STATIC ANALYSIS OF A CONCURRENT PROGRAMMING LANGUAGE BY ABSTRACT INTERPRETATION

Maryam Zakeryfar

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy (Computer Science)
Concordia University
Montréal, Québec, Canada

March 2014

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mrs. Maryam Zakeryfar**

Entitled: **Static Analysis of a Concurrent Programming Language by Abstract Interpretation**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Deborah Dysart-Gale _____ Chair

Dr. Weichang Du _____ External Examiner

Dr. Mourad Debbabi _____ Examiner

Dr. Olga Ormandjieva _____ Examiner

Dr. Joey Paquet _____ Examiner

Dr. Peter Grogono _____ Supervisor

Approved by _____

Dr. V. Haarslev, Graduate Program Director

_____ _____ _____

Christopher W. Trueman, Dean

Faculty of Engineering and Computer Science

Abstract

Static Analysis of a Concurrent Programming Language by Abstract Interpretation

Maryam Zakeryfar, Ph.D.

Concordia University, 2014

Static analysis is an approach to determine information about the program without actually executing it. There has been much research in the static analysis of concurrent programs. However, very little academic research has been done on the formal analysis of message passing or process-oriented languages. We currently miss formal analysis tools and techniques for concurrent process-oriented languages such as Erasmus . In this dissertation, we focus on the problem of static analysis of large Erasmus programs. This can help us toward building more reliable Erasmus software systems.

Reasoning about non-deterministic large Erasmus program using static analyzer is hard. These kinds of programs can quickly exhaust the computational and memory resources of the static analyzer tool. We use Abstract Interpretation to reason about Erasmus programs. To use the Abstract Interpretation theory, we introduce a lattice for Erasmus communications and an Event Order Predictor algorithm to statically determine the order that events happen in an Erasmus program. By using fixed-point theory of lattice, we compute a safe approximation of reachable states of an Erasmus programs.

We also offer a Resettable Event order Vector for Erasmus processes to realistically implement our vector for large Erasmus programs using bounded space. We believe that our formal approach is also applicable to other types of process-oriented programs and MPI programs.

Acknowledgements

It all started with my mother's dream. All her life she had aspired to higher learning but life had clipped her wings; her dream of piercing through the clouds of un-knowing became my inspiration for pursuing higher education. Keeping her dream alive inspired me to fly, allowing me to share the joy with her. It was a wonderful journey.

I was truly blessed to work under the guidance of a brilliant professor: Dr Peter Grogono, who helped me pursue my goal with respect, humour and great patience. I also want to thank my defence committee: Dr Joey Paquet, Dr Olga Ormandjieva, Dr Mourad Debbabi, and Dr Weichang Du for their thoughtful and meticulous feedback, shedding light where there was darkness. Finally, I am very grateful to my colleagues in the lab, Nima Jafroodi and Tamer Abdou for keeping me on track.

Despite living so far away, my family was always by my side. My mother inspired me with her wisdom; my father continuously fuelled my dream encouraging me to push my limits, to work hard, to dream big. The unconditional love of my siblings Mehri, Mehrnaz and Daniel, provided the wind under my wings, propelling me to new heights. None of this could have happened without them and I cannot thank them enough.

I met my husband Georges on a beautiful summer day in Montreal. Ever since that day he has travelled alongside me, showing me his love and expressing joy in my accomplishment. This thesis is dedicated to him.

Contents

Program Listings

List of Tables

List of Figures

Chapter 1

INTRODUCTION

Static analysis of programs has been a significant topic for all categories of programming languages. In this thesis, we investigate the static analysis of a process-oriented programming language called: Erasmus . This chapter gives an overview of the structure of the thesis and its contributions. Section 1.1 gives a short introduction to the problem statement of the thesis. Section 1.2 reviews the thesis contributions. Section 1.3 provides the thesis organization.

1.1 Problem Statement

The purpose of the static analysis is to detect properties of programs without actually executing them. This can help programmers in finding bugs early in the software development phase. There has been much research in the static analysis of computer programs for both sequential and concurrent programs. However, most of the existing research [12, 1, 59, 42, 31, 46, 41, 45, 52, 16] has been carried out on object-oriented and CSP languages and is not targeted for message passing or process-oriented languages. Looking at the existing works and research in the static analysis of concurrent programs, we see that despite the importance of large-scale process-oriented programs, very little academic research has been done on the formal analysis of this type of programs. We currently lack formal analysis tools and techniques to analyze the communication properties of concurrent applications such as Erasmus –a process-oriented language– and message passing programs.

Static analysis of small Erasmus programs seems trivial, However, since one of the main goals of Erasmus is to facilitate writing large programs, we need to consider an Erasmus program with many processes and cells. The main challenge arises when a process-oriented application is large and generates many non-deterministic calls. Such applications can quickly exhaust the computational as well as memory resources of the static analysis tools [26].

We use Abstract Interpretation (AI) to reason about large Erasmus programs. To the best of our knowledge, there has been no previous research in using AI in formal analysis of process-oriented programs. Cousot [14] tried applying AI concepts on CSP but the amount of computation needed to do the analysis made the applicability of his work to CSP languages very difficult. Siegel [57] also emphasized the need of applying AI on message-passing programs:

> Other interesting avenues of research include applications of static analysis and Abstract Interpretation to MPI or other message-passing systems. These approaches could potentially reason without bounds on parameters or process counts. Yet very little research has been done in this area [57].

Developing the right approach for reasoning about concurrent Erasmus programs is an important step toward building reliable Erasmus software systems.

1.2 Contributions

Generally, concurrent programs are non-deterministic in their execution order, unless the programmer carefully writes a deterministic concurrent program. In this thesis, we target both deterministic and non-deterministic Erasmus programs and make the following contributions:

- Since our focus in this research is communication analysis, the existing domains in abstract interpretation theory such as intervals, convex hull and polyhedra, etc [15] are not useful for us. Therefore, we offer a novel communication domain whose con-

cretization and abstraction function are introduced. The abstract domain can be added to the existing abstract domains in Abstract Interpretation theory [15].

- To build a lattice for event communications for POP programs, we propose an Event Order Predictor (EOP) where the events orders are determined statically. Our approach is inspired by Vector Clock and it predicts the order of events for all possible paths. Event Order Predictor may be applicable to the formal analysis of other concurrent programs. From the result of our first experiment, we realize that implementing EOP for non-deterministic Erasmus programs is not practically possible. Therefore, we replace the vector with a Resettable Event Order Vector.

- We construct a Resettable Event Order Vector for Erasmus programs that contain **loop**, **select**, **loopselect** and **case** control structures. The Reset conditions and EOP algorithms are provided based on the properties of Erasmus language.

- To reason about Erasmus programs we propose a generic approach for static analysis of concurrent Erasmus programs that contain **loop**, **select**, **loopselect** and **case** control structures. By using AI theory, we compute the fixed point of only reachable states of the program. Therefore, we partially avoid the issue of state explosion. We also use EOP to realistically implement our vector using bounded space. This, reduces the size of vector. We prove the correctness of our EOP using mathematical theory. We define a vector order for Erasmus Cell. This way, we encapsulate the details of processes within a Cell. This, reduces the length of the vector. Detecting dead process(es) and omitting the dead process from the vector also reduces the length of Event Order Vector.

1.3 Thesis Organization

The thesis is structured as follows: Chapter 2 reviews the theoretical background of the research. It explains the communication of processes in Erasmus programming language, the mathematical foundations to understand Abstract Interpretation technique and two popular algorithms for dynamically building a partial order of events in distributed systems:

Lamport Clock and Vector Clock. Chapter 3 presents our novel communication domain. The concretization and abstraction functions are also introduced in this chapter. Chapter 4 presents the Event Order Predictor(EOP) for process-oriented programs. The implementation details of Event Order Vector is also provided in this chapter. Chapter 5 offers a reasoning approach to reason about non-deterministic synchronous Erasmus programs that contain **loop**, **select** and **case** statements. The chapter presents a practical evaluation to evaluate our reasoning methodology in detecting deadlock and successful communication in Erasmus programs. A comparison of the new approach with the existing tools and techniques is provided at the end of the chapter. Chapter 6 shows our experiment in partially implementing the concepts of EOP in Erasmus analysis. The goal of the chapter is to investigate the possibility of reasoning about deterministic concurrent synchronous programs by Event Order predictor and Abstract Interpretation theory. This experiment makes understand the limitations of the EOP approach. To overcome the efficiency limitation, we propose a Resettable vector in our EOP approach. Chapter 7 summarizes the closely related works and compares them to our work. Finally, in chapter 8 thesis conclusions and possible future works are provided.

Chapter 2

BACKGROUND

This chapter covers fundamental elements that are used in this research. Section 2.2 presents an overview of most widely used verification techniques. Section 2.3 explains the existing approaches for static analysis of programs. Section 2.4 presents the syntax of Erasmus programming language. In Section 2.5, the basic definitions and terminologies for the Abstract Interpretation are presented. Section 2.7 explains two popular algorithms to generate partial order of events in distributed systems. Last Section 2.8 summarizes the material discussed in this chapter.

2.1 Glossary of Terms

A list of Glossary of terms that is used throughout this chapter is given in Table 2.1. The explanations of terms are given on the right.

2.2 Verification Techniques

Choosing the right technique(s) for the verification of Erasmus program is a non-trivial problem. First, the chosen technique should scale to large and complex programs. Second, it should address the issue of the state explosion problem. In the following section, a survey of the existing verification techniques will be presented.

Term	Explanation
Correctness	Correctness of a a program is asserted when it is said that a program is correct with respect to a specification. For example, a program is correct when it does or does not have certain properties.
Halting problem	Given a description of a computer program, decide whether the program finishes running or continues to run forever.
Undecidability	There is no universal procedure to determine whether an arbitrary program does or does not possess the property. Example: Halting problem.
Deadlock	Deadlock in concurrent programs happens when a process is waiting for a resource that is being held by another process that is also waiting for another resource. The result is that both processes freeze.
Starvation	Starvation occurs when a process is waiting for a resource that keeps getting given to other processes. Therefore, the process starves.
Fairness	Granting that each process will be given the access to a resource equally (according to each process priority).
Abstract Interpretation Terms	
Forbidden Zone	The set of states in which certain properties do not hold.
Safety Property	A property of a program that ensures that no possible executions of the program can reach a forbidden zone.
Concrete semantics	Semantics that describe the possible behaviours of programs during their execution.
Collecting semantics	A class of properties of program executions derived from concrete semantics. The purpose of a "collecting" semantics is to collect information about the program by interpreting it.
Concretization function	A function that maps abstract properties to concrete properties.
Abstract function	A function that maps concrete properties to abstract properties.
Fixed Point	A point that is mapped to itself within an order-preserving self map.

Table 2.1: Glossary of Terms.

2.2.1 Theorem-proving

In general, theorem-proving can be applied to the semantics of a language to show that programs have particular properties. One such property might be correctness, assuming that this can be formalized. Program properties in sequential program can be delivery

of correct results and termination; whereas in concurrent programs the properties of interference freedom, deadlock freedom and fairness are critical. In this technique which is known as formal verification, a specification notation with formal semantics, along with a deductive apparatus for reasoning, are used for analysis of the program. The history of theorem-proving as a means of establishing properties, goes back to 1960s and 70s, when Hoare [34] suggested a logical basis for reasoning about the properties of a program. The original idea was from Floyd [24]. Floyd [25] also proposed a basic for formal definition of the meaning of program similar to [34]. *The Science of Programming* [29] is the very first text book that used proof techniques for program development. The book is constructed based on some of techniques suggested by Dijkstra [20]. In this book, Dijkstra identifies some of the weaknesses of programming languages and considered a program to be a mathematical structure.

Verifying programs by theorem-proving is an activity that requires insight as well as significant mathematical calculations. However, the fully automatic verification of the program using theorem proving is an undecidable problem and therefore impossible to implement.

2.2.2 Model Checking

Another verification technique to establishes program properties is model checking. This technique is mostly used after the program has been abstracted to a finite state system. It consists of constructing a system model along with specifications to ensure the system model conforms to the specification. This model must preserve the properties of the program that we wish to establish. Ideally, it should abstract out everything that we are not interested in. For instance, the model can include only communication statements but ignore other statements. Therefore, the model can check communication properties but cannot check, e.g., overflow. Model checking explores all the possible states and transitions of a particular program. Also, this verification technique can be automated. Model checking can be summarized in three general phases as follows:

- **Modelling**: Converting a design into formalism so mathematical computation can be performed.

- **Specification**: Stating the properties that belong to the system. Model should respect these properties.
- **Verification**: Verifying the model against specification using mathematical computation and logical deduction.

The history of model checking in software systems goes back to the early 1980s when Queille and Sifakis presented *CESAR* as an interactive system for analysis of properties of parallel systems [53]. The main idea was validating the system against a particular set of specifications expressed as formulas. The specification language of *CESAR* was branching time logic and fixed points of monotonic predicate transformers considered as a specification [40]. After automatic translation of program's description into interpreted Petri nets, each formula of the specifications is evaluated. At about the same time, Clarke and Emerson also published their article using branching time temporal logic [10]. Nowadays, almost any system has a vast number of states. The size of a model depends on the number of possible states, and this number typically increases exponentially with the number of processes and variables [63]. This is a major obstacle for model checking called *state space explosion* problem. Many techniques have been developed to reduce the complexity of the state in model checking. Binary Decision Diagram, Partial Order Reduction, Compositional Reasoning and Abstraction are some of the well-known techniques for this purpose. Another issue of model-checking technique is the need of encoding the program properties into temporal logic formulas which relies on the experienced user. Therefore, the adjustment of technique is limited for an inexperienced user [17].

2.2.3 Static Analysis

The main goal of static analysis is to compute the set of states that arise during the execution of the program by providing the set of initial states. The first step in static analysis is to express the semantics of the program as a set of equations. Next step is to solve the equations iteratively over some abstract domain. If the abstract domain satisfies the ascending-chain condition, an iterative technique can produce the fixed-point of the equation as the most precise solution for the equations. Difficulties may arise when the domain

contains infinite strictly-increasing chains which make the solution obtained imprecise. The analyzer provides a positive or negative result for a property: we don't know whether it is true or false. The tool has to be calibrated with systems for which we know the answers. Therefore, the result of the static analysis can be one of the *true positive*, *true negative*, *false positive*, *false negative*

	Positive	Negative
True	True Positive	True Negative
False	False Positive	False Negative

Table 2.2: Outcomes of Static Analyzer Tool (SAT). Columns (Negative and Positive) are SAT outcomes. Rows (True and False) shows the situations where SAT results are true or false.

True positive indicates a correct result, accurately informing us of the existence or non-existence of a property such as *deadlock* in the program. *True negative* means that the system does not have the property and the analyzer says that it doesn't. These two correspond to the answers that we would like to get. The other entries contain answers that we do not want but may have to tolerate: *False negative* means that the property is present in the program, but that the analysis indicates that it is not. False negative is a serious problem (worst answer). For example, the analyzer might declare program P to be deadlock-free although it actually isn't. On the other hand, *false positive* means the program does not have the property but the analysis shows that it does. This is not as harmful as *false negative* but may serve as a warning signal

Result Type	**Explanation**
True Positive(TP)	Program has defect and Analyzer says it does.
False Positive (FP)	Program does not have defect, but Analyzer says it does.
False Negative (FN)	Program does have defect, but Analyzer says it does not.
True Negative (TN)	Program does not have defect and Analyzer says it does not.

Table 2.3: Summary of Static Analysis Tool Result Types.

2.3 Classical Static analysis Techniques

Static analysis is performed on the programs without executing them. Finding the right approach for static analysis of Erasmus program is challenging. However, the chosen approach can be a combination of existing approaches. In classical static analysis four main approaches to program analysis are introduced [51] :

- Data Flow Analysis
- Constraint Based Analysis
- Type and Effect Systems
- Abstract Interpretation

There are a lot of commonalities among these four approaches that motivated us to look into each one of them. We explored the possibility of choosing one approach as a the primary approach and utilizing the power of other approaches in our analysis.

2.3.1 Data Flow Analysis

In this analysis, the information about the possible values of program variables at each program point is collected. These information is used to build a data flow equation system for each node of control flow graph of the program. Then, the equation is solved repeatedly until it reaches the fixed point. Data flow analysis was first introduced by Kildall [36]. By applying lattice theory to the static analysis, he brought a mathematical basis to the analysis.

2.3.2 Constraint Based Analysis

Constraint based analysis is a type of control flow analysis. This analysis can be performed in two steps. The first phase emits constraints that a solution to an intended analysis needs to satisfy. The second phase solves the constraints. The advantage of this approach is that the analysis presents itself in an intuitive form and allows for reusable constraint solving software, independent of a particular analysis. A disadvantage of the approach is that the

resulting system of constraints might have high complexity[51]. Constraint based Analysis has been used by many researchers [65, 32, 33, 48]. A more detailed information about this analysis is given in [2].

2.3.3 Type and Effect Systems

In type checking technique, types are used to analyze the behaviours of programs and to enforce the absence of some dynamic errors [50]. The technique is only applicable to typed programming languages. A more detailed development of type and effect systems can be found in chapter 5 of [51]. This technique is useful for analyzing Erasmus programs since Erasmus already has a "type and effect" system: protocols. However, industrial programmers have not responded well to elaborate type systems and their complex formalisms, and we decided to look for more user-friendly approaches.

2.3.4 Abstract Interpretation

The idea of Abstract Interpretation is based on abstracting the behaviour of the program and performing the analysis on the abstract level instead of concrete level. Cousot and Cousot added the solid mathematical foundation to classical static analysis [13]. This theory is based on two fundamental keys: First, building the abstract semantic from concrete semantic and the correspondence between them through *Galois connections*, second computation of fixed point of the abstract semantics. When infinite abstract domains are considered, to ensure the scalability of fixed point analysis, this theory offers widening operators (to get fast convergence) and narrowing operators (to improve the accuracy of the resulting analysis) [11]. Abstract Interpretation technique is sound but not complete; for example, if a property about the program can be proved using Abstract Interpretation, then it is correct (sound), but we can not be certain that all properties about the program can be determined using this technique (not complete). Section 2.5 provides a complete description of Abstract Interpretation.

2.4 Erasmus

Erasmus is a process-oriented programming language that aims to reduce the complications of writing concurrent programs by using only sequential processes that communicate by exchanging messages. The main goal of the Erasmus project is to improve the structure and maintainability of large programs. Erasmus programs contains cells and processes where cells provide structure and processes describe actions within the cells. Cells may may contain processes and processes may contain cells. Processes communicate by exchanging messages through channels and a process can share variables only with other processes in the same cell. Protocol is another part of Erasmus which defines communication type. The protocol concept is similar to the Interface concept in Java language. However, a Java interface only specifies the methods that are provided but does not specify allowed sequences of calls. Erasmus protocol provides the methods as well as the sequences of method calls. Channels are for facilitating the communication of processes or cells. Each channel is connected with a protocol that defines the types of the messages that may be sent through the channel and their processes. Processes have also ports that are connected to channels. A port can be server or a client depends on its direction [30]. We model synchronous communication in Erasmus programming language, defined as follows:

> Two processes, P_1 and P_2, *communicate synchronously* if:
>
> 1. they are both connected to a channel, C; and **either**:
> 2. P_1 sends a message on C and waits until P_2 receives the message; **or**
> 3. P_1 expects to receive a message on C and waits until it has received the message from P_2.

An example of the structure of cells and channel communications in Erasmus programming language is provided in Figure 2.1. *The Sleeping Barber* is a classic inter-process communication problem between multiple processes [19].

2.4.1 The Sleeping Barber

The Sleeping Barber Problem was first introduced by Dijkstra [19].

A barbershop has a single barber and a waiting-room with a number of chairs. If the barber is not busy, he sleeps. If there is a customer, he wakes up and cuts the customer's hair. A new customer acts as follows:

- If the barber is asleep, the customer wakes him up and has his hair cut.
- If the barber is busy, and there are empty chairs in the waiting-room, the customer waits in a chair.
- If the barber is busy and there are no free chairs, the customer goes away.

When the barber finishes with a customer, any waiting customer is served immediately. If there are no waiting customers, the barber goes to sleep.

Figure 2.1 shows the structure of the barber program with cells and processes. In Figure 2.1, the cells have heavy borders and processes have light borders.

The following codes in Listing 2.1 and Listing 2.2 are the Erasmus implementation of Sleeping Barber problem. In Listing 2.1 we represent a customer by a Text, consisting of the customer's name and an annotation. Client process is a process that first sends a query and then receives a response. A minus sign (-) before the protocol name, indicates that the process it is a client. The server has a plus sign (+) before the protocol name. The server

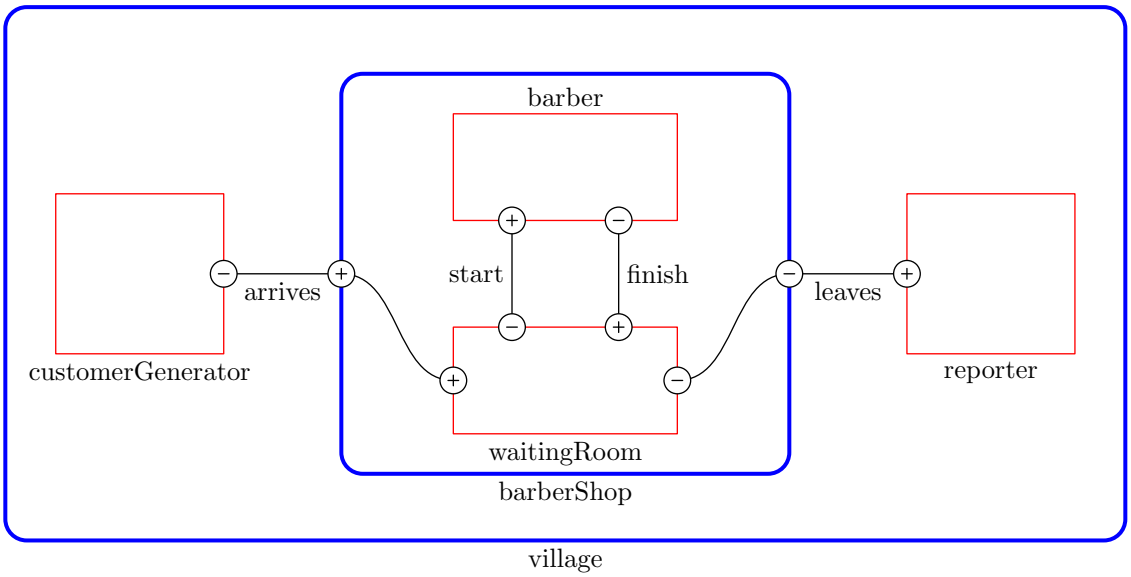


Figure 2.1: Barber Design Diagram

first receives the query from the client and then it sends the response.

The heart of the solution is the process waitingRoom, which simulates the waiting-room. This process has four ports: arrives for customers arriving; leaves for customers leaving; start for customers going to the barber; and finish for customers who have had their hair cut. It shares the variable sleeping with process barber.

Waiting-room has a finite number of chairs. We use an indexed value to represent the chairs in the waiting-room. Initially, each chair is empty. The rest of this process is a loop that performs an action when a customer arrives or when a customer is finished. The first case is split into two, depending on whether the barber is sleeping or busy.

The barber's shop is represented by a Cell that contains instances of barber and waitingRoom, the ports, and the shared variable. The process customerGenerator creates a stream of customers. and the process reporter issues a message as each customer leaves the shop. Finally, the village generates and reports on customers, and has a barber shop.

Listing 2.1: Sleeping barber problem- Part 1

```
prot = protocol {customer: Text }
NUM_CHAIRS = constant 3;
barber = process start: +prot; finish: -prot{
   customer: Text;
   loop
   {
      customer := start.customer;
      finish.customer := customer // "had hair cut";
   }}

waitingRoom = process
   arrives: +prot;
   leaves: -prot;
   start: -prot;
   finish: +prot ; {
         chairs: Word indexes Text;
         customer: Text ;

         for (c:=0 ;c<= NUM_CHAIRS ; c+=1)
         {
                  chairs[c] := "";
         }
         loop select policy random;
         {
                  ||   start.customer := arrives.customer
                  ||   customer := arrives.customer;
                 any name in range chairs such that name =""
                    name := customer;
                          else
                                   leaves.customer := customer // ' went away';
                  ||
                                   leaves.customer := finish.customer;
         any name in range chairs such that name <> ""
         {
       start.customer := name;
       name :=""
       }
         } }
```

Listing 2.2: Sleeping barber problem- Part2

```
barberShop = process arrives: +prot; leaves: −prot {
        start, finish: prot;
        barber( start, finish );
        waitingRoom(arrives, leaves, start, finish)
}
customerGenerator = process start: −prot {
   for (custnum := 0; custnum≤ 20; custnum+=1)
   {
       start.customer := 'C' // custnum
   } }

reporter = process finish: +prot {
        loop
        {
                scrln( finish.customer // '\n')
        } }

village = cell {
        arrives, leaves: prot;
        customerGenerator(arrives); reporter( leaves );
        barberShop(arrives, leaves) }
village ()
```

2.5 Abstract Interpretation

Before we explain the Abstract Interpretation (AI) theory and its use in static analysis of Erasmus programs, we give a brief introduction to mathematical foundations(i.e: Lattice theory and Fixed point) that are essential to understand the technique.

