns a process-local handle, `$collator`, to the collate queue. Operations on the collate queue can only be performed through the handle. A collate in the queue is implemented as a data structure, `$gcollate_state`. Similarly, a process can only access a collate through a process-local handle, `$collate_state`.

The CIVL-C data structure representing the collate queue `$gcollator` (left) and the collate `$gcollate_state` (right) is given in Fig. 9.6. The left data structure is simply a wrap of a CIVL-C sequence of collates and two integers: the number of processes and the queue size. The right data structure represents a collate. In a collate, the array, `status`, keeps the track of whether every process have copied its snapshot to this collate. If `status[`p`]` is true, process p has copied its snapshot to this collate. The field `state` holds the program state that is merged from the snapshots that have been copied to this collate. The two arrays , `requirements` and `guarantees`, store the corresponding absence assertions to this collate for each process. The field `isPre` labels the collate for whether it is associated to a collective pre-state. The field `isTarget` labels the collate for whether it is associated to a collective state of the verifying function. Finally, `id` holds an identifier for the function associated with this collate.

The `$state` is a CIVL-C built-in type representing program states. The `state` field in a collate will keep being updated via being merged with the snapshots from processes. Once `status[`p`]` is true for every process p, the collate is completed, the `state` field becomes equivalent to the collative state, to which the collate is associated.

Following the collate management algorithm (described in §7.3.2), an object (collate) of `$gcollate_state` type will be created and enqueued by the first process that reaches a collective location, i.e., the entry or exit of a contracted function. The `id`

```
struct _gcollator {
  int nprocs;
  int queue_length;
  $gcollate_state queue[];
};
```

```
struct _gcollate_state {
  _Bool status[];
  $state state;
  _Bool requirements[];
  _Bool guarantees[];
  _Bool isPre;
  _Bool isTarget;
  int id;
};
```

Figure 9.6: CIVL-C implementation for collate and collate queue

field is initialized during the creation. When every following process reaches a collective location and copies its snapshot to the collate, the process compares the identifier of the reached function with the `id` in the collate. A mismatch means that there is a collective-style function in the program that is not called by all processes collectively.

The other four fields are used by the absence assertion checking algorithm, which will be described in the next subsection.

Around these data structures, the `collate` library provides a set of functions for the clients to interact with them. Among those functions, there is a pair for processes to copy their snapshots to proper collates and to free collates collectively:

1. ```
   $collate_state $collate_arrives($collator c, $scope scope,
       _Bool isPre, _Bool isTarget, _Bool requirement,
       _Bool guarantee, int id);
   ```

2. ```
   void $collate_departs($collator c, $collate_state cs);
   ```

When a process reaches a collective location associated with a collective-style function g, the process calls the `$collate_arrives` function to save its snapshot to the proper collate in a collate queue. The queue is given by the `$collator` handle. In addition to the snapshot, the partially evaluated path-requirement and -guarantee of g of the process are stored in the collate. This `$collate_arrives` function returns a handle to the corresponding collate. To free a collate, each process calls `$collate_departs` with a handle to the collate collectively. The function does not guarantee that the

collate object is actually destroyed. But a process shall not access a collate once it is "freed" by a call to `$collate_departs`.

A snapshot of a process is also a `$state`. Initially, the `state` field in a collate holds the snapshot of the first "arrived" process. Then, whenever a process "arrives", its snapshot will be merged into the `state` field.

In CIVL, a program state is mainly a tree of *dynamic scopes* (dyscopes) and a group of call stacks. A dynamic scope is a runtime instance of a lexical program scope and is assigned a unique ID. A call stack belongs to a process. An entry of a call stack is a reference to a dyscope.

Figure 9.7 shows a CIVL state of a CIVL-C program transformed from a general MPI program. Recall that the transformation template is given in Fig. 8.5. The state belongs to a run with two MPI processes. Note that in addition to the two processes that run the `_mpi_process`, the state includes a "root" process p0 that was spawned at the time the CIVL-C program launches. The "parent-of" relation in between dyscopes relates to their lexical scopes. If a dyscope d is the parent of another dyscope d', the lexical scope of d is the parent of the lexical scope of d'. For example, in Fig. 9.7, **d0** is a runtime instance of the root scope s_0 and **d1** is a runtime instance of the lexical scope s_1 of the `main` function. So s_0 is the parent of s_1 and **d0** is the parent of **d1**. For a lexical scope, it can have multiple runtime instances in a state.

In a snapshot of a process p taken from a state ω, the dyscope tree contains only the dyscopes reachable by p in ω. There is only one call stack in the snapshot, which belongs to p.

Note that the `$collate_arrives` function also takes a `$scope` parameter, which shall refer to the dyscope where the process invokes the call to this function. Let h be such `$scope` parameter. In the snapshot of a process taken at a call to `$collate_arrives`, no descendant dyscopes of h needs to be included in the snapshot.

The `$state` field in a collate is initialized by the snapshot of the first process that "arrives" the collate. We call the `$state` object in an incomplete collate a *partially merged state*. Merging a snapshot ω_p to a partially merged state ω_c includes the

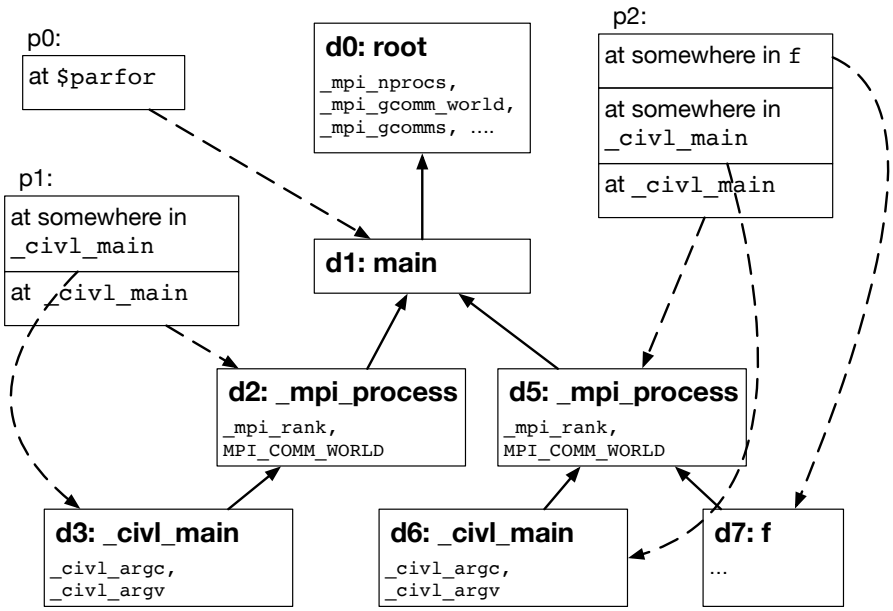

Figure 9.7: A CIVL state of a CIVL-C program transformed from an general MPI program, which contains a function `f`. The boxes labeled by **d0, d1, ...** are dyscopes. Arrows between dyscopes denote the "parent-of" relation. Boxes under "p0, p1, p2" are call stack entries. Dashed arrows mark the referred dyscope of every call stack entry.

following steps:

1. Re-numbering dyscope IDs in ω_p to be consistent with the ω_c. For a dyscope d in ω_p, if there is a dyscope d' in ω_c such that d and d' have the same ID (denoted, $d = d'$), let d keeps its ID. Otherwise, re-numbering the ID of d to be a new unique one that does not exist in ω_c.

2. Let d and d' be two dyscopes in the re-numbered ω_p and ω_c, respectively. We call d or d' the *least common ancestor* of the two trees in ω_p and ω_c if

   - $d = d'$,
   - any leaf dyscope in the dyscope tree of ω_p is a descendent of d, and any leaf dyscope in the dyscope tree of ω_c is a descendent of d', and
   - there is not a pair of dyscopes δ and δ' such that δ is d or a descendent of d, δ' is d' or a descendent of d', and δ and δ' satisfy the both two conditions above.

   The merging is then carried out by appending all child dyscopes of d to d'.

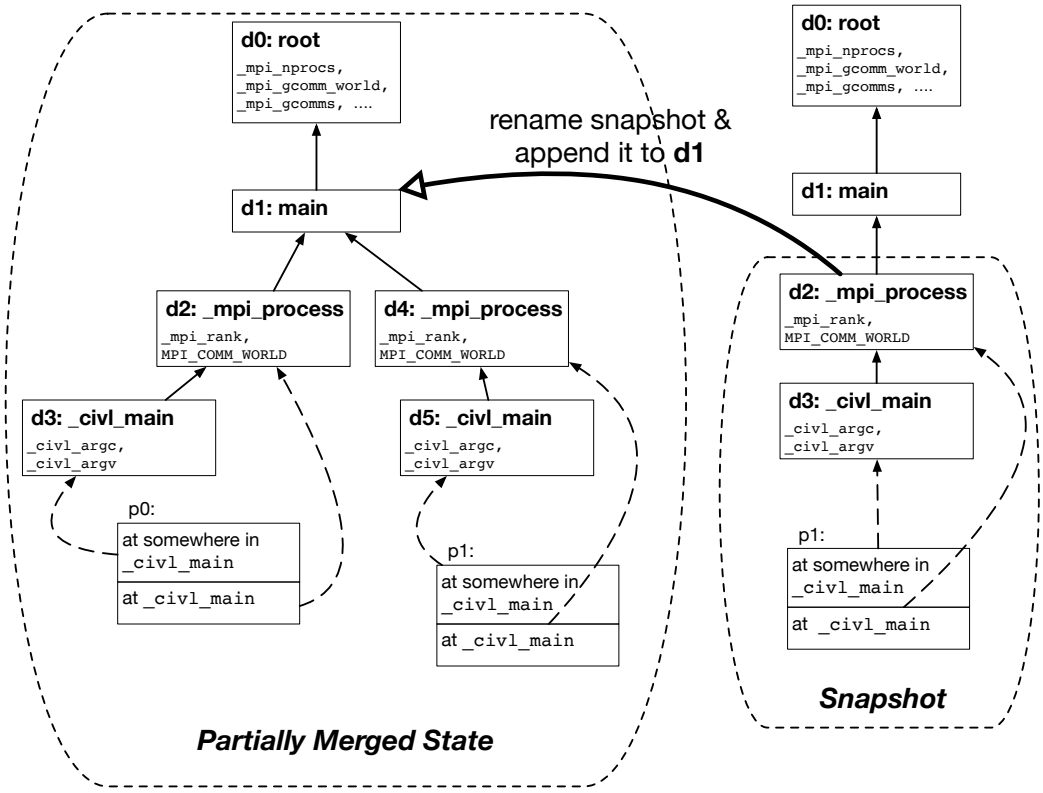

Figure 9.8: Merging a snapshot to a state. The snapshot (on the right) was taken from the state in Fig. 9.7 for process p1.