2.5.1 Mathematical Foundations

In this section, we review some of the mathematical foundations of Abstract Interpretation technique [13, 5, 35]. Before we explain the theory of lattices, we define the following terms.

Definition 2.1. Partial order, Poset

A binary relation $\sqsubseteq$ on a set P is a *partial order* if and only if it satisfies for all x, y, $z \in P$ the following conditions:

1. $x \sqsubseteq x$ (reflexivity)

2. $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$ (anti-symmetry), and

3. $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$ (transitivity). [35]

The ordered pair $\langle P, \sqsubseteq \rangle$ is called a *poset* (partially ordered set) when $\sqsubseteq$ is a partial order on P.

Definition 2.2. smallest (minimal) / greatest element (maximal)
Let $\langle P, \sqsubseteq \rangle$ be a poset. An element $y \in P$ is a smallest element of P if there is no element $x \in P$ that satisfies $x \sqsubseteq y$ and if $x \neq y$. Similarly, an element $y \in P$ is a greatest element of P if there is no element $x \in P$ that satisfies $y \sqsubseteq x$ and $x \neq y$.
We should note that a smallest or greatest does not necessarily exist and if it does exist, it is not necessarily unique.

Definition 2.3. upper bound/lower bound
Let $S \subseteq P$ be a subset of a poset $\langle P, \sqsubseteq \rangle$. *Upper bound* of S is an element $x \in P$ which is greater than or equal to every element of S. Similarly, *lower bound* of S is an element $x \in P$ which is smaller than or equal to every element of S.
For example if we define set S to be a set of some natural numbers: $\{4, 20, 44, 88, 99\}$, then 1 and 4 are the lower bounds of S. But 10 is not the lower bound. Also we can say 99 and 100 are upper bounds of S.

Definition 2.4. least upper bound (lub), greatest lower bound (glb)
Let $S \subseteq P$ be a subset of a poset $\langle P, \sqsubseteq \rangle$ and y be all upper bounds of S. An element $x \in P$ is called least upper bound of S, if

- x is an upper bound of S, and
- $\forall y \in S$ we have $x \sqsubseteq y$

Similarly, an element $x \in P$ is called greatest lower bound of S, if

- x is an lower bound of S, and
- $\forall y \in S$ we have $y \sqsubseteq x$

Definition 2.5. least upper bound operation $\sqcup$

This operation that is also called "join" is defined as the unique *least upper bound* with respect to a partially ordered set. We can say the "join" of two elements, x and y:

$$x \sqcup y \ for \ lub(x, y)$$

Example: let set S $(12, 20)$ be a set of natural numbers. $\sqcup S = 20$

Definition 2.6. greatest lower bound operation $\sqcap$

A greatest lower bound operation also called as "meet" is defined as unique *greatest lower bound* of a partial order set. We can say the "meet" of two elements, x and y:

$$x \sqcap y \ for \ glb(x, y)$$

For example if we perform the *meet* operation on a set of numbers $(12, 20)$ 12 is the greatest lower bound value. we note that 11 can be a lower bound but not the "greatest lower bound".

Based on the notations above, a complete *Lattice* is a partially ordered set which all subsets have unique *minimal, maximal elements* and greatest-lower-bound and least upper bound operations. The formal definition follows:

Definition 2.7. Complete *Lattice*

A *complete lattice*,

$$L = \langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$$

is including a set D and a partial ordering, $\sqsubseteq$, on D a smallest element $\bot$, a greatest element $\top$, a least upper bound operation, $\sqcup$ and a greatest lower bound operation $\sqcap$.

A set of subsets of a finite *lattice* form a complete *lattice*.

Definition 2.8. Semi lattice

A partially ordered set that has a lug (a least upper bound) or glb (a greater lower bound) for any nonempty finite subset.

Definition 2.9. bounded lattice

A lattice that has a min/max element.

Definition 2.10. Monotone

Let $\langle P, \sqsubseteq_p \rangle$ and $\langle Q, \sqsubseteq_q \rangle$ be two ordered sets. Let $\alpha : P \to Q$ be a mapping.
This mapping is monotone, if $x \sqsubseteq_p y$ implies $\alpha(x) \sqsubseteq_q \alpha(y)$

example: If we think of α as an electrical circuit mapping input to output, α is monotonic if increasing the input voltage causes the output voltage to increase or stay the same.

Definition 2.11. Galois Connection

Galois connection is a specific relation between two partially ordered sets in order theory. Suppose $\langle A, \sqsubseteq \rangle$ and $\langle B, \leq \rangle$ are two partially ordered sets and Suppose $F : A \to B$ and $G : B \to A$. Then $(F; G)$ is a Galois connection of A and B iff for all $x \in B$ and $y \in A$,

$$F(x) \sqsubseteq y \equiv x \leq G(y) :$$

F is called the lower adjoint. G is the upper adjoint.

2.5.1.1 Fixed Point

In our analysis we will use Fixed Point and Lattice theory to define some properties of programs. In this section we start with basic definition of Fixed point. Then we explain *Knaster-Tarski* theorem and *Kleene fixed-point* theorem. More detailed information about these theories can be found in [28].

Definition 2.12. Fixed Point

A fixed point of a function $f(x_0) = x_0$ is a point x_0 such that

$$f(x_0) = x_0$$

In another word, a fixed point is a point that the function does not change any more.

Theorem 2.1. *Knaster Tarski*

Let $\langle D, \sqsubseteq \rangle$ be a a complete lattice and let let $f : D \to D$ be a monotonic function on $\langle D, \sqsubseteq \rangle$. Then:

- *The set of fixed points of f in D is also a complete lattice under $\sqsubseteq$.*
- *Since a complete lattice cannot be empty, f has at least one fixed point.*

Definition 2.13. Ascending Kleene chain

Let $\langle D, \sqsubseteq \rangle$ be a a complete lattice and let $f : D \to D$ be a monotonic function on $\langle D, \sqsubseteq \rangle$. We can obtain a chain by iterating f on the least element $\perp$ as follow:

$$\perp \leq f(\perp) \leq f(f(\perp)..... \leq f^n(\perp)$$

Theorem 2.2. *Kleene fixed-point theorem*

Let D be a complete partial order, and let $f : D \to D$ be a monotone function. Then f has a least fixed point, which is the supermum of the ascending Kleene chain of f.

If f has a least fixed-point, lfp(f), that can be computed as $\lim_{n\to\infty} f^n(\perp)$.

$$lfp(f) : \lim_{n\to\infty} f^n(\perp)$$

2.5.2 Static Analysis of Program by Fixed Point Theory

Using theoretical bases that we introduced above, we explain how Fixed Point theory can be effective in static analysis of a given program. We define a behaviour of a program as a set of reachable states that might happen during the execution of the program. Assuming that Σ is a set of all reachable states of a program and $s \to s'$ stands for a state transition: we introduce the followings:

$$F \quad : \quad \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$$

in which $\mathcal{P}(\Sigma)$ is the powerset of Σ, i.e., the set of all subsets of Σ.

A set of descendants of the initial state s_0:

$$S \quad = \quad \{\, s \mid s_0 \longrightarrow ... \longrightarrow s \,\}$$

We know that $\mathcal{P}(\Sigma)$ is a complete lattice with bottom element $\emptyset$ and top element Σ. Given an initial state s_0, we define F for an arbitrary set of states S by

$$F(S) \quad = \quad \{s_0\} \cup \left\{ \; s' \mid s \in S \wedge s \longrightarrow s' \; \right\}.$$

Following convention, we define $F^0(S) \equiv S$ and $F^n(S) \equiv F(F^{n-1}(S))$. $F^0(s_0)$ is the initial state and for adequately large n, $F^n(s_0)$ will be the set of all states reachable from s_0.

To guarantee that this process is practical, we need to prove that we reach an n such that $F^{n+1}(S_0) = F^n(S_0)$. To prove this we need to define a lattice on set of states of the program. That is:

$$S_0 = F^0(S_0) \sqsupseteq F^1(S_0) \sqsupseteq F^2(S_0) \sqsupseteq \cdots$$

with this condition, we can take advantage of Knaster-Tarski theorem that F has fixed-points and that the sequence above converges to the least fixed-point, $\mathrm{lfp}(F)$.

2.5.3 Galois Connection in Abstract Interpretation

A *model* of a system S is a simpler system M that retains some of the properties of S. For example, we might construct a model M with the property that S deadlocks if and only if M deadlocks. This might be very difficult, so we might accept a simpler condition: if M cannot deadlock, then S cannot deadlock. This leaves open the possibility of a *false positive*: M might predict the possibility of deadlock, even though S cannot deadlock in practice. In this sense, models *approximate* the behaviour of systems. Suppose, for example, that Σ is the set of all states that the program can be in. Then Σ is ordered by the subset relation, $\subseteq$. Let $S \subseteq \Sigma$. Then if $|S| = 1$, we know exactly which state the program is in, which is the best we can expect. But if $S = \Sigma$, we know nothing at all about the program. Thus $S_1 \subseteq S_2$ says that S_1 has greater information than S_2. We suppose that we can also provide an ordering for the models. We will denote this order by $\leq$. Now we know that S and M are partially ordered sets. Elements of S are concrete values and elements of M are abstract values. Let $a : S \to M$ be a monotonic function called "abstract function", and let

$c : M \to S$ be a monotonic function called "concretion function".

$$\begin{array}{lrcl} \text{Abstraction function:} & a & : & S \to M \\ \text{Concretion function:} & c & : & M \to S \end{array}$$

Figure 2.2 provides this view. We move from system to model using a and from model to system using c. We should note that even though the system S and the model M are both ordered, orderings are not the same ($\leq$ and $\subseteq$, respectively). We would like to have $c(a(s)) = s$, but this is not possible in general because information is lost during abstraction. However, we can choose a pair (a, c) that forms a *Galois Connection*.

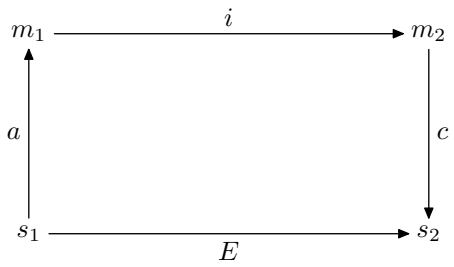


Figure 2.2: A system S and its model M

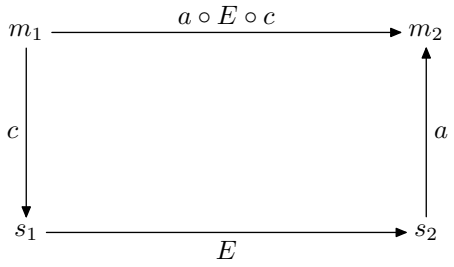


Figure 2.3: Fixed-point approximation

Static Analysis using Abstract Interpretation is a way of executing the abstraction of a program instead of the concrete program. To make sure the approximation is safe, we should make sure that the significant properties of concrete model are included in abstract model. Abstraction function a maps a set of concrete values into abstract values. Concretion function c maps elements from an abstract domain to the concrete one. Now that we have the abstract model of our program, instead of computing an approximation to the least fixed-point of E, we can compute $E' = \text{lfp}(a \circ E \circ c)$, which should be easier because the

calculations are performed in the model space. The Analyzer computes the abstract fixed point by an iterative computation.

2.5.3.1 Example

The example below is originally obtained from [37]. In this example we illustrate the use of Galois connection and Fixed Point theory in partial order of elements. The example is shown in Figure 2.4. We have the followings:

- A system(concrete) domain D which is defined as a partial order $\langle 2^Z, \subseteq \rangle$.
- A model (abstract) domain of D: $\hat{D}$ which is defined as partial order $\langle Sign, \sqsubseteq \rangle$

We define two functions that map the system(concrete) to the model(abstract) and vice versa. These functions are defined in Figure 2.5. The functions are:

$$\begin{aligned} \text{Abstraction function:} \quad & \alpha \quad : \quad D \rightarrow \hat{D} \\ \text{Concretion function:} \quad & \gamma \quad : \quad \hat{D} \rightarrow D \end{aligned}$$

To prove this abstraction is safe– meaning that by reasoning about the abstract program we can learn about the behaviour of concrete program– programs α and γ have to satisfy that they form a relation called *Galois connection*. That is:

$$\alpha(x) \subseteq \hat{y} \Leftrightarrow x \leq \gamma(\hat{y}) :$$

Therefore, we can perform the analysis in the the abstract domain $\langle Sign, \sqsubseteq \rangle$, instead of the concrete domain. As we explained earlier in this section, we consider the behaviour of the program as a set of its reachable states. These states can be computed by fixed point of function F that is an arbitrary set of states of program in domain D.

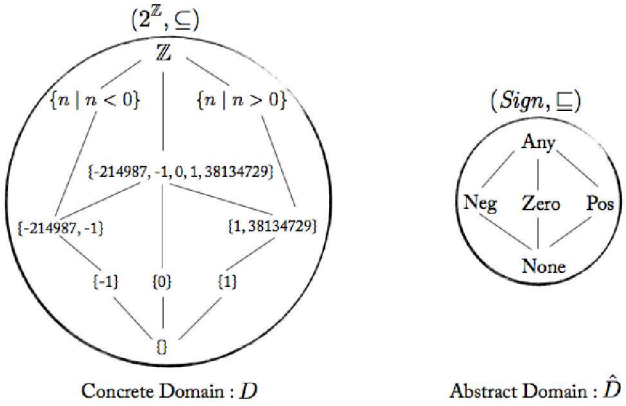


Figure 2.4: Concerete Domain $\langle D, \subseteq \rangle$ and Abstract Domain$\langle \hat{D}, \subseteq \rangle$ [37].

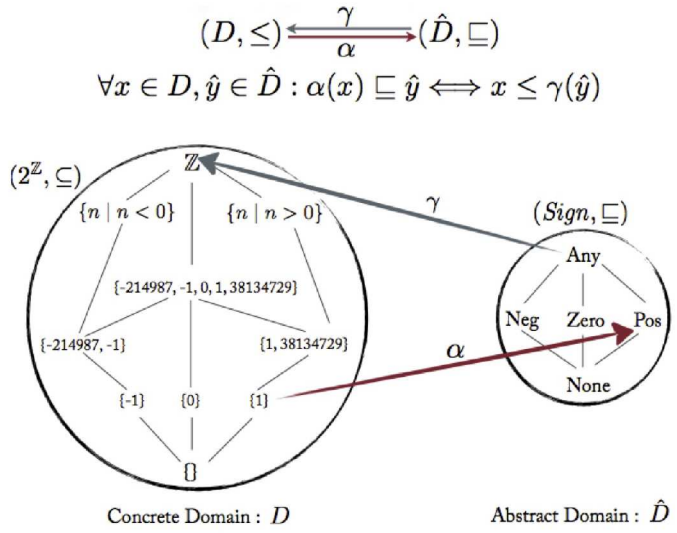


Figure 2.5: Abstraction function (α) maps elements from $\langle D, \subseteq \rangle$ to $\langle \hat{D}, \subseteq \rangle$. Concretization function (γ) maps elements from $\langle \hat{D}, \subseteq \rangle$ to $\langle D, \subseteq \rangle$ [37].

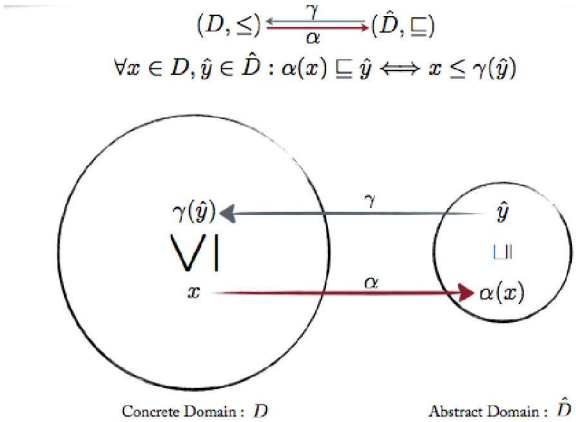


Figure 2.6: α and γ form a Galois connection [37].

2.6 A Lattice for Erasmus

The first step toward abstracting Erasmus programs is to create an abstract domain for the analysis. What is of interest for Erasmus programs is the order in which things happen and, in particular, circumstances that could prevent them happening. Consequently, we are looking for a mathematical structure S: a set of events, with properties of a complete Lattice. Since communication in Erasmus is synchronous, we can consider a *communication* between processes to be a single event. Formally:

> An *event* is the simultaneous execution of a send command in one process and a matching receive command in a distinct process.

In this definition, "matching" means that the two messages must be the same: If P sends x, then Q must receive x. Also, the definition implies that a process cannot send messages to itself, because to do so it would have to perform a send command and a receive command simultaneously. Additionally, we need to define a relation between events that happen in a given Erasmus program. To find this relation, we looked into some of the existing algorithms to build a partial order of synchronous events. Next section explains these algorithms.

2.7 Algorithms for partial ordering of events in a distributed system

Determining the order, in which events happen in concurrent programs, has been an fundamental difficulty in concurrent programming. A trivial solution to this problem is to assign a number representing the current time to a permanent record of the execution of each event. However, practically it is not feasible for concurrent processes to access a precise clock of the system [22]. Another possible solution can be introducing a separate clock in every process, synchronized as much as necessary with timestamps of the system. Lamport [39] extended these ideas and Duggan [21] has provided a simple exposition of them.

2.7.1 Lamport Logical Clock

One of the very first algorithms to determine the order of events in distributed system is Lamport algorithm [39]. This algorithm captures the order of event numerically based on

Happened-Before-Relation. The algorithm has been used to analyze behaviours of concurrent systems. Before we explain the algorithm, we review the followings definitions:

Definition 2.14. Causality

Causality is the relation between two events when first event is the cause of the second event. In another word, the second event is the consequence of the first event.

HBR is the relation between two events and it is denoted by $\rightarrow$. This relation is originally formulated by Leslie Lamport [39]. HBR captures the causal relationships between events:

- If a and b are the events in the same process, and a is executed before b then $a \rightarrow b$
- If a is the sending of a message by one process and b is the receive then $a \rightarrow b$

HBR relation has the same property as strict partial order relation. Strict partial order $<$ is a binary relation that is irreflexive, transitive and asymmetric. The formal definition of HBR is follows:

Definition 2.15. Happened Before Relation(HBR) $\rightarrow$

- $\forall a \ a \nrightarrow a$ (irreflexivity) ;
- If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$ (transitivity) ;
- $\forall a, b$ if $a \rightarrow b$ then $b \nrightarrow$ a (asymmetric)

Figure 2.7 show the HBR relation between events of processes P, Q and R.

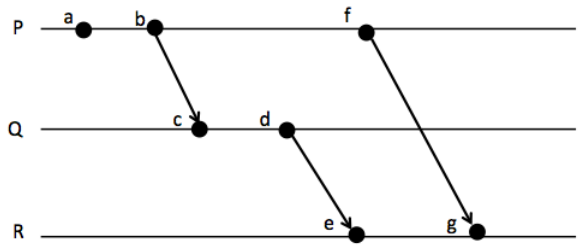


Figure 2.7: HBRs between events in processes P, Q and R are: $a \rightarrow b$, $b \rightarrow f$, $c \rightarrow d$, $e \rightarrow g$, $b \rightarrow c$, $d \rightarrow e$, $f \rightarrow g$, $a \rightarrow f$.

Definition 2.16. Concurrent Relation $\|$

If two events are not *causally* related, then they are *concurrent*.

For all pair of events a, $b \in E$ and $a \neq b$

if $a \nrightarrow b$ and $b \nrightarrow a$ then $a \parallel b$.

In order to build a partial order of events in a distributed system, we need to measure the times that events happen. However, since we are only interested in ordering events, we do not need to use time units (hour, minute, second). Instead, we can use a positive integer to just show the order of the events. This is called " logical time". The *smaller* integers denotes *earlier* times and *larger* integers denotes *later* times. " Logical Clock" is the clock that keeps such logical time. Lamport introduced a system of logical clocks to make the HBR relation between events possible. This logical clock is an incrementing counter maintained in each process. The formal definition of Lamport Logical Clock follows.

Definition 2.17. Lamport Logical Clock

A function C_i is a logical clock that assigns a number $C_i(e)$ to any event e in process i.

The Clock Condition:if $a \rightarrow b$ then $C(a) \leq C(b)$.

Definition 2.18. Lamport Clock Algorithm

Using above notations, the Lamport algorithm is as follows:

- The program keeps a "trace of clock" of every event that occurs as it executes.
- Each process maintains an integer value (logical clock): C_i.
- Logical clock(C_i) is initially zero for all the processes.
- C_i of each event is attached as a timestamp to the record of execution of the event.
- Rule 1: Whenever a new event e occurs in process i, increment the process clock, C_i, thus: $C_i := C_i + 1$.
- Rule 2:
  - If event e is sending a message, attach timestamp $ts = C_i$ with the message.

– If the event e is receive, compare the value of the sender clock to the receiver clock; advance the receiver clock to: $C_i = \text{Max } \{C_i, ts\} + 1$

Limitations of Lamport Logical Clock

One of the problems of the Lamport Logical Clock is that $C(a) \leq C(b)$ does not necessarily imply $a \nrightarrow b$. Another drawback for Lamport clock is that HBR algorithm is not applicable to synchronous programs. Fidge [22] was the first to claim this issue. He proposed a solution for this problem. His solution is to exchange timestamps when synchronous communication happens. We address this solution to both issues in the next section by introducing Vector Clock.

2.7.2 Vector Clock (VC)

VC is the data structure that contains timestamps for all (known) processes. The data structure is defined as below:

Vector Clock

Let n be the number of processes in a distributed system, then, the timestamp C_i of an event a of process i is a vector of n logical clocks.

The vector Clock is represented as an array with an integer clock value for all the processes.

$$[C_1, C_2, C_3, ..., C_n]$$

Definition 2.19. Vector Clock algorithm

Vector Clock algorithm is to generate a partial order of events in a distributed system [22, 43]. The algorithm applies Lamport's logic clock [39] to synchronous programs. Vector clock theory first was suggested by Fidge and Mattern [22, 43]. The main idea of the algorithm is that every time communication happens in synchronous programs, the process should exchange their timestamps. The algorithm maintains a "logical clock" for every atomic action happens in the program. Every time a process has an atomic action it should increment its logical clock value in the vector. Every time a communication happens the clock updates its values. The update rule is as following:

- Process that sends the message (Sender):
  - Increments its own logical clock in the vector by one.
  - Sends its message with the entire vector to the receiver.
- Process that receives the message (Receive):
  - Receives the message (if the communication is valid),
  - Increments its own logical clock in the vector by one.
  - Compares each value of the elements in Sender Vector to its own value of logical clock and updates each element in its vector by taking the maximum value of two vectors.

Compare to Lamport Clock, VC preserves more information about events. One of the advantages of the VC is that we can compare two vector timestamps of events a and b in different possible ways. These comparisons are presented in Table 2.4. Another advantages of VC is that comparing vector timestamps we can tell if two events are causally related: if $VC(a) < VC(b)$ then $a \rightarrow b$. In another word, event a causes event b if and only if $VC(a) < VC(b)$. The proof for this is presented in [22]. The properties of VC is defined as follows:

Order	Comparison	Condition
Equal	$VC_a = VC_b$	iff $\forall i$, $VC_a[i] = VC_b[i]$
Not Equal	$VC_a \neq VC_b$	iff $\exists i$, $VC_a[i] \neq VC_b[i]$
Less Than or Equal To	$VC_a \leq VC_b$	iff $\forall i$, $VC_a[i] \leq VC_b[i]$
Not Less Than or Equal To	$VC_a \nleq VC_b$	iff $\exists i$, $VC_a[i] \nleq VC_b[i]$
Less Than	$VC_a < VC_b$	iff ($VC_a \leq VC_b$ and $VC_a \neq VC_b$)
Concurrent	$VC_a \parallel VC_b$	iff $VC_a \nleq VC_b$ and $VC_b \nleq VC_a$)

Table 2.4: Comparing two vector timestamps [67]. a and b are referred to the process events. VC_a is referred to the Vector Clock value for event a. The Vector Clock value of event a in process i is denoted by $VC_a[i]$. Same applied to event b.