3. Finally, copying the sole call stack in ω_p to ω_c. Re-number the ID of the process of the call stack to its rank corresponding to the collate.

Example 9.4. Figure 9.8 shows merging a snapshot (on the right), which is taken from the state in Fig. 9.7 for process p1, to a *partially merged state* (on the left), on which processes are re-numbered by their ranks and are at some collective locations. The dyscope **d1** is the least common ancestor of the two dyscope trees on the left and right, respectively. Dyscopes **d2** and **d3** on the right will then be re-numbered to **d6** and **d7**, respectively. The process p0 on the right will then be re-numbered in correspondence to its rank in the collate. □

According to such state merging process, the "shared" dyscopes, i.e., the ones reachable by all processes, are coming from the snapshot of the first "arrived" process.

The MPI communicators reside in these "shared" dyscopes. Following the collate management algorithm, message channels in the MPI communicators in a collate shall then be updated whenever a process p, which has not arrived the collate, executes a send or a receive operation.

To update the message channels in collates, the `collate` library provides a pair of functions:

- `$collate_enqueue($collator c, $comm comm, $message m)`
- `$collate_dequeue($collator c, $comm comm, $message m)`

For every collate that has not "arrived" by the running process in the queue referred by `c`, the `$collate_enqueue` function enqueues the `$message m` to the proper message channel of the communicator referred by `comm` residing in the merged state of the collate. Note the source and destination information of a `$message` are encoded in itself. Vice versa for the `$collate_dequeue` function.

9.3.2 Implementing Absence Assertions

Absence assertions in the MPI contract language are translated to CIVL-C *abstract functions*. An abstract function is a mathematical uninterpreted function.

- `\absentof \sendfrom(`p`, `t`) \after \exit(`p`) \until \exit` is translated to an boolean abstract function `$absent_sendfrom(`p`, `t`)`;
- `\absentof \sendto(`p`, `t`) \after \enter \until \enter(`q`)` is translated to an boolean abstract function `$absent_sendto(`p`, `t`, `q`)`;
- `\absentof \exit \after \enter \until \exit(p)` is translated to an boolean abstract function `$absent_exit(`p`)`.

We define these abstract functions in a CIVL-C library, `absence_assertion.cvh`, which is another extension to CIVL. In addition to these abstract functions, the library provides a set of functions for reasoning about the absence assertions. For example, there is an `$aa_not_exists(_Bool set, _Bool absence)` function. It takes two boolean expressions: `set` is the conjunction of a set of absence assertions and `absence`

is a single absence assertion. The function returns true iff the `absence` assertion is not an element of the `set`. For instance, if for some integers a, b, c, n, r, p,

`set` is $\forall i \in [n].\forall t \in \mathbb{Z}.$ `$absent_sendto(`i`, `t`, `r`)` $\wedge$ `$absent_exit(`p`)`

and `absence` is `$absent_sendto(`a`, `b`, `c`)`,

the function returns $(\forall i \in [n].\, i \neq a) \vee (\forall t \in \mathbb{Z}.\, t \neq b) \vee r \neq c$.

Absence assertions are checked for the freedom of the errors defined in Fig. 7.2 and 7.4 when processes reach specific locations. We use the CIVL-C function `check_guarantee` that checks the freedom of **PGV** as an example. Figure 9.9 shows the simplified CIVL-C code of this `check_guarantee` function. The function is executed immediately after a process performs a send operation. At line 6-10, the code searches the collate of the collective pre-state of the verifying function by iterating the given collate queue. By the collate management algorithm, the search will succeed because the goal collate will not be freed until all processes have reached the end of the verifying function. Line 16-20 checks for each process q that has not entered the verifying function, if the guarantee of the function of the running process (identified by `place` in the communicator) contains

`\absentof \sendto(dest, tag) \after \enter \until \enter(`q`)`,

the guarantee is violated.

9.3.3 Transformation

A *contract transformer* was developed and added to CIVL for transforming a C/MPI/CIVL-C program along with ACSL function contracts to a pure CIVL-C program. The transformer takes a parameter f, which is the name of the verifying function, and then transforms all the annotated functions in the original program in two ways:

1. For the function f, the transformer copies f to f`_origin` and creates a new function f`_driver`. The f`_driver` initializes all the variables with respect to the state-requirement of f, then calls f`_origin` to launch the exploration of the

```
1  void check_guarantee(int place, int dest, int tag, int nprocs,
2                       $collator c) {
3    $collate_state cs;
4    int queue_size = $collate_queue_size(c, place);
5
6    for (int i = 0; i < queue_size; i++) {
7      cs = $collate_get_in_queue(c, i);
8      if ($collate_state_is_pre(cs) && $collate_state_is_target(cs))
9        break;
10   }
11   // cs must have hold the handle to the collate for the collective
12   // pre-state of the verifying function.
13   _Bool element;
14   _Bool set = $collate_get_guarantee(cs, place);// get my guarantee
15
16   for (int i = 0; i < nprocs; i++)
17     if (!$collate_arrived(cs, i)) {
18       element = $absent_sendto(dest, tag, i);
19       $assert($aa_not_exists(element, set),
20        "the guarantee of the verifying function is violated!");
21     }
22 }
```

Figure 9.9: The simplified code of the CIVL-C function that will be called every time a send operation was performed by a process in order to check for the free of **PGV**.

original function body. The state-guarantee of f is checked in f`_driver` after the control returns from f`_origin`.

2. For every annotated function g (including f), the transformer replaces the definition of g with a CIVL-C code that 1) asserts the state-requirement of g; 2) refreshes the objects specified in `assigns` clauses; 3) assumes the state-guarantee of g.

Note that the verifying function f will be transformed in both of the ways. This results in three different functions that all originate from f: f`_driver`, f`_origin` and the f with a new body.

The transformation relies on a number of basic CIVL-C primitives, as well as helper libraries that provide contract-specific data types and functions. A selected set of such primitives are listed below:

- `$value_at(expr, state, pid)` is an expression that represents the evaluation of expr on state by the process of pid. The `\on`, as well as the `\old` construct from ACSL, will be translated to this expression.
- `$run` stmt is a CIVL-C statement that creates a new process to execute stmt.
- `$with(state)` stmt is a CIVL-C statement that executes the given stmt starting from the given state.
- `$havoc(pointer)` "refreshes" an object by assigning a unique and unconstrained symbolic constant to the object referred by the pointer.
- `$mem` is a type representing a set of memory locations.
- `$mem_havoc(`m`)` assigns a unique and unconstrained symbolic constant to every object in the memory locations in m.
- `$mem_contains(`m_0`, `m_1`)` returns true iff every element in the memory location set m_0 is also an element of the memory location set m_1.
- `$write_set_push()` informs the verifier to start to save all the memory locations that are modified during the execution. This is implemented by pushing a new empty memory location set to a stack that is maintained by every process. The stack is called the *write set stack*. During the execution, a write operation that is performed by a process p on a memory location m will cause m be added into the top entry of the write set stack of p unless the stack is empty.
- `$write_set_pop()` pops and returns the top entry of the write set stack of the executing processes. When the write set stack is empty, the verifier will not save any memory location.
- `$mpi_snapshot(comm, scope, isPre, isTarget, requirement, guarantee, fid)` is a helper function for the transformer that wraps the `$collate_arrives` function. In addition to a call to `$collate_arrives`, this function contains CIVL-C code that checks absence assertion violation and performs POR. Dually, there is a `$mpi_unsnapshot(collate_state)` function.

  We remark that the comm parameter is of `MPI_Comm` type. In our extended implementation for MPI contracts, an MPI communicator was extended with an extra field holding a collate queue (i.e., a `$gcollator`). This is be in consistent with the theory that a collate queue is associated with a communication universe. As a consequence, collate queues in different MPI communicators are naturally independent.

All the MPI specific constructs in the MPI contract language are translated to special CIVL system functions. A system function in CIVL is a function that is

implemented in the language that writes CIVL, i.e., Java 8 [57], instead of CIVL-C. For example, the `\mpi_agree(`e`)` expression in contract will be translated to a call to the system function `$mpi_agree(`e`)`. The system function returns true iff every process evaluates e to a same value at the current state.

Driver Function Generation. For a given verifying function `f`, `f` must be assigned a function contract. The transformer moves the original body of `f` to the new function `f_origin`. Then generates a new function `f_driver` with respect to the contract. Figure 9.10 summarizes the idea of generating `f_driver` with a template.

In the template, the pseudocode above the horizontal line is the original verifying function `f` with its contract. The `T, T1, T2` and `T3` represent arbitrary types. The ψ and ϕ are boolean expressions. The Δ represents a list of lvalue expressions. The Γ and Υ are sets of absence assertions.

The pseudocode below the horizontal line is the generated `f_driver` function. The MPI communicator `comm` associated to the collective behavior in the contract is initialized by duplicating the `MPI_COMM_WORLD` (line 4). Two variables `$mpi_comm_rank` and `$mpi_comm_size` are created to replace the two constants `\mpi_comm_rank` and `\mpi_comm_size`, respectively. Formal parameters are declared as local variables. These local variables, as well as the global variables, are initialized by unconstrained symbolic constants (line 8).

For a pointer type formal parameter v, one cannot assume that v can be safely dereferenced unless the contract "requires" `\valid(`v`)`. For such v, a unique object o of the referenced type of v will be declared and "refreshed". The expression `\valid(`v`)` in the state-requirement is then replaced by v `== &`o.

We remark that such a transformation for `\valid` expressions in state-requirements is only sound under an assumption that there is no *pointer aliasing*, i.e., a pointer type formal parameter is assumed not to refer to a program variable or an object referred by another pointer unless it is explicitly expressed in the state-requirement.

The state-requirement is `$assume`d in a pair of barriers so that the state from which this `$assume` statement is executed will be equivalent to a pre-state of `f`. Similar

for the case of `$assert`-ing the state-guarantee.

Although for `f`, there is no need to construct its collective pre- and post-states, there are still `$mpi_snapshot` calls that create collates for `f`. This is because that we use these two calls to represent the events of entering and exiting of `f`, respectively. In addition, the created collective pre-state of `f` will be used to evaluate the `\old` expressions in the state-guarantee.

The call to the `f_origin` is surrounded by a pair of `$write_set_push` and `$write_set_pop`. By doing so, any memory location that is modified during the execution of the original function body will be saved in `m` (line 17). Then the validity of the frame condition can be verified by testing if `m` is a subset of Δ.

Finally, the collates created by calls to `$mpi_snapshot` will be collectively "destroyed" through the calls to `$mpi_unsnapshot` (line 26-27).

Summarizing Function Behavior From Contract. For every contracted function, including the verifying function, its' original function body will be replaced by the transformer with a CIVL-C code that summarizes the function from the function contract. The template for generating such a new function body from a contract is given in Fig. 9.11.

In this template, a pair of collective pre- and post-state of the function `f` are created through calls to `$mpi_snapshot`. The state-requirement and -guarantee will be evaluated on these two states respectively once they are completed. For each MPI process, it shall not wait at the location until a collective state is completed because if doing so, the semantics of the program gets changed. Therefore, an MPI process will "fork" a new "deamon" process that will asynchronously waits for the completion of a collective state and then evaluates the state-requirement or -guarantee on the state. The "deamon" process is created by the `$run` statement (line 10 and 21). The "deamon" process will be blocked by the `$when` statement (line 11 and 22), in which the guard is a system function `$collate_complete(`c`)` that returns true iff the collate referred by c is completed. Once the guard evaluates to true, the "deamon" process asserts (or assumes) the state-requirement (or the state-guarantee) from the completed

```
T3 g, ... ;
/*@  \mpi_collective(comm, kind):
       requires ⋀Γ;
       requires ψ;
       assigns Δ;
       ensures φ;
       ensures ⋀ϒ; */
T f(T1 a, T2 * b, ... )  { ... }
```

```
void f_driver() {
  int $mpi_comm_rank, $mpi_comm_size;
  MPI_Comm comm;
  MPI_Comm_dup(MPI_COMM_WORLD, &comm);
  MPI_Comm_rank(comm, &$mpi_comm_rank);
  MPI_Comm_size(comm, &$mpi_comm_size);
  T1 a, T2 *b, T2 b_obj ... ;
  $havoc(&g); $havoc(&a); $havoc(&b); $havoc(&b_obj); ...
  MPI_Barrier(comm);
  $assume(ψ);
  MPI_Barrier(comm);
  $collate_state _cs_pre = $mpi_snapshot(comm, $here, $true, $true,
    ⋀Γ, ⋀ϒ, ''f'');

  $write_set_push();
  T $result = f_origin(comm, a, b, ... );
  $mem m = $write_set_pop();
  $assert($mem_contains(Δ, m));

  $collate_state _cs_post = $mpi_snapshot(comm, $here, $false, $true,
    ⋀Γ, ⋀ϒ, ''f'');
  MPI_Barrier(comm);
  $state* _s_pre = $collate_get_state(_cs_pre);
  $assert(φ);
  MPI_Barrier(comm);
  $mpi_unsnapshot(_cs_pre);
  $mpi_unsnapshot(_cs_post);
}
```

Figure 9.10: Transformation template for generating `f_driver` from a function `f` with its contract. The upper shows the original annotated function `f` and the lower shows the generated `f_driver`.

collective state. The collective state is given by a pointer, which points to the `$state` object in a collate (line 9 and 20). The "deamon" process terminates automatically after the assertion (or assumption).

In addition to making assertions and assumptions, the objects listed in `assigns` clauses will be "refreshed" by the `$havoc` calls.

A special variable `$result` is created to represent the returned value of `f`. The expression `\result` in the contract will be replaced by `$result`.

The running process will be blocked (at line 24) if it has to wait for some other processes p that have not "entered" this function. Recall that the entering-function event is represented by the transition that an MPI process marks itself on the collate associated to the collective pre-state. This code realizes the guarantees in the contract of the form: `\absentof \exit \after \enter \until \exit(`p`)`.

```
T3 g, ... ;
T f(MPI_Comm comm, T1 a, T2 *b, ... )  {
  int $mpi_comm_rank, $mpi_comm_size;
  MPI_Comm_rank(comm, &$mpi_comm_rank);
  MPI_Comm_size(comm, &$mpi_comm_size);

  $collate_state _cs_pre = $mpi_snapshot(comm, $here, $true, $false,
    ⋀Γ, ⋀ϒ, ''f'');
  $state* _s_pre = $collate_get_state(_cs_pre);
  $run
    $when ($collate_complete(_cs_pre))
      $with (*_s_pre) $assert(ψ);

  T $result;
  ∀d ∈ Δ. $havoc(&d);
  $havoc(&$result);

  $collate_state _cs_post = $mpi_snapshot(comm, $here, $false, $false,
    ⋀Γ, ⋀ϒ, ''f'');
  $state* _s_post = $collate_get_state(_cs_post);
  $run
    $when ($collate_complete(_cs_post))
      $with (*_s_post) $assume(ϕ);
  $when($collate_arrived(_cs_pre, p));
  $mpi_unsnapshot(_cs_pre);
  $mpi_unsnapshot(_cs_post);
}
```

Figure 9.11: The transformation template for summarizing `f` with respect to its contract.

Chapter 10

EVALUATION

The evaluation of our approach for verifying message-passing program against function contracts is based on the MPI function contract system described in Chapter 9. We applied our implementation to a number of C/MPI/CIVL-C collective-style functions, including functions from CIVL's MPI library, implementations of advanced algorithms for MPI collective functions and collective-style functions carved out from MPI scientific computing applications.

Two of the aforementioned collective-style functions are selected to be presented with details in §10.1. The overall experimental results and performance are discussed about in §10.2.

10.1 Running Examples

Real MPI implementations, such as MPICH [107], use advanced algorithms for MPI collective functions. And more such algorithms keep being designed and implemented [59, 108]. An MPI contract system can be effective for verifying the correctness of these algorithms since 1) it can verify function implementations in isolation; and 2) different implementations of one MPI collective function can share a function contract. To illustrate such effectiveness, we show the verification of two `MPI_Allgather` implementations in §10.1.1.

The implementation of an MPI collective function, or more generally, of an MPI collective-style function can be complicated though, their complicatedness has no effect on the verification of their caller functions for a contract system. Thanks to the modular verification feature of contract systems, the behavior of a call to a contracted function only depends on its function contract. We illustrate such an advantage of our

MPI contract system in §10.1.2 by presenting an example that uses the `MPI_Allgather` function.

10.1.1 Allgather Implementations

CIVL's implementation. In CIVL, simple and deterministic algorithms are used to implement MPI collective functions. Figure 10.1 shows CIVL's implementation of the `MPI_Allreduce` function, which is just a call to the `MPI_Reduce` followed by a call to the `MPI_Bcast`.

Both the `MPI_Reduce` and `MPI_Bcast` are annotated with function contracts. Therefore, for the purpose of verifying the functional correctness of the `MPI_Allreduce` function, their definitions are not needed. Since all these functions are MPI collective functions, they are guaranteed not to interfere with each other. So there is no need to specify a requirement or guarantee for these functions.

To verify `MPI_Allreduce` with the contracts, the user needs to specify a bounded number of processes. The contract system then is able to verify that `MPI_Allreduce` satisfies its function contract for the bounded number of processes and arbitrary message data, data size, data type and reduction operation.

Next, we briefly explain these contracts.

For the contract of `MPI_Bcast`, the state-requirement states that

1. `root` must be in between 0 and the size of the communicator;
2. the type signature (i.e., the product of the `count` and the `datatype` extent) must be non-negative;
3. all processes must have the same values for `root` and the type signature, respectively;
4. the pointer `buf` must point to an allocated memory region, of which the size shall not be less than the type signature.

The state-guarantee ensures that eventually all processes will have the same value in their memory regions referred by their `buf`s. In addition, only those memory regions

of the non-root processes will be modified. Such a frame condition is expressed with a behavior named by "`nonroot`".

For the contract of `MPI_Reduce`, the state-requirement is similar to the one of `MPI_Bcast` except that it is stronger over the values of `count` and `datatype`. It requires that all processes must have same values for `root` and `datatype`, respectively. In fact, the author of this dissertation was not aware of that `MPI_Reduce` needs a stronger state-requirement than `MPI_Bcast` until the contract system reports an error. The state-guarantee ensures that 1) nothing will be modified for all the non-root processes; and 2) the memory region referred by `recvbuf` on the root process will eventually be assigned the result of reduction over all the memory regions referred by `sendbuf`s on all the processes.

The contract of `MPI_Allreduce` is similar to the one of `MPI_Reduce` except that `MPI_Allreduce` ensures all processes will eventually hold the reduction result.

Recursive Doubling. Real MPI implementations use more optimized algorithms for `MPI_Allreduce`, such as the *recursive doubling* algorithm. Figure 10.2 is a CIVL-C realization of a recursive doubling `MPI_Allreduce` pseudocode given in [96]. The function contract of the `MPI_Allreduce` in this figure is redundant and hence omitted.