VC Property:

Considering events a and b, we have the followings:

- $VC(a)$ denote the time at which event a occurs; the value of Vector Clock for event a

- $VC(b)$ denote the Vector Clock of event b.
- if a happens before b; $a \rightarrow b$ then $VC(a) < VC(b)$.

2.7.2.1 Scenario example

Vector Clocks are useful for Erasmus programs that are able to make choices. Making choices increases the possibility of inconsistent information because of multiple paths. We have adapted the following scenario from Fink [23] to illustrate the usage of Vector Clock. The Listing 2.3 is based on Erasmus syntax and notations. Erasmus syntax is provided in the Appendix of the thesis.

Philosophers Meeting Scheduling Problem.

- Four philosophers, named P, Q, R and S, are planning to meet next week for a philosophy seminar.
- The planning starts with P suggesting they meet on Wednesday.
- Later, Q discuss alternatives with R and they decide on Thursday instead.
- Q also exchanges email with S, and they decide on Tuesday.
- P pings everyone again to find out whether they still agree with Wednesday's suggestion.
- R claims to have settled on Thursday with Q.
- S claims to have settled on Wednesday with Q.

In above scenario no one is able to determine the order in which these communications happened, and none of P, Q, R and S, know whether Tuesday or Thursday is the correct choice.

Listing 2.3: Philosophers Meeting Scheduling Problem

```
prot = protocol { day: Word }
P = process p1, p2, p3: −prot; p4, p5, p6: +prot {
        day: Word;
        /* sends the chosen day to other philosophers. */
        p1.day := 'Wednesday'
        p2.day := 'Wednesday'
        p3.day := 'Wednesday'
        /* receives the chosen day from other philosophers. */
        day := p4.day
        day := p5.day
        day := p6.day
}
Q = process q3, q4: −prot; q1, q2: +prot {
        day: Word;
        day := q1.day  −− receives 'Wednesday' from P
        select {
                || q3.day := 'Thursday' −−sends 'Thursday' to R
                || day := q2.day   −− receives 'Tuesday' from S
        }
        q4.day := day  −− sends back the chosen day to P [Thursday]
}
R = process r1: −prot; r2, r3: +prot {
        day: Word;
        day := r2.day     −− receives 'Wednesday' from P
        day:= r3.day     −−  receives  'Thursday' from Q
        r1.day := day −− sends back the chosen day to P [Thursday]
}
S = process s2, s3: −prot; s1: +prot {
        day: Word;
        day := s1.day −− receives 'Wednesday' from P
        s2.day := 'Tuesday'  −− sends 'Tuesday' to Q
        s3.day := day  −− sends back the chosen day to P [Wednesday]
}
```

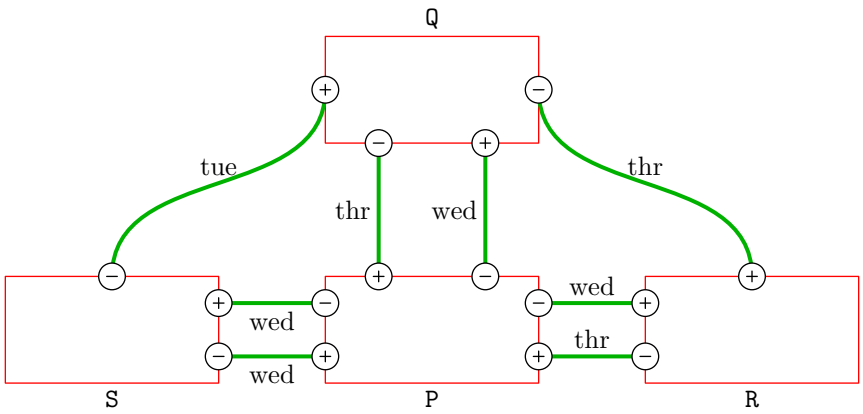


Figure 2.8: Philosophers Meeting Scheduling Problem.

Vector clock is actually the solution for the above problem. If the clock value and the sender of the message are piggy-backed on the actual message (here the message is the meeting day), then philosophers would be able to know the latest information that is exchanged.

2.7.2.2 Limitation of Vector Clock

Erasmus is designed to improve writing large and complex concurrent programs. Therefore, while choosing the right technique to perform the analysis of the program, we should take into account the size of the programs that can be fairly large. A main drawback of the application of VC algorithm is the length of the vectors will grow proportionally with the number of processes. Clearly, if we have n processes, the size of the vector clock must be at least n [8]. In addition, to implement VC algorithm, the application requires unlimited space since vector clocks grow unboundedly. The key challenge is to reduce the length and size of Vector Clock. A number of researchers have recognized this problem and have looked for ways of solving it. For example Arora et al [4] provided a bounded-space implementation of the Vector Clock component. Their idea is to use a resettable vector clocks (RVC) instead of VC and provide a realistic implementation of RVC.

2.8 Chapter Summary

This chapter presented an overview of existing approaches for verification and static analysis of concurrent programs. The syntax of Erasmus language were explained in the third section of the chapter. Abstract Interpretation framework and the related mathematical theorems (e.g. lattice, fixed-point, etc) were explained this chapter. We also reviewed two well known algorithms: Lamport Logical Clock and Vector Clock for dynamic partial order of events in distributed systems. The imitations of these algorithms are also discussed at the end of the chapter.

Chapter 3

DESIGN A NOVEL COMMUNICATION ABSTRACT DOMAIN

In this chapter, we present an abstract interpretation framework for a synchronous process-oriented programming language. The framework is built based on the concepts of Erasmus programming language, but it might also be applicable to other process-oriented or message passing programs.

Since our focus in this research is communication analysis, the existing domains in abstract interpretation theory such as intervals, convex hull and polyhedra, etc [15] are not useful for us. Therefore, the framework uses a novel communication abstract domain whose concretization and abstraction function are introduced in this chapter.

The chapter is organized as follows: Section 3.1 explains **case**, **loop** and **select** controls in Erasmus programs and the communication violations in these type of Erasmus programs. Section 3.2 defines a concrete semantic domain. A fixed-point algorithm to computer the reachable states of the program is presented in Section 3.2.2. Section 3.3 defines an abstract domain which approximates the concrete semantic domain. Transfer operators are presented in Section 3.4. Finally, Section 3.6 summarizes the materials discussed in this chapter.

3.1 Preliminaries

Nondeterminism is a fundamental concept in concurrency. There are various meanings of nondeterminism in computer science. *External nondeterminism* happens when given a set of the fixed input, a concurrent program behaves differently in each execution and it produces different results due to the external factors that effect the program behaviour. However, nondeterminism has a very specific definition in CSP, the basis of Erasmus . If a process makes a choice based on the environment, the choice is called *"deterministic"*, because an inspector who is aware of the environment can tell what the process will do. But if a process makes a choice not based on the environment, the choice is called *"nondeterministic"* because the inspector cannot predict the outcome. Nondeterminism happens in Erasmus when the code contains conditional blocks. In this section, we discuss Erasmus programs that contain control structures (**loop** , **select** and **case**) and introduce the notations that is used in this chapter.

3.1.1 Conditional Erasmus program

Consider an Erasmus program shown in Figure 3.1. The Erasmus code is provided in Listing 3.1. The program consists of processes *Electricity*, *Alarm* and *ChemicalPlant*. Assume that the process *Alarm* is connected to two channels. One is the channel that *ChemicalPlant* is also connected to it and the other one is the same channel that *Electricity* is connected. That is, processes can communicate. We define variable *level* in *Alarm* process to show how critical is an emergency situation. For instance, if an emergency situation happens with a $level \geq 0.5$, the *alarm* should send a message to the *ChemicalPlant* process with a shutdown message. If the level of criticalness of an emergency situation is smaller than 0.5, then alarm sends a message to the *Electricity* process to shutdown the electricity. We assume that process Alarm receives the variable *level* from an external process through channel c_2. This program is *nondeterministic* because the response depends on the [value of the] level which is not known to the system environment.

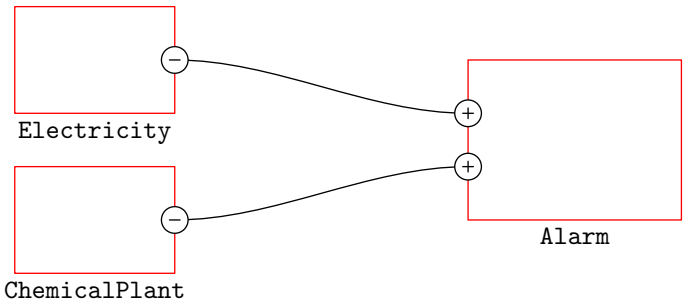


Figure 3.1: Chemical Plant Emergency Alarm System.

Listing 3.1: Chemical Plant Emergency Alarm System

```
Prot = protocol { level: Real,, message: Word }

Electricity = process q: −Prot {
        action: Word;
        action := q.message;
}

ChemicalPlant = process r: −prot {
        action: Word;
        action = r.message;
}

Alarm = process c: +prot, c1: +prot c2: +prot {
{
        level = c2.level;
        if level ≥ 0.5 then
                c.message := 'Plant shutdown'
        else if level < 0.5
                c1.message= 'Electricity shutdown'
        else exit;
}
```

3.1.2 Select and Loop Select Statements

Select statement makes a deterministic choice of communication based on the environment. The effect of the statement will be determined during run-time and not compile time. In Erasmus the policies for selection are fair, ordered, and random which are defined by a enumeration type: Policy = enum {Fair, Ordered, Random}. Writing **loop** before **select** indicates that the **select** statement should be executed repeatedly until an exit statement

has been executed. An example of Erasmus program with **select** statement is shown in Listing 3.2.

Listing 3.2: Erasmus program with loop select and conditions

```
prot = protocol { w: Word }

P = process p1: −prot; p4: +prot
{
word: Word;
loop{
        p1.w := 42
        word := p4.w
        }
}

Q = process q1,q3: +prot ; q2,q4: −prot
{
word: Word;
canAnswer:Boolean:=false;
loop select {
        || word := q1.w ;
                case {
                |canAnswer = true | q4.w := 45
                ||   q2.w := 45 }
        || word := q3.w ; q4.w := 44
        }
}

R = process r2: −prot; r3: +prot
{
 word: Word;
loop {
        r2.w := 42;
        word:= r3.w
        }
}
controller = cell{
        e1,e2,e3,e4: prot;
        R (e2,e3);
        Q (e1,e2,e3,e4);
        P (e1,e4);
}
controller ();
```

```
process A:
0: n := 1
1: loop select {
2: || snd(b, n)
3: || rec(c, n)
}

process B:
0: n := 1
1: loop select {
2: || snd(b, n)
3: || rec(c, n)
}
```

Figure 3.2: A Sender, A, and a receiver, B. Program does not deadlock.

3.1.3 Deadlock

In this section, we just explain how deadlock can happen when we have **loop select** and **loop** statements in a given Erasmus code. Figure 3.2 shows two processes: A and B. For simplicity, the example is not written in exact syntax of Erasmus and its based on CSP notation. In particular, we use $snd(c, n)$ to send a number n on channel c. Similarly, $rcv(c, n)$ receives n on channel c. The processes are assumed to run concurrently. A sends one number to B and receives one number from B. Process B also sends one number to A and receives one number from B. The key point of this example is that we do not want to fix the order of processing because we do not know which operation will be executed first, send or receive. We note that this program does not deadlock because in Erasmus synchronous program, a send operation is not completed unless there is a receive operation matching the send (or vice versa). Consequently, when the program terminates, we expect *either* A to send its message and then receive a message from B (or vice versa).

Now, we do a slight modification to the previous example. The modification is that we omit the **select**. The new program is shown in Figure 3.3. The process A sends the message and then waits for B to receive the message. Meanwhile, process B sends a message to A. We note that since both processes are waiting for a receive statement, they can not progress anymore and program deadlocks.

```
process A:
0: n := 1
1: loop {
2: snd(b, n);
3: rec(c, n)
}

process B:
0: n := 1
1: loop {
2: snd(b, n);
3: rec(c, n)
}
```

Figure 3.3: A Sender, A, and a receiver, B. Program does deadlock.

3.2 Concerete Semantic Domain for Erasmus

Abstract Interpretation provides formal methods to approximate semantics. A semantics is a mathematical characterization of a possible behaviour of the program. The first step toward static analysis of Erasmus programs is to create a concrete and abstract domain for our analysis. The precision and performance of the resulting static analyzer are almost entirely determined by the choice of these domains. In theory of Abstract Interpretation various domains are defined. Cousot [15] categorized these domains in two groups. Numerical domains and Symbolic domains. Numerical domains are including intervals, affine equalities, convex hull and polyhedra whereas symbolic domains include abstraction of sequences, trees and graphs, binary decision diagrams, word and tree automata, pointer analysis.
Table 3.1 shows a list of possible domains for analysis of Erasmus . In this research, since our focus is communication analysis, therefore the existing domains are not useful for us and we need to define an abstract domain for communication between processes and cells in Erasmus programs.

3.2.1 A Communication Lattice

Abstract Interpretation is based on theory of *lattice*. The first step toward abstracting Erasmus programs is to find a partial order over program states or events. Having a lattice

Abstract Domain	Analysis
Type	Compute type of each value.
Concrete	Compute all of the possible states.
Array Bounds	Compute array range.
Communication	Compute if each sent/receive corresponds to receive/send.
Range Check	Compute range of values in program.
Real-Time	Compute the time taken for tasks.

Table 3.1: Possible Abstract Domains for Erasmus

makes it easier to interpret the program because we can build our mathematic model based on the lattice and take advantage of existing theorems about lattice. (e.g., fixed-point theory of lattice). Later, we can get the approximation of least fixed-point of the program and limit the number of computation needed by using *widening/narrowing* techniques, if necessary. Although we can use interval analysis for Erasmus , it would not be very helpful because we will ignore the non-communication statements. Our interest in Erasmus program is the order in which events happen and in particular, circumstances that could prevent them happening. For example, we would like to know if processes can continue executing or they might deadlock. Referring to the grammar (see attachments), "communications" appear as $p.v$. In a left context (lvalue), $p.x$ receives and stores in x; in a right context, $p.e$ sends the value of expression e. Statements that do not communicate are removed from the program. This will leave empty control structures, e.g., a loop that does not communicate will now be empty, and can be removed. What is left is a program with only control structures (case, loop, select) and communications. This is the "abstract program" that we analyze. Since calculations are discarded, AI techniques using intervals, polyhedra, etc., are of no use to us.

We need to create a mathematical structure in our program with same properties as *partial order*. The Followings explain our attempt to create *partial order* for communications happen in an Erasmus program. Let E be an Erasmus program, we will ignore the non-communication related statements and consider only the following components:

- E consists of *processes* P, Q, R.

- Each process executes statements in a sequence determined by its control structures.
- The control structures are sequence, conditional, and repeat.

Definition 3.1. An *event* is either a send statement or a receive statement.

According to Lamport's Happened Before Relation (HBR) [39] theory we say that *send* happens before *receive* in communication of processes. Therefore, the relation between the two events *send* and *receive* is: $e_1 \leq e_2$ means "event e_1 occurs before e_2. This binary relation $\leq$ satisfies *irreflexivity*, *asymmetry* and *transitivity*. However, this relation is not a partial order, because a partial order must be reflexive and antisymmetric. If an event e_1 must precede an event e_2 we say "e_1 happens before e_2". Formally, HB ("happens before") is a relation, HB $\subseteq E \times E$, defined formally as follows.

Definition 3.2. $(e_1, e_2) \in$ HB if either:

- e_1 and e_2 are in the same process and if either e1 textually precedes e2 or e1 and e2 are different occurrences of an event in a loop, in which case the HB relation is determined by loop iterations; or:
- e_1 and e_2 are in different processes, e_1 is a send statement, e_2 is the corresponding receive statement.

HB relation in loop iterations

Consider the following code, in which we write communication events as e_i:

$$e_1;\ \textbf{loop}\ \{\ e_2\ \};\ e_3$$

When this code is executed, e_2 will occur zero or more times. We write the successive occurrences of e_2 as e_2^1, e_2^2, ..., e_2^n, using superscripts to distinguish loop iterations. Using this convention, for all $n \geq 1$:

- If e_j occurs within a loop, $e_j^n \rightarrow e_j^{n+1}$;
- If e_i occurs before a loop, and e_j occurs within the loop, then $e_i \rightarrow e_j^n$;
- If e_j occurs within a loop, and e_k occurs after the loop, then $e_j^n \rightarrow e_k$.

Since communication in Erasmus is synchronous, we define a *constraint* to be a pair of *send* event and a matching *receive*.

Definition 3.3. Constraint

"A *constraint* is the simultaneous execution of a *send* command in one process and a matching *receive* command in a distinct process."

For simplicity, below examples is not written in exact syntax of Erasmus and its based on CSP notation **Example**: The constraint is denoted by c: $P \xrightarrow{x} Q$ in Figure 3.3 and in Figure 3.4, the program E_2 has two constraints $P \xrightarrow{x} Q$ and $Q \xrightarrow{y} R$.

Listing 3.3: Program E_1 with constraint $P \xrightarrow{x} Q$

$P =$ **process** $\{snd(x)\}$
$Q =$ **process** $\{rcv(x)\}$

Listing 3.4: Program E_2 with constraints $P \xrightarrow{x} Q$ and $Q \xrightarrow{y} R$

$P =$ **process** $\{snd(x)\}$
$Q =$ **process** $\{$**select** $\{rcv(x) \mid rcv(y)\}$ $\}$
$R =$ **process** $\{snd(y)\}$

Constraints have the following properties:

- Each *send* or *receive* operations in a constraint have a event order value.
- The event order value of the first event(send) is always smaller than the second event(receive).
- The constraint is inconsistent if event order value of the *send* is greater than the *receive*.
- An inconsistent constraint implies that the program will deadlock.

Definition 3.4. A *trace* $\langle c_1, c_2, \ldots, c_n \rangle$ is simply an ordered list of events constraints in which no constraint happens before a constraint that precedes it in the list.

Property:

A trace $T = \langle c_1, c_2, \ldots, c_n \rangle$ if $j > i$ then $c_j \not\rightarrow c_i$.

We can not say the property that $c_i \rightarrow c_{i+1}$ because c_i and c_{i+1} might be concurrent. A trace is an abstract representation of the execution of a program. A program may have one or more possible traces. Examples:

- Program E_1 (Figure 3.3) has one trace: $\left\langle (P \xrightarrow{x} Q) \right\rangle$.
- Program E_2 (Figure 3.4) has one trace: $\left\langle (P \xrightarrow{x} Q), (Q \xrightarrow{y} R) \right\rangle$.

We define a partial ordering for event constraints (traces) denoted by $\sqsubseteq$. The definition follows:

Definition 3.5. Trace $T_1 = \langle\, c_1, c_2, \ldots, c_m \,\rangle$ is a *prefix* of trace $T_2 = \langle\, c'_1, c'_2, \ldots, c'_n \,\rangle$ (that is, $T_1 \sqsubseteq T_2$) if and only if $m \leq n$ and $c_i = c'_i$ for $1 \leq i \leq m$.

For example, $\langle\, c_1, c_2 \,\rangle \sqsubseteq$ (is a prefix of) $\langle\, c_1, c_2, c_3 \,\rangle$ but is not a prefix of $\langle\, c_1, c_3, c_2 \,\rangle$. Trace 1 has less information than trace 2. We claim that $\sqsubseteq$ is a complete lattice with following properties:

- The traces are ordered by $\sqsubseteq$.
- $\langle\ \rangle$ provides no information whatsoever about the program.
- If $T \sqsubseteq T'$, then T' tells us more about the program than T.
- The greatest element $\top$ contains all of the events that can occur in a program.
- Meet or greatest lower bound operation ($\sqcap$) operator is the trace that has less events constraints. e.g: $\langle\, c_1, c_2 \,\rangle \sqcap \langle\, c_1, c_2, c_4, c_5 \,\rangle = \langle\, c_1, c_2 \,\rangle$ and When the traces have nothing in common, the meet is empty, e.g., $\langle\, c_1, c_2, c_3 \,\rangle \sqcap \langle\, c_4, c_5, c_6 \,\rangle = \langle\ \rangle$.

$\sqsubseteq$ is a partial order:

- Reflexive: A sequence T can be a prefix of itself. Thus $T \sqsubseteq T$.
- Antisymmetric: Suppose $T_1 \sqsubseteq T_2$ and $T_2 \sqsubseteq T_1$. Then every event in T_1 is also contained in q_2 and *vice versa*. Consequently, $T_1 = T_2$.
- Transitive: If T_1 is a prefix of T_2, and T_2 is also a prefix of T_3, then clearly T_1 is also prefix of T_3.

Prefix of traces are ordered by the $\sqsubseteq$ relation. For example we have:

- $\langle e1, e2 \rangle$ is subset $\sqsubseteq$ of $\langle e1, e2, e3 \rangle$. Meaning that $\langle e1, e2, e3 \rangle$ tells us more about the program than $\langle e1, e2 \rangle$.
- The bottom element of the lattice is an empty set: $\bot = \emptyset$.
- Top element $\top$ contains all of the events that can occur in a program. e.g. $:\top = \langle e1, e2, e3 \rangle$
- $A \sqcap B$ is the information provided by both A and B e.g. : $\langle e1, e2 \rangle \sqcap \langle e1, e2, e3 \rangle = \langle e1, e2 \rangle$

The behaviour of a program is a set of all the possible states that can occur during execution of the program. General steps for the analysis of the code are: fixed-point calculation of program semantics and detecting communication violations. The following section, explains the fixed-point computation of program semantics.

3.2.2 Fixed-Point calculation of program semantics

We define the behaviour of a system as a set of all the reachable states of the program. We calculate all the reachable states from a initial state of the program. We adapt the approach of Deutsch [18] to calculate the fixed-point. We introduce a function F for an arbitrary set of states S by

$$F(S) \quad = \quad \{s_0\} \cup \left\{ \; s' \mid s \in S \wedge s \longrightarrow s' \; \right\}.$$

we define $F^0(S) \equiv S$ and $F^n(S) \equiv F(F^{n-1}(S))$. Eventually, we will have $F^{n+1}(S) \equiv F^n(S)$ where $F^n(s_0)$ will be the set of all states reachable from s_0.

$$\begin{aligned} F^0(s_0) &= \text{the initial state} \\ F^1(s_0) &= \text{the states that can be reached in one step} \\ &\cdots \\ F^n(s_0) &= \text{the states that can be reached in } n \text{ steps} \end{aligned}$$

We note that:

- The state of the program is the event trace of the particular program point.
- The event trace is a sequence of events constraints.
- Each event constraint includes the event order value of a *send* and its matching *receive* operation. Example: in program E_1 we have $s_0 = \langle\, (e_1, e_2)\,\rangle = \langle\, ([1,0],[1,1])\,\rangle$
- The event order of a process is represented as a tuple of values. If we have n processes in the program we have a tuple of size n.
- The algorithm produces least fixed-point of all the states of the program.
- We detect the communication errors by inspecting S; least fixed-point of the program.

If each process adds only a finite number of values to its order component, the total number of distinct event orders is finite. This means that, in principle, we can find them all by static analysis using abstract interpretation. Algorithm 3.1 is to calculate the fixed-point of event order values of all processes of the program. We will define and further explain "event order values" in Chapter 6.

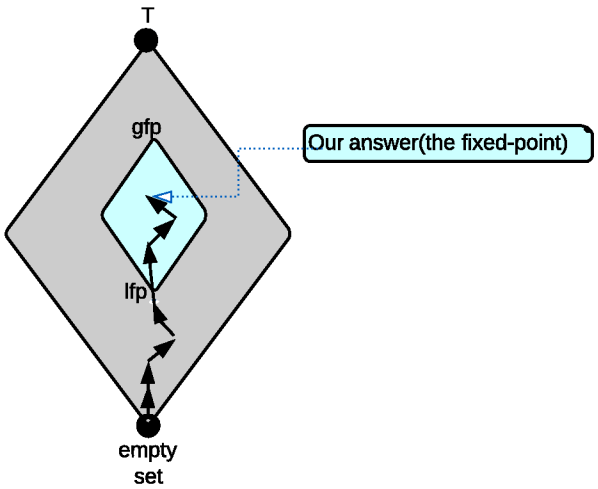


Figure 3.4: The lattice includes all of the states of the program. Fixed-point is the reachable states that can occur in the program. lfp is the least fixed-point of the lattice and gfp is the greatest fixed-point.