In this implementation, we used CIVL-C primitives to abstract details that are irrelevant to the algorithm:

- The `$mpi_malloc` is a helper function written in CIVL-C. It allocates a memory region of the size of the given type signature. In the case of that the value of `datatype` is arbitrary, instead of elaborating all possible concrete choices, this function uses an uninterpreted function *extent*(`datatype`) to represent the extent of the `datatype` and allocates a memory region of size *extent*(`datatype`) $*$ `count`.
- In ACSL, reduction operations can be expressed with different constructs according to the reduction operation. For example, summing up all the n elements in an array a can be expressed as `\sum(0,` n`-1, \lambda int i;` a`[i])` in ACSL. However, for the cases where the reduction operation is arbitrary, specifying a reduction operation in ACSL will inevitably elaborate all possible concrete choices. However, elaboration will make the verification much more expensive. Hence, we

```
/*@ \mpi_collective(comm, COL):
    requires 0 <= root < \mpi_comm_size && 0 <= count * \mpi_extent(datatype)
          && \mpi_agree(root) && \mpi_agree(count * \mpi_extent(datatype))
          && \mpi_valid(buf, count, datatype);
    ensures \mpi_agree(\mpi_region(buf, count, datatype));
    behavior nonroot:
      assumes \mpi_comm_rank != root;
      assigns \mpi_region(buf, count, datatype); */
int MPI_Bcast(void * buf, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm);

/*@ \mpi_collective(comm, COL):
    requires 0 <= count*\mpi_extent(datatype)
          && \mpi_valid(sendbuf, count, datatype)
          && \mpi_valid(recvbuf, count, datatype)
          && \mpi_agree(root) && \mpi_agree(count) && \mpi_agree(datatype);
          && 0 <= root < \mpi_comm_size;
    behavior root:
      assumes \mpi_comm_rank == root;
      assigns \mpi_region(recvbuf, count, datatype);
      ensures \mpi_equals(\mpi_region(recvbuf, count, datatype),
                          \mpi_reduce(sendbuf, count, datatype, op)); */
int MPI_Reduce(void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);

/*@ \mpi_collective(comm, COL):
    requires count >= 0 && \mpi_valid(sendbuf, count, datatype)
          && \mpi_valid(recvbuf, count, datatype)
          && \mpi_agree(count) && \mpi_agree(op) && \mpi_agree(datatype);
    assigns \mpi_region(recvbuf, count, datatype);
    ensures \mpi_equals(\mpi_region(recvbuf, count, datatype),
                        \mpi_reduce(sendbuf, count, datatype, op));*/
int MPI_Allreduce(void * sendbuf, void * recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) {
  MPI_Reduce(sendbuf, recvbuf, count, datatype, op, 0, comm);
  MPI_Bcast(recvbuf, count, datatype, 0, comm);
  return 0;
}
```

Figure 10.1: Annotating function contracts for all the collective-style functions used by CIVL's implementation of `MPI_Allreduce`.

used a CIVL-C system function `$memapply` to implement the `reduce` function that performs process local reduction operation:

```
void reduce(void * inout, void * in, int count, int size,
            MPI_Op op) {
  $memapply(inout, ($operation)op, in, count, size, inout);
}
```

The `$memapply` function performs an "element-wise" `$operation` on two memory regions referred by `inout` and `in`, respectively, and writes the result back to `inout`. Here "element-wise" means that the operation is applied to every pair of `size / count` bytes memory regions that are referred by `(char*)inout + i * (size / count)` and `(char*)in + i * (size / count)`, respectively, for $0 \leq i <$ `count`.

When the `op` has a non-concrete value, `$memapply` uses an uninterpreted function to represent the general reduction result.

With these abstractions, this function implementation can be verified by the MPI contract system with the same configuration as the previous implementation.

10.1.2 Paralle Vector Product

The two `MPI_Allreduce` implementations presented in §10.1.1 have different degrees of complicatedness. But the implementation is irrelevant to the functional correctness of a caller function of `MPI_Allreduce` as long as `MPI_Allreduce` is annotated with a contract. We present such a collective-style caller function to `MPI_Allreduce` in Fig. 10.4. The data structures used in this figure are showed in Fig. 10.3. This C code was carved out from the HYPRE [37] open source project.

The `hypre_ParVectorInnerProd` function lets every process computes the vector product of two sequences of elements in parallel and then sums up the vector products on all the processes.

The collective behavior in the function contract of `hypre_ParVectorInnerProd` is associated to the MPI communicator stored in the structure referred by `x`. The state-requirement states that

1. the structures referred by `x` and `y` have been allocated,
2. the `hypre_Vector` structures referred by the fields `local_vector` of `*x` and `*y` have been allocated,

```
1  void MPI_Allreduce(void * sendbuf, void * recvbuf, int count,
2                     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) {
3    int datasize = sizeofDatatype(datatype) * count;
4    void * global = $mpi_malloc(count, datatype);
5    int pof2, rem, int nprocs, rank, myrank, mask = 1;
6
7    MPI_Comm_size(comm, &nprocs); MPI_Comm_rank(comm, &rank);
8    pof2 = log2(nprocs); pof2 = pow2(pof2);
9    rem = nprocs - pof2;
10   memcpy(global, sendbuf, datasize);
11   if (rank < 2 * rem) {
12     if (rank % 2 == 0) {
13       MPI_Send(global, count, datatype, rank + 1, 0, comm);
14       myrank = -1;
15     } else {
16       MPI_Recv(recvbuf, count, datatype, rank - 1, 0, comm, MPI_STATUS_IGNORE);
17       reduce(global, recvbuf, count, datasize, op);
18       myrank = rank / 2;
19     }
20   } else
21     myrank = rank - rem;
22   if (myrank != -1)
23     while (mask < pof2) {
24       int dst, newdst = myrank ^ mask;
25
26       if (newdst < rem)
27         dst = newdst * 2 + 1;
28       else
29         dst = newdst + rem;
30       MPI_Sendrecv(global, count, datatype, dst, 0,
31                    recvbuf, count, datatype, dst, 0, comm, MPI_STATUS_IGNORE);
32       reduce(global, recvbuf, count, datasize, op);
33       mask = mask << 1;
34     }
35   if (rank < 2 * rem)
36     if (rank % 2 != 0)
37       MPI_Send(global, count, datatype, rank - 1, 0, comm);
38     else
39       MPI_Recv(global, count, datatype, rank + 1, 0, comm, MPI_STATUS_IGNORE);
40   memcpy(recvbuf, global, datasize);
41   free(global);
42 }
```

Figure 10.2: An `MPI_Allgather` implementation based on the recursive doubling algorithm.

```
1 typedef struct {
2   HYPRE_Complex *data;
3   HYPRE_Int size;
4   HYPRE_Int num_vectors;
5   ...
6 } hypre_Vector;
```

```
1 typedef struct hypre_ParVector_struct {
2   MPI_Comm comm;
3   hypre_Vector *local_vector;
4   ...
5 } hypre_ParVector;
```

Figure 10.3: The data structures that are used in Fig. 10.4. Insignificant fields are omitted.

3. the sequences of elements referred by the fileds `data` of `x->local_vector` and `y->local_vector` have been allocated, and
4. the size of an element sequence must be positive.

The state-guarantee states that 1) nothing will be modified by this function, and 2) the returned value will equal to the sum over all the dot products computed by all the processes.

This function relies on the `hypre_SeqVectorInnerProd` sequential function to perform process local dot product operations. In order to fully verify the functional correctness of `hypre_ParVectorInnerProd`, one must also verify the function `hypre_SeqVectorInnerProd` independently. The contract of this sequential function is purely in ACSL and is easy to understand. We remark that in order to verify this function for arbitrary size of the element sequences, we need to annotate the loop at line 23-24 with a loop invariant. The loop invariant describes the property of the loop and will be verified in an inductive way during the symbolic execution. Once the loop is verified as satisfying the loop invariant, the effect of the loop to the program can be directly derived from the loop invariant. Detailed description of CIVL's support on sequential loop invariants can be found in [77].

By verifying `hypre_SeqVectorInnerProd` and `hypre_ParVectorInnerProd` against their contracts separately, the functional correctness of both functions are thus verified. Of course, for `hypre_ParVectorInnerProd`, the result is limited for a bounded number of processes.

```
#define hypre_ParVectorComm(vector) ((vector) -> comm)
#define hypre_ParVectorLocalVector(vector) ((vector) -> local_vector)
#define hypre_VectorData(vector) ((vector) -> data)
#define hypre_VectorSize(vector) ((vector) -> size)
#define hypre_VectorNumVectors(vector) ((vector) -> num_vectors)
#define hypre_conj(value) value
#define SIZE ((x->size)*(x->num_vectors))
#define PAR_SIZE ((x->local_vector->size)*(x->local_vector->num_vectors))
#define NPROCS \mpi_comm_size
/*@ requires \valid(x) && \valid(y) && 0 < SIZE
          && \valid(x->data + (0 ..  SIZE-1)) && \valid(y->data + (0 ..  SIZE-1));
    assigns \nothing;
    ensures \result == \sum(0, SIZE-1, \lambda int j; y->data[j] * x->data[j]); */
HYPRE_Real hypre_SeqVectorInnerProd(hypre_Vector *x, hypre_Vector *y) {
  HYPRE_Complex *x_data = hypre_VectorData(x);
  HYPRE_Complex *y_data = hypre_VectorData(y);
  HYPRE_Int i, size = hypre_VectorSize(x);
  HYPRE_Real result = 0.0;
  size *=hypre_VectorNumVectors(x);
  /*@ loop invariant 0 <= i <= size
              && result == \sum(0, i-1, \lambda int j; y_data[j] * x_data[j]);
      loop assigns result, i; */
  for (i = 0; i < size; i++)
    result += hypre_conj(y_data[i]) * x_data[i];
  return result;
}

/*@ \mpi_collective(hypre_ParVectorComm(x), COL):
    requires \valid(x) && \valid(y) && 0 < PAR_SIZE &&
             \valid(x->local_vector) && \valid(y->local_vector) &&
             \valid(x->local_vector->data + (0 ..  PAR_SIZE-1)) &&
             \valid(y->local_vector->data + (0 ..  PAR_SIZE-1));
    assigns \nothing;
    ensures \result == \sum(0, NPROCS-1, \lambda int k;
                    \on(k, \sum(0, PAR_SIZE-1, \lambda int t;
                      x->local_vector->data[t] * y->local_vector->data[t]))); */
HYPRE_Real hypre_ParVectorInnerProd(hypre_ParVector *x, hypre_ParVector *y) {
  MPI_Comm comm = hypre_ParVectorComm(x);
  hypre_Vector *x_local = hypre_ParVectorLocalVector(x);
  hypre_Vector *y_local = hypre_ParVectorLocalVector(y);
  HYPRE_Real result = 0.0;
  HYPRE_Real local_result = hypre_SeqVectorInnerProd(x_local, y_local);
  hypre_MPI_Allreduce(&local_result, &result, 1, HYPRE_MPI_REAL,
        hypre_MPI_SUM, comm);
  return result;
}
```

Figure 10.4: A function that computes vector product in parallel using MPI.