Algorithm 3.1 Calculate fixed-point of event order values(EOV) of all the processes

Require: Processes P_1 and P_2 ready to communicate.
Ensure: S: fixed-point of all distinct EOVs.

```
 1: S := 0; // set of EOVs, 0 is the initial EOV
 2: stable := true;
 3: repeat
 4:
 5:     stable := true;
 6:     v1 := new state of EOV of P1
 7:     if v1 is not in S: then
 8:         S := S U v1;
 9:         stable := false;
10:     end if
11:     update P1 state;
12:     v2 := new state of EOV of P2
13:     if  v2 is not in S: then
14:         S := S U v2;
15:         stable := false;
16:     end if
17:     update P2 state;
18: until stable
```

3.3 Abstract Semantic

Definition 3.6. An *Abstract Trace* is a set of constraints in which different occurrences of the same constraint are considered equal.

Abstract Trace is ordered by the subset relation, which is a complete partial order. There is a lattice of abstract trace, with following properties:

- The sets are naturally ordered by $\subseteq$.
- The least element $\perp$ ($\emptyset$) is the bottom element. it provides the least information about the program.
- The greatest element $\top$ as the set of all event constraints.
- Meet or greatest lower bound operation ($\cap$) of two abstract trace is the set that has less number of constraints.

A binary relation $\subseteq$ on a set of events is a *partial order* because it satisfies reflexivity, anti-symmetry and transitivity.

Example: abstract trace is ordered by the subset relation:

- $\{e1, e3\} \subseteq \{e1, e2, e3\}$.
- $\bot = \emptyset$
- $\top = \{e1, e2, e3\}$
- $\{e1, e3\} \sqcap \{e1, e2, e3\} = \{e1, e3\}$

3.4 Operators and Transfer Functions

Abstraction and Concretion functions Let $\mathcal{E}$ be the set of all abstract traces with typical member s and $\mathcal{Q}$ be the set of all traces with typical member q. We define two functions as follows:

- (A) *Abstraction:* $a : \mathcal{Q} \to \mathcal{E}$.

$$a(q) = \{\ e \mid e \in q\ \}.$$

- (C) *Concretization:* $c : \mathcal{E} \to \mathcal{Q}$.

$$c(s) = q \text{ such that } a(q) = s \text{ and } q \text{ is minimal.}$$

3.4.0.1 Galois Connection

We claim that a and c constitute a Galois connection. To show this, we must show that $c(a(q)) \sqsubseteq q$ (sequence ordering) and $a(c(s)) \subseteq s$ (subset ordering).This

To show $c(a(q)) \sqsubseteq q$:

- By (A), $a(q) = \{\ e \mid e \in q\ \}$. q may have superscripts, but $a(q)$ does not.
- Therefore $c(a(q)) = c\,(\{\ e \mid e \in q\ \})$.
- By (C), $c\,(\{\ e \mid e \in q\ \})$ is a minimal sequence q' such that $a(q') = a(q)$.
- Thus $c(a(q)) = q'$, where q and q' have the same abstraction but q' is minimal.

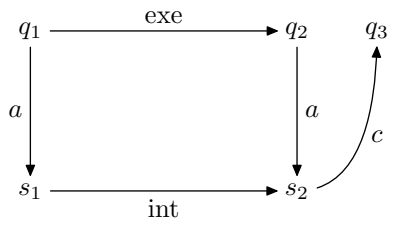
- Any difference between q and q' must be due to the fact that q may have superscripts but q' does not, and therefore $q' \sqsubseteq q$.
- Since $q' = c(a(q))$, we have $c(a(q)) \sqsubseteq q$.

To show $a(c(s)) \subseteq s$:

- By (C), $c(s) = q$ where q is such that $a(q) = s$ and q is minimal.
- Therefore, $a(c(s)) = a(q) = s$.
- Consequently, $a(c(s)) \subseteq s$.

We introduce two more functions:

- $\text{exe} : \mathcal{Q} \to \mathcal{Q}$ represents execution of the program, mapping a sequence to a sequence.
- $\text{int} : \mathcal{E} \to \mathcal{E}$ represents abstract interpretation, mapping an event set to an event set.



We consider execution of the program from an initial sequence q_1 to a final sequence q_2. Abstract interpretation takes us from an initial event set s_1 to a final event set s_2. We have

$$\begin{aligned} q_2 &= \text{exe}(q_1) && \text{(execution)} \\ s_2 &= \text{int}(s_1) && \text{(interpretation)} \\ s_1 &= a(q_1) && \text{(abstraction)} \\ s_2 &= a(q_2) && \text{(abstraction)} \end{aligned}$$

We can derive the correctness of the interpretation from these equations as

$$\begin{aligned} \mathrm{int}(a(q_1)) &= \mathrm{int}(s_1) \\ &= s_2 \\ &= a(q_2) \\ &= a(\mathrm{exe}(q_1)). \end{aligned}$$

We see that abstract interpretation may give us a result sequence q_3 with *fewer* events than the actual computed sequence q_2 but it cannot give a result with *more* events. In this sense, it is a "safe" approximation. In practice, we would start from an initial event, $\langle\, e_0 \,\rangle$, for which the abstraction is the set $a(\langle\, e_0 \,\rangle) = \{e_0\}$. Then we compute the fixed point of the abstract interpreter:

$$S = \lim_{n\to\infty} \mathrm{int}^n(\{e_0\})$$

which means in practice that we compute $S_n = \mathrm{int}^n(\{e_0\})$ until $S_{n+1} = S_n$, which means that we have arrived at the fixed point, S. (This process must terminate because the number of events is finite.) The concretization of this set, $c(S)$, is the sequence that we want.

3.5 Example

For the communication analysis of Erasmus programs, it's unnecessary to inspect every details of Erasmus code, since only a subset of statements influence the communication behaviour of the program. Therefore, we create an abstract version of each process that shows only the amount of detail which is relevant to communication behaviour. Then we perform the analysis on abstract code that has communication instructions.We use the Abstract Interpretation to remove the statements that have no effect in the communication of the process. We also remove the **loop** control from the processes. As shown in Listing 3.6, E' is an abstract version of processes in E. This abstraction of processes is shown in Figure 3.5. We note that numbers at the right (0), (1) (2) are program states.

Listing 3.5: Program E

```
P = process −p1, +p4
{       loop (0)
        {
                p1.snd; (1)
                p4.rcv
        } }
Q = process +q1, −q2, +q3, −q4
{
   loopselect (0)
   {
      || q1.rcv;
        if canAnswer then (1) q4.snd else (2) q2.snd
      || q3.rcv; (3) q4.snd
   }}
R = process +r2, −r3
{       loop (0)
   {
      r2.rcv; (1)
      r3.snd
   }}
```

Listing 3.6: Abstract Processes: E'

```
P = process −p1, +p4
{       p1.snd; (1)
         p4.rcv
}
Q = process +q1, −q2, +q3, −q4
{
  loopselect (0)
      || q1.rcv;
        if canAnswer then (1) q4.snd else (2) q2.snd
      || q3.rcv; (3) q4.snd
}
R = process +r2, −r3
{       r2.rcv; (1)
        r3.snd
}
```

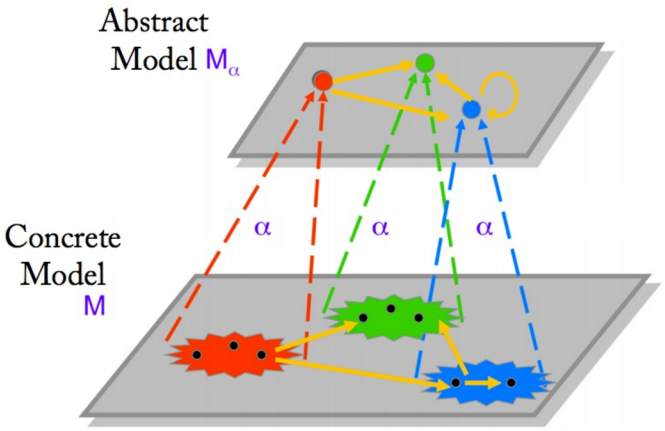


Figure 3.5: Abstraction of processes in Program E. The concrete model is Program E and the abstract model is program E'. The image is adapted from [37]

3.5.1 Galois Connection

In this section we prove that our abstraction from the System(E) to the model(E') is safe. For the theoretical background of Abstract Interpretation refer to Chapter 2. The definitions used in this section are presented in Table 3.2. The example used in the table is based on the programs E and E' presented in Listings 3.6 and 5.1. Table 3.3 shows the naming conventions for each statements in programs E and E'. We will use these names later in this section.

Using the programs E and E' as an example:

$$
\begin{aligned}
\text{Let} \quad q &= \langle\, e_1^1, e_3^1, e_4^1, e_2^1, e_1^2, e_3^2, e_4^2, e_2^2, \ldots, e_1^n, e_3^n, e_4^n, e_2^n \,\rangle \\
\text{then} \quad a(q) &= \{e_1, e_3, e_4, e_2\} \\
\text{and} \quad c(a(q)) &= c\,(\{e_1, e_3, e_4, e_2\}) \\
&= \langle\, e_1, e_3, e_4, e_2 \,\rangle \\
&\sqsubseteq \langle\, e_1^1, e_3^1, e_4^1, e_2^1, e_1^2, e_3^2, e_4^2, e_2^2, \ldots, e_1^n, e_3^n, e_4^n, e_2^n \,\rangle \\
&= q.
\end{aligned}
$$

Term	Explanation	Example
event	A send statement or a receive statement.	Process P in program E' has two events: $e_1 = p1.snd$ and $e_2 = p4.rcv$. Consequently, process P of program E might include the events: $e_1^1, e_2^1, e_1^2, e_2^2, e_1^3, e_2^3$. We note that e^x is the xth successive occurrences of an event in a loop.
event set	A set of events in which different occurrences of the same event are considered equal.	Process P of program E' has one event set: $\{e_1, e_2\}$. Process P of program E has one event set: $\{e_1, e_2\}$.
event sequence	An ordered list of events in which no event happens before an event that precedes it in the list.	Sequence of event in process P of program E is $q = \langle\, e_1, e_2 \,\rangle$

Table 3.2: Definitions

event	**statement**
e_1	p1.snd
e_2	p4.rec
e_3	q1.rec
e_4	q4.snd
e_5	q2.snd
e_6	q3.rec
e_7	q4.snd
e_8	r2.rcv
e_9	r3.snd

Table 3.3: Naming convention for events in program E

Going the other way:

$$
\begin{aligned}
\text{Let} \quad & s = \{e_1, e_3, e_4, e_2\} \\
\text{then} \quad & c(s) = q \text{ such that } a(q) = s \text{ and } q \text{ is minimal} \\
\text{i.e.,} \quad & c(s) = \langle\, e_1, e_3, e_4, e_2 \,\rangle \\
\text{and} \quad & a(c(s) = a\left(\langle\, e_1, e_3, e_4, e_2 \,\rangle\right) \\
& = \{e_1, e_3, e_4, e_2\} \\
& \subseteq \{e_1, e_3, e_4, e_2\} \\
& = s.
\end{aligned}
$$

We have proved that the abstraction from program E to program E' is a safe approximation. Meaning that by reasoning about program E' we can learn about the behaviour of program E. This makes our analysis approach applicable to the large scale Erasmus programs. In the next chapters we explain how we reason about program E using AI theory.

3.6 Chapter Summary

The goal of this research is to perform a communication analysis on process-oriented programs. Existing domains such as intervals, convex hull and polyhedra, etc [15] are not useful for communication analysis. Therefore we need to create a novel communication abstract domain. This chapter presented a novel communication abstract domain. We introduced a lattice of communication events for both concrete and abstract domains. Transfer operations and fixed-point analysis are also presented.

The lattice presented in this chapter is based on theory of happened before relation of Lamport [39] and Vector Clock. The next chapter discusses this theory and explains advantages and disadvantages of this approach. It then introduces an event order predictor that builds a partial order of synchronous events of a POP program without actually running it.

Chapter 4

DESIGN AN EVENT ORDER PREDICTOR

Following the previous chapter's goal : building a lattice for event communications for POP programs, in this chapter, we propose an Event Order Predictor (EOP) where the events orders are determined statically. Our approach is inspired by Vector Clock and it predicts the order of events for all possible paths.

This chapter is organized as follows: Section 4.1 presents a An Event Order Predictor for POP programs. The implementation details of Event Order Vector is also provided in this section. Section 4.1.4 proposes a Vector Clock for Erasmus cells. Section 5.4 provides a brief summary of the chapter.

In this chapter we use j , k to denote processes. A list of symbols that is used throughout this chapter is given in Table 4.1.

4.1 An Event Order Predictor

Using AI theory for static analysis of POP programs, it is essential to know the order that events happen in POP programs. There has been many research to dynamically define the order of events in distributed systems (i.e : Vector Clock Algorithm, Resettable Vector Clock). While considerable research has been devoted to the use of Vector Clock in dynamic analysis, we have not found any existing research that applies the concept of Vector Clock in the static analysis of concurrent programs. For example, Vo [64] used both Vector Clock [22] and Lamport Clock [39] algorithms to develop a dynamic verifier for Message Passing

Symbols	Explanation
j, k, P, Q, R	processes
e_i	The individual events using different values of i for distinct events
e_i^x	The xth successive occurrences of an event in a loop.
$DE(x, j)$	xth distinguished event of process j
m	messages
e_1 hb e_2	e_1 happened before e_2
q	sequence of events $\langle e_1, e_2, \ldots, e_n \rangle$
VC	vector clock
RVC	resettable vector clock
EO	event order
EOV	event order vector
AI	Abstract Interpretation

Table 4.1: Nomenclature

programs. Cain et al. [7] built an algorithm using Vector-timestamp to verify a multi-threaded program. Resettable Vector Clock (RVC) was proposed by several researchers [61, 44, 49, 58, 4]. The main idea is that instead of estimating the size of the VC; which practically can be either too small or too large; only a necessary number of bits for the size of VC is considered and then every time the counter that stores the value of VC, is about to overflow, VC resets to zero. The reset rule varies in each research, depending on the applicability and purpose of the program. For instance, Mostefaoui and Theel [49] addressed this issue by proposing a reset rule; when VC reaches a predetermined limit, the clocks of the processes resets to zero. The problem of their solution is that it is not possible to compare timestamps of two events that happen before and after resetting the clock and therefore, the causality relation in the application is changed. Arora et al. [4] argue that the existing RVC rules do not satisfy the *fault-tolerance* property. *Fault-tolerance* is the ability of a system to respond gracefully to an unexpected failure. For implementing RVC, when the system is recovered after a failure, the casual relation of events should not be affected by the failure. Arora et al. [4] propose a realistic implementation of RVC that satisfy the *fault-tolerance* property. They suggest to reuse the timestamp of events in phase-based applications. When a process moves from one phase to another, it resets its own local clock. RVC are useful for Erasmus programs with two properties: first, there is a possibility of in-

consistent information because of multiple paths. Second, the program works in "phases", allowing resetting at the end of a phase.

To the best of our knowledge, there has been no previous research that determines the order of events without executing the program. Since some Erasmus applications are also phase-based, we partially adapt the approach of Arora et al. [4] to implement our Event Order Predictor (EOP). Our Event Order Predictor might be also applicable to the formal analysis of other concurrent programs.

4.1.1 Event Order

We define a data structure, inspired by Vector Clock, that contains a counter number for event orders for all (known) processes. The data structure is defined as below:

Definition 4.1. Event Order Vector

Let n be the number of processes in a POP system, C_i is the order event counter predictor for process i for a particular event. The Event Order Vector is represented as an array with an integer order value for all the processes.

$$[C_1, C_2, C_3, ..., C_n]$$

Throughout the whole chapter, we use EOV to refer to the vector of event orders of the processes of a program. There is an entry (counter) in the Event Order Vector for each process of the program. Every time a communication happens, the entry for this process is incremented.

Definition 4.2. EOV Ordering

The comparison of two VOCs follows the same order as Vector Clock timestamp as it is provided in Table 2.4 on page 29.

Definition 4.3. EOV Property

Let a and b be two process events. $EOV(a)$ is referred o the vector order value for event a. Same applies to b. If $EOV(a) < EOV(b)$ then $a \longrightarrow b$. In another word, event a causes

event b if and only if $EOV(a) < EOV(b)$.

The event order value can give us some information about the relation betweens events in the programs. For example, an event a with EOV of [1,0] happens before another event b with EOV of [2,1] (we show a happens before b by: $a \rightarrow b$). Reason: This follows the same property of Vector Clock if Vector Clock of event $a <$ Vector Clock of event b then $a \rightarrow b$. Since $[1, 0] < [2, 1]$ then $a \rightarrow b$. However, this counter value in a long run of the program will overflow. To avoid the problem of counter overflow, we reset the value of the counter in specific conditions to implement a bounded EOV. However, the challenge in implementing EOV for Erasmus application is to guarantee that the order of events will not be affected after resetting the counters. To make sure that events order will not be changed when we reset the vector, the program has to satisfy some specific conditions that we explain in the following sections. The terms used throughout this section are defined in Table 4.2.

Terms	**Definition**
Order bound	The maximum value for EOV, denoted by ℓ.
Phase bound	The maximum number of times that a program is able to reset its counter, denoted by reset.
Time-frame	Maximum time-frame that message m sent from process j, will be received by another process k, denoted by M.
Non-blocking reset	The method is called every time a process moves from one phase to another phase, without blocking other processes.
reset[]	An integer array to keep track of the number of the times each process resets its EOV value.
Client of EOV	A program that every time it performs a *send* or *receive* operations, it also calls the send and receive operations of EOV.
Operations of EOV	These methods are called by the client of EOV. These methods are presented in Algorithms 4.3, 4.4 and 4.5 on page 62.

Table 4.2: Nomenclature

4.1.1.1 Communication Pattern

In order to implement EOV for Erasmus programs the program should behave according to the communication pattern. The pattern says that all messages that originate within the

time-frame reach their destination within the time-frame. Following is the communication pattern.

Listing 4.1: Communication Pattern [4]

$DE(x, j)$: *xth distinguished event of process* j

$(\forall k, j, x : x \in N :$

$(\exists e_k \in k :: (DE(x, j)\ hb\ e_k\ ) \wedge$

$\neg\ (DE(M, j)\ hb\ e_k))$

$\wedge\ (\forall m_k :: ((DE(x, j)\ hb\ Send(m_k))$

$\vee\ (Receive(m_k)\ hb\ DE(M, j)))))$

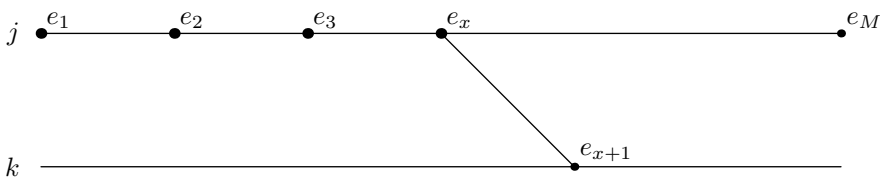


Figure 4.1: Communication Pattern: j and k are two processes. $DE(1, j) = e_1$, $DE(2, j) = e_2$ and $DE(x + 1, k) = e_{x+1}$ We note that $e_1\ hb\ e_2\ hb\ e_3\ hb\ e_x\ hb\ e_{x+1}$. All the events started in process j should be delivered by process k within the time-frame M distinguished events of j.

Definition 4.4. *comm*(M, ℓ)

An Erasmus program satisfies *comm*(M, ℓ) iff (1) it satisfies the communication pattern in Listing 4.1 and (2) between any two adjacent distinguished events, the number of send or receive or local events is less than ℓ, $\ell \in Z^+$

According to [4], any phase-based application that satisfies the above conditions can use the approach providing the suitable values of M and ℓ by considering bounds on the process delay and message delay. In Erasmus , we try to avoid any dependencies on speed and delay, therefore considering bounds on the process speed and message delay is problematic for Erasmus . Instead, we use Abstract Interpretation technique to establish the values of M and ℓ, the conditions for the Theorem 4.1.

Theorem 4.1. *Substitutability of* VC *with bounded-space* RVC

Let P be a client of EOV *and P_D satisfy the communication pattern $comm(M, l)$ for some $M, l \in Z^+$. Then a bounded-space* EOV *is substitutable for VC in P.*

4.1.2 Reset Conditions for Erasmus

In this section, we present our approach in implementing EOV for Erasmus programs. The input of EOV Algorithm 4.3 is two variables: order-bound denoted by ℓ, maximum time-frame for delivery of a message in a process denoted by M. We present our approach in calculating these variables from a given Erasmus code in the following sections. We use the example given in Figure 3.6 to illustrate our approach.

4.1.2.1 Order-bound value

Order bound is the maximum size of the Order Event Vector that a program needs. Algorithm 4.1 shows how to compute this value from a given Erasmus code. The reason we compute the maximum value of order-bound of each processes is because not all the processes have the same number of events. Therefore we need to compute the order-bound for every processes and taking the maximum of these order-bound values is the safest upper bound for the size of the Order Event Vector that is shared between all of the processes.

Algorithm 4.1 Order-bound Computation algorithm

Require: N processes within program E'
Ensure: order-bound of the program : ℓ
1: var order-bound : array [1..N] of integer , var c-bound = 0;
2: **for all** process lists of program $j \in E'$ **do**
3: **for all** c : communication statements $\in j$ **do**
4: order-bound[j]++;
5: **end for**
6: **end for**
7: $\ell = 2$ * getMax(order-bound[], N)

4.1.2.2 Time-Frame

Algorithm 4.2 is written based on the communication pattern that is presented in Listing 4.1. To compute a time-frame in which all messages originated within this time-frame reach to their destination within this time-frame, we count the number of distinguished events

from a send statement in one process to the matching receive in other process(es). The maximum value of these time-frames is a loose upper bound for the value M.

Algorithm 4.2 Time-Frame Computation algorithm

Require: N processes within program E'
Ensure: Time-Frame of the program : M
1: var time-frame : array [1..N] of integer , var M = 0;
2: **for all** process lists of program $j \in E'$ **do**
3: **for all** c : communication statements $\in j$ **do**
4: **if** c: send statement **then**
5: **for all** $\forall$ k: $k \neq j$, $k \in$ process lists of program E' **do**
6: rec = FindMatch(send: c).
7: x: number of distinguish events from c to rec.
8: time-frame[j] = x;
9: **end for**
10: **end if**
11: **end for**
12: **end for**
13: M = getMax(time-frame[], N)

4.1.3 The Reset Rule

Before we explain the reset rule in Erasmus program, we explain the concept of "phase" in Erasmus .

4.1.3.1 Phase-based applications

In phase-based applications, a process executes a phase and begins its next phase when the previous phase is fully completed. Communications is Erasmus programs are synchronized and therefore, the phases should be synchronized. According to [47] phases synchronization should guarantee:

- No process begins its $(k+1)^n$ phase until all the processes have completed their k^n phase $k \geq 0$
- No process will be permanently blocked from executing its $(k+1)^n$ phase if all process have completed their k^n phase $k \geq 0$

We note that the Erasmus phase definition is strictly for processes that contains **loop** and **loop select** statements. We define phase as below:

A phase is a set of distinguished events with respect to other process(es) in a **loop select** statement. When the process finish executing the **loop select** statements, it should goes back to its original state. In another word, each loop execution is a new phase.

Example: in the heating simulation system that is presented in Section 5.2.7 on page 75, process Pump completes one phase when it receives the order value information and sends the temperature details to Thermostat. The next phase is either repeating the same statements in phase one or to execute the second block of the **loop select**.

Listing 4.2: Process Pump in heating simulation

```
Pump = process c: +Clock; m: −Measure; t: −Therm
    loop select
    || time: Integer := c.tick;
       if time % PUMP_INT = 0 then
         t.temp := m.temp
       end
    || c.stop; t.stop; exit
    end
```

Reset Rule: When a process moves from one phase to another, it should reset its counter by calling the nonblocking-reset() method in Algorithm 4.5 on page 62. Algorithms 4.3 on page 62 shows the operations for implementing Event Order Predictor. These algorithms are partially adapted from Arora et al. [4].