10.2 Experiment

We evaluate our MPI contract system implementation by using it to specify and verify a set of C/MPI/CIVL-C collective-style functions.

This experiment contains two parts. In the first part, we ignore the checking of absence assertions in the contracts. In other words, 1) the first part assumes that no interference can happen for every collective-style function; and 2) it does not verify the guarantees of the verified functions. In the second part, we enable full validity verification for a selected subset of the experiments.

For each collective-style function in the experiment, we prepared a set of negative examples, each of which contains a specific kind of contract violation, such as state-guarantee violation, too weak state-requirement to ensure the state-guarantee or a kind of absence assertion violations. Our system can precisely report the expect error for each of these negative examples. For brevity, we do not show the results of all the negative examples.

Instead of directly verifying the full validity of every contracted function, we separated the verification only for state-requirement and -guarantees in the first part for two reasons. First, the absence assertions are introduced for reasoning about the case of interference. Most of our examples use only MPI collective functions and wildcard-free send/receive operations. They fall into the category where no interference can happen naturally. Second, full validity verification involves checking path predicates (i.e., absence assertions). CIVL was originally designed for verifying properties that can be expressed by state predicates only. So its sophisticated POR algorithm is not sound for full validity verification. In our contract system implementation, we enforce the soundness by informing the CIVL model checker to stop applying POR when a statement that may be visible to the absence assertions is enabled. This is a coarse approximation in that the model checker eventually has to explore a large number of commutative executions. Therefore, in the second part of the experiment, examples cannot be scaled to the same level as themselves in the first part.

CIVL uses automated theorem provers to reason about the logic formulas that

are still non-trivial after a series of simplifications applied by CIVL. Each prover is assigned a timeout so that it will not run infinitely in undecidable cases. A call to a prover is expensive. Although the longer the timeout, the higher the possibility of that a prover can eventually solve a formula, CIVL has to waste more time in waiting for the reasoning of unsolvable formulas. Therefore, in this experiment, we carefully set the timeout for different provers.

1. Z3 and CVC4 are used in a sequence for all kinds of the reasoning with a 2 seconds timeout.
2. CVC4 will be invoked only if Z3 is not able to solve a formula within 2 seconds.
3. Why3 will only be used for checking assertions and is only invoked when neither Z3 nor CVC4 solves a formula within 2 seconds. Why3 is assigned a default 5 seconds timeout. In fact, Why3 itself is not an automated theorem prover but a platform that translates the input and sends it to various provers including Z3 and CVC4. We use Why3 for two of its advantages over our direct translation to Z3 and CVC4: 1) in addition to the translation, Why3 introduces extra transformations and axioms for specific theories; 2) Why3 supplies libraries for rich theories such as permutation and multi-set.

The experiment was run on an iMac with 3.5 GHz Intel Core i7 and 32 GB 1600 MHz DDR3.

Table 10.1 and 10.2 show the experimental results of the first part.

Examples in Table 10.1 are the implementations of MPI collective functions in CIVL's MPI library. These implementations are all based on simple and intuitive algorithms. For example, the implementation of `bcast` is letting the root process send messages to all processes in a fixed order. Although the implementations are naive, their contracts are general and conform to the MPI standard.

Reasoning about arbitrary aggregate type objects can be expensive. Observing the differences in the statistics of `bcast`, `gather` and `gatherv`. The naive implementation of `gather` can be seen as a reverse of the one of `bcast`: the root process receives messages from all processes in a fixed order. These two examples have very different performances. For the runs of them with a same number of processes, the verification of `gather` generates almost 9 times more prover calls than the one of `bcast`. This is

because the value eventually in the receiving buffer of the `gather` function is a partitioned array, in which each part was sent from a process. Considering the fact that both the content and size of each part are arbitrary, the general representation of such a value has to involve a number of quantified formulas. In `gatherv`, the "gathered" partitioned array is even more general in that the order of each part is arbitrary. This explains why verifying `gatherv` takes twice the time of verifying `gather`.

The contract system is effective in re-using verification results. For example, the `allgather` example is implemented by combining `gather` and `bcast`. During the verification, the behaviors of `gather` and `bcast` are derived from their contracts respectively. In other words, by assuming that `gather` and `bcast` satisfy their contracts, the verification results of them are directly re-used. According to the statistics in the table, it is easier for the contract system to verify `allgather` than `gather`.

A function contract can be re-used for different implementations. In Table 10.2, `allreduce_dr` and `reduceScatter` are implementations of MPI collective functions based on advanced algorithms. Implementations are different though, `allreduce_dr` and `allreduce` share the same function contract.

Rest of the examples in Table 10.2 are functions from different MPI scientific computing applications. In these examples, each process is responsible for performing some computation locally. Process-local functions can be specified by pure ACSL function contracts. Verifying and re-using pure ACSL contracts is just a special case of dealing with MPI contracts for our contract system. Taking `diff1dExchange` and `diff1dIter` as an example. The `diff1dExchange` is the "ghost cell exchange" function. The `diff1dIter` is the loop body of the iterative computational procedure of a parallel 1d-diffusion solver. The `diff1dIter` is a combination of a call to `diff1dExchange` and a call to a process-local computing function. The behaviors of both calls are summarized from their contracts. The verification result of the process-local function is not showed in the table since it is irrelevant to our topic. Similar for `diff2dExchange` and `diff2dIter`.

```
1  int N, M, L, *A, *B, *C, size, rank;
2  void matmat() {
3    int tmpC[L], tmpA[M];
4
5    MPI_Bcast(B, M*L, MPI_INT, 0, MPI_COMM_WORLD);
6    MPI_Scatter(A, M, MPI_INT, tmpA, M, MPI_INT, 0, MPI_COMM_WORLD);
7    if (rank != 0)
8      memcpy(A, tmpA, M * sizeof(int));
9    matmat_local();
10   memcpy(tmpC, C, L * sizeof(int));
11   MPI_Gather(tmpC, L, MPI_INT, C, L, MPI_INT, 0, MPI_COMM_WORLD);
12 }
```

Figure 10.5: The definition of the `matmat` function that collectively performs matrix multiplication.

For functions from the 1d-diffusion solver, the contracts have no specific assumption on the pattern of distribution. That is, the left neighbor of a process can be any other process. Ditto for the right neighbor. However, for the functions from the 2d-diffusion solver, we strengthen the contracts to assume a checkerboard distribution. This is the compromise we made to the complexity of proving these functions for arbitrary parameters. For this example, there are 2d arrays with arbitrary contents and sizes, inside which partitions are moving around. The general representation we mentioned before for arbitrarily partitioned arrays is powerful in expressiveness but makes reasoning difficult.

The `matmat` example is a collective matrix multiplication implementation. It is a typical example for showing that dividing MPI programs by collective-style functions are effective. Figure 10.5 shows the function definition. The function definition can be considered as being composed of three collective-style functions and a process-local procedure. Taking the process local function `matmat_local`, which computes one row of the result, as a special case of collective-style functions, the verification of this function definition is thus divided into four sub-problems.

Function contracts sometimes can help the developer to ignore the details of communication but focus on the correctness of the algorithm. For instance, the

```
1  int id, n; // id: process rank, n: sorting array
2  T myvalue; // element hold by the process
3  MPI_Comm comm;
4
5  /*@ \mpi_collective(comm, P2P):
6      requires \mpi_agree(i);
7      requires 0 <= i < n && id == \mpi_comm_rank && n == \mpi_comm_size;
8      assigns myvalue;
9      ensures (id % 2 == 0 && id < n-1 ==> myvalue <= \on(id+1, myvalue))
10         || (id % 2 == 1 && id < n-1 ==> myvalue <= \on(id+1, myvalue));
11     ensures \sum(0, n-1, \lambda int t; \on(t, myvalue) == myvalue ? 1 : 0) ==
12           \sum(0, n-1, \lambda int t; \old(\on(t, myvalue)) == myvalue ? 1 : 0);
13 */
14 void oddEvenParIter(int i); // i is the loop identifier
```

Figure 10.6: Function contract of the function in the `OddEvenSort` example.

`oddEvenSort` example comprises the loop body of a parallel odd-even sorting algorithm. The correctness of this algorithm is not so obvious and parallelism makes it even harder for developers to prove it. Figure 10.6 shows the function contract of this example. Each process holds an element of an unsorted sequence. Every i-th element in the sequence can be expressed as `\on(i, myvalue)`. In the contract, the state-guarantee at line 9-10 expresses the sortedness property; the one at line 11-12 expresses the permutation property. The contract only describes input and output of the function while leaves communication details invisible. To prove the sortedness and permutation of the complete algorithm, one can focus on the induction on the contract.

In the second part of the experiment, we enable the contract system to verify the full validity of the examples listed in Table 10.3. In order to let the experiment cover all kinds of violations, we implemented a `gather_w` function using wildcard receives, which is intended to deliver the same functionality as `gather`. Then we assume that other examples (i.e., `allgather` & `matmat`) use `gather_w` instead of `gather`.

In `gather_w`, a root process receives messages from others in a non-deterministic order with wildcards. Due to the use of wildcard receives, `gather_w` can be interfered by

name & description	#procs	#states	#trans	#prover	time(s)
`bcast`: root sends in a roll and the rests do a receive, implemented in CIVL's MPI library	2	1, 121	3, 500	7	6
	3	3, 057	8, 235	14	8
	4	5, 805	15, 632	23	12
	5	9, 797	26, 608	34	17
	6	15, 545	42, 832	47	25
`gather`: root receives in a roll and the rests do one send, implemented in CIVL's MPI library	2	2, 608	7, 246	47	9
	3	6, 791	18, 226	102	16
	4	14, 788	38, 659	187	28
	5	31, 541	79, 997	297	54
	6	71, 850	175, 925	428	123
`gatherv`: root receives in a roll and the rests do one send, implemented in CIVL's MPI library	2	1, 266	3, 692	41	8
	3	3, 053	8, 640	85	13
	4	5, 852	16, 398	140	27
	5	10, 121	28, 029	215	78
	6	16, 612	45, 512	310	236
`scatter`: root sends in a roll and the rests do one receive, implemented in CIVL's MPI library	2	2, 466	7, 007	43	8
	3	6, 332	17, 438	101	14
	4	13, 509	36, 428	188	27
	5	28, 067	73, 801	275	46
	6	62, 483	158, 481	386	108
`alltoall`: each proc performs #procs sends and receives	2	786	2, 038	17	6
	3	1, 382	3, 424	45	7
	4	2, 153	5, 111	67	10
	5	3, 114	7, 135	89	14
	6	4, 287	9, 532	115	19
`alltoall2` each proc collectively calls `scatter` #procs times, implemented in CIVL's MPI library	2	933	2, 986	19	7
	3	1, 769	5, 905	32	10
	4	3, 011	10, 351	66	15
	5	4, 779	16, 972	84	31
	6	7, 174	26, 560	150	69
`allgather`: each proc collectively calls `gather` then `bcast`, implemented in CIVL's MPI library	2	2, 119	6, 235	23	11
	3	3, 690	10, 699	34	15
	4	5, 794	16, 512	44	21
	5	8, 944	24, 820	55	31
	6	14, 176	37, 915	65	50
`allreduce`: each proc collectively calls `reduce` then `bcast`, implemented in CIVL's MPI library	2	841	2, 958	3	6
	3	1, 368	4, 803	3	6
	4	1, 903	6, 763	3	7
	5	2, 446	8, 838	3	9
	6	2, 997	11, 028	3	10

Table 10.1: The experimental results of verifying CIVL's implementations of the MPI collective functions with respect to state-requirements and -guarantees.

name & description	#procs	#states	#trans	#prover	time(s)
`allreduce_dr`: an implementation of the recursive doubling algorithm given by a pseudo code in [96]	2	648	1, 904	4	7
	3	1, 060	3, 006	4	7
	4	1, 562	4, 349	4	8
	5	2, 017	5, 615	4	10
	6	2, 486	6, 942	4	11
`reduceScatter`: an implementation of an algorithm optimized for non-commutative reduction operations [15]	2	713	1, 905	13	7
	3	1, 378	3, 163	42	11
	4	2, 147	4, 679	70	16
	5	3, 102	6, 489	111	25
	6	4, 248	8, 629	166	50
`diff1dExchange`: the exchange ghost cells function in a 1d-diffusion implementation	2	1, 771	5, 083	19	7
	3	6, 979	19, 164	40	12
	4	25, 546	68, 328	69	25
	5	93, 086	244, 148	102	84
	6	345, 202	891, 940	135	382
`diff1dIter`: one time step implementation for a 1d-diffusion solver	2	2, 186	6, 354	58	19
	3	8, 646	23, 677	161	85
	4	31, 492	82, 514	380	275
	5	114, 108	287, 467	829	731
`diff2dExchange`: $n \times n$ checkerboard ghost cell exchange	2×2	2, 010	4, 787	52	43
	3×3	5, 154	12, 817	135	115
`diff2dIter`: one time step for a 2d-diffusion solver with $n \times n$ checkerboard distribution	2×2	1, 727	5, 126	66	95
`dotProd`: parallel vector product function extracted from [37]	2	712	2, 315	26	39
	3	1, 127	3, 629	51	82
	4	1, 546	5, 010	84	141
	5	1, 995	6, 458	125	220
	6	2, 434	7, 973	174	213
`matmat`: collective matrix multiplication, each process computes for one row	2	1, 155	3, 829	37	35
	3	1, 849	6, 096	40	41
	4	2, 551	8, 469	46	49
	5	3, 261	10, 948	52	57
	5	3, 979	13, 533	56	64
`oddEvenSort`: one loop iteration in a parallel odd-even sorting implementation	2	1, 115	3, 505	4	6
	3	2, 141	6, 699	6	8
	4	3, 947	12, 458	8	12
	5	6, 320	20, 217	10	13
	6	10, 713	34, 757	12	19

Table 10.2: The experimental results of verifying MPI collective-style functions with respect to state-requirement and -guarantees. The experiments are under an assumption that no interference can happen.

name	#procs	#states	#seen	#trans	#prover	time(s)
`gather_w`	2	5, 953	5	19, 597	48	12
	3	218, 766	302	764, 096	123	208
`allgather`	2	9, 848	6	32, 855	31	22
	3	164, 155	93	547, 753	92	378
`matmat`	2	3, 116	1	11, 102	39	37
	3	23, 254	17	83, 359	48	75
	4	342, 653	285	1, 229, 132	110	704
`oddEvenSort`	2	1, 824	0	6, 327	4	6
	3	8, 706	8	30, 164	6	10
	4	96, 961	115	338, 889	10	66
`diff1dIter`	2	5, 668	5	19, 745	35	10
	3	492, 163	567	1, 779, 177	182	505
`diff1dExchange`	2	3, 485	3	11, 300	21	8
	3	114, 173	150	388, 751	66	75
`dotProd`	2	1, 711	1	6, 058	6	7
	3	7, 127	5	25, 431	12	15
	4	36, 450	23	130, 936	18	43
	5	225, 223	119	811, 950	24	389

Table 10.3: The experimental results of verifying full validity of MPI collective-style functions.

statements following itself. Therefore, the function contract of `gather_w` shall include a path-requirement asserting that no process shall send a message with a specific tag to root after exiting `gather_w` until the root process exits. Otherwise, the contract system cannot prove the validity of this function with respect to a function contract.

For the same reason, in `allgather` and `matmat` examples, contracts of the collective functions shall be strong enough to make sure that the calls to `gather_w` will never be interfered. For example, in `allgather`, a call to `bcast` follows a call to `gather_w` hence the contract of `bcast` shall guarantee that it cannot interfere `gather_w`.

For the rest of the examples in Table 10.3, interference is impossible since only deterministic send and receive operations or MPI collective functions are used. So for these examples, in addition to what have been verified in the first part, their guarantees are verified in the second part.

Most of the examples in the second part can hardly be scaled to more than 3

processes within 1, 000 seconds. Compare to the corresponding result in the first part, the full validity verification of an example has to explore much more executions that are commutative to the ones already explored in the first part. In theory, commutative executions will eventually run into a same state. This explains why the numbers of prover calls of two corresponding runs in the first part and the second part respectively are close though the states they explored differ significantly. State-guarantees are barely repeatedly checked for commutative executions. For example, the contract system explores only 3, 947 states for `oddEvenSort` with 4 processes in the first part while it explores almost 25 times more states for `oddEvenSort` with the same setting in the second part. The difference in the number of prover calls is only 2.

We can conclude that the current POR algorithm is sound for full validity verification but can hardly claim its effectiveness in performance. Either better POR algorithm or new approach for check absence assertions is needed.

Chapter 11

SUMMARY AND FUTURE WORK

11.1 Dissertation Summary

The correctness of MPI applications is critical. Formal verification techniques have been showed effective in finding standard property violations in MPI programs but there is little contribution in specifying and verifying the functional correctness of MPI programs.

Procedure contracts have been widely used for formally specifying sequential programs. Existing contract languages, such as ACSL for C or JML for Java, provide rich primitives to express various functional correctness properties. A contract specifies a sequential procedure by describing its input and output. But for parallel MPI programs, the behavior of a procedure depends not only on its input but also on other components running in parallel. Hence procedure contracts cannot be directly applied to MPI programs.

The first contribution made by this dissertation is a new theory for adapting procedure contracts for general message-passing programs. To illustrate the theory precisely, a toy message-passing programming language, MiniMP, was invented (Chapter 4). A specification language for specifying functional and communication correctness of MiniMP was designed (Chapter 5).

Hoare logic is the basis of the sequential procedure contract approach in that a procedure with a contract is represented by a Hoare triple. With the inspiration given by Hoare logic and various its extensions, a MiniMP procedure with its specification is represented by a collective triple (Chapter 5). Specially, not every but only collective-style MiniMP procedures can be specified as valid collective triples. The collective-style procedure based approach preserves the advantages of its sequential origination:

1) it provides formal specification for message-passing programs; 2) it is suitable for all MiniMP programs and is able to divide a program into several small enough procedures in most cases. In addition, it abstracts a certain level of communication details away from functional correctness reasoning.

Similar to Hoare logic, a set of inference rules for collective triples are defined in Chapter 6. A proof of soundness of these rules are given. But the rules are not complete. That means not every collective triple can be proved valid by only applying the rules with a set of boolean assumptions. Mostly, these rules are used for decomposing a collective triple into a number of sub-triples, each of which specifies a sub-procedure.

To verify the validity of collective triples, a model checking and symbolic execution based verification approach was introduced (Chapter 7). By leveraging the automation of model checking and symbolic execution, a collective triple can be automatically verified in a modular and composite way.

Another main contribution of this dissertation is bringing the theory from a toy language to MPI (Chapter 9). This includes a contract programming language for MPI collective-style functions and a contact system implementation. The contract language for C/MPI programs was extended from ACSL. The extension comprises a set of constructs for concurrency and MPI. The contract system was implemented in a general verification framework called CIVL. It takes in C/MPI programs, where function contracts are annotated, and verifies whether collective-style functions satisfy their contracts automatically.