Theorem 4.2. *The reset rule suffices to guarantee VC Property.*

Proof. : See [4]. □

4.1.4 Event Order Predictor for Cells

An Erasmus program consists of cells and processes. The cells define the structure of the program, and the processes define its behaviour. The internal representation of cell is generally hidden from from the rest of the program by the cell wall. A cell may contain cells, processes, and variables. Below is an example:

Algorithm 4.3 Event Order Predictor Algorithm [4]

Require: Process list, M, l
Ensure: event order vector value for each process.
var orders:array [1..N] of integer { to store latest event order value of processes.}
var reset:array [1..N] of integer { to keep track of number event order counter resets of processes.}
initially:
(∀ j: $j \neq k$: orders .j[k] := 0 , reset.j[k] := 0)
reset-bound = max($3 * M + 1$)
order-bound = ℓ
if send event (j, e_j , m_j , flag) **then**
    increment-vector(j)
end if
{ attach latest reset vector and event order vector of process j with the message.}
reset.m_j , orders.m_j := reset.j , orders.j;
{ attach latest reset vector and event order of process vector j with the event.}
reset.e_j , orders.e_j := reset.j , orders.j;
if receive event(j, e_j, m_l , flag) **then**
    ($\forall k : k \neq j$): {if m_l has the newer information}
    if (reset.j[k] < reset.$m_l[k]$ ∧ reset.j[k] + M + 1 > reset.$m_l[k]$) ∨ (reset.j[k] > reset.$m_l[k]$ ∧ reset.j[k] ≥ reset.$m_l[k]$ +(reset-bound - M) **then**
        reset.j[k], orders.j[k] := reset.$m_l[k]$, orders.$m_l[k]$;
    end if
else
    if reset.j[k] = reset.$m_l[k]$ **then**
        orders.j[k] := max (orders.j[k], orders.$m_l[k]$) {if m_l and j have the same information}
    else
        {Do not update orders.j[k], reset.j[k]}
    end if
    increment-vector(j)
end if
reset.e_j , orders.e_j := reset.j , orders.j;

Algorithm 4.4 increment-vector [4]

Require: process j
Ensure: increment orders.j[j]
orders .j[j] := (orders.j[j] + 1) mod order-bound

Algorithm 4.5 nonblocking-reset [4]

Require: process j
Ensure: reset the counters at orders .j[j] { orders .j[j] is the order vector for process j at index j.}
reset.j[j] := (reset.j[j] + 1) mod reset-bound {store the number of resets}
orders .j[j] := 0; {Reset the vector}

Algorithm 4.6 happened-before

Require: e_j, e_k

Ensure: boolean: casual relation of events e_j and e_k

1: (reset.$e_j[j]$ = reset.$e_k[j]$ $\wedge$ orders .e$_j[j]$ $\leq$ orders .$e_k[j]$)
2: $\vee$ (reset.$e_j[j]$ < reset.$e_k[j]$ $\wedge$ reset.$e_j[j]$ + n > reset.$e_k[j]$)
3: $\vee$ (reset.$e_j[j]$ > reset.$e_k[j]$ $\wedge$ reset.$e_j[j]$ $\geq$ reset.$e_k[j]$ + m)

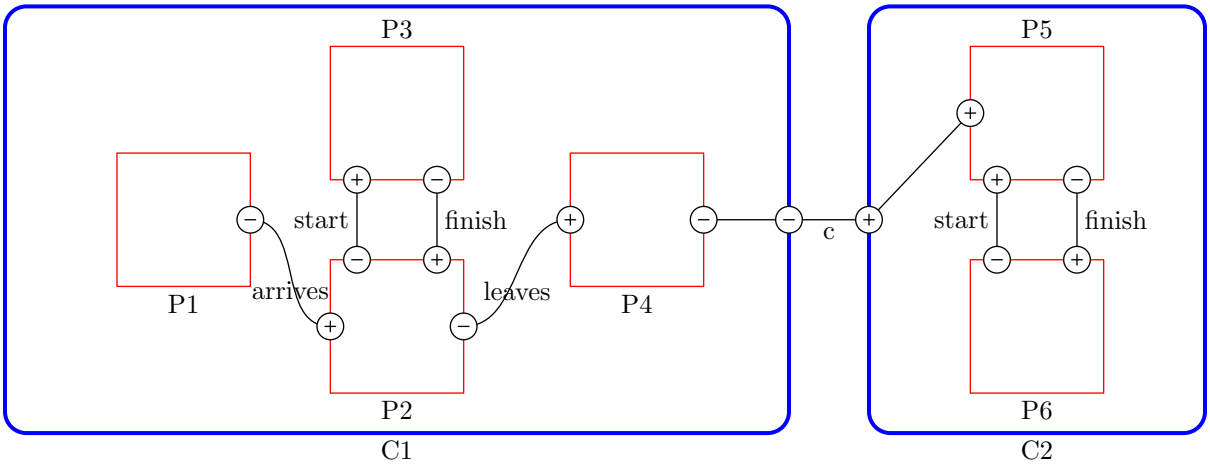


Figure 4.2: Cell ownership in Erasmus.

In the previous section, we assigned an event order for each process of an Erasmus program. Here, we define EOV for Erasmus cell. In Figure 4.2 we have two cells, C_1 and C_2. C_1 contains processes P_1, P_2, P_3 and P_4 whereas cell C_2 has two processes, P_5 and P_6. We define the size of cell EOV the number of processes within the cell. For example, EOV size for Cell C_1 in Figure 4.2 is four, since there are four processes within the cell C_1. Also, the EOV size of C_2 is two.

- State of a cell is EOV value of a the process that is directly communicating with the cell.
- The EOV is represented as a tuple of values. If we have n processes in the cell, we have a tuple of size n.

4.2 Chapter Summary

In this chapter, we discussed the application of Event Order Predictor in POP programs. By using Event Order Predictor we are able to determine the order of events that happen in the program without actually executing it. We have proposed an Event Order Vector specifically for Erasmus programs. The main challenges in implementing EOV is to guarantee that the order of events will not be affected after resetting the counters. The theoretical conditions, implementation details and algorithms were offered in the chapter. We also proposed an Event Order Predictor for Erasmus Cells. In the next chapter, we will explain how we reason about a POP program by using AI theory and Event Order Predictor.

Chapter 5

EVALUATIONAL ANALYSIS

In the previous chapters, we explained AI theory and EOP application for POP programs. In this chapter, we use these two techniques to reason about synchronous Erasmus programs. The chapter is organized as follows: Section 5.1 explains the reasoning steps. Section 5.2 presents an a practical evaluation to evaluate our reasoning methodology. These evaluations are presented in Sections 5.2.1, 5.2.5 and 5.2.7. Section 5.3 compares our reasoning approach with the existing tools and techniques. Section 5.4 provides a summary of the chapter.

5.1 Reasoning

We reason about non-deterministic large Erasmus programs that contain **loop**, **select** and **case**. The methodology of the reasoning are as followings:

- Get an Erasmus code that contains **loop**, **select** and **case**.
- First level abstraction of the code: remove statements that are not related to the communication.
- Second level abstraction of the code: abstract the loop controls.
- Static analysis of the code: compute the values of M and ℓ using Algorithms 4.1 and 4.2.
- Prove the abstraction is safe and the values are applicable to the original code.

- Update EOP operations provided in Algorithm 4.3 with values of M and ℓ.
- Build a lattice of event orders values for the code.
- Compute the fixed-point of the lattice using Algorithm 3.1.
- Inspect the fixed-point to detect communication errors. We can detect a communication violation using vector timestamps by comparing the timestamp of a sender of the message to the timestamp of the receiver of the message. According to vector clock definition, If the sender's timestamp is less than the receiver time vector, a (potential) causality violation has occurred. Properties of the vector colock are explained in Section 2.19 in Chapter 2 of the thesis.

In the following sections we evaluate our reasoning methodology on detecting two properties of a given Erasmus program: successful communication and deadlock in a program.

5.2 Practical Evaluation

5.2.1 Cyclic Communication Pattern

The scenario in this section is developed based on an Erasmus program shown in Figure 5.1. The program has three processes: P, Q and R. These processes are implemented in Listing 5.1 on page 67. Numbers in parentheses at the left denote program points(states). Messages are not specified: we write just p.snd to mean "send on port p and p.rcv to mean "receive on port p. A program is at its original state when its at program point (0). The communications of processes are as followings:

Process P alternately sends a request on channel e_1 to process Q and receives a reply from Q on channel e_4. Process Q receives a request from P and either answers it immediately on channel e_4 or forwards it to process R on channel e_2. Process Q also passes replies from R back to P using channels e_3 and e_4. Process R alternately receives a request on channel e_2 and replies on channel 3. In the next section, we abstract the program E and prove that according to Abstract Interpretation theory, the abstraction is safe.

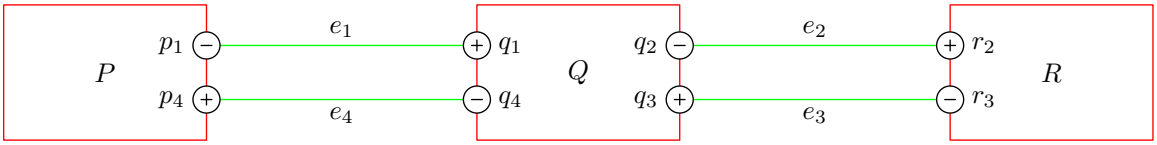


Figure 5.1: A chain of processes, P, Q and R

Listing 5.1: Program E

```
P = process −p1, +p4
{       loop (0)
        {
                p1.snd; (1)
                p4.rcv
        } }
Q = process +q1, −q2, +q3, −q4
{
   loopselect (0)
   {
      || q1.rcv;
        if canAnswer then (1) q4.snd else (2) q2.snd
      || q3.rcv; (3) q4.snd
   }}
R = process +r2, −r3
{       loop (0)
   {
      r2.rcv; (1)
      r3.snd
   }}
```

5.2.2 Abstraction

In chapter 3 we have shown that the abstraction of program E to program E' is a safe approximation. We use program E' to compute the values of M and ℓ to implement bounded EOV for program E. M is the maximum time-frame that message m sent from process P, will be received by another process Q and ℓ is the size of the event order vector. In the next sections we explain our approach for computing these values from an Erasmus program.

5.2.3 EOV Operations

We show that above Cyclic Communication Pattern (program E) satisfy the communication pattern with $M = 2$ and $\ell = 4$

Lemma 1. Program E satisfies the $comm(2, 4)$.

Proof. Program E satisfies $comm(2, 4)$ with $\ell = 4$ because after running the Algorithm 4.1, we see that for every process of program E, there is maximum of two communication statements. Process P has two statements: p1.snd and p4.rcv. Process Q either run q1.rcv; q4.snd or else q1.rcv; q2.snd or q3.rcv; q4.snd. In any case, the maximum number of communication statements that can happen in process Q is 2. Process R has also maximum two communication statements: r2.rcv; r3.snd. Therefore, the upper bound for the size of EOV is 4.

Program E satisfies $comm(2, 4)$ with the time-frame: $M = 2$. We show that for every event in program E all the message that are send within this time-frame will be received before the end of the time-frame. We us the Algorithm 4.2 to compute the time-frame for the delivery of every send in all of the processes. The maximum of these values give us the best upper bound of the time-frame, which is 2 in this case. □

Reset Rule: When all the process finish executing the loop simultaneously, each process should reset its own vector.

5.2.4 Reasoning

To reason about the above program E, without executing the program, we use the EOV operations provided in Algorithm 4.3 to compute the event order values of each processes. Every time there is a *send* or *receive* operations in Erasmus code, the program should call the *send* and *receive* operations of *EOV*.

The Event Order Vector length for the program E is equal to the number of the processes of the program. Therefore, the length of the EOV for program E is three (We have three processes: P, Q and R). Each index of the EOV points to the latest counter value of a

P	Q	R
$P_0 \xrightarrow{e_1} P_0$	$Q_0 \xrightarrow{e_3} Q_1$	$R_0 \xrightarrow{e_8} R_1$
$P_1 \xrightarrow{e_2} P_0$	$Q_0 \xrightarrow{e_3} Q_2$	$R_1 \xrightarrow{e_9} R_0$
	$Q_0 \xrightarrow{e_6} Q_3$	
	$Q_1 \xrightarrow{e_4} Q_0$	
	$Q_2 \xrightarrow{e_5} Q_0$	
	$Q_3 \xrightarrow{e_4} Q_0$	

Figure 5.2: Transition table for processes P, Q, and R

process. This is shown Table 5.1. We define an array $vc : [P, Q, R]$ as a Event Order Vector. Initially the Vector Value for all the processes is $[0, 0, 0]$. We show this initial state with $\langle P_0, Q_0, R_0 \rangle$.

Index	**Process**
0	P
1	Q
2	R

Table 5.1: EOV indexes for program E (Listing 5.1)

.

In the followings, we use the naming conventions that are presented in Table 3.3 on page 52. Example: we use e_1 instead of p1.snd. We look for events that can occur from state $\langle P_0, Q_0, R_0 \rangle$. One possibility is that event e_1 can occur, causing P to transit from state 0 to state 1, then event e_3 causes Q to transit from state 0 to state 1. But Q has two options to do a transition from state 0. The eligible candidates are:

$$P_0 \xrightarrow{e_1} P_1 \qquad Q_0 \xrightarrow{e_3} Q_1 \qquad Q_0 \xrightarrow{e_3} Q_2$$

Figure 5.2 shows all the possible states of the program E. We use this table to analyze the behaviour of the program. In program E processes communicate in two scenarios. We explain both scenarios and reason if communications can happen without any error.

Scenario 1:

Process P send a request to process Q, if process Q is able to answer to the request, it will

send the response to process P. Process P receive the response. All the processes go back to their initial state: $\langle P_0, Q_0, R_0 \rangle$. There is no communication with process R. The path for this scenario is $\langle\, e_1, e_3, e_4, e_2 \,\rangle$.

Fixed-point Computation:

We define an event constraint $c = (v_1, v_2)$ where v_1 is the EOV value of the sender of the event. (e.g. $([P_1, Q_0, R_0])$ and v_2 is the EOV value of its matching receive.(e.g. $[P_1, Q_1, R_0]$). We then define the state of the program as the event trace of the particular program point. The event trace is a sequence of events constraints. Example: $\langle\, (c) \,\rangle$. As it is explained in Definition 3.5, the event traces make a partial order: $\sqsubseteq$.

Now, we define function $F(\mathbf{S})$ where $\mathbf{S}$ is the event trace for processes P, Q, R.

$$F(S) \quad = \quad \{s_0\} \cup \left\{\, s' \mid s \in S \wedge s \longrightarrow s' \,\right\}.$$

The states can only be added, never removed, as we iterate F. Consequently, for any $\mathbf{S}$, we have $\mathbf{S} \sqsubseteq F(\mathbf{S})$. We can use Kleene's theorem to find the least fixed-point of F as $F^{\infty}(\perp)$, where $\perp = ([0, 0, 0])$. The least fixed-point gives, for each program point, the set of all states that can be reached by executing the program. Using Algorithm 3.1 the result of computing the fixed-point of EOV values of the processes is :

$s_0 = \langle\, [0, 0, 0] \,\rangle$
$F^0(s_0) = \langle\, ([1, 0, 0], [1, 1, 0]) \,\rangle$
$F^1(s_0) = \langle\, ([1, 0, 0], [1, 1, 0]), ([1, 2, 0], [2, 2, 0]) \,\rangle$
$S_1 = \langle\, ([1, 0, 0], [1, 1, 0]), ([1, 2, 0], [2, 2, 0]) \,\rangle$

Figure 5.3: Finding the fixed-point of program E- Scenario 1.

Now processes P, Q, R are all simultaneously in a new phase, so, we reset the counter.

Scenario 2:

Process P send a request to process Q, Process Q is not able to response to this request. Therefore, it will forward the request to process R. Process R receive the request and send the response back to process Q. Q forwards this response to P. P receives the response.

All the processes are back to their initial state: $\langle P_0, Q_0, R_0 \rangle$. The path for this scenario is $\langle\, e_1, e_3, e_5, e_8, e_9, e_6, e_4, e_2 \,\rangle$. We calculate the fixed-point of Event Order Vector value of the program using Algorithm 3.1. Therefore, we have:

$s_0 = \langle\, [0,0,0] \,\rangle$
$F^0(s_0) = \langle\, ([1,0,0],[1,1,0]) \,\rangle$
$F^1(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1]) \,\rangle$
$F^2(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1]),([1,2,2],[1,3,2]) \,\rangle$
$F^3(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1]),([1,2,2],[1,3,2]),([1,4,2],[2,4,2]) \,\rangle$
$S_2 = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1]),([1,2,2],[1,3,2]),([1,4,2],[2,4,2]) \,\rangle$

Figure 5.4: Finding the fixed-point of program E- Scenario 2.

Now processes P, Q, R are all in the simultaneously in a new phase, therefore, we reset the counter. Figure 5.5 shows the timestamp for all the event. The combination of S_1, S_2 is the reachable states for the program E in the specified paths. $S = \{S_1 \cup S_2\}$. The next step is to inspect this set to find a potential communication error.

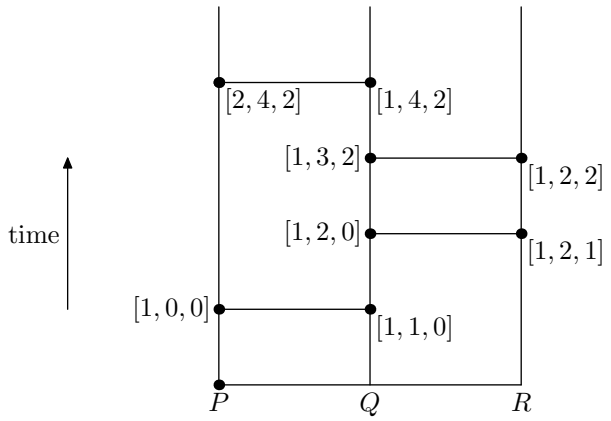


Figure 5.5: Scenario 2: Graph of events of program E.

Analysis:

Scope of analysis: Successful communication:

Condition: All the processes are making progress and for each process in the program, a match for send/receive commands exists in another distinct process(es).In addition, there is no deadlock in the system. These conditions are further explained in Section 6.3 in the

chapter 3 of the thesis. From the final state sets, we infer:

- All the processes have advanced their counter. This means there is no dead process in the program.
- For every pair of sender and receiver of the message, EOV value of sender has been always smaller than EOV value of the receiver. This ensures us that for every send statement, a match of receive exists in another distinct process(es).
- Total number of distinct EOVs in fixed-point set is finite. Therefore, the program terminates.
- The program can run without communication error in the specified two paths.

5.2.5 Cyclic Communication Pattern - Deadlock

No we update only process P of program E. In this version, process P does not require to wait for the reply. It can perform two transitions: it can either send to p_1 ($p1.snd$) or read from P_4 ($p4.rcv$). The code for process P follows:

```
 P = process -p1, +p4
loopselect (0)
 {
|| p1.snd
|| p4.rcv
 }
```

The processes Q and R are the same as listed in Listing 5.1. We name this version of the program: E_1. In this section, we use the same notations that we introduced for previous evaluation. The reset rule is the same as the rule in program E.

Reset Rule: when all the process finish executing the loop simultaneously, each process should reset its own counter.

5.2.6 Reasoning

For this example, the least fixed-point will be the set that contains all of the trace events shown in Figure 5.6 and Figure 5.7. Computing the fixed-point requires computing the successor states of every state that can occur in an actual execution. In particular, we will have to compute the states σ that follow $\langle P_0, Q_0, R_0 \rangle$. In doing so, we will discover two invalid constraints in the third iteration of computing fixed-point: $F^3(s_0)$. These constraints are: $([1,2,2], \emptyset)$ and $(\emptyset, [2,4,0])$. These constraints do not qualify the condition of " timestamp of the sender of the message is always smaller than the timestamp of the receiver. " This state has no successors and is dead-locked states, because

P is waiting to send on q_1 or receive on q_4

Q is waiting to send on q_2

R is waiting to send on q_3

and no process can advance.

$s_0 = \langle\, [0,0,0] \,\rangle$
$F^0(s_0) = \langle\, ([1,0,0],[1,1,0]) \,\rangle$
$F^1(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[2,2,0]) \,\rangle$
$F^2(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[2,2,0),([3,2,0],[3,3,0]) \,\rangle$
$F^3(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[2,2,0),([3,2,0],[3,3,0]),([3,4,0],[3,4,1]) \,\rangle$
$F^4(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[2,2,0),([3,2,0],[3,3,0]),([3,4,0],[3,4,1]),([3,4,2],[3,5,2]) \,\rangle$
$F^5(s_0) = \langle([1,0,0],[1,1,0]),([1,2,0],[2,2,0),([3,2,0],[3,3,0]),([3,4,0],[3,4,1]),([3,4,2],[3,5,2]),$
$([3,6,2],[4,6,2])\rangle$
$S_1 = \langle([1,0,0],[1,1,0]),([1,2,0],[2,2,0),([3,2,0],[3,3,0]),([3,4,0],[3,4,1]),([3,4,2],[3,5,2]),$
$([3,6,2],[4,6,2])\rangle$

Figure 5.6: Finding the fixed-point of program E_1- Scenario 1. The path is $\langle\, e_1, e_3, e_4, e_2, e_1, e_3, e_5, e_8, e_9, e_6, e_7, e_2 \,\rangle$. At the 5th iteration, all the processes finished one phase and they are in a new phase. So the counter should be reset.

$s_0 = \langle\, [0,0,0] \,\rangle$
$F^0(s_0) = \langle\, ([1,0,0],[1,1,0]) \,\rangle$
$F^1(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1]) \,\rangle$
$F^2(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1),([1,3,0],[2,3,0]) \,\rangle$
$F^3(s_0) = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1),([1,3,0],[2,3,0]),([1,2,2],\emptyset)(\emptyset,[2,4,0]) \,\rangle$
$S_2 = \langle\, ([1,0,0],[1,1,0]),([1,2,0],[1,2,1),([1,3,0],[2,3,0]),([1,2,2],\emptyset)(\emptyset,[2,4,0]) \,\rangle$

Figure 5.7: Finding the fixed-point of program E_1- Scenario 2. The path is $\langle\, e_1, e_3, e_5, e_8, e_9, e_6, e_1, e_2 \,\rangle$. Deadlock at timestamp [1,2,2] and [2,4,0].

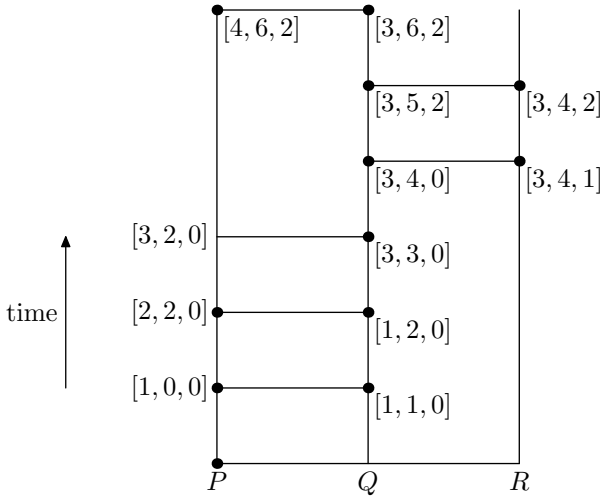


Figure 5.8: Graph of events of program E_1. Since all of the processes are in their original states, we reset the counter.

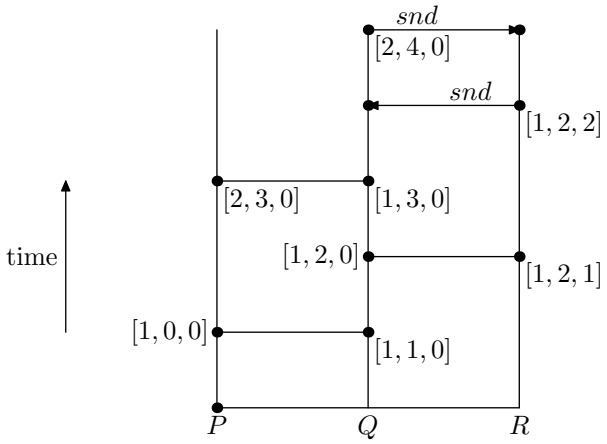


Figure 5.9: Graph of events of program E_1. Deadlock happens at timestamps [1,2,2] and [2,4,0]. P is waiting to send on p_1, Q is waiting to send on q_2.

5.2.7 Heating Simulation: Successful Communication:

We use a Heating simulation program to evaluate our approaches in implementing EOV for Erasmus program and to perform a static analysis on the program. Figure 5.10 shows the overall plan of the system. The system includes seven major processes as followings:

- Clock: sends the time to the three processes that perform a task at regular intervals:

Thermostat, Pump, and Reporter.