This system was evaluated with a set of MPI functions (Chapter 10). The evaluation shows that the MPI contract language is expressive enough to specify various properties of MPI functions, such as the correctness of data distribution, sortedness and permutation. A contract can be shared by various implementations of a function. Verification results can be effectively re-used. Moreover, by re-using the contracts of verified functions, irrelevant details in those function definitions are abstracted away. The contract system is able to handle arbitrary inputs except for the number of processes. Without considering interferences, the satisfaction to state-requirements and

-guarantees by all the tested MPI functions can be verified with up to 6 processes within a reasonable amount of time. However, full validity verification cannot be scaled to more than three processes due to the coarseness of modified POR algorithm for checking path predicates.

11.2 Future Work

The current contract system can be improved in the following directions.

Separating State and Path Predicate Verification. By observation, for a contracted MPI function, the verification of the validity of a state-guarantee is barely dependent to verifying path-guarantees and interference freedom.

We have showed in the evaluation in Chapter 10 that without considering interference and absence assertion, the contract system is effective. On the other hand, if not proving the state-guarantees, first, there will be less prover calls. Second, state-requirements, as well as the state-guarantees of the called functions, can be simplified accordingly. This can result in much simpler path conditions, which are periodically reasoned about during a verification run.

The verification for path-guarantee and interference freedom is mainly about the communication patterns. Isolating such verification enables the possibility of applying other approaches or other model checkers, which are specific for communication correctness or have been optimized for checking path predicates.

Implementing Collective Loops. The contract system uses loop invariants to perform unbounded verification for sequential (or process-local) loops. MPI applications may also contain "parallel loops", of which the body is a collective-style procedure. Such as the iterative computation process in parallel diffusion solvers or the iterative sorting process in the parallel odd-even sorting algorithm. The current contract system is not able to deal with such parallel loops.

In [103], Siegel et al. introduced a symbolic execution based approach for proving loop invariants of parallel loops. The approach takes care of the complicated situation that multiple processes can be in different iteration levels at a state during a collective

execution of a parallel loop. However, for MPI programs, according to the loop rule defined in our inference system in Chapter 6, if one can prove that for any two consecutive iterations, no interference can happen, the loop is functionally equivalent to a new loop that is obtained by wrapping the old loop body with a pair of "bulk-synchronous" barriers. For the new loop, the loop body can be treated as a collective-style procedure and the loop invariants can be proved by induction on a contract of the body.

Parametric Verification with Global Invariants. Global invariants are used by static verifiers [4, 13, 18, 25] to perform parametric verification for concurrent programs. The basic idea was explained in §**??** that one must show that the global invariant cannot be violated at any location by any statement of a program. Particularly, [78] shows how to use a global invariant to statically prove properties for a simple message-passing program.

The main challenge in the use of global invariants is that for a program, especially a concurrent program, a strong enough global invariant is notoriously hard to be found. We observe that, in [78], a big portion of the global invariant is for expressing communication correctness. At the same time, one of the advantage of our MPI contract approach is to abstract away communication details. In the future, we will exploit the possibility of combining global invariants with function contracts for MPI program verification. By annotating contracts to collective-style functions, there will be less communication details that are visible to the global invariant. Thus, hopefully, the user can pay less attention on the side of communication correctness when attempting to figure out a proper global invariant for a message-passing program.

BIBLIOGRAPHY

[1] ACSL: ANSI/ISO C Specification Language. http://frama-c.com/acsl.html. Accessed Sep. 6, 2019.

[2] Ada Reference Manual 2012. http://docs.adacore.com/live/wave/arm12/html/arm12/arm12.html. Accessed Sep. 6, 2019.

[3] SPARK 2014 Reference Manual. http://docs.adacore.com/spark2014-docs/html/lrm/. Accessed Sep. 6, 2019.

[4] S. Amani, J. Andronick, M. Bortin, C. Lewis, C. Rizkallah, and J. Tuong. COMPLEX: A Verification Framework for Concurrent Imperative Programs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 138–150, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3018610.3018627.

[5] W. Araujo, L. C. Briand, and Y. Labiche. Enabling the Runtime Assertion Checking of Concurrent Contracts for the Java Modeling Language. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 786–795, New York, NY, USA, 2011. ACM. http://doi.acm.org/10.1145/1985793.1985903.

[6] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[7] E. A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, Feb. 1975.

[8] T. Ball, B. Hackett, S. Lahiri, S. Qadeer, and J. Vanegue. Towards Scalable Modular Checking of User-Defined Properties. In *Verified Software: Theories, Tools and Experiments (VSTTE 2010)*. Springer Verlag, August 2010. https://www.microsoft.com/en-us/research/publication/towards-scalable-modular-checking-of-user-defined-properties-2/.

[9] J. Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK, 2012. http://www.altran.co.uk,UK.

[10] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://doi.org/10.1007/11804192_17.

[11] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-30569-9_3.

[12] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177, 2011.

[13] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007. http://doi.org/10.1007/978-3-540-69061-0.

[14] J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of process algebra*. Elsevier, 2001.

[15] M. Bernaschi, G. Iannello, and M. Lauria. Efficient Implementation of Reduce-scatter in MPI. In *Proceedings of the 10th Euromicro Conference on Parallel, Distributed and Network-based Processing*, EUROMICRO-PDP'02, pages 301–308, Washington, DC, USA, 2002. IEEE Computer Society.

[16] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. J. Naughton, III, H. P. Pritchard, M. Schulz, and G. R. Vallee. A Survey of MPI Usage in the U.S. Exascale Computing Project. Technical Report 790, Oak Ridge National Lab., June 2018. https://doi.org/10.2172/1462877.

[17] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded Model Checking. *Advances in computers*, 58(11):117–148, 2003.

[18] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing. https://doi.org/10.1007/978-3-319-66845-1_7.

[19] I. Board. Ieee Standard Classification for Software Anomalies. *IEEE Std*, 1044, 1993.

[20] G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Proceedings of the 7th Annual IEEE/ACM International*

Symposium on Code Generation and Optimization, CGO '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. http://dx.doi.org/10.1109/CGO.2009.32.

[21] M. Christakis and K. Sagonas. Detection of asynchronous message passing errors using static analysis. In R. Rocha and J. Launchbury, editors, *Practical Aspects of Declarative Languages*, pages 5–18, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[22] Clang static analyzer. http://clang-analyzer.llvm.org. Accessed Sep. 6, 2019.

[23] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[24] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, Sep. 1976. http://doi.org/10.1109/TSE.1976.233817.

[25] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin / Heidelberg, 2009. http://dx.doi.org/10.1007/978-3-642-03359-9_2.

[26] S. Conchon, M. Iguernelala, and A. Mebsout. A Collaborative Framework for Non-Linear Integer Arithmetic Reasoning in Alt-Ergo. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 161–168, Sep. 2013. http://doi.org/10.1109/SYNASC.2013.29.

[27] *Correctness'17: Proceedings of the First International Workshop on Software Correctness for HPC Applications*, New York, NY, USA, 2017. ACM. https://correctness-workshop.github.io/2017/.

[28] *Correctness'18: Proceedings of the Second International Workshop on Software Correctness for HPC Applications*, Los Alamitos, CA, USA, nov 2018. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/Correctness.2018.00004.

[29] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An overview of the mcrl2 toolset and its recent advances. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 199–213, Berlin, Heidelberg, 2013. Springer-Verlag.

[30] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, pages 233–247, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33826-7_16.

[31] P. Cuoq, B. Monate, A. Pacalet, and V. Prevosto. Functional dependencies of c functions via weakest pre-conditions. *International Journal on Software Tools for Technology Transfer*, 13(5):405–417, Oct 2011. "https://doi.org/10.1007/s10009-011-0192-z".

[32] R. B. Dannenberg and G. W. Ernst. Formal Program Verification Using Symbolic Execution. *IEEE Transactions on Software Engineering*, SE-8(1):43–52, Jan 1982. 10.1109/TSE.1982.234773.

[33] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[34] J. B. Drake, P. W. Jones, and G. R. Carr. Overview of the Software Design of the Community Climate System Model. *IJHPCA*, 19:177–186, 2005. https://journals.sagepub.com/doi/pdf/10.1177/1094342005056094.

[35] A. Droste, M. Kuhn, and T. Ludwig. MPI-Checker: Static Analysis for MPI. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 3:1–3:10, New York, NY, USA, 2015. ACM. http://doi.acm.org/10.1145/2833157.2833159.

[36] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 411–420, May 1999. http://doi.org/10.1145/302405.302672.

[37] R. D. Falgout and U. M. Yang. *hypre*: A Library of High Performance Preconditioners. In P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, editors, *Computational Science — ICCS 2002*, pages 632–641, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-47789-6_66.

[38] J.-C. Filliâtre and A. Paskevich. Why3 — Where Programs Meet Provers. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-37036-6_8.

[39] R. W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993. https://doi.org/10.1007/978-94-011-1793-7_4.

[40] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma. Precise predictive analysis for discovering communication deadlocks in mpi programs. *ACM Trans. Program. Lang. Syst.*, 39(4):15:1–15:27, Aug. 2017.

[41] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In C. Courcoubetis, editor, *Computer Aided Verification*, pages 438–449, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[42] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama. Report of the HPC Correctness Summit. Technical report, USDOE Office of Science (SC), 2017. https://arxiv.org/abs/1705.07478.

[43] M. J. C. Gordon. *Mechanizing Programming Logics in Higher Order Logic*, pages 387–439. Springer New York, New York, NY, 1989. https://doi.org/10.1007/978-1-4612-3658-0_10.

[44] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[45] L. Hatton. The T experiments: Errors in scientific software. *IEEE Computational Science & Engineering*, 4(2):27–38, Apr. 1997.

[46] L. Hatton. Defects, scientific computation and the scientific method. In A. M. Dienstfrey and R. F. Boisvert, editors, *Uncertainty Quantification in Scientific Computing*, volume 377 of *IFIP Advances in Information and Communication Technology*, pages 123–138. Springer Berlin Heidelberg, 2012.

[47] L. Hatton and A. Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785–797, Oct. 1994.

[48] M. Hentschel, R. Bubel, and R. Hähnle. The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, 2018.

[49] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. Mpi runtime error detection with must: Advances in deadlock detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 30:1–30:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. http://dl.acm.org/citation.cfm?id=2388996.2389037.

[50] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[51] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[52] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[53] P. Huchant, E. Saillard, D. Barthou, and P. Carribault. Multi-valued expression analysis for collective checking. In R. Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 29–43, Cham, 2019. Springer International Publishing. https://doi.org/10.1007/978-3-030-29400-7_3.

[54] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 989:2011 N1570: Programming Languages – C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, Apr. 2011.

[55] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[56] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag. http://dl.acm.org/citation.cfm?id=1947873.1947902.

[57] Java SE 8. https://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html. Accessed Sep. 6, 2019.

[58] H. Johansen, A. Rodgers, N. A. Petersson, D. McCallen, B. Sjogreen, and M. Miah. Toward Exascale Earthquake Ground Motion Simulations for Near-Fault Engineering Analysis. *Computing in Science Engineering*, 19(5):27–37, 2017. http://doi.org/10.1109/MCSE.2017.3421558.

[59] Q. Kang, J. L. TrÃdff, R. Al-Bahrani, A. Agrawal, A. Choudhary, and W. keng Liao. Scalable algorithms for mpi intergroup allgather and allgatherv. *Parallel Computing*, 85:220 – 230, 2019.

[60] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 489–507, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0013032.

[61] S. Kauer and J. F. H. Winkler. Mechanical Inference of Invariants for FOR-loops. *J. Symb. Comput.*, 45(11):1101–1113, Nov. 2010.

[62] D. Khanna, S. Sharma, C. Rodríguez, and R. Purandare. Dynamic Symbolic Verification of MPI Programs. In K. Havelund, J. Peleska, B. Roscoe, and

E. de Vink, editors, *Formal Methods*, pages 466–484, Cham, 2018. Springer International Publishing. https://doi.org/10.1007/978-3-319-95582-7_28.

[63] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. https://doi.org/10.1145/360248.360252.

[64] T. Kleymann. Hoare Logic and Auxiliary Variables. *Form. Asp. Comput.*, 11(5):541–566, Dec. 1999.

[65] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, pages 1–35, New York, NY, USA, 2013. Springer-Verlag New York, Inc. http://dx.doi.org/10.1007/978-3-642-39799-8_1.

[66] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. Marmot: An mpi analysis and checking tool. In *Advances in Parallel Computing*, volume 13, pages 493–500. Elsevier, 2004. https://doi.org/10.1016/S0927-5452(04)80063-7.

[67] W. Krenn and B. K. Aichernig. Test Case Generation by Contract Mutation in Spec#. *Electronic Notes in Theoretical Computer Science*, 253(2):71 – 86, 2009. http://www.sciencedirect.com/science/article/pii/S157106610900406X.

[68] D. Kroening and M. Tautschnig. Cbmc – c bounded model checker. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[69] S. Lahiri. Security audit using extended static checking: Is it cost-effective yet? Technical Report MSR-TR-2012-103, microsoft, October 2012. https://www.microsoft.com/en-us/research/publication/security-audit-using-extended-static-checking-is-it-cost-effective-yet/.

[70] L. Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14(1):21–37, Jun 1980. https://doi.org/10.1007/BF00289062.

[71] G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*, pages 175–188. Springer US, Boston, MA, 1999. https://doi.org/10.1007/978-1-4615-5229-1_12.

[72] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-17511-4_20.

[73] K. R. M. Leino, P. Müller, and J. Smans. *Verification of Concurrent Programs with Chalice*, pages 195–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. https://doi.org/10.1007/978-3-642-03829-7_7.

[74] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking java programs via guarded commands. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 110–111, London, UK, UK, 1999. Springer-Verlag. http://dl.acm.org/citation.cfm?id=646779.704969.

[75] R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM*, 18(12):717–721, Dec. 1975.

[76] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida. Protocol-based Verification of Message-passing Parallel Programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 280–298, New York, NY, USA, 2015. ACM. http://doi.acm.org/10.1145/2814270.2814302.

[77] Z. Luo and S. F. Siegel. Symbolic Execution and Deductive Verification Approaches to VerifyThis 2017 Challenges. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 160–178, Cham, 2018. Springer International Publishing.

[78] Z. Luo and S. F. Siegel. Towards deductive verification of message-passing parallel programs. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 59–68, Nov 2018. https://doi.org/10.1109/Correctness.2018.00012.

[79] Z. Luo, M. Zheng, and S. F. Siegel. Verification of mpi programs using civl. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, pages 6:1–6:11, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3127024.3127032.

[80] Z. Manna and A. Pnueli. Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs. *Sci. Comput. Program.*, 4(3):257–289, Dec. 1984.

[81] A. Mathuriya, Y. Luo, R. C. Clay, III, A. Benali, L. Shulenburger, and J. Kim. Embracing a New Era of Highly Efficient and Productive Quantum Monte Carlo Simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 38:1–38:12, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3126908.3126952.

[82] K. L. McMillan. *Symbolic Model Checking*, pages 25–60. Springer US, Boston, MA, 1993. https://doi.org/10.1007/978-1-4615-3190-6_3.

[83] K. L. McMillan. Lazy Annotation for Program Testing and Verification. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 104–118, Berlin, Heidelberg, 2010. Springer-Verlag.

[84] Z. Merali. Error: why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010.

[85] Message Passing Interface Forum. MPI: A Message-Passing Interface standard, version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, June 2015.

[86] B. Meyer. Eiffel: A language and environment for software engineering. *J. Syst. Softw.*, 8(3):199–246, June 1988.

[87] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, Oct. 1992.

[88] G. Newton. Proving Properties of Interacting Processes. *Acta Informatica*, 4(2):117–126, Jun 1975.

[89] W. Oortwijn, S. Blom, and M. Huisman. Future-based Static Analysis of Message Passing Programs. In D. Orchard and N. Yoshida, editors, *Proceedings PLACES 2016*, pages 65–72, 4 2016. https://doi.org/10.4204/EPTCS.211.7.

[90] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I*. *Acta Informatica*, 6(4):319–340, Dec 1976.

[91] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, July 1982.

[92] R. Palmer, G. Gopalakrishnan, and R. M. Kirby. Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '07, pages 43–53, New York, NY, USA, 2007. ACM. http://doi.acm.org/10.1145/1273647.1273657.

[93] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm. AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 157–168, New York, NY, USA, 2014. ACM. http://doi.acm.org/10.1145/2577080.2577084.

[94] J. C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, July 2002. http://doi.org/10.1109/LICS.2002.1029817.

[95] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy*, 82:90 – 97, 2015. http://www.sciencedirect.com/science/article/pii/S030645491400379X.

[96] M. Ruefenacht, M. Bull, and S. Booth. Generalisation of Recursive Doubling for Allreduce. *Parallel Comput.*, 69(C):24–44, Nov. 2017.

[97] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear Loop Invariant Generation using Gröbner Bases, 2004.

[98] S. F. Siegel. Efficient verification of halting properties for mpi programs with wildcard receives. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, pages 413–429, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-30579-8_27.

[99] S. F. Siegel. Verifying parallel programs with mpi-spin. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 13–14, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[100] S. F. Siegel and G. S. Avrunin. Modeling wildcard-free mpi programs for verification. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 95–106, New York, NY, USA, 2005. ACM. http://doi.acm.org/10.1145/1065944.1065957.

[101] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 61:1–61:12, New York, 2015. ACM. http://doi.acm.org/10.1145/2807591.2807635.

[102] S. F. Siegel and T. K. Zirkel. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science*, 5(4):395–426, 2011. https://doi.org/10.1007/s11786-011-0100-7.

[103] S. F. Siegel and T. K. Zirkel. Loop Invariant Symbolic Execution for Parallel Programs. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 412–427, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-27940-9_27.

[104] E. W. Stark. A Proof Technique for Rely/Guarantee Properties. In *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 369–391, Berlin, Heidelberg, 1985. Springer-Verlag.

[105] M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-Flow Analysis for MPI Programs. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society. http://dx.doi.org/10.1109/ICPP.2006.32.

[106] T. Takaoka. Parallel Program Verification with Directed Graphs. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, SAC '94, pages 462–466, New York, NY, USA, 1994. ACM. http://doi.acm.org/10.1145/326619.326811.

[107] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, Feb. 2005.

[108] J. L. TrÃdff. On Optimal Trees for Irregular Gather and Scatter Collectives. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2060–2074, Sep. 2019.

[109] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, pages 66–79, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-70545-1_9.

[110] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010. http://www.sciencedirect.com/science/article/pii/S0010465510001438.

[111] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, Dec 1992. https://doi.org/10.1007/BF00709154.

[112] M. T. Vandevoorde and J. V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *In Proceedings of the 1994 ACM/SIGSOFT Foundations of Software Engineering Conference*, pages 121–127, 1994. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.7940.

[113] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[114] U. Yang, R. Falgout, and J. Park. Algebraic Multigrid Benchmark, Version 00, 8 2017. https://www.osti.gov//servlets/purl/1389816.

[115] F. Ye, J. Zhao, and V. Sarkar. Detecting MPI Usage Anomalies via Partial Program Symbolic Execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18,

pages 63:1–63:5, Piscataway, NJ, USA, 2018. IEEE Press. https://doi.org/10.1109/SC.2018.00066.

[116] H. Yu. Combining symbolic execution and model checking to verify mpi programs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 527–530, New York, NY, USA, 2018. ACM. http://doi.acm.org/10.1145/3183440.3190336.

[117] M. Zheng, J. G. Edenhofner, Z. Luo, M. J. Gerrard, M. S. Rogers, M. B. Dwyer, , and S. F. Siegel. CIVL: Applying a general concurrency verification framework to c/pthreads programs (competition contribution). In *TACAS 16: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Proceedings*, TACAS '16, Eindhoven, The Netherlands, Apr 2016. Springer-Verlag.

[118] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. CIVL: Formal verification of parallel programs. In *ASE 2015: 30th IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, ASE '15, Piscataway, NJ, USA, Nov 2015. IEEE Press.

[119] W. Zheng and G. Bundell. Test by Contract for UML-Based Software Component Testing. In *International Symposium on Computer Science and its Applications*, pages 377–382, Oct 2008. http://doi.org/10.1109/CSA.2008.66.