- Thermostat : reads the temperature of a room and uses the measurement to turn the furnace on or off
- Pump: simulates a pump that moves water from the furnace to the rooms.
- Furnace: heats the water if it is switched on.
- Room 1: Water flows through the edges m3, t1, t2, and t3. It is heated by the furnace and cools as it gives some of its heat to the rooms.
- Room 2: Different instance of process Room 1, running the same code.
- Reporter requests information from the rooms periodically and stores it in a file.

Client process is a process that first sends a query and then receives a response. In this simulation, processes Clock, Thermostat, Pump and Reporter are from the type client. A minus sign (-) before the protocol name, indicates that the process it is a client. The server has a plus sign (+) before the protocol name. The server first receives the query from the client and then it sends the response. Listing 5.2 presents the code of this simulation written in Erasmus syntax.

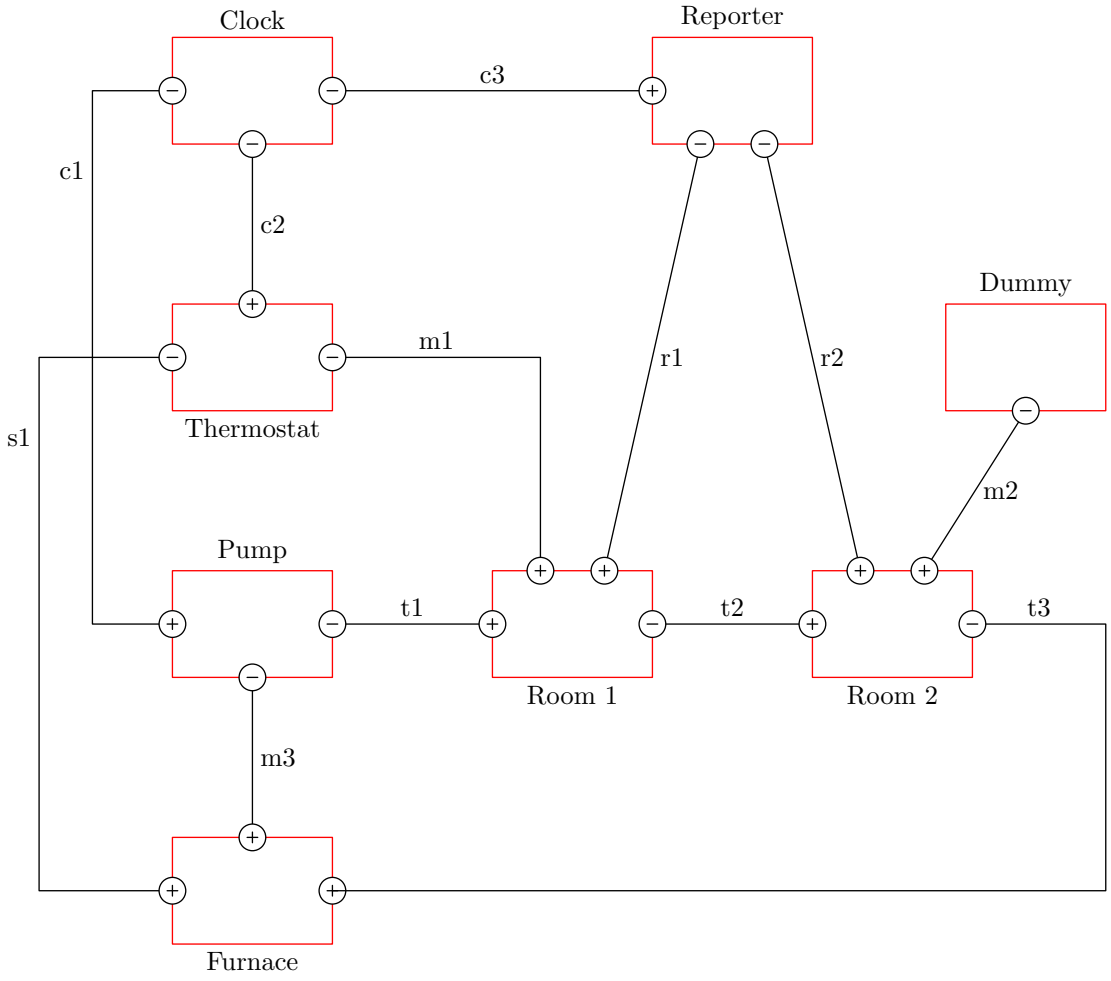


Figure 5.10: Heating Simulation

Listing 5.2: Heating Simulation- Part 1

```
Clock = protocol *tick: Integer; stop end
Print = protocol *↑desc: Text end
Therm = protocol *temp: Float; stop end
Measure = protocol *↑temp: Float; stop end
Switch = protocol *(on | off); stop end
MasterClock = process c1, c2, c3 : −Clock |
        for now in 0 to MAXTIME do
          c1.tick := now;
          c2.tick := now;
          c3.tick := now;
        end;
        c1.stop;
        c2.stop;
        c3.stop
end
Reporter = process c: +Clock; r1, r2, r3: −Print |
        rep: OutputFile := file_open_write (" results .txt" );
         loopselect
        || time: Integer := c.tick;
           if time % REP_INT = 0 then
             msg: Text := format(text time, 5);
              file_write (rep, msg + r1.desc + r2.desc + r3.desc + \n );
             sys.err := msg + r1.desc + r2.desc + r3.desc + \n
           end
        || c.stop;
            file_close (rep);
exit end
end
Thermostat = process c: +Clock; s: −Switch; m: −Measure |
         loopselect
        || time: Integer := c.tick;
           if time % THERM_INT = 0 then
             temp: Float := m.temp;
             if temp < THERM_SET − THERM_TOL
               then s.on
              elif temp > THERM_SET + THERM_TOL
               then s. off
             end
           end
        || c.stop; exit
        end
end
```

Listing 5.3: Heating Simulation- Part 2

```
Pump = process c: +Clock; m: −Measure; t: −Therm |
       loopselect
       || time: Integer := c.tick;
          if time % PUMP_INT = 0 then
            t.temp := m.temp
          end
       || c.stop; t.stop; exit
       end
end
Furnace = process s: +Switch; m: +Measure; t: +Therm; r: +Print |
             temp: Float := 20;
             running: Bool := false;
             state: Bool indexes Char;
             state[false] := " ";
             state[true] := "*";
             loopselect
             || s.on;
                if not running
                  then running := true
                  end
             || s.off;
                if running
                  then running := false
                end
             || m.temp := temp;
                if running
                  then temp += HEAT_INC;
                end
             || temp += WATER_UNIT * (t.temp − temp);
                temp += TANK_ENV * (ENV_TEMP − temp)
             || r.desc := format(temp, 7, 2);
             || t.stop; exit
             end
end
```

Listing 5.4: Heating Simulation- Part 3

```
Room = process name: Text; m: +Measure; p: +Print; pred: +Therm; succ: −Therm |
        temp: Float := 20;
        loopselect
        || m.temp := temp
        || p.desc := format(temp, 6, 2)
        || inp: Float := pred.temp;
           temp += WATER_ROOM ∗ (inp − temp);
           temp += ROOM_ENV ∗ (ENV_TEMP − temp);
           inp += ROOM_WATER ∗ (temp − inp);
           succ.temp := inp
        || pred.stop; succ.stop; exit
        end
end

Control = cell
        c1, c2, c3: Clock;
        MasterClock(c1, c2, c3);
        r1, r2, r3: Print;
        Reporter(c1, r1, r2, r3);
        s1: Switch;
        m1, m2, m3: Measure;
        Dummy(m2);
        Thermostat(c2, s1, m1);
        t1, t2, t3: Therm;
        Pump(c3, m3, t1);
        Furnace(s1, m3, t3, r1);
        Room("Room 1", m1, r2, t1, t2);
        Room("Room 2", m2, r3, t2, t3);
Control()
```

5.2.8 Reset Condition

We show that above Heating Simulation (HS) satisfy the communication pattern with $M = 2$ and $\ell = 8$

Lemma 3. HS satisfies the $comm(2, 8)$.

Proof. HS satisfies $comm(2, 8)$ with $\ell = 8$ because after running the Algorithm 4.1, we see that for every process of HS, there is maximum of four communication statements which gives us the upper bound of 8 for the size of our EOV. HS satisfies $comm(2, 8)$ with the time-frame: $M = 2$. We show that for every event in HS all the message that are send within

this time-frame will be received before the end of the time-frame. We us the Algorithm 4.2 to compute the time-frame for the delivery of every send in all of the processes. The maximum of these values give us the best upper bound of the time-frame, which is 2 in this case. □

Reset Rule: when a process finish executing the loop, it should reset its own counter.

5.2.9 Reasoning

Now we apply the Algorithm 4.3 to calculate the vector counter value for all events of the program. We note that that the length of the EOV is equal to the number of the processes (seven in this case). Each index of the EOV points to the latest counter value of a process. This is shown Table 5.2. We have:

array vc: [Clock, Thermostat, Room1, Furnace, Pump, Room2, Reporter]

Index	**Process**
0	Clock
1	Thermostat
2	Room1
3	Furnace
4	Pump
5	Room2
6	Reporter

Table 5.2: EOV indexes for Heating Simulation system

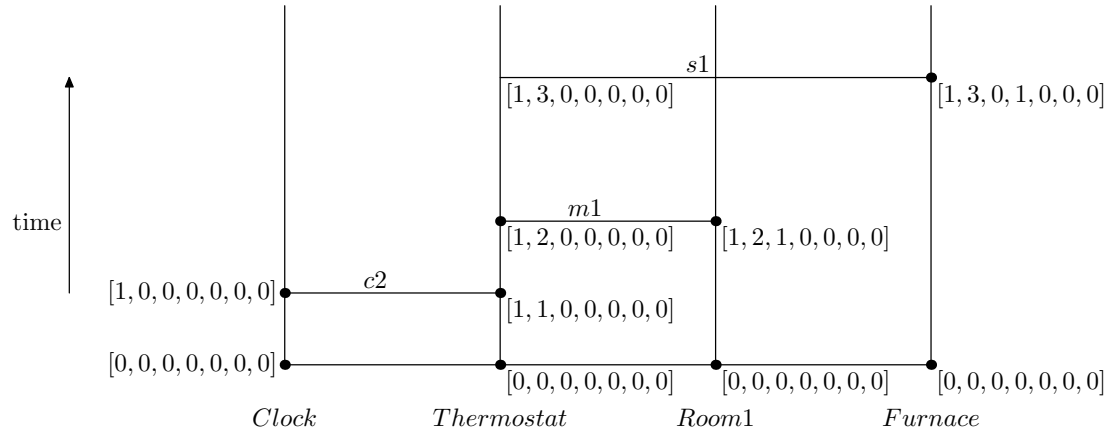


Figure 5.11: Graph of events of Heating Simulation program using EOV. The path is $\langle\, c2, m1, s1 \,\rangle$

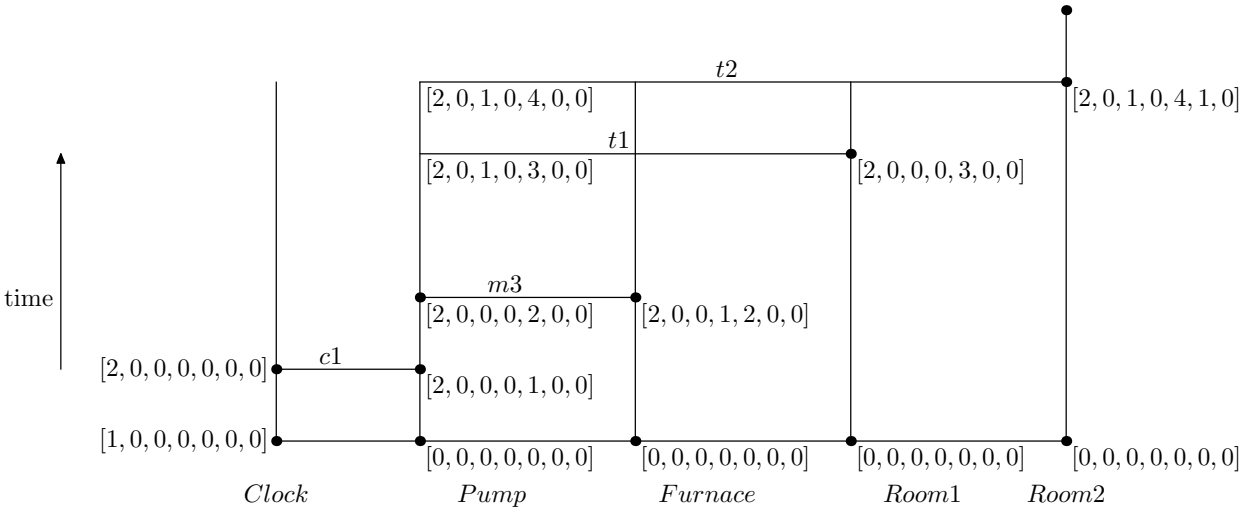


Figure 5.12: Graph of events of Heating Simulation program using EOV. The path is $\langle\, c1, m3, t1, t2 \,\rangle$

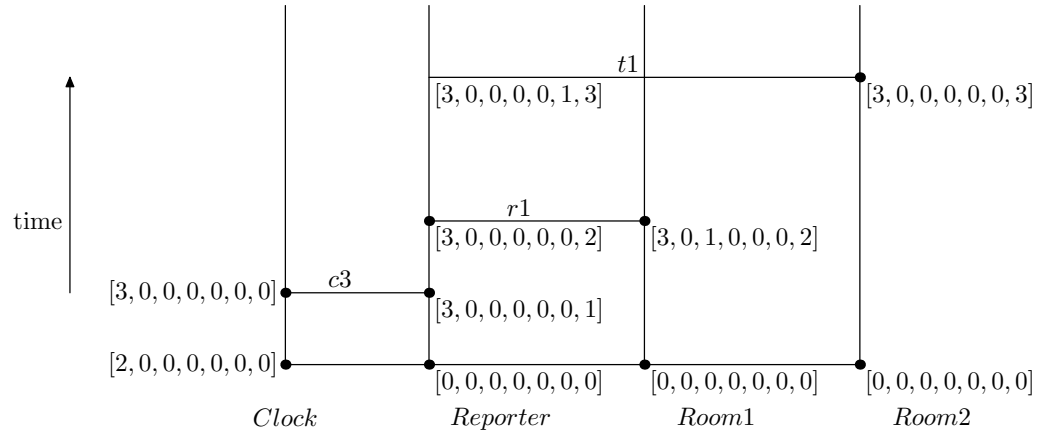


Figure 5.13: Graph of events of Heating Simulation program using EOV. The path is $\langle\, c3, r1, r3 \,\rangle$.

In HS program, events can happen in different orders. Here, we explore this path of events: $\langle\, c2, m1, s1, c1, m3, t1, t2, c3, r1, r3 \,\rangle$. To complete the analysis, all the possible path should be examined. Every time there is send or receive statement in the program, the *EOV operations* will be called. This operations are presented in Algorithm 4.3 on page 62. The next step is to calculate the reachable states of the program using Algorithm 3.1 on page 46. We present the fixed-point result in three phases. When all of the processes finish executing the loop and go back to their original states, they move to a new phase. The reason we present the fixed-point of the system in three phases is to facilitate the understanding of the system and the fixed-point data.

Fixed-points:

Set S_1 contains the fixed-point of EOV of the program in phase 1. The fixed-point computation is shown in Figure 5.14.

$s_0 = \langle\, [0,0,0,0,0,0,0] \,\rangle$
$F^0(s_0) = \langle\, ([1,0,0,0,0,0,0],[1,1,0,0,0,0,0]) \,\rangle$
$F^1(s_0) = \langle\, ([1,0,0,0,0,0,0],[1,1,0,0,0,0,0]),([1,2,0,0,0,0,0],[1,2,1,0,0,0,0]) \,\rangle$
$F^2(s_0) = \langle([1,0,0,0,0,0,0],[1,1,0,0,0,0,0]),([1,2,0,0,0,0,0],[1,2,1,0,0,0,0]),$
$([1,3,0,0,0,0,0],[1,3,0,1,0,0,0])\rangle$
$S_1 = \langle([1,0,0,0,0,0,0],[1,1,0,0,0,0,0]),([1,2,0,0,0,0,0],[1,2,1,0,0,0,0]),$
$([1,3,0,0,0,0,0],[1,3,0,1,0,0,0])\rangle$

Figure 5.14: Finding the fixed-point of HS (1)

The processes Thermostat, Room1 and Furnace call the non-blocking reset method (Algorithm 4.5) to resets their own counter. This is shown in Figure 5.11. Set S_2 has the fixed-point of EOV values of all the events of the program in phase 2. The fixed-point computation is shown in Figure 5.15.

$s_0 = \langle\, [0,0,0,0,0,0,0,0] \,\rangle$
$F^0(s_0) = \langle\, ([2,0,0,0,0,0,0,0], [2,0,0,0,0,1,0,0]) \,\rangle$
$F^1(s_0) = \langle\, ([2,0,0,0,0,0,0,0], [2,0,0,0,0,1,0,0]), ([2,0,0,0,0,2,0,0], [2,0,0,1,2,0,0]) \,\rangle$
$F^2(s_0) = \langle ([2,0,0,0,0,0,0,0], [2,0,0,0,0,1,0,0]), ([2,0,0,0,0,2,0,0], [2,0,0,1,2,0,0]),$
$([2,0,1,0,3,0,0], [2,0,0,0,0,3,0,0]) \rangle$
$F^3(s_0) = \langle ([2,0,0,0,0,0,0,0], [2,0,0,0,0,1,0,0]), ([2,0,0,0,0,2,0,0], [2,0,0,1,2,0,0]),$
$([2,0,1,0,3,0,0], [2,0,0,0,0,3,0,0]), ([2,0,1,0,4,0,0], [2,0,1,0,4,1,0]) \rangle$
$S_2 = \langle ([2,0,0,0,0,0,0,0], [2,0,0,0,0,1,0,0]), ([2,0,0,0,0,2,0,0], [2,0,0,1,2,0,0]),$
$([2,0,1,0,3,0,0], [2,0,0,0,0,3,0,0]), ([2,0,1,0,4,0,0], [2,0,1,0,4,1,0]) \rangle$

Figure 5.15: Finding the fixed-point of HS (2)

The processes Pump, Room1, Room2 and Furnace call the non-blocking reset method to resets their own counter. This is shown in Figure 5.12.
Finally, set S_3 contains the fixed-point of EOV values of all the events of the program in phase 3. The fixed-point computation is shown in Figure 5.16.

$s_0 = \langle\, [0,0,0,0,0,0,0,0] \,\rangle$
$F^0(s_0) = \langle\, ([3,0,0,0,0,0,0,0], [3,0,0,0,0,0,0,1]) \,\rangle$
$F^1(s_0) = \langle\, ([3,0,0,0,0,0,0,0], [3,0,0,0,0,0,0,1]), ([3,0,0,0,0,0,0,2], [3,0,1,0,0,0,2]) \,\rangle$
$F^2(s_0) = \langle ([3,0,0,0,0,0,0,0], [3,0,0,0,0,0,0,1]), ([3,0,0,0,0,0,0,2], [3,0,1,0,0,0,2]),$
$([3,0,0,0,0,1,3], [3,0,0,0,0,0,0,3]) \rangle$
$S_3 = \langle ([3,0,0,0,0,0,0,0], [3,0,0,0,0,0,0,1]), ([3,0,0,0,0,0,0,2], [3,0,1,0,0,0,2]),$
$([3,0,0,0,0,1,3], [3,0,0,0,0,0,0,3]) \rangle$

Figure 5.16: Finding the fixed-point of HS (3)

The processes Reporter, Room1, Room2 and Clock call the non-blocking reset method to resets their own counter. This is shown in Figure 5.13. The combination of S_1, S_2 and S_3 is the reachable states for the program HS in the specified path. $S = \{S_1 \cup S_2 \cup S_3\}$. The next step is to inspect this set to find a potential communication error.

Analysis: We need to check the counter values of every pair of matching send and

receive statement. If in any of the pairs, the EOV of sender is smaller than EOV of receiver, then there is a potential error. By looking at each pair of EOV for sender and receiver of events in S, we observer that all the values of the EOV are greater than 0. This means that all the processes are making progress and there is no dead process. In addition, the condition EOV of sender $<$ EOV of receiver has never become true during our analysis. Therefore, the analysis shows that the communication in the run $\langle\, c2, m1, s1, c1, m3, t1, t2, c3, r1, r3 \,\rangle$ is safe and contains no error. We can observe the other possible paths for the system with the same approach.

5.3 Comparisons

5.3.1 Static analysis of process-oriented programs

There has been much research in the static analysis of computer programs for both sequential and concurrent programs. However, most of the existing research has been carried out on object-oriented programs and is not targeted for message passing or process-oriented languages. Despite the similarities of MPI and Erasmus language, we are not able to use the existing static analyzer tools that are build for MPI programs because of some major communications differences of both languages. The existing MPI analysis tools are mainly targeted to detect communication violations such as deadlock, race analysis and symmetry analysis of message passing programs. As stated in Section 2.4, processes in Erasmus language do not have shared memory, thus race condition cannot occur between processes in Erasmus . Therefore, finding race condition and shared memory related bugs are not relevant to our investigation. To the best of our knowledge, no work has been done on using Abstract Interpretation with Event Order concepts to build a static analyzer for concurrent programs. However, this confirms the novelty of our research and also shows that our results may be applicable to the formal analysis of other concurrent programs.

Cousot [14] applied Abstract Interpretation concepts on some properties of CSP language. However, to our knowledge, the future work of his research — characterizing the other properties of CSP language by fixed point — has not been further developed. To analyze CSP programs, Taylor [60] created an algorithm to calculate the set of all possible pairs

of synchronized processes, but even after reducing the complexity of the algorithm by symbolic execution, his algorithm is still exponential. Later on, Mercouroff [45] used abstract interpretation technique in his algorithm to approximate the set of process communications. However, this abstraction is applied to a number of communications in each channel and is not sufficient to resolve the well-known issue of *state-explosion* problem. Siegel [57] also emphasized this need in the conclusion of his paper about formal analysis of message passing.Our ultimate goal is to create a framework for static analysis of process-based programs such as Erasmus .

5.3.2 Event Order Predictor

The closest work to us in implementing EOP for concurrent languages is Arora et al[4]. We have adapted the RVC operations and some of the proof of theories from their work. However, since our goal is to implement Event Order Predictor for Erasmus and use it for reasoning of the program, there are some major differences between our work and theirs. The major differences follow. In Arora et al [4],

- EOP determines the order of events of a POP program without running the program. RVC is for dynamic detection of order of events.
- The implemented RVC is targeted for asynchronous programs, Erasmus programs are synchronous.
- The time-frame and order-bound values are computed based on the assumption that each each process does the same communications with other processes. In Erasmus every process can have different communications with different number of processes. This has effected our approach in calculating variables of order-bound and time-frame.
- When there is a time-out situation (a large delay), the program calls a global-reset method. In Erasmus we avoid time-out and because we try to avoid any dependencies on speed and delay.

5.4 Chapter Summary

In this Chapter, we offered our reasoning approach to reason about nondeterministic synchronous Erasmus programs that contain **loop**, **select** and **case** statements. We evaluated our reasoning approach in detecting deadlock and successful communication in Erasmus programs. A comparison of our approach with existing tools and techniques is provided at the end of the chapter.

Chapter 6

IMPLEMENTATION

In this chapter, we investigate the possibility of reasoning about deterministic concurrent synchronous programs by Event Order predictor and Abstract Interpretation theory. We use the experiment to understand the EOP concept and the applicability of it for analysis of Erasmus programs. The analysis result detects some communication errors such as circular wait that cause deadlock in deterministic programs with finite number of processes. The static analysis in this chapter is targeted strictly for deterministic Erasmus programs. Generally, concurrent programs are non-deterministic in their execution order, unless the programmer carefully writes a deterministic concurrent program. We partially implement some of the modules for Erasmus static analysis in this chapter.

The chapter is structured as follows: Section 6.1 describes Event Order Predictor program. Section 6.2 explains our Event Order predictor program. Section 6.3 explains the static analysis of Erasmus . Scope of the analysis is provided in this section. In Section 6.3.1, we use an example to illustrate our approach in finding a circular wait in an Erasmus program. Section 6.4 presents our observations from this proof of concept. Section 6.5 concludes the chapter and discusses the future work.

6.1 Event Order Predictor algorithm

The terms used throughout this section are defined in Table 6.1.

Terms	Definition
HB	Lamport[40] happens before relation.
e_i	an individual event.
causality	The relation between two events where the first event (e_1) is the cause and the second event (e_2) is the consequence of the first event. We show this by: $(e_1, e_2) \in$ HB and $e_1 \rightarrow e_2$.
not causal	if e_1 is not the cause of e_2 then: $(e_1, e_2) \notin$ HB and $e_1 \not\rightarrow e_2$.
concurrent	e_1 and e_2 are *concurrent* if and only if $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$.
VC	Vector Clock.
EOV	Event Order Vector.

Table 6.1: Nomenclature

Event Order Predictor(EOP) is an algorithm to generate a partial order of events of POP programs. Unlike Vector Clock algorithm that is used dynamically to detect causality violations, EOP is meant to be used statically. Communications is Erasmus are synchronous and therefore, we are interested to detect synchronous communication errors. In addition, we would like to perform a static analysis –examine the source code and derive some properties of the program–.

For deeper understanding of EOP concept, we apply EOP algorithm to a synchronous Erasmus program. We scan the Erasmus code and every time we find a *send* statement and its matching *receive*, we assign an event order vector value to both statements. The algorithm is shown in Listing 6.1. We note that the sender order value is always smaller than receiver's order value. The example we use is the *barber problem* that is presented in Section 2.4.1 of Chapter 2 of thesis. Figure 6.1 shows the events order in a barber program using using EOP algorithm. The following is the scenario used for the events presented in Figure 6.1:

A Scenario in Barber Example

1. The system has four processes: *CustomerGenerator*, *WaitingRoom*, *Barber* and *Reporter*.
2. The *CustomerGenerator* sends one customer to *WaitingRoom*.
3. *Barber* is asleep so *WaitingRoom* sends one customer to *Barber*.
4. *Barber* receives the customer and performs a hair cut.
5. *Barber* finishes the hair cut and sends a finish message to *WaitingRoom* and goes to sleep.
6. *WaitingRoom* sends the message leave to *Reporter*.
7. *CustomerGenerator* sends another customer to *WaitingRoom*.
8. *WaitingRoom* waits for *Reporter* to receive the message.
9. *WaitingRoom* receives the customer from *CustomerGenerator*.

In above scenario, steps 6 and 7 are concurrent.

Algorithm 6.1 Original Vector Clock Algorithm [27]

Require: Process P_j
Ensure: Vector Clock value of the events of process P_j

```
1: var v:array [1..N] of integer
2: initially (∀ i: i ≠ j: v[i] := 0) ∧ (v[j] := 1)
3: if send event (s, send,t) then
4:       t.v := s.v
5:       t.v[j] := t.v[j]+1
6: end if
7: if receive event(s, receive(u),t) then
8:       for all (i := 1 to N)  do
9:            t.v[i] := max(s.v[i], u.v[i])
10:      end for
11:      t.v[j] := t.v[j]+1
12: end if
13: if  internal event(s, internal, t) then
14:      t.v := s.v
15:      t.v[j] := t.v[j]+1
16: end if
```

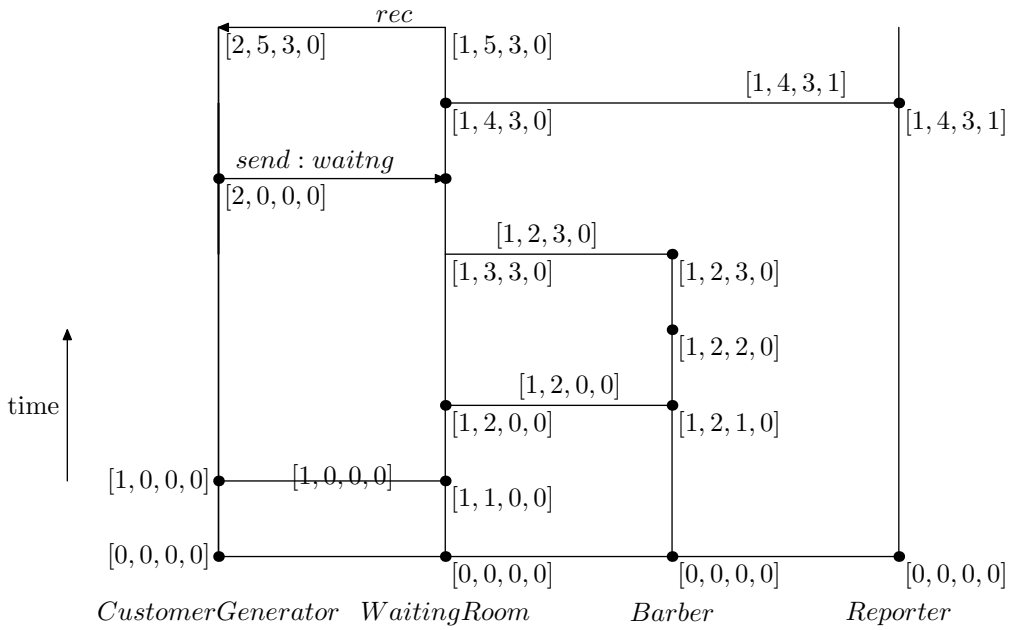


Figure 6.1: Graph of events of barber program using EOP algorithm. Events at [2,0,0,0] and [1,4,3,0] are concurrent. The sender's order value is always smaller than receiver's order value.

6.2 Implementation details

We performed a slight modification to the algorithm. That is, if the communication is successful, the order counter value of *sender* of the message is equivalent to the *receiver*'s value. Our modification to the EOV Clock algorithm is summarized in Rule 1. The rest of the algorithm is as the original algorithm explained in Section 6.1.

Rule 1: when a successful communication happens between two processes, both sender and receiver of the event should update their local order value to the maximum value of their orders. The sender order value is equivalent to the receiver order value.

The main advantage of this approach is that by looking at the events orders generated by Event Order Predictor we can recognize successful communications. And therefore, we can observe the potential communication violations in the program.

We define a Statement class to assign a Order Event value to all the *send* or *receive* operations in a given Erasmus code. Statement class has three attributes: Order Event Vector value, process id, type of the operation. The classes for this implementation are provided in Appendix A of thesis.

We simulate the behaviour of barber problem using the new algorithm. Figure 6.3 shows the result of our simulation of barber Erasmus program using our modified EOV

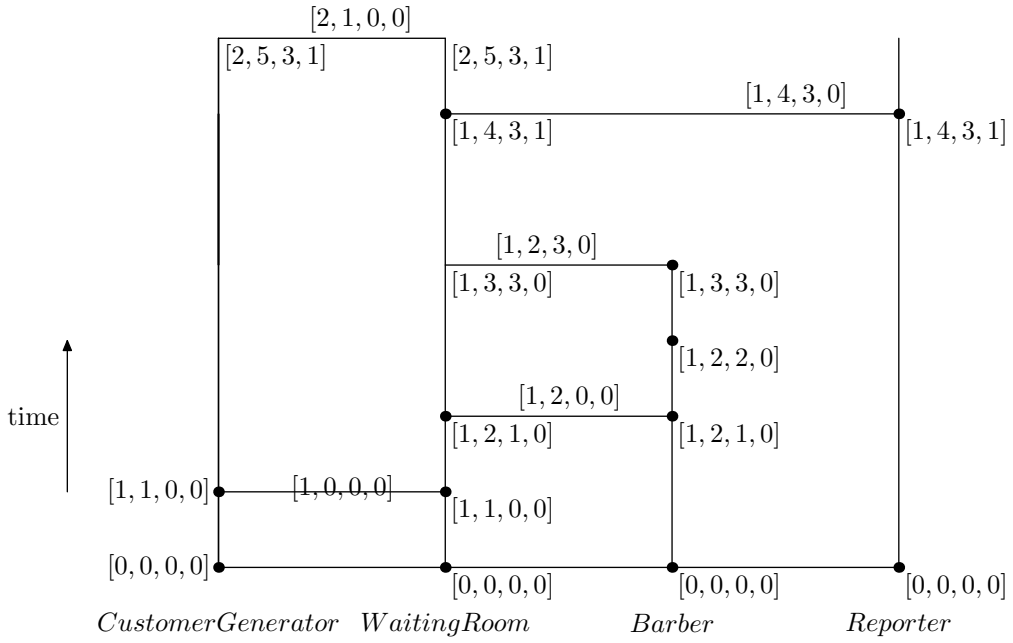


Figure 6.2: Graph of events of barber program using EOP algorithm. If a synchronous communication is successful, order value of the sender is equal to the receiver's order.

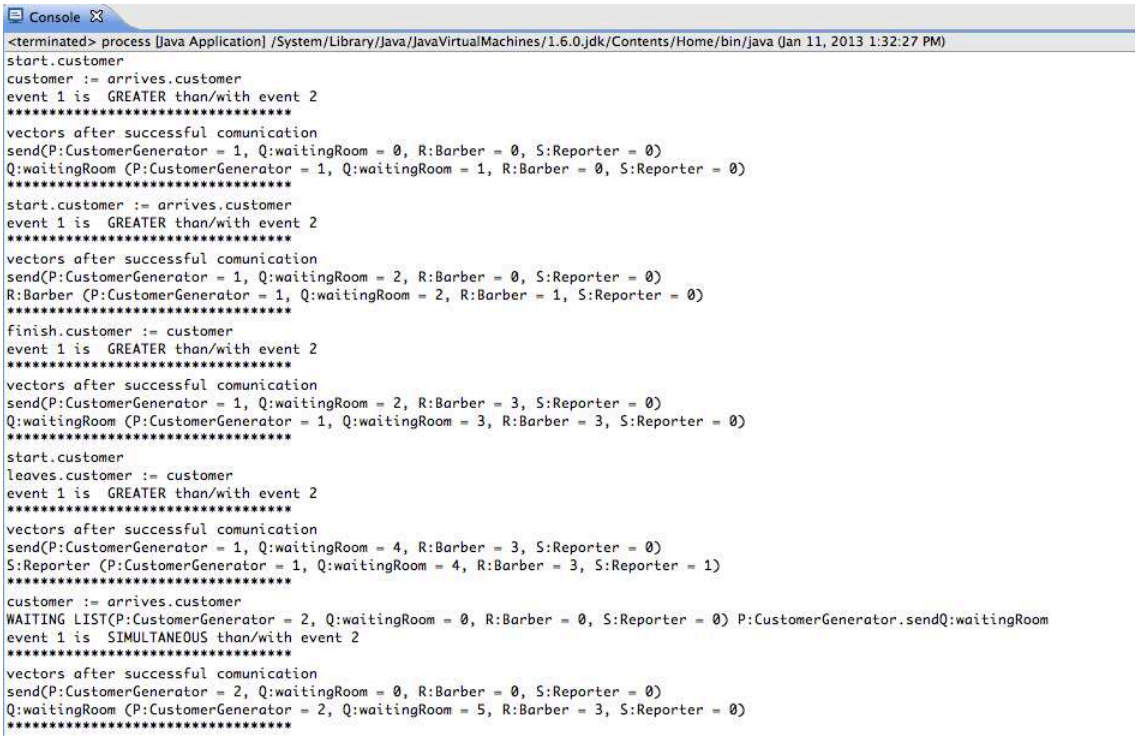


Figure 6.3: Event Order Vector values of events in barber program generated by Analyzer.java program.

6.3 Static analysis of Erasmus

Communications are essential in Erasmus . When one or more processes cannot make any progress, we say deadlock has occurred. Deadlock can happen in Erasmus programs for different of reasons. Table 6.2 shows the situations that potentially cause process(es) to stop making progress in determinist Erasmus programs.

Situation	**Example**
Channel conflict	process A sends a message throughout channel π_1 to process B but process B receives this message from channel π_2.
Type conflict	process A expects a message of type T_1 but process B offers a message of type T_2.
Communication fault	process A waits for process B to respond to its request. Or process B waits for process A to send a request.
Sequence error	process A sends x followed by y, but process B receives y followed by x.
Circular dependency chain	process A_1 waits for process A_2; A_2 waits for . . . A_n; A_n waits for A_1.
Dead process	process A has no communications with other process(es).

Table 6.2: Deadlock Situations in Erasmus

In this analysis we investigate on detecting *dependency chain* and *communication fault* that cause deadlock. Communication fault in an Erasmus program happens when a process waits to receive a message that is never sent or when a process waits for another process to receive its message but the message will be never received. For example, consider an Erasmus program consisting of processes P and Q and assume that the processes are connected to the same channel π. That is, they can communicate. Process P sends an integer number and waits for process Q to receive the message. Process Q also sends a message containing a string value from channel π to process P. In this case, process P is waiting for process Q to receive the message and process Q is waiting for process P to receive its message. None of the processes can make progress. This is a cause for deadlock. This example is shown in Listing 6.1.

To detect this fault we write an algorithm to examine the source code and detect pairs of *send* and *receive* statements that have the same interface specifications. These specifications are: *type, protocol, channel*. If the algorithm finds a *send* statement, it perform a search algorithm on the code until it finds the matching receive. otherwise, it adds the message to *pending list* (vice versa for *receive*). At the end of the analysis, if the *pending list* is not empty, it shows a communication fault.

Listing 6.1: Communication Fault in Erasmus

```
prot = protocol { users: Integer; word: Word}

P = process p1: −prot; p2: +prot
{
        p1.users := 23;   //send
        word := p2.w;   //receive
}
Q = process q1: −prot; q2: +prot
{
        q2.word := "start"; //send
        word := q1.w;   //receive
  }
```

6.3.1 Circular Dependency Chain

Most processes have more than one port and are connected to more than one process. *Circular dependency chain* occurs in systems that contain cycles. Even though, practically not many distributed systems have cycles, but it is still possible for programmers to write a cyclic Erasmus program. Therefore, we need to also check for the possibility of occurring circular wait in these kind of systems. Consider the system shown in Figure 6.4. Th system has three processes: P, Q and R. Processes P and Q are connected to channel C_1. Q and R are connected to channel C_2. and R and P are connected to channel C_3. Therefore, processes can communicate.

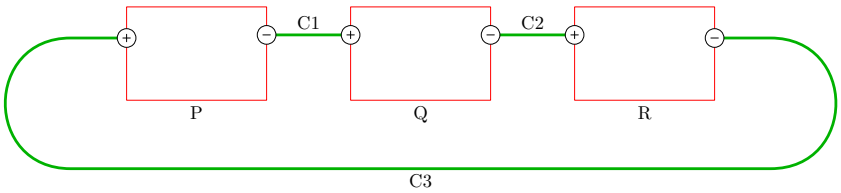


Figure 6.4: Circular wait in program E_1 causes deadlock. None of the processes can be completed since each process is waiting for another process that is also waiting.

We give the code for the processes, showing only the principle control structures and communication. For simplicity, we use CSP notations to describe the send and receive statements; C1!x means "send x on port C_1" and C1?x means "receive x on port p". The program contains three processes. P,Q, R as its shown in Figure 6.5. Each process perform

the following transitions:

- Process P can perform two transitions: it first sends message x to C_1 and then reads message y from C_2.
- Process Q sends message y to C_2 and then reads message z from C_3.
- Process R sends message z to C_3 and then reads message x from C_1.

This program deadlocks because of the circular wait problem. Process P is waiting for process Q to receive a message, process Q is waiting for R to perform receive operation and process R is waiting for process Q to receive message z. So since all of the processes are waiting for other processes to complete an operation, none of the processes can make any progress.

```
P = process { C1 ! x; C3 ? z }
Q = process { C2 ! y; C1? x }
R = process { C3 ! z; C2 ? y }
```

Figure 6.5: Program E_1 with events $P \xrightarrow{x} Q$ and $Q \xrightarrow{y} R$ and $R \xrightarrow{z} P$.

Our analysis to detect *circular wait* by using Event Order Vector in a given Erasmus program, has five direct phases:

- Get the Abstract Syntax Tree of a given Erasmus program.
- Abstract the program; omitting constructs not involving communication.
- Construct a partial order of events from the abstract program using Event Order predictor algorithm.
- Collect program specifications from the Abstract Syntax Tree.
- Interpret the result and detect *circular wait*.

Instead of operating directly on Erasmus code, we operate on Abstract Syntax Tree (AST) of an Erasmus program. While the source code is translated to the AST, the Desi compiler construct Control Flow Graph. Figure 6.6 show all the phases of the analysis. We explain

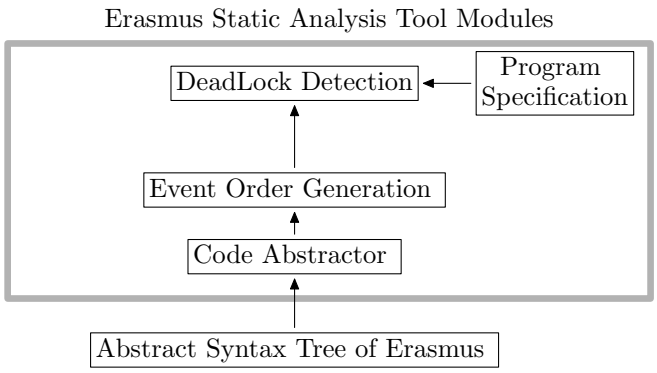


Figure 6.7: Modules of the analysis

each phase in the following sections. Figure 6.7 shows the internal modules of Erasmus Static Analysis Tool (ESAT).

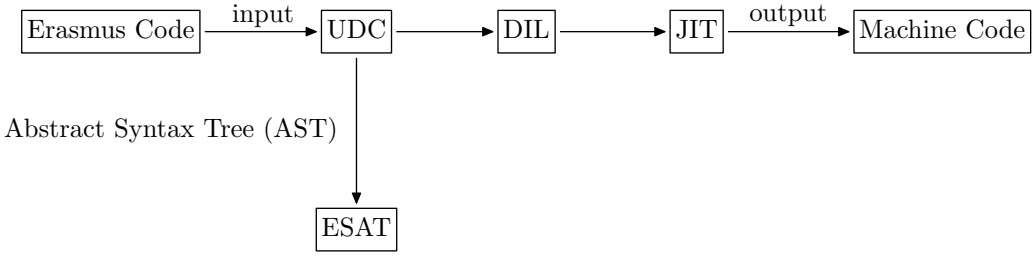


Figure 6.6: Static analysis is performed on Abstract Syntax Tree instead of the source code. UDC : Desi compiler, JIT : DIL compiler ESAT: Erasmus Static Analyzer

6.3.2 Event Order Vector

In this phase, we use the abstract version of the original code that has only communication-related statements. First, we scan the code and then we assign a Event Order value (based on Event Order predictor algorithm) to each statement of the program. The implementation of this algorithm is provided in Listing A.2. The method name is GenerateOrders().

6.3.3 Program Specification

It is necessary to get the program specifications from the source code. The specification we are looking for is specifically the sequence of operations in each process within the program. Later, we check these specifications against the partial order of events generated by event

order value of each statement. We justify the behavioural specification of the program by using Lamport's *Happened Before* Relation (HBR) [39]. According to HBR theory if a and b are two events in the same process, and a comes before b, we say a happened before b and we show this by: $a \rightarrow b$. In program E_1 presented in Figure 6.5 the we define the specification as followings:

Specification: Let T_i, T_i' and T_i'' be sequence of operations in processes P, Q and R without considering their communications with other processes. We define the followings:

$T_1 \equiv C_1!x$

$T_2 \equiv C_3?z$

$T_1' \equiv C_2!y$

$T_2' \equiv C_1?x$

$T_1'' \equiv C_3!z$

$T_2'' \equiv C_2?y$

According to the sequence order of each operations in the program we have:

$T_1 \rightarrow T_2$

$T_1' \rightarrow T_2'$

$T_1'' \rightarrow T_2''$

Event Order Vector : We use the Event Order Predictor program to define the time at which events occur. Using the method GenerateOrders() in Listing A.2 we assign a Event Order Vector value for each operations of the processes P, Q and R of the program E_1, considering their synchronous communications with other processes. According to the Event Order predictor algorithm, when a communication is synchronous, the send and receive

operations happen at the same time. We have:

$$
\begin{aligned}
T_1 \equiv C1!x &\equiv (P=1, Q=2, R=0) \\
T_2 \equiv C3?z &\equiv (P=2, Q=2, R=1) \\
T_1' \equiv C2!y &\equiv (P=2, Q=2, R=2) \\
T_2' \equiv C_1?x &\equiv (P=1, Q=2, R=0) \\
T_1'' \equiv C_3!z &\equiv (P=2, Q=2, R=1) \\
T_1'' \equiv C_2?y &\equiv (P=2, Q=2, R=2)
\end{aligned}
$$

The combination of the sequence order of the operations and the partial order of synchronous events gives a contradiction:

$T_1 \rightarrow T_2$

$T_2' \rightarrow T_1'$

$T_1' \rightarrow T_2'$

$T_1'' \rightarrow T_2''$

This contradicts the Definition 2.15: Happend Before Relation is asymmetric.

$$if\ T_1' \rightarrow T_2'\ then\ T_2' \nrightarrow T_1'\ (asymmetric)$$

Therefore, the specification of the program implies that this equation is false and therefore, there is no possible ordering of the events in time. So the communication is not successful.

6.4 Proof of Concept Results

The modules of this chapter are developed using Java programming language. The implementation goal was to gain deeper insight into the Event Order Vector operations and to investigate the possibility of performing a static analysis on Erasmus program using Event Order Vector. We tested these modules with various deterministic Erasmus programs, each with a finite number of processes. The observations from this experiment are as followings:

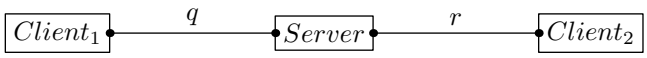


Figure 6.8: Three processes communicating over two channels. $Client_2$ is waiting for *Server* to become available.

Observation 1: This type of analysis is not capable of detecting starvation in the scenario shown in Figure 6.8. The Erasmus code is presented in Listing 6.2. In this scenario we have three processes that communicate over channels r and q. $Client_1$ sends a request to *Server* process. $Client_2$ also sends a request to *Server*. *Server* process uses a loop *select* statement to make a non-deterministic choice of communication between $Client_1$ and $Client_2$. Since there is no policy identifier in the select, the compiler assumes *Ordered* policy. According to *Ordered* policy of Erasmus programs, the first communication (earliest) in a select branch executes first. Consequently, $Client_2$ starves to access the *Sever*.

Listing 6.2: Starvation in Erasmus

```
Client1 = process q: Prot {
        MAX: Integer = 1000;
        loop i := 0;
        {
          | i <MAX|
                w: Word;
                q.w = 'start'
                i +=1;
        }
}
Client2 = process r: Prot {
        w: Word;
        r.w = 'start' ;
}
Server = process q: Prot; r: Prot
        loop select
        w1: Word;
        w2: Word;
        {
            |w1:=q.w;
            |w2:=r.w;
        }
```

Solution: There are three policies for *select* statements in Erasmus programs. These policies are *Fair, Ordered and Random* To avoid starvation in these kind of Erasmus programs, the programmer should use *Fair* for the select policy to give the chance of execution

to all the select branches.

Observation 2: The size of the Event Order Vector is directly proportional to the number of processes in the program which can effect the efficiency of the analyzer.

6.5 Summary

This chapter presented our investigation is using Abstract Interpretation and Event Order Predictor to reason a bout deterministic Erasmus program. The modules of the static analyzer are implemented in this chapter. The next step is to adjust the modules so it can be applicable for non-deterministic Erasmus programs. We believe Using a resettable Vector for implementing Event Order Predictor as it is explained in Chapter 3 will potentially lead us to create an analyzer the overcomes the efficiency limitation, thus being suitable for non-deterministic programs.

Chapter 7

RELATED WORKS

In general, works related to ours can be divided in two categories: research that aim to create new techniques to perform static analysis on concurrent programs; and research directed towards Abstract Interpretation technique. We can also view existing research on the static analysis of message passing programs as complementary to our work. In the following section, we provide a summary of some of the existing static analysis tools and their functionalities. We group them into three major categories: static analysis tools that used Abstract Interpretation technique, static analysis tools targeted for CSP languages and static analysis tools for MPI programs. A summary of these tools is provided in Table 7.1.

7.1 Static Analysis Tools by Abstract Interpretation

While doing our research, we found some static analysis tools for sequential and OOP programs that is built based on Abstract Interpretation. We only list the name and description of few of them shortly here.

- "Astrée" [12] is a static analyzer designed to automatically prove the absence of run time errors in programs written in the C programming language. The analyzer is sound, semantic-based and automatic.

- "Thesee" [46] is another static analyzer for synchronous embedded C software which

is built based on Astrée [12]. The analyzer was successfully applied to prove the absence of run-time errors in large critical control/command software from Airbus. The major analysis is done on threads communicating through a shared memory with weak consistency and scheduled according to strict priorities.

- "Polyspace" [1] is a commercial static analyzer tool that implements Abstract Interpretation. The tool is available for C/C++ and Ada languages. It detects run-time errors directly in code and includes file and class-level software component verification.

7.2 Static Analysis Tools for CSP languages

To the best of our knowledge, these are some of the CSP analyzer tools available:

- "SLAP" [52] is a static live-lock analyzer of processes. The main technique used in this work is to flag the processes that can potentially cause live-lock using Failures-Divergences refinement (FDR).

- "Dialyzer" [16] is an Erlang static analysis tool that detects some software defects such as obvious type errors, unreachable code, redundant tests, unsafe virtual machine byte code, etc. However, it also checks the program for some communication errors. Communication issues arise when a program contains a *message never understood* and messages that may not be handled. The approach used in Dialyzer to find these issues is a *behavioural type system.*

- Christakis et al. [9] designed and implemented an analysis for asynchronous message passing by constructing the communication graph. Their work is integrated to Dialyzer.

7.3 Static Analysis Tools for Message Passing Programs

Message passing have been used more and more for the development of process communications in distributed systems. The communications in MPI are very similar to process-oriented languages like ErasmusṪhis similarity motivated us to review some of the existing

works done in the static analysis of MPI programs. Based on our research partial order reduction and symbolic reasoning are the most used techniques in these existing works. The closest works in this area to us are as follows:

- Vakkalanka et al. [62] and Siegel [55] used partial order reduction to make the program into smaller classes of behaviours before verification. "MPI-Spin" [55] combines model checking and symbolic execution to verify MPI programs. It detects deadlock and other message passing properties related to successful communication.

- Bronevetsky [6] made a compiler analysis framework that extends data to parallel message passing applications on arbitrary numbers of processes. The goal of his framework was to provide extra information about the properties of parallel message-passing applications and to improve the quality of debugging performance.

- Sharma et al. [54] proposed a solution that automatically detects functionally irrelevant barriers in MPI programs using partial order reduction The suggested algorithm named Fib is based on the POE Algorithm[62] with some improvements. Fib is capable of detecting deadlocks in MPI languages.

- Wang et al. [66] created a communicator-sensitive collective communication analyzer to detect synchronization error in MPI programs. Kreaseck et al.[38] also performed a depth analysis of MPI Programs.

7.4 Chapter Summary

In this chapter, we discussed some of the existing static analysis tools in three main categories, including Abstract Interpretation , CSP and MPI programs.

Concurrency Analysis Tools			
Category	Name	Functionality	Technique
OOP	Astrée [12]	Threads Communication	Abstract interpretation
	JULIA [59]	Java Bytecode Analyzer	Abstract interpretation
	CIBAI [42]	Modular Java Analyzer	Abstract interpretation
CSP	DIALYZER [16]	Message Communication	process calculus and type system
	MEB/CEB [41]	Program Slicing	Synchronized Control Flow Graph
	Mercouroff [45]	Communication Errors	Abstract Interpretation
	SLAP [52]	Live-lock	BDD-based and SAT-based
MPI	SPIN [56]	Deadlock, Communication Errors	Model Checking, Symbolic Execution
	Bronevetsky [6]	Communication Errors	CFG

Table 7.1: Summary of some of the existing research in concurrency analysis tools

Chapter 8

CONCLUSION AND FUTURE WORK

This chapter summarizes the research in this thesis by providing conclusions and possible future work. In Section 8.1, we provide the conclusions from the research in this thesis. Section 8.3 reviews some possible future work.

8.1 Research Contributions

This thesis addressed the issue of static analysis of synchronous process-oriented programs using formal methods. More specifically, we focused on static analysis of a process-oriented language called Erasmus̈We developed a static analysis approach by using Abstract Interpretation framework. Since our focus in this research is communication analysis, the existing domains in abstract interpretation theory are not useful for us. Therefore, we offered a novel communication domain. This abstract domain can be added to the existing abstract domains in Abstract Interpretation theory [15].

Next, we offered a lattice for Erasmus programs. Having a lattice makes it easier to interpret the program because we can build our mathematic model based on the lattice and take advantage of existing theorems about lattice. (e.g., fixed-point theory of lattice). An algorithm to compute the fixed-point from a lattice of Erasmus communications was presented in the thesis. To build a lattice for event communications for POP programs, we proposed an Event Order Predictor (EOP) where the events orders are determined statically. This approach is inspired by Vector Clock and it predicts the order of events for all possible

paths. Event Order Predictor may be applicable to the formal analysis of other concurrent programs. We partially implemented EOP for deterministic Erasmus programs. By doing this experiment, we realized that implementing EOP for non-deterministic Erasmus programs is not practically possible. Therefore, we replaced the vector with a Resettable Event Order Vector.

We introduced reset conditions for Event Order Vector and a reset method that should be called when a given program moves from one phase to another phase. By doing this, we were able to set a boundary for the size of the Event Order Vector and overcome the efficiency limitation of our analysis approach, thus being suitable for large programs with loop. To reason about Erasmus programs we proposed a generic approach by using AI and EOP for static analysis of concurrent Erasmus programs that contain **loop**, **select**, **loopselect** and **case** control structures.

8.2 Research Limitations

As Anderson[3] explains in his article about "The Use and Limitations of Static Analysis Tools to Improve Software Quality", metrics to evaluate the static analyzers generally work in opposition to each other and therefore theoretically the possibilities of building an analyzer that would have outstanding precision and excellent *recall* with high *scalability*, given enough time and access to enough processing power, is less likely possible. We need to strikes a proper balance between *soundness* and *precision* and *scalability* in our analysis. In this research, by applying Abstract Interpretation technique, we accept that the ESAT analysis is *sound* but not *complete* . Another limitation of ESAT is it does not handle dynamic creation of processes.

8.3 Directions For Future Research

Our work suggests several directions for future work. These directions are as follows:

8.3.1 Applying AI concepts to Erasmus

One future direction of this research is related to applying AI concepts to Erasmus program. In this research, we have introduced some abstractions for sequential and loop programs. However, discovering more abstractions can reduce the amount of computation required for analysis of the programs. In addition, introducing **widening** and **narrowing** operators in Abstract Interpretation theory based on Cell concept in Erasmus can enhance the analyzer thus being suitable for more complex programs.

8.3.2 Applying EOP concepts to Erasmus

Currently, EOP does the event ordering at the process level. By using the concept of Cell in Erasmus and and allocating the EOP to cells, we can enhance the scalability of the analyzer. Investigating Erasmus Cell with EOP is a possible direction for future work of this research.

8.3.3 Fully Implementation of the Analyzer

We have partially implemented some of the modules of our static analyzer and EOP algorithms. The fully implementation of the approach is a project related to this research that will be carried out in the future. Fully automatic derivation of the Erasmus Static Analysis Tool from the Abstract Syntax Tree of the Erasmus code should be part of the future implementation.

Appendix A

Java Classes for implementation

A.1 OrderVector and Statement Classes

Listing A.1: OrderVector and Statement Classes

```
public class OrderVector extends HashMap<String, Integer> {
        public void incrementOrder(String pUnit);
        public String[] getOrderedIDs();
        public Integer[] getOrderedValues();
        public String toString();
        public Integer get(Object key); }

public class Statement {
        private OrderVector vc;
        private String name;
        private int prcoess_id; }
```

A.2 Event Order Predictor Class

Listing A.2: Event Order Predictor Algorithm

```
1  public class Analyzer {
2  VectorClock updateVector(OrderVector send, OrderVector rec) {
3          OrderVector vqUpdated =  rec;
4          rec = OrderVector.max(send, rec);
5          rec.incrementOrder(rec.p);
6          vqUpdated = rec;
7          logger.info("␣communication␣is␣successful");
8          return vqUpdated;  }
9
10 public void GenerateOrders()
11 {
12     for each process P in program
13                 OrderVector[]: array [1...N] of  integer
14                 vectors[] : array [1...N] of OrderVector
15                 initially: OrderVector[i] = 0
16     for each communication event in P
17         if (event  e is send)
18         {
19                 vectors[i].incrementOrder(P);
20                 Statement send = new Statement(e, i,vectors[i]);
21                 Statement rcv = FindMatch(pending_rcvs, s1);
22                 if (Match is not found)
23                         Add send statement to the pending send list;
24                 else
25                 {
26                         vectors[rec.p] = updateVector(send.vc, rec.vc);
27                         sender.vc = vectors[rec.p].clone();
28                         Remove receive from pending receive list;
29                 }
30         }
31 }
```

Appendix B

Erasmus Grammar

B.1 Terminal symbols

also and andb any assert case cell constant domain else enum exit external false for if iff implicit implies impliesb import in indexes loop mod nand nandb nor norb not op or orb policy private process protocol public range revimp revimpb routine select such that true type xor xorb

Intrinsic NumericLiteral TextLiteral iden

‘#’ ‘%’ ‘%=’ ‘(’ ‘)’ ‘*’ ‘*=’ ‘+’ ‘+=’ ‘,’ ‘-’
‘-=’ ‘->’ ‘.’ ‘..’ ‘/’ ‘//’ ‘//=’ ‘/=’ ‘:’ ‘:=’
‘;’ ‘<’ ‘<<’ ‘<<=’ ‘<=’ ‘<==’ ‘<=>’ ‘<>’ ‘=’
‘==>’ ‘>’ ‘>=’ ‘>>’ ‘>>=’ ‘>>>’ ‘>>>=’ ‘?’ ‘@’
‘[’ ‘[]’ ‘]’ ‘^’ ‘and=’ ‘andb=’ ‘implies=’ ‘impliesb=’
‘nand=’ ‘nandb=’ ‘nor=’ ‘norb=’ ‘or=’ ‘orb=’ ‘revimp=’
‘revimpb=’ ‘xor=’ ‘xorb=’ ‘{’ ‘|’ ‘}’ ‘~’

B.2 Non-terminal symbols

ArithmeticOp *Assert* *AssignOp* *Assignment* *BinaryOp* *BitOp* *BooleanLiteral* *BooleanOp* *Case* *CellDefinition* *ComparisonOp* *Comprehension* *ConstantDefinition* *Declaration* *Definition* *EnumDefinition* *Exit* *ForAny* *Guard* *GuardedSequence* *Import* *Invocation* *Literal* *Loop* *Lvalue* *MapLiteral* *Module* *Parameters* *ProcessDefinition* *ProtocolDefinition* *ProtocolExpression* *RoutineDefinition* *Rvalue* *Select* *Sequence* *ShiftOp* *Signal* *Statement* *TextOp* *Type* *TypeDefinition* *TypeParameters* *UnaryOp* *UnguardedSequence*

B.3 Rules

Module = { [**public** | **private**] (*Import* | *Definition* | *Invocation*) } .

Import = **import** { TextLiteral }‘,’ [‘;’] .

Definition = [**public** | **private**] (*ConstantDefinition* | *TypeDefinition* | *EnumDefinition* | *ProtocolDefinition* | *RoutineDefinition* | *ProcessDefinition* | *CellDefinition*) .

ConstantDefinition = iden [‘:’ *Type*] ‘=’ **constant** *Rvalue* ‘;’ .

TypeDefinition = iden ‘=’ **type** [*Type*] ‘;’ .

EnumDefinition = iden ‘=’ **enum** ‘{’ { iden }‘,’ ‘}’ .

ProtocolDefinition = iden ‘=’ **protocol** ‘{’ *ProtocolExpression* ‘}’ .

ProcessDefinition = iden ‘=’ **process** *Parameters* (*Sequence* | **external** TextLiteral ‘;’) .

CellDefinition = iden ‘=’ **cell** *Parameters* (*Sequence* | **external** TextLiteral ‘;’) .

RoutineDefinition = (iden | **op** (*UnaryOp* | *BinaryOp*)) '=' [**implicit**] **routine** *TypeParameters Parameters* (*Sequence* | '=' Intrinsic ';' | **external** TextLiteral ';') .

Type = iden

| *Type* **indexes** *Type*

| **mod** NumericLiteral .

ProtocolExpression = ['^'] iden [':' iden]

| '*' *ProtocolExpression*

| { *ProtocolExpression* }‹';'

| { *ProtocolExpression* }‹'|'

| '(' *ProtocolExpression* ')' .

TypeParameters = '<' { iden }‹',' '>' .

Parameters = { *Declaration* }‹';' ['->' { *Declaration* }‹';'] .

UnguardedSequence = '{' { *Statement* } '}' .

GuardedSequence = '{' { *Guard* { *Statement* } } '}' .

Sequence = *UnguardedSequence*

| *GuardedSequence* .

Guard = '|' [*Rvalue*] '|' .

Statement = *Sequence*

| *Assert* [';']

| *Assignment* ';'

| *Case*

| *Declaration* ‘;’

| *Exit*

| *ForAny*

| *Invocation*

| *Loop*

| *Select*

| *Signal* .

Assert = **assert** ‘(’ *Rvalue* [‘,’ *Rvalue*] ‘)’ .

Assignment = *Lvalue AssignOp Rvalue* .

Case = **case** [*Rvalue*] *GuardedSequence* .

Declaration = { iden }‹,› [‘:’ [‘+’ | ‘-’ | ‘@’] *Type*] [‘:=’ *Rvalue*] .

Exit = **exit** .

ForAny = (**for** | **any**) { *Comprehension* }**also** [**such that** *Rvalue*] *Statement* [**else** *Statement*] .

Invocation = iden ‘(’ { *Rvalue* }‹,› [‘->’ { *Lvalue* }‹,›] ‘)’ .

Loop = **loop** { *Declaration* }‹;› *Statement* .

Select = [**loop**] **select** [**policy** iden ‘;’] { *Declaration* }‹;› *GuardedSequence* .

Signal = iden { ‘[’ *Rvalue* ‘]’ } ‘.’ *Rvalue* .

Comprehension = ‘(’ *Assignment* ‘;’ *Rvalue* ‘;’ *Assignment* ‘)’

| iden **in** (**domain** | **range**) *Rvalue*

| iden **in** *Type* .

Lvalue = iden { ‘[’ *Rvalue* [‘..’ *Rvalue*] ‘]’ } [(‘.’ | ‘?’) *Rvalue*] .

Rvalue = *Lvalue*

| *Literal*

| *UnaryOp Rvalue*

| *Rvalue BinaryOp Rvalue*

| *Rvalue* **if** *Rvalue* **else** *Rvalue*

| *Rvalue* **in domain** *Rvalue*

| *Invocation*

| ‘(’ *Rvalue* ‘)’ .

Literal = *BooleanLiteral*

| NumericLiteral

| TextLiteral

| *MapLiteral* .

BooleanLiteral = **true**

| **false** .

MapLiteral = ‘[]’ .

AssignOp = ‘:=’

| ‘+=’

| ‘-=’

| ‘*=’

| ‘/=’

| ‘%=’

| ‘//=’

| ‘<<=’

| ‘>>=’

| ‘>>>=’

| ‘and=’

| ‘nand=’

| ‘or=’

| ‘nor=’

| ‘xor=’

| ‘implies=’

| ‘revimp=’

| ‘andb=’

| ‘nandb=’

| ‘orb=’

| ‘norb=’

| ‘xorb=’

| `'impliesb='`

| `'revimpb='` .

UnaryOp = `'+'`

| `'-'`

| `'~'`

| `'#'`

| **not** .

BinaryOp = *ArithmeticOp*

| *ComparisonOp*

| *BooleanOp*

| *ShiftOp*

| *BitOp*

| *TextOp* .

ArithmeticOp = `'+'`

| `'-'`

| `'*'`

| `'/'`

| `'%'` .

ComparisonOp = `'='`

| `'<>'`

| ‘<’

| ‘<=’

| ‘>’

| ‘>=’ .

BooleanOp = **and**

| **nand**

| **or**

| **nor**

| **xor**

| **implies**

| **revimp**

| **iff**

| ‘<=>’

| ‘==>’

| ‘<==’ .

ShiftOp = ‘<<’

| ‘>>’

| ‘>>>’ .

BitOp = **andb**

| **nandb**

| **orb**

| **norb**

| **xorb**

| **impliesb**

| **revimpb** .

TextOp = '//' .

Bibliography

[1] Polyspace static analysis tools [online]. 1994. URL: `http://www.mathworks.com/products/polyspace/index.html` [cited 3/8/13].

[2] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, November 1999.

[3] Paul Anderson. The use and limitations of static-analysis tools to improve software quality. *The Journal of Defense Software Engineering*, 21(6), 2008.

[4] Anish Arora, Sandeep Kulkarni, and Murat Demirbas. Resettable vector clocks. *J. Parallel Distrib. Comput.*, 66(2):221–237, 2006.

[5] Garrett Brinkhoff. *Lattice Theory*, volume 25. American Mathematical Society, 3d edition, 1967.

[6] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Code Generation and Optimization*, CGO '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Harold W. Cain and Mikko H. Lipasti. Verifying sequential consistency using vector clocks. In *Parallel Algorithms and Architectures*, SPAA '02, pages 153–154, New York, NY, USA, 2002. ACM.

[8] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[9] Maria Christakis and Konstantinos Sagonas. Detection of asynchronous message passing errors using static analysis. In *Practical Aspects of Declarative Languages*, PADL '11, pages 5–18, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[11] Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems and Structures*, 37(1):24 – 42, 2011.

[12] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. In M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP '05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, 2005. Springer.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles Of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[14] Patrick Cousot and Radhia Cousot. Semantic analysis of communicating sequential processes. In J.W. de Bakker and J. van Leeuwen, editors, *Seventh International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 119–133, Berlin, Germany, 1980. Springer.

[15] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. Logical abstract domains and interpretations. In *The Future of Software Engineering*, pages 48–71. Springer-Verlag, Heidelberg, 2010.

[16] Fabien Dagnat and Marc Pantel. Static analysis of communications for Erlang. In *In Proceedings of 8th International Erlang/OTP User Conference*, Sweden, November 2002.

[17] Mourad Debbabi, Fawzi Hassaïne, Yosr Jarraya, Andrei Soeanu, and Luay Alawneh. *Verification and Validation in Systems Engineering*. Springer Berlin Heidelberg, 2010.

[18] Alain Deutsch. Static verification of dynamic properties [online]. November 2003. URL: `http://nesl.ee.ucla.edu/courses/ee202a/2005f/papers/Static_Verification.pdf` [cited 2/8/13].

[19] Edsger W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, Technological University, Eindhoven, The Netherlands., 1965.

[20] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[21] Dominic Duggan. *Enterprise Software Architecture and Design: Entities, Services, and Resources*. Wiley-IEEE Computer Society., 2012.

[22] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10:56–66, 1988.

[23] Bryan Fink. Why vector clocks are easy [online]. January 2010. URL: `http://basho.com/why-vector-clocks-are-easy/` [cited 6/8/13].

[24] Robert W. Floyd. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, July 1963.

[25] RobertW. Floyd. Assigning meaning to programs. In *Program Verification*, volume 14 of *Studies in Cognitive Systems*, pages 65–81. Springer Netherlands, 1967.

[26] Center for Assured Software. On analyzing static analysis tools [online]. 2011. URL: `http://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf` [cited 3/8/13].

[27] Vijay K. Garg. *Elements of distributed computing*. Wiley-IEEE Press, New York, NY, USA, 1st edition, 2002.

[28] Andrzej Granas and James Dugundji. *Fixed Point Theory*. Springer, 2003.

[29] David Gries. *The Science of Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[30] Peter Grogono and Brian Shearing. Concurrent software engineering: preparing for paradigm shift. In *Proceedings of the 2008 C3S2E conference*, C3S2E '08, pages 99–108, New York, NY, USA, 2008. ACM.

[31] Christian Haack, Marieke Huisman, and Clement Hurlin. Permission-based separation logic for multithreaded Java programs. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 15:13–23, 2011. URL: `http://doc.utwente.nl/76303/`.

[32] Nevin Heintze and David Mcallester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science*, LICS '97, pages 342–351, 1997.

[33] Fritz Henglein. Simple closure analysis. *DIKU Semantics Report D-193, DIKU, University of Copenhagen, Universitetsparken*, 1, March 1992.

[34] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[35] Ralf Huuck. *Software Verification for Programmable Logic Controllers.* PhD thesis, Christian-Albrechts-Universitat zu Kiel, Kiel, Germany, 2003.

[36] Gary A. Kildall. A unified approach to global program optimization. In *Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[37] Soonho Kong. Bug catching: Automated program verification and testing Abstract Interpretation [online]. November 2011. URL: `http://www.cs.cmu.edu/~emc/15414-f11/syllabus.html` [cited 3/8/13].

[38] Barbara Kreaseck, Michelle Mills Strout, and Paul Hovland. Depth analysis of MPI programs. In *Proceedings of the First Workshop on Advances in Message Passing*, AMP 2010, June 2010.

[39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[40] Leslie Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *Principles of Programming Languages*, POPL '80, pages 174–185, New York, NY, USA, 1980. ACM.

[41] Michael Leuschel, Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. The MEB and CEB static analysis for CSP specifications. In Michael Hanus, editor, *Logic-Based Program Synthesis and Transformation*, volume 5438 of *Lecture Notes in Computer Science*, pages 103–118. Springer Berlin Heidelberg, 2009.

[42] Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 283–298. Springer-Verlag, 2007.

[43] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[44] Sigurd Meldal, Sriram Sankar, and James Vera. Exploiting locality in maintaining potential causality. In *Principles of Distributed Computing*, PODC '91, pages 231–239, New York, NY, USA, 1991. ACM.

[45] N. Mercouroff. An algorithm for analyzing communicating processes. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 312–325. Springer Berlin / Heidelberg, 1992.

[46] Antoine Miné. Static analysis of run-time errors in embedded critical parallel c programs. In *Programming Languages and Systems/ Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 398–418, Berlin, Heidelberg, 2011. Springer-Verlag.

[47] Jayadev Misra. Phase synchronization. In *Information Processing Letters*, volume 38, pages 101–105, Amsterdam, The Netherlands, April 1991. Elsevier North-Holland, Inc.

[48] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* Ph.d. thesis, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1997.

[49] A. Mostefaoui and O. Theel. Reduction of timestamp sizes for causal event ordering. *IRSA*, page 21, November 1996.

[50] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer-Verlag.

[51] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[52] Joel Ouaknine, Hristina Palikareva, A. W. Roscoe, and James Worrell. Static livelock analysis in CSP. In *Concurrency Theory*, CONCUR '11, pages 389–403, Berlin, Heidelberg, 2011. Springer-Verlag.

[53] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[54] Subodh Sharma, Sarvani Vakkalanka, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. A formal approach to detect functionally irrelevant barriers in MPI programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 265–273, Berlin, Heidelberg, 2008. Springer-Verlag.

[55] Stephen F. Siegel. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 44–58. Springer Berlin Heidelberg, 2007.

[56] Stephen F. Siegel and George S. Avrunin. Verification of halting properties for MPI programs using nonblocking operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'07, pages 326–334, Berlin, Heidelberg, 2007. Springer-Verlag.

[57] Stephen F. Siegel and Ganesh Gopalakrishnan. Formal analysis of message passing. In *Verification, Model Checking, and Abstract Interpretation*, VMCAI '11, pages 2–18, Berlin, Heidelberg, 2011. Springer-Verlag.

[58] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1):47–52, August 1992.

[59] Fausto Spoto. The nullness analyser of JULIA. In *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '10, pages 405–424, Berlin, Heidelberg, 2010. Springer-Verlag.

[60] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM*, 26(5):361–376, May 1983.

[61] Francisco Torres-Rojas and Mustaque Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distrib. Comput.*, 12(4):179–195, 1999.

[62] Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *Computer Aided Verification*, CAV '08, pages 66–79. Springer-Verlag, 2008.

[63] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.

[64] Anh Vo. *Scalable Formal Dynamic Verification of MPI Programs Through Distributed Causality Tracking*. PhD thesis, School of Computing, University of Utah, August 2011. URL: `http://books.google.ca/books?id=RIErYAAACAAJ`.

[65] Mitchell Wand. A simple algorithm and proof for type inference. In *Fundamental Informaticae 10*, pages 115–122, 1987.

[66] Panfeng Wang, Yunfei Du, Hongyi Fu, Xuejun Yang, and Haifang Zhou. Static analysis for application-level checkpointing of MPI programs. In *High Performance Computing and Communications*, HPCC '08, pages 548–555, Washington, DC, USA, 2008. IEEE Computer Society.

[67] Tong Lai Yu. Operating system concept and theory (lecture notes) [online]. March 2010. URL: http://cse.csusb.edu/tong/courses/cs660/notes/index.php [cited 3/8/13